CS 264A – Automated Reasoning: Theory and Applications

Spring 2015
Course Project - Due 11:55pm Sunday, June 7

Description

In class we have examined in depth how modern SAT solvers work. Accordingly, we have seen, among others, a description of a generic algorithmic structure shared by modern SAT solvers that deploys two powerful techniques, *unit resolution* and *clause learning*. In this project you will master your knowledge about SAT solvers by having a hands-on experience. In particular, to quickly build a SAT solver, we will provide you with a new abstraction that utilizes a notion of SAT state and some primitives. Your task will be to implement those primitives, which are not only useful to build a modern SAT solver but also crucial for the efficiency of knowledge compilers and model counters. As such, we will assess correctness and performance of your implementation in the context of a knowledge compiler and a model counter. In the rest of this document we will discuss the details of your task together with the requirements of the project.

What you have to do

Before describing the details of your task, let us recall how a modern SAT solver works. Algorithm 1 presents a modern SAT solver that is based on unit resolution and clause learning. This algorithm starts at *decision level* $L = 1$ and repeatedly performs the following process. A literal ℓ is chosen and added to the decision sequence D, after incrementing the decision level L (we say that ℓ has been decided at level L). After deciding the literal ℓ, unit resolution is performed on $\Delta \wedge \Gamma \wedge D$. If no conflict is found, another literal is decided. Otherwise, an asserting clause α is identified. A number of decisions are then erased until we reach the assertion level of clause α, at which point α is added to Γ. The solver terminates under one of two conditions: either a conflict is found under an empty decision sequence D (Line 5), or all literals are successfully decided (Line 12). In the first case, the input CNF must be unsatisfiable. In the second case, the CNF is satisfiable with D as a satisfying assignment.

Algorithm 1: `SAT`(Δ)

Input: Δ : a CNF
Output: true if Δ is satisfiable; false otherwise

```
1  Γ ← {} // learned clauses
2  D ← ⟨⟩ // decision sequence
3  while true do
4      if unit resolution detects a contradiction in Δ ∧ Γ ∧ D then
5          if D = ⟨⟩ then return false
6          α ← asserting clause for (Δ, Γ, D)
7          m ← the assertion level of α
8          D ← the first m − 1 decisions of D
9          Γ ← Γ ∪ {α} // learning clause α
10     else
11         if ℓ is a literal where neither ℓ nor ¬ℓ are implied by unit resolution from Δ ∧ Γ ∧ D then D ← D; ℓ
12         else return true
```

Function: *decide_literal*$(\ell, S = (\Delta, \Gamma, D, I))$
$D \leftarrow D; \ell$ `// add a new decision to` D
if *unit resolution detects a contradiction in* $\Delta \wedge \Gamma \wedge D$ **then**
  return *an asserting clause for* (Δ, Γ, D)
$I \leftarrow$ literals implied by unit resolution from $\Delta \wedge \Gamma \wedge D$
return *success*

Function: *undo_decide_literal*$(\ell, S = (\Delta, \Gamma, D, I))$
erase the last decision ℓ from D
$I \leftarrow$ literals implied by unit resolution from $\Delta \wedge \Gamma \wedge D$

Function: *at_assertion_level*$(\alpha, S = (\Delta, \Gamma, D, I))$
$m \leftarrow$ assertion level of α
if *there are* $m - 1$ *literals in* D **then return** true
else return false

Function: *assert_clause*$(\alpha, S = (\Delta, \Gamma, D, I))$
$\Gamma \leftarrow \Gamma \cup \{\alpha\}$ `// add learned clause to` Γ
if *unit resolution detects a contradiction in* $\Delta \wedge \Gamma \wedge D$ **then**
  return *an asserting clause for* (Δ, Γ, D)
$I \leftarrow$ literals implied by unit resolution from $\Delta \wedge \Gamma \wedge D$
return *success*

Figure 1: Some SAT solver primitives.

We will next abstract the primitives used in SAT solvers, viewing them as operations on what we shall call a SAT state. Not only these primitives will help you produce a modern SAT solver quickly but also will be used in the context of a knowledge compiler and a model counter.

Definition 1 *A SAT state is a tuple* $S = (\Delta, \Gamma, D, I)$ *where* Δ *and* Γ *are sets of clauses,* D *is a sequence of literals, and* I *is a set of literals.*

Here, Δ is the input CNF, Γ is the set of learned clauses, D is the decision sequence, and I are the literals implied by unit resolution from $\Delta \wedge \Gamma \wedge D$. Initially, the SAT state is equal to $S = (\Delta, \{\}, \langle\rangle, \{\})$.

In this project you will implement some SAT solver primitives that operate on a SAT state. Then using your implementation, you will be able to build a SAT solver, a knowledge compiler and a model counter.

Figure 1 depicts some of the primitives you will implement. Note that the figure only illustrates an abstraction, not suggesting a particular implementation scheme. It is your task to decide on the details. Two key points will be how to implement unit resolution and how to construct an asserting clause.

What we provide in the folder `code/primitives/`

All primitives you need to implement are given in a header file called `sat_api.h`. This file also contains some structures that you must use to manipulate variables, literals, and clauses. You can fill in those structures however you want. But you cannot change their names. Similarly, you cannot change the signature of any function provided in `sat_api.h`. We also provide another file, `sat_api.c`, in which you can find an incomplete implementation of the functions you will implement. When you complete implementing the functions in `sat_api.c`, you can obtain a static `C` library, `libsat.a`, that packages your implementation using the given Makefile. Then all you need will be to test your library in three different settings: SAT solving, knowledge compilation and model counting, which are explained next.

Using the primitives to build a SAT solver

Once you implement the primitives in Figure 1, you can easily build a SAT solver using the pseudocode in Algorithm 2. Notice that Algorithm 2 is recursive. Yet, it uses the same idea as in Algorithm 1.

What we provide in the folder `code/sat_solver/`

We have already written the code for Algorithm 2, which can be found in `main.c`. It is only missing the library `libsat.a` that implements the SAT primitives. We also provide a Makefile. So, once you get the `libsat.a`, you can simply run the `make` command to compile an executable called `sat`, which will decide whether a given CNF file is satisfiable or not.

Algorithm 2: `SAT`(S)

Input: S : a SAT state $(\Delta, \{\}, \langle\rangle, \{\})$
Output: *success* if Δ is satisfiable; $\bot$ otherwise

```
1  ret ← ⊥
2  if unit resolution does not find a contradiction in Δ then
3  |   I ← literals implied by unit resolution from Δ
4  |   ret ← SAT_AUX(S)
5  return ret

6  Procedure SAT_AUX(S)
7  |   find a literal ℓ where neither ℓ nor ¬ℓ are implied in S
8  |   if there is no such literal ℓ then return success
9  |   ret ← decide_literal(ℓ, S)
10 |   if ret is a success then ret ← SAT_AUX(S)
11 |   undo_decide_literal(ℓ, S)
12 |   if ret is a learned clause then
13 |   |   if at_assertion_level(ret, S) then
14 |   |   |   ret ← assert_clause(ret, S)
15 |   |   |   if ret is a success then return SAT_AUX(S)
16 |   |   |   else return ret
17 |   |   else return ret
18 |   else return success
```

Note that we will use DIMACS format for input CNF files. For instance, the following CNF (which is over binary variables $1, 2, 3, 4$)

$$(1 \vee \neg 2 \vee 3) \wedge \neg 1 \wedge (\neg 3 \vee 4)$$

can be represented as follows:

```
c this is a comment line
% this is another comment line, starts with % not lowercase c
0 yet another comment line, this time it starts with 0 (zero)
ccc one more comment line, then we will have the problem line
p cnf 4 3
1 -2 3 0
-1 0
c a comment line is possible in between clause lines
-3 4 0
cc a comment line is possible after the last clause line
% another comment line
0
```

In general, a CNF file may start with a number of comment lines, where each such line must begin with one of lowercase `c`, the percent sign `%`, or the number `0`. Next, we must have what is known as "problem line", which begins with lowercase `p`, followed by `cnf`, followed by the number of variables n, followed by the number of clauses m. This is followed by clause lines. A clause line is defined by listing clause literals one by one, where a negative literal is preceded by a `-` sign. The end of a clause is defined by `0`. Note variables are indexed from 1 to n. There may also be comments in between clause lines, and after the last clause line.

Using the primitives to build a knowledge compiler and a model counter

Another context where you can see the usefulness of your implementation is knowledge compilation and model counting. Similar to building a SAT solver, you will be able to build a knowledge compiler and a

model counter instantly by having implemented the SAT primitives. Indeed, we will test both correctness and performance of your implementation in the context of a knowledge compiler and a model counter. A detailed description of the compiler we will use can be found in [1]. Here we will only give a high-level description of the compiler, which should be enough for this project.

Figure 2 illustrates the general picture that you should be aware of. Accordingly, the compiler takes as input a CNF and a decision vtree for the CNF. Then, it compiles the given CNF into a Decision-DNNF by utilizing some SAT solver primitives. A pseudocode of this compilation procedure is given in [1]. One can easily convert this compilation algorithm to (weighted) model counting algorithm.

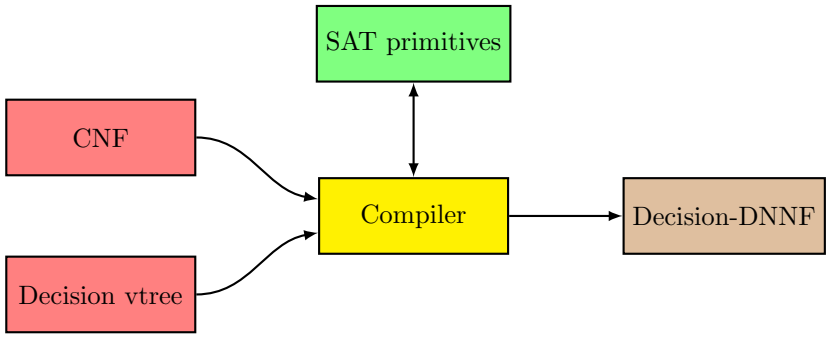

Figure 2: The pipeline of the compiler.

What we provide in the folder `code/c2D_code/`

This directory contains the code for an upcoming open-source package that can be used for both knowledge compilation and model counting. The package only misses the library `libsat.a`. Similar to building a SAT solver, all you need to do is to link your library `libsat.a` with the provided code and test your implementation using the `c2D` package.

What we provide in the folder `code/benchmarks/`

Some CNF benchmarks that you can use to test your implementation, which contain challenging instances as well as easy and moderate ones. In the evaluation, we will also use some other benchmarks.

What we provide in the folder `code/executables/`

Two sample executables: `sat` for the SAT solver, and `c2D` for the knowledge compiler/model counter. We obtained those executables by just linking our own SAT library `libsat.a` with what we provide in this project. The executable `sat` just takes a CNF file, and outputs `SAT` if the CNF is satisfiable, `UNSAT` otherwise. For the executable `c2D`, you can get help about its usage by running the help command (i.e., `$ ./c2D -h`). To compile a CNF into a Decision-DNNF, you will typically prompt the following:

```
$ ./c2D -c <cnf_file> -C -E
```

Here the optional `-c` specifies the input CNF file, which must be written in DIMACS format. The optional `-C` is specified to count the models of the resulting NNF, and the optional `-E` is specified to check if the compiled NNF entails the input CNF. You can think of those two optionals as a way of sanity checking. Also, during its execution, `c2D` first constructs a decision vtree. Although the details of constructing decision

vtrees are not important, you should at least have a way to generate such vtrees. For that, we provide you with different options, which can be seen by running the help command. You may want to play with those options to recognize the effect of vtree construction in the compilation process.
As mentioned earlier, you can use `c2D` as a model counter directly (i.e., without compiling it into a Decision-DNNF, which is known to be tractable for model counting). For that, you can run the following command:

```
$ ./c2D -c <cnf_file> -W
```

where the optional `-W` tells `c2D` to count the models of the CNF.

You can use the sample executable to get a sense of how fast your implementation is. Indeed, `c2D` is a highly-optimized executable. So, it is fine if your implementation does not turn out to be as efficient as `c2D`.

Time frame

We suggest the following time frame.

Weeks	Tasks
Week 4	Familiarize yourself with the project
Week 5	Implement constructing SAT states
Week 6	Implement unit resolution/clause learning
Week 7	Implement unit resolution/clause learning
Week 8	Finish implementing the primitives in `sat_api.h`
Week 9	Testing/Optimization
Week 10	Testing/Optimization and write the project report

Grading

Your project grade will be based on the following three criteria.

Correctness (60%) The correctness of your implementation will be checked in the context of a knowledge compiler/model counter. For that, we will look at the structure of the constructed circuits and also their model counts. To do that, we will run your implementation as explained above (either as a knowledge compiler or a model counter).

Performance (30%) The performance of your implementation will be determined in the context of a knowledge compiler/model counter by a class-wide competition. Each implementation that passes the correctness test will enter the competition against other correct submissions. The results will be based on how fast your implementation is.

Report (10%) You should write a report that is no longer than 5 pages. Your project report should contain the following:

(a) Description of key points of your implementation for SAT solver primitives. In particular, you should describe which data structures you used, and how you implemented unit resolution and clause learning.

(b) An experimental evaluation of your implementation on the benchmarks provided in the directory `code/benchmarks/sampled/`. For each CNF instance, you should report compilation time (with 1-hour time limit), the model count of the CNF, and the size of the constructed NNF circuit (node and edge counts).

Submission & Rules

(1) You should submit a single zip file containing your project report (a pdf) and your implementation for SAT primitives (in a folder called `primitives`) through CCLE.

(2) When we get your implementation, we will create a static `C` library called `libsat.a` that implements the SAT solver primitives. For that, you must provide a Makefile. You can use the Makefile we provide (in the directory `code/primitives`). In case creating the library requires more than a simple `make` command, you must provide a README file that explains in detail how to obtain `libsat.a` from your source code.

(3) The code we provided should run under both Linux and Mac OS. However, we will build your project using `gcc` compiler on a 64-bit **Linux** machine using the `-std=c99`, `-O2`, and `-finline-functions` optionals. To get any correctness or performance points, the source code you turn in must successfully run under those settings.

(4) There is no late submission.

(5) You can either work individually or form a team of size two. In the latter case, exactly one member of the team should submit a solution, and the project report must contain names of both team members. The deadline to declare teams already past due.

References

[1] Umut Oztok and Adnan Darwiche. A Top-Down Compiler for Sentential Decision Diagrams. In *IJCAI*, 2015. To appear.