CS 264A – Automated Reasoning: Theory and Applications

Spring 2015
Course Project - Due 11:55pm Friday, June 5

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 model counters and knowledge compilers. As such, we will assess correctness and performance of your implementation in the context of a knowledge compiler. 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: $\Delta$ : a CNF
Output: true if $\Delta$ is satisfiable; false otherwise
1  $\Gamma \leftarrow \{\}$ // learned clauses
2  $D \leftarrow \langle\rangle$ // decision sequence
3  while true do
4      if unit resolution detects a contradiction in $\Delta \wedge \Gamma \wedge D$ then
5          if $D = \langle\rangle$ then return false
6          $\alpha \leftarrow$ asserting clause for $(\Delta, \Gamma, D)$
7          $m \leftarrow$ the assertion level of $\alpha$
8          $D \leftarrow$ the first $m-1$ decisions of $D$
9          $\Gamma \leftarrow \Gamma \cup \{\alpha\}$ // learning clause $\alpha$
10     else
11         if $\ell$ is a literal where neither $\ell$ nor $\neg\ell$ are implied by unit resolution from $\Delta \wedge \Gamma \wedge D$ then $D \leftarrow D;\ell$
12         else return true
```

```
Function: $unit\_resolution(S = (\Delta, \Gamma, D, I, \alpha))$
$I \leftarrow$ literals implied by unit resolution from $\Delta \wedge \Gamma \wedge D$
if unit resolution detects a contradiction then
    $\alpha \leftarrow$ an asserting clause for $(\Delta, \Gamma, D)$
    return false
return true
```

```
Function: $undo\_unit\_resolution(S = (\Delta, \Gamma, D, I, \alpha))$
erase implied literals in $I$ at the highest decision level
```

```
Function: $decide\_literal(\ell, S = (\Delta, \Gamma, D, I, \alpha))$
$D \leftarrow D; \ell$ // add a new decision to $D$
if $unit\_resolution(S)$ = true then return true
else return false
```

```
Function: $undo\_decide\_literal(S = (\Delta, \Gamma, D, I, \alpha))$
erase the last decision $\ell$ from $D$
$undo\_unit\_resolution(S)$
```

```
Function: $add\_asserting\_clause(S = (\Delta, \Gamma, D, I, \alpha))$
$\Gamma \leftarrow \Gamma \cup \{\alpha\}$ // add learned clause to $\Gamma$
if $unit\_resolution(S)$ = true then
    $\alpha \leftarrow nil$ // clear the conflict
    return true
else return false
```

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

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.

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

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

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 and a knowledge compiler.

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 `satapi.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 `satapi.h`. We also provide another file, `satapi.c`, in which the functions you will implement are explained in detail. When you complete implementing the functions in `satapi.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 two different settings: SAT solving and knowledge compilation, 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 missing the library `libsat.a` that implements the SAT primitives and also another function explained in `main.c`. We also

Algorithm 2: `SAT`(S)

```
    Input: $S$ : a SAT state $(\Delta, \Gamma, D, I, \alpha)$
    Output: true if $\Delta \wedge \Gamma \wedge D$ is satisfiable; false otherwise
1   $ret \leftarrow$ false
2   if $unit\_resolution(S)$ then $ret \leftarrow$ SAT_AUX(S)
3   $undo\_unit\_resolution(S)$
4   return $ret$

5   Procedure SAT_AUX($S$)
6       find a literal $\ell$ where neither $\ell$ nor $\neg\ell$ are set
7       if there is no such literal $\ell$ then return true
8       $ret \leftarrow$ false
9       if $decide\_literal(\ell, S)$ then $ret \leftarrow$ SAT_AUX($S$)
10      $undo\_decide\_literal(S)$
11      if $ret$ = false then
12          if $at\_assertion\_level(S)$ and $add\_asserting\_clause(S)$ then return SAT_AUX($S$)
13          else return false
14      else return true
```

provide a Makefile. So, once you get these implementations, you can just type `make` to compile an executable called `sat`, which will decide whether a given CNF file is satisfiable or not.

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
c each comment line starts with lowercase c
ccc one more comment line, then we will have the problem line
p cnf 4 3
1 -2 3 0
-1 0
-3 4 0
cc a comment line is possible after all clauses are defined
% even a redundant stuff which is not a comment line is possible
```

More generally, a CNF file may start with a number of comment lines, where each such line must begin with lowercase "c". 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 m clause lines. There cannot be any comments in between clause lines. There may be some redundant stuff after m clause lines. A clause line is defined by listing clause literals one by one, where a negative literal is preceded by a minus sign. The end of a clause is defined by `0`. Note variables are indexed from 1 to n.

Using the primitives to build a knowledge compiler

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

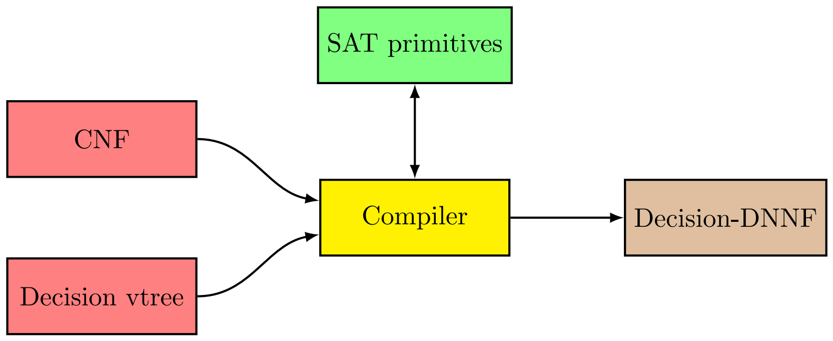

Figure 2: The pipeline of the compiler.

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].

What we provide in the folder `code/knowledge_compiler`

This directory will be completed later and will have three main components.

Compiler An incomplete compiler implementation that 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 compiler.

Benchmarks Some CNF benchmarks (`iscas85` and `iscas89`). You can test your implementation using these benchmarks, which contain challenging instances as well as easy and moderate ones. In the evaluation, we will also use some other benchmarks.

Executable A sample executable (`c2d`) for the compiler. This is for you to see what it would look like when you complete the implementation. To get help about various options of `c2d`, you can type the following:

```
$ ./c2d -help
```

To compile a CNF file into a Decision-DNNF, you will typically prompt the following:

```
$ ./c2d -in_cnf <cnf_file> -in_vtree <vtree_file>
```

Here the optional `in_cnf` specifies the input CNF file, which must be written in DIMACS format. The optional `in_vtree` specifies the vtree file. Although the details of constructing decision vtrees are not important, you should at least have a way to generate such vtrees. For that, we will provide you with an executable `cnf2vtree`, which will construct a decision vtree for a given CNF. You can find the details on this by typing the following:

```
$ ./cnf2vtree -help
```

You can also use the sample executable to get a sense of how fast your compiler is. Indeed, `c2d` is a highly-optimized compiler. So, it is fine if your compiler is not 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 `satapi.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. For that, we will look at the structure of the constructed circuits and also their model counts. To do that, we will run your compiler as follows:

```
$ ./c2d -in_cnf <cnf_file> -in_vtree <vtree_file> -check_entailment -count
```

Here, the optional `check_entailment` checks if the constructed NNF is indeed decomposable and also entails the input CNF. The optional `count` counts the number of models of the NNF. To see if your compiler correctly counts the number of models of a CNF, you can simply compare your result against the result produced by the sample executable.

Performance (30%) The performance of your implementation will be determined in the context of a knowledge compiler by a class-wide competition. Each compiler that passes the correctness test will enter the competition against other correct submissions. The results will be based on how fast your compiler 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 compiler on benchmarks that we will specify later. For each CNF instance, you should report compilation time, model count of the CNF, and size of the constructed NNF circuit.

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) We will build your project using `gcc` compiler on a 64-bit linux machine. 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 is Friday, May 1^{st}, 11:55pm, by email.

References

[1] Umut Oztok and Adnan Darwiche. A Top-Down Compiler for Sentential Decision Diagram. In *IJCAI (to appear)*, 2015.