DCC831 Theory and Practice of SMT Solving
2024.2

Course Project

Due: Fri, Jan 24, 2025

This is assignment is to be done in teams of 2 people for undergraduate students and individually for post-graduate students. The grade for the assignment will be given on an individual basis. Team members will be asked to submit an evaluation of how well they and their teammate performed. Each evaluation is confidential and will be incorporated in the calculation of the project grade.

You can pick your project from those listed below, or you can propose your own as long as it has a similar level of difficulty. In this case you will also be responsible for providing a reasonable set of test cases, again similarly to the projects specified below.

All three projects below involve the implementation of an automated solver. Read their description in order since descriptions for one project may refer to the description of the preceding projects. Correctness – more appropriately, soundness – of the solver is a necessary requirement for all submissions. We will run your solver on additional benchmarks as needed to test its soundness. Solver completeness is also a requirement for all projects. However, the more ambitious ones will be treated more leniently with respect to completeness and performance.

The project descriptions below are meant to get you started. Further information on them can be provided as needed.

Project 1: Implement CDCL with two watched literals

Requirements

For this project, you are to write a SAT solver that implements Conflict-Driven Clause Learning (CDCL). We have studied CDCL as a proof system with the rule set Propagate, Pure, Decide, Conflict, Explain, Backjump, Fail, Learn, Forget, and Restart. You should now be ready to write a program that can do the same for much larger formulas in a completely automated fashion. Your SAT solver should accept a text file in DIMACS CNF format. Its output should be the SAT competition variation of the format. Specifically, it should be a sequence of two lines. The first line should be `s SATISFIABLE` or `s UNSATISFIABLE`. The next line should be blank unless the first line is `s SATISFIABLE`. In that case, it should be `v` M, where M is a space-separated list of non-contradictory literals, where each literal is just a positive or negative (non-zero) integer.

To bridge the gap between abstract and practical CDCL, you will have to put some thought into how you want to store and manipulate the clause set Δ as well as the assignment M. There are many open-source SAT solvers, written in different languages: MicroSat (written in C), MiniSat (written in C++), Sat4J (written in Java), as well as numerous less well-known implementations in others languages such as Python. You are free to implement the SAT solver in your language of choice so long as it is not too esoteric. Please check *beforehand* if you are unsure about whether a specific language would be accepted.

Your solver should have at least the following features:

1. support for the input and output format mentioned above;
2. CDCL solving with clause learning and non-chronological backjumping;
3. an efficient implementation of unit propagation using *two watched literals* (see below).

It can be a bit challenging to implement a full CDCL solver without a reference implementation. There is a Python implementation online that meets most of our requirements and appears to be a concise and well-documented project. You are allowed to take inspiration from that project or other CDCL implementations you may find. If you do that, however, you must credit the source and also implement at least one additional feature with respect to those listed above. The additional feature should be one that requires a non-trivial modification of the source material. Examples include (but are not limited to):

- the Variable State Independent Decaying Sum (VSIDS) heuristic, to pick an unassigned variable for Decide;
- a Luby policy, or an arithmetic policy for Restart;
- a time and space-efficient implementation using Numpy arrays instead of less efficient implementations such as dictionaries or maps.

If you have an idea for some other modification in alternative to the three above, please propose it to the instructor and get their approval before you start implement it.

Testing and evaluation

The SATLIB website has a number of problems in DIMACS format that you might find useful for testing.

We will provide a selection of benchmarks that you should use for your final evaluation. For that, you should collect information such as solver answer and runtime in millisecond, and compare those values with those obtained by running the MiniSat solver on the same benchmarks.

You can choose the value of the timeout for your tests. Make sure to report that value and information on the platform you ran your evaluation on (processor, RAM, operating system).

References

To learn more about the VSIDS heuristic and the two watched literals implementation of unit propagation, you can read Sections 2 and 3 of the paper that introduced them.

```
   ⟨Sort⟩  ::=  Real | Bool
⟨SortedVar⟩  ::=  ( ⟨Symbol⟩ ⟨Sort⟩ )
  ⟨Numeral⟩  ::=  numeral (0, 1, 2, . . . )
  ⟨Decimal⟩  ::=  decimal numeral (e.g., 0.08, 1.43, . . . )
 ⟨Rational⟩  ::=  ( / ⟨Numeral⟩ ⟨Numeral⟩ )
 ⟨Function⟩  ::=  not | and | or | => | = | ite | + | - | * | <= | < | >= | >
⟨Expression⟩ ::=  true | false | ⟨Decimal⟩ | ⟨Rational⟩ | ⟨Symbol⟩
               |  ( ⟨Function⟩ ⟨Expression⟩+ )
  ⟨Command⟩  ::=  ( declare-const ⟨Symbol⟩ ⟨Sort⟩ )
               |  ( declare-fun ⟨Symbol⟩ () ⟨Sort⟩ )
               |  ( define-fun ⟨Symbol⟩ ( ⟨SortedVar⟩* ) ⟨Sort⟩ ⟨Expression⟩ )
               |  ( assert ⟨Expression⟩ )
               |  ( set-logic QF_LRA )
               |  ( check-sat )
               |  ( get-model )
```

Figure 1: SMT-LIB syntax for QF_LRA

For a pictorial explanation of unit propagation using watched literals, you can take a look at the slides titled "BCP - How to improve it?" and "BCP - Two Watched Literals" in this presentation.

Project 2: Implement a CDCL($\mathcal{T}_{\mathsf{LRA}}$) solver from existing software

Requirements

In this project, you will implement an SMT solver based on the CDCL(T) proof system and specialized on the quantifier-free theory of linear real arithmetic. You will write code to:

1. parse an input SMT problem;
2. convert the resulting formula to a clause form Δ;
3. combine a CLDL solver with a solver for quantifier-free LRA to check the satisfiability of Δ.

Parsing

Your solver should accept problems in SMT-LIB 2.6 format. The format allows one to construct problems as scripts that tell the solver which theory (or more precisely, SMT-LIB logic) you want to use, any additional symbols you want to declare, and the set of formulas you want to check for satisfiability. Note that you do not need to master the entire SMT-LIB language since you will only need to support the fragment of SMT-LIB 2 generated by the grammar in Figure 1.

There are many open-source libraries that expose an API for parsing and manipulating SMT-LIB syntax. If you are working in Python, you may want to use pySMT for that. We will use to

```
;; example.smt2
(set-logic QF_LRA)
(declare-fun x () Real)
(declare-fun y () Real)
(declare-fun z () Real)
(assert (or (< x 0) (> x 0)))
(assert (or (> x 0) (= z 0)))
(assert (= (+ x y) z))
(check-sat)
(get-model)
```

Figure 2: Example SMT-LIB script for QF_LRA.

pySMT in the rest of this section. If you use a different programing language you may adapt any open-source parser for SMT-LIB 2.6 or develop one yourself for the fragment of interest here.

Consider the SMT-LIB shown script in Figure 2 and suppose it is stored in a file called example.smt2. To parse the file and extract its query as a single formula, you need just a few pySMT commands:

```
$ python3
>>> from pysmt.smtlib.parser import SmtLibParser
>>> script = SmtLibParser().get_script_fname("example.smt2")
>>> formula = script.get_last_formula()
```

If you print the content of variable `formula`, you will get the following textual representation in pySMT of the conjunction of formulas asserted in the SMT-LIB script:

```
(((x < 0.0) | (0.0 < x)) & ((0.0 < x) | (z = 0.0)) & ((x + y) = z))
```

Converting to clause form

Once you have parsed the query as a single formula, you are to transform it to clause form using transformations like those studied in the course. The query from example.smt2 is in conjunctive normal form so its clause form would be something like

$$\Big\{\{x < 0.0,\ 0.0 < x\}, \{0.0 < x,\ z = 0.0\}, \{x + y = z\}\Big\}$$

The actual internal representation of clauses and clause set is up to you. Conceptually though you need to be able to access the individual clauses in the clause set and, for each clause, access the individual literals in the clause.

Once you have the clause set in memory, you can generate its Boolean abstraction by replacing each atom in the clause set by a positive integer, making sure that occurrences of the same atom are abstracted by the same integer. For example, in the clause set above, you could use the following abstraction $(x < 0.0) \mapsto 1$, $(0.0 < x) \mapsto 2$, $(z = 0.0) \mapsto 3$, $(x + y = z) \mapsto 4$, which gives you the propositional clause set

$$\Big\{\{1, 2\}, \{2, 3\}, \{4\}\Big\}$$

This abstraction will be useful when interfacing with a CDCL solver in order to check the satisfiability of the clause set at the propositional level.

```
from os import environ
from pysmt.shortcuts import get_env, Symbol, Real, LT, Equals, Plus, Solver
from pysmt.typing import REAL
from pysmt.logics import QF_LRA

x = Symbol("x", REAL); y = Symbol("y", REAL); z = Symbol("z", REAL)

l1 = LT(x, Real(0)); l2 = LT(Real(0), x); l3 = Equals(z, Real(0))
l4 = Equals(Plus(x, y), z)

get_env().factory.add_generic_solver\
  ( "cvc5"
  , [ environ["HOME"] + "/local/debug-cvc5/bin/debug-cvc5" ]
  , [ QF_LRA ]
  )

with Solver(name = "cvc5", logic = "QF_LRA") as s:
  s.add_assertion(l1); s.add_assertion(l2)
  s.add_assertion(l3); s.add_assertion(l4)

  if s.solve():
    print("sat"); print(s.get_model())
  else:
    print("unsat")
```

Figure 3: Python script for communicating with an SMT solver.

Combining a CDCL solver and a theory solver

You can obtain a CDCL($\mathcal{T}_{\mathsf{LRA}}$) solver by suitably combining an off-the-shelf CDCL SAT solver and a theory solver for QF_LRA. The combination can be achieved as seen in the lectures by having the SAT solver produce a satisfying assignment for the Boolean abstraction of the clause set, if such an assignment exists, and then checking that the corresponding theory literals are jointly satisfiable in $\mathcal{T}_{\mathsf{LRA}}$. You can do the latter by translating the current assignment back into theory literals, asserting them individually to a solver for $\mathcal{T}_{\mathsf{LRA}}$, and then asking the solver to check their satisfiability in $\mathcal{T}_{\mathsf{LRA}}$. If the theory solver replies positively, the assignment found by the SAT solver is consistent with the theory and can be returned. Otherwise, you can ask the theory solver for an *unsatisfiable core*, a minimal subset of assertions that is already unsatisfiable in the theory. You can then *block* the current (failed) assignment in the SAT solver by adding the negation of the unsat core as a clause. This will allow the CDCL solver to perform conflict analysis and backjump to an earlier point in the search.

We do not expect you to write a solver for the theory of linear real arithmetic. Instead, we ask that you use an existing SMT solver as a subroutine. Figure 3 shows a Python script that demonstrates how you can send to the cvc5[1] SMT solver the literals corresponding, for instance, to the Boolean assignment $1 := \mathsf{true}, 2 := \mathsf{true}, 3 := \mathsf{true}, 4 := \mathsf{true}$ and obtain either a satisfying assignment for the variable or `unsat`.

pySMT prints `unsat` since $x < 0.0$ and $0.0 < x$ cannot both be true. Therefore, you need to

[1]cvc5 can be downloaded from https://github.com/cvc5/cvc5/releases/

extend the initial clause set with a blocking clause for that assignment. An unsat core for the set of asserted literal is just $\{x < 0.0,\ 0.0 < x\}$. So you can send the blocking clause $\{-1,\ -2\}$ to the SAT solver in addition to the original clauses. The SAT solver might then return the assignment $1 := \mathsf{true}, 2 := \mathsf{true}, 3 := \mathsf{true}, 4 := \mathsf{true}$ at this point. This Boolean assignment is consistent with the theory, and will allow cvc5 to generate the following satisfying assignment for the variables x, y and z.

```
sat
x := -1.0
y := 1.0
z := 0.0
```

Note that, although the running example above is based on pySMT, you are not required to use it, or use Python as the implementation language. If you choose a different language, the way you communicate with the SMT solver will depend on the solver and its API for that language. All SMT solvers that support SMT-LIB allow you to communicate with them by sending and receiving text in SMT-LIB 2.6 format on the standard input and output channels. So you can still use an SMT solver even if it does not have an API in the language of your choice. The same applies to SAT solvers, which all support the DIMACS format.

Testing and evaluation

Benchmarks for QF_LRA are available through SMT-LIB. We will provide a selection of benchmarks that you should use for your final evaluation. For that, you should collect information such as solver answer and runtime in millisecond, and compare those values with those obtained by running the cvc5 solver directly on the same benchmarks.

You can choose the value of the timeout for your tests. Make sure to report that value and information on the platform you ran your evaluation on (processor, RAM, operating system).

References

The documentation for pySMT is available online.

Project 3: Implement a CDCL($\mathcal{T}_{\mathsf{UF}}$) solver from scratch

Requirements

For this project, you will implement an SMT solver for the quantifier-free fragment QF_UF of the theory $\mathcal{T}_{\mathsf{UF}}$ of equality with uninterpreted functions. This project is essentially a version of Project 2 where, however, you implement the theory solver from scratch. In this case though the theory is $\mathcal{T}_{\mathsf{UF}}$, which requires a simpler solver than $\mathcal{T}_{\mathsf{LRA}}$. As before, you can assume that the query will be provided in SMT-LIB format. You can also assume that the query will be presented with respect to a signature Σ, where the set of logical symbols corresponds with the SMT-LIB specification: $\{\neg, \wedge, \vee, \Rightarrow\}$, the set of sorts is, for instance, $\{\mathsf{Bool}, \mathsf{Univ}\}$, and the set of function symbols is $\{\top, \bot, \doteq_{\mathsf{Bool}}\} \cup \{\doteq_{\mathsf{Univ}}\ f_1, \ldots, f_n\}$. The rank of each function f_i will have the form

$$\langle \overbrace{\mathsf{Univ}, \ldots, \mathsf{Univ}}^{n \text{ times}}, \mathsf{Univ} \rangle$$

```
(set-logic QF_UF)
(declare-sort Univ 0)
(declare-fun l1 () Univ)
(declare-fun l2 () Univ)
(declare-fun rev (Univ) Univ)
(declare-fun app (Univ Univ) Univ)
(assert (= (rev (rev l1)) l1))
(assert (= (rev (rev l2)) l2))
(assert (= (rev (app (rev l1) (rev l2)))
           (app (rev (rev l2)) (rev (rev l1)))))
(assert (= (rev (app l2 l1))
           (app (rev l1) (rev l2))))
(assert (not (= (rev (rev (app (rev l1) (rev l2))))
                (app (rev l1) (rev l2)))))
(check-sat)
```

Figure 4: Example SMT-LIB script for QF_UF.

$$
\begin{array}{rcl}
\langle \textit{Function} \rangle & ::= & \langle \textit{Symbol} \rangle \mid \texttt{not} \mid \texttt{and} \mid \texttt{or} \mid \texttt{=>} \mid \texttt{=} \mid \texttt{ite} \\
\langle \textit{Expression} \rangle & ::= & \texttt{true} \mid \texttt{false} \mid \langle \textit{Symbol} \rangle \mid \texttt{(} \ \langle \textit{Function} \rangle \ \langle \textit{Expression} \rangle^{+} \ \texttt{)} \\
\langle \textit{Command} \rangle & ::= & \texttt{(} \ \texttt{declare-fun} \ \langle \textit{Symbol} \rangle \ \texttt{(} \ \texttt{Univ}^{*} \ \texttt{)} \ \texttt{Univ} \ \texttt{)} \\
& \mid & \texttt{(} \ \texttt{assert} \ \langle \textit{Expression} \rangle \ \texttt{)} \\
& \mid & \texttt{(} \ \texttt{set-logic} \ \texttt{QF\_UF} \ \texttt{)} \\
& \mid & \texttt{(} \ \texttt{check-sat} \ \texttt{)} \\
& \mid & \texttt{(} \ \texttt{get-model} \ \texttt{)} \\
& \mid & \texttt{(} \ \texttt{get-unsat-core} \ \texttt{)}
\end{array}
$$

Figure 5: SMT-LIB syntax for QF_UF.

where $n \geq 0$.[2] An example of such a query in the SMT-LIB syntax is provided in Figure 4.

Just like in Project 2, you can use an existing library to parse SMT-LIB. In any case, you may assume that the syntax of the SMT-LIB input is effectively restricted to the language generated by the grammar in Figure 5.

Once you parse the problem you can convert it to CNF, then abstract the theory atoms as Boolean literals, and use your SAT solver to produce an assignment. When you wish to check the satisfiability in $\mathcal{T}_{\mathsf{UF}}$ of the interpretation returned by the SAT solver, you can send the corresponding conjunction of theory literals to your theory solver. Since we will not test your theory solver independently, you are free to decide how it will receive the conjunction of theory atoms, and how it will report the results to the SAT solver. That said, your theory solver should produce two kinds of output. If the literals it receives are satisfiable in $\mathcal{T}_{\mathsf{UF}}$, it should produce an interpretation that satisfies them all. Otherwise, it should produce an unsatisfiable core, an unsatisfiable subset of the input literals.

More concretely, your theory solver should:

[2] Note that actual benchmarks may other sorts in addition or in place of Univ.

1. quickly compute the congruence closure that arises from a set of equalities. This involves writing an efficient union-find data structure with optimizations such as path compression;
2. produce a satisfying assignment for every $\mathcal{T}_{\mathsf{UF}}$-satisfiable set of equalities and disequalities;
3. identify a small unsatisfiable subset of an unsatisfiable set of equalities and disequalities.

Nieuwenhuis and Oliveras in Fast Congruence Closure and Extensions, address these challenges. Section 2 of their paper describes the necessary terminology. Section 3 defines the data structures and functions necessary for efficiently computing a congruence closure. Section 4 describes how to produce the aforementioned small unsatisfiable subset. Though the authors only mention path compression in their paper, Sedgewick and Wayne explain it in slides 29 through 32 of their presentation.

There is a chance that implementing the approach suggested by Nieuwenhuis and Oliveras proves too challenging. In that case, you may refer to Nelson and Oppen's original paper on Fast Decision Procedures Based on Congruence Closure and implement their algorithm. They do not explicitly describe how to compute an unsatisfiable subset of an inconsistent set of equalities and disequalities. However, you can leverage the property that every unsatisfiable set of $\mathcal{T}_{\mathsf{UF}}$-literals has a unsatisfiable subset that contains zero or more equations and *exactly one* disequation.

References

To reiterate the references suggested above:

- `https://www.cs.princeton.edu/courses/archive/spr08/cos226/lectures/01UnionFind.pdf` is a review of union-find with path compression. It is not necessary to understand Sedgewick and Wayne's implementation. You should however understand the purpose of union-find and the reason for considering path compression.
- `https://www.cs.upc.edu/~oliveras/IC.pdf` is how we suggest that you implement your theory solver.
- `https://web.eecs.umich.edu/~weimerw/2011-6610/reading/nelson-oppen-congruence.pdf` can be used as a fallback in case Nieuwenhuis and Oliveras' implementation proves difficult to implement.

Benchmarks

There are no readily available benchmarks for the restricted version of the quantifier-free theory of equality with uninterpreted functions mentioned above, but you can filter out benchmarks from the QF_UF logic in SMT-LIB.

All projects: What to submit

You are to submit your full code, including any scripts or build files necessary for compiling and/or executing your solver.

The code should be accompanied by the following documentation.

1. Comments for each major data structure (class, record, etc.) and for each non-trivial function, procedure or method in the source code, explaining its role in the context of the solver.
2. A `readme` file with detailed instructions on how to install and execute your solver **on Linux**.
3. A report in PDF format of up to 10 pages describing at a high level what you implemented and how, as well as results of an experimental evaluation of your solver. If you need a few more pages for the report do not waste time with formatting tricks. Do not pad up the report if you need less than 10 pages.