Boolean Satisfiability - Project Report

Combinatorial Problem Solving - MIRI - UPC

Xavi Arnal (xavier.arnal@estudiantat.upc.edu)

October 18, 2021

Contents

1 Introduction

Unlike with the other combinatorial problem-solving technologies seen in the course thus far, the language of CNF formulae is not expressive by default — for instance, the constraint on the amount of NOR gates to use, that in the first project was achieved through subclassing `IntMinimizeSpace` and a one-line `cost` function, now becomes three classes on a separate file to implement the sorting networks necessary for cardinality constraints.

Of course, this supports the thesis that CNF formulae are not expressive *by default* — as the sorting networks only have to be implemented once, after which (within the framework of this project) cardinality constraints are a perfectly valid and well-defined part of the language. This informs how this project is structured and should probably inform how it is read, which is why this introduction exists. With this in mind, the project is divided into two sections:

- **The satisfiability module.** This sections deals with modeling and solving[1] CNF formulae. Though originally meant to be a set of fairly thin python bindings for Kissat, this section grew to house all of the extensions to the CNF language necessary for properly (without artifacts) modeling the NLS problem. The first section of this report deals with the structure (more so than the implementation) of this module.
- **The rest of top-level files.** These handle IO and use the satisfiability module to model and solve the NLS problem. They are strongly analogous to the files from previous projects, modulo small details like the fact that boolean satisfiability doesn't include any native form of minimization. The second section and on deal with this part of the project.

1.0.1 Disclaimers

The code is not written for speed. Of course, how the code is written has no bearing on how fast Kissat runs, but Kissat runs fast enough that it makes it possible that the overhead in the generation of formulae is not comparatively insignificant. No problem instance takes more than two seconds to solve, but perhaps this could've been a single second.

Satisfiability modules in python already exist. Using them seemed to circumvent the point of the project, however, so this was avoided.

2 The Satisfiability Module

2.1 General structure

The module is largely divided into three sections:

- **Modeling.** These contain the basic tools for modeling problems with CNF formulae: the data structures relating to variables, literals, and the formulae themselves, as well as some primitive constructs (namely, partial reification and formula composition).
- **Premade models.** Contains a number of premade formulae (with some parameters) that can be $\wedge$-ed to the end of any existing formula to add an extra constraint. This is where general constraints (e.g. cardinality) are modeled in CNF, such that these constraints can be used directly from the client.
- **Solving.** Contains a simple interface to use Kissat.

We go into detail for each of these sections:

[1] finding, or showing the nonexistence of, satisfying assignments for a CNF formula

2.2 Modeling

2.2.1 Variables and literals

This section corresponds to the classes `Variable` and `Literal` in `modeling/primitives.py`. Both of these are short extensions of `int`, to be interpreted in the same way as the input/output of Kissat — some integer i corresponds to a variable x_i (perhaps as a literal), while $-i$ corresponds to the literal $\neg x_i$. This has a number of benefits:

- It facilitates straightforward integration with Kissat.
- It provides a natural identification of x_i as a variable and x_i as a literal, which is desirable for ease-of-use.
- It doesn't impede easy implementation of methods you'd expect from a `Literal` class — namely, a method to get the underlying variable (this is simply $\ell \mapsto abs(\ell)$), and a method to get truthiness/polarity ($\ell \mapsto \ell > 0$).

Note that, in fact, the existence of these classes is not *necessary* for the program, mostly due to the fact that the only SAT solver it integrates with is Kissat.

2.2.2 CNF formulae

This subsection corresponds to the classes in `modeling/problem.py`.

`SatFormula` a priori corresponds to a simple CNF formula — the clauses would be stored as lists of literals in the attribute `local_clauses`. However, it also has an attribute `subformulae`, which contains[2] other instances of `SatFormula` — these are to be understood as being extra constraints on the variables of the parent formula. To clarify, say an instance of `SatFormula`, with `local_clauses` making up a CNF formula C, has two other `SatFormula` instances in `subformulae`, corresponding to CNF formulas C_1, C_2. Then, the parent formula is to be understood as representing the CNF formula $C \wedge C_1 \wedge C_2$.

This sort of bookkeeping of what-formula-is-a-subformula-of-what-other-formula isn't at all necessary, but it nicely translates recursive definitions from the domain (CNF formulae) to the implementation. For instance, a look at the implementation of `SortingNetwork` (in `general.py`, from `modeling/subproblems/cardinality/`) will reveal that a sorting network of order n contains as subformulae two sorting networks of order $n-1$ and a merge network of order n, which mirrors the definition given in class.

`SatProblem` is an extension of `SatFormula` that provides the necessary structure for modeling actual problems — namely, creates and keeps track of variables, and implements some IO utilities.

Keeping in mind the consideration about subformulae from `SatFormula.subformulae`, the role of `SatProblem` is to lie at the root of a tree of subformulae. Every formula in this tree will need to have a reference to this root at some point if it wants to create its own auxiliary variables.

2.2.3 Partial reification

Consider a variable r, and a CNF formula $C = c_1 \wedge \cdots \wedge c_n$, where each c_i is a disjunction. Let $c_i' = \neg r \wedge c_i$, and $C' = c_1' \wedge \cdots \wedge c_n'$. Then, C' is C *partially reified* by r — if it is used as a subformula in some other formula, then C is only enforced if r is set to true. This is easy to check in the language of propositional logic: by definition, $c_i' \equiv (r \implies c_i)$, and thus $C' \equiv (r \implies C)$.

This construct is implemented in the class `PartiallyReified`, in `modeling/reification.py`. It extends `SatFormula` and behaves as a sort of wrapper around another instance of `SatFormula`, prepending $\neg r$ to each of its clauses.

[2] this is not enforced during runtime (and compile-time enforcement is not an option)

2.3 Premade models

Premade models are all extensions of `SatFormula`, to be used as subformulae, expressing some particular constraint on the variables of a parent formula, perhaps using their own helper variables or subformulae. For instance, consider the implementation of `Equal`:

```
class Equal(SatFormula):
def __init__(self, root: SatProblem, l1, l2):
    super().__init__()
    self.add_clause(+l1, -l2)
    self.add_clause(-l1, +l2)
```

Note that the constructor takes in variables/literals (from the parent problem) that it will put an extra constraint on. Moreover, although `Equal` does not use it, all premade models take in a parameter `root` in their constructor — this is a reference to the `SatProblem` at the root of the subformula tree that this particular premade model lies in, which is necessary for creating auxiliary variables.

We go through the premade models implemented in the project. These are simply the ones strictly necessary to model the NLS problem, as well as some extra simple models for testing. All of these are located within the directory `modeling/subproblems`.

- **Binary relations**. These are `Equal`, `Distinct`, and `Implies`, in `binary_relations.py`. They constrain two literals, each in the way that their name suggests.
- **Binary operators**. These are `Nor` and `Nand`, in `binary_operators.py`. Their constructors take three literals i_1, i_2, and o, and post constraints that ensure that $o = \mathsf{NOR}(i_1, i_2)$ and $o = \mathsf{NAND}(i_1, i_2)$, respectively.
- **Unary cardinality constraints**. These are `AtLeastOne`, `AtMostOne`, and `ExactlyOne`, in `cardinality/unary.py`. Each take an array of variables as input, and post the constraint that at least one (resp. at most one, exactly one) of these variables must be true.

  `AtLeastOne` has a native encoding in CNF, namely, $x_1 \vee \cdots \vee x_n$ means that at least one of $x_1, \ldots, x_n$ is true. `AtMostOne` is represented by means of a Heule encoding, explained in the course material. `ExactlyOne` is simply the conjunction of `AtLeastOne` and `AtMostOne`, over the same array of literals.
- **General cardinality constraints**. These are `AtLeast`, `AtMost`, and `Exactly`, located in `cardinality/general.py`. Each take an array of variables as input, and post the constraint that at least k (resp. at most k, exactly k) of these variables must be true, where k is an (integer) parameter passed to the constructor.

  The implementation of this relies on the helper classes `Comparator`, `MergeNetwork`, and `SortingNetwork`, which implement the concepts described in the course material. Notably, the "native" usage of these constraints only allows for sorting/posting cardinality constraints on arrays of variables the length of which is a power of 2 — when this isn't the case, the input and output of `SortingNetwork` are zero-padded accordingly.

  Note that, unlike with unary cardinality constraints, `Exactly` is not the conjunction of `AtLeast` and `AtMost` — because `AtLeast` and `AtMost` each use their own sorting network. Moreover, these sorting networks follow the suggestion of the course material pertaining to the implementation of comparators — for instance, `AtLeast` uses comparators with three clauses instead of six, which is all that's necessary for lower-bound constraints.

2.4 Solving

All code related to solving is in the `solving` directory. This includes a `Results` class, which stores the result of an attempt (successful or not) to satisfy a CNF formula, a `SatSolver` class, which provides an abstract interface for SAT solving, and a class `Kissat`, extending `SatSolver` and using an installation of Kissat[3] to solve SAT problems. The implementation of all of these things is straightforward, with the exception of timeouts, which we will cover now.

2.4.1 Handling timeouts

A possible result of an attempt to satisfy a CNF formula is a timeout — indeed, `Result` has an attribute `timed_out`. However, it is not so straightforward to *enforce* these timeouts — at least I was unable to find a parameter for Kissat that specified a maximum search time. For this reason, `SatSolver` has three methods:

- `_solve` is the abstract method that each non-abstract subclass of `SatSolver` should implement — it takes a `SatProblem` and returns a `Result`.
- `solve` spawns a thread and uses it to execute and get the return value of `_solve` — if this execution takes less time than the timeout specified. Otherwise, it kills the thread and returns `Result.timed_out()`.
- `_solve_helper` is arguably an implementation artifact — it just calls `_solve` and puts the return value in a (multiprocessing) queue passed by argument. This is the method that `solve` calls, because getting the return value from a thread directly isn't a native capability in python's multiprocessing module.

Of course using threading to handle timeouts isn't ideal, but it is the only *general* way to do it if there is no guarantee that each SatSolver will have a nice way to specify maximum search time. If some specific SatSolver *does* have a nice way to specify maximum search time, its corresponding class may simply override `solve` instead of `_solve` to avoid the inelegant threading solution.

Along these lines, it is possible to do this with Kissat — because it is called through a command line, one might use the command `timeout` (while in an unix system) to specify how long Kissat can run. However, it is possible (and it does happen) that this timeout kicks in while Kissat is printing out a satisfying assignment, meaning that the python method that calls Kissat and processes its output has to handle this possibility — this isn't ideal, both because it makes the code significantly more complicated, and because there isn't really any way to distinguish a timeout from an issue with the output format.

[3] the path of which has to be specified in `config.py`