Summer School on Formal Techniques

Boolean (Un)Satisfiability

Lab 1

June, 2022

Explanation

These exercises are designed to provide a deeper understanding of the operation of Boolean satisfiability (SAT) solvers, especially when applied to unsatisfiable formulas. A key requirement is that solver be able to generate a proof of unsatisfiability in such cases.

The provided problems range in how much time and effort is required, and whether any programming is involved. Each problem has an associated *level*, according to the following standard:

I: Simple pencil-and-paper exercises designed to provide a concrete examples for the concepts presented. Doing these will help you gain confidence in the concepts being presented

II: More challenging pencil-and-paper exercises, or algorithmic and experimental activities. These may running solvers on some benchmarks.

III: Deeper explorations. These may require devising new algorithms, writing code, and performing experiments that go beyond the core lecture material.

All file names are specified in this document are given as path names of the form *ROOT*`/DIR/FILE` where *ROOT* indicates the root of the directory structure, *DIR* is the subdirectory and *FILE* is the file name.

Using the Provided Programs

Here is how to use the provided tools. In the following, we assume *FILE* is the common base name for a set of files having different extensions.

KISSAT

- Running without proof generation

  ROOT`/tools/kissat/build/kissat` *FILE*`.cnf`

- Running with proof generation

  ROOT`/tools/kissat/build/kissat --no-binary` *FILE*`.cnf` *FILE*`.drat`

TBSAT

- Running in direct mode without proof generation, and generating up to K solutions

  ROOT`/tools/tbuddy/bin/tbsat/tbsat -i` *FILE*`.cnf -m` K

- Running in direct mode with proof generation

  ROOT`/tools/tbuddy/bin/tbsat/tbsat -i` *FILE*`.cnf -o` *FILE*`.lrat`

- Running in bucket mode with proof generation

  ROOT`/tools/tbuddy/bin/tbsat/tbsat -b -i` *FILE*`.cnf -o` *FILE*`.lrat`

DRAT-TRIM

- Checking proof

  ROOT`/tools/drat-trim/drat-trim` *FILE*`.cnf` *FILE*`.drat`

- Transforming a DRAT proof into an LRAT proof

  ROOT`/tools/drat-trim/drat-trim` *FILE*`.cnf` *FILE*`.drat -L` *FILE*`.lrat`

LRAT-CHECK

- Checking proof

  ROOT`/tools/drat-trim/lrat-check` *FILE*`.cnf` *FILE*`.lrat`

The Pigeonhole Problem PHP(n)

The formula PHP(n) encodes the impossible problem of assigning $n + 1$ pigeons to n holes such that 1) each pigeon is assigned to some hole and 2) no hole contains more than one pigeon. It is described in Slide #14 of Lecture 1b. The SAT encoding uses variables $p_{i,j}$ for $1 \leq i \leq n$ and $1 \leq j \leq n + 1$, where $p_{i,j}$ is 1 when pigeon j is assigned to hole i. It makes use of the cardinality constraints described in Slide #18 of Lecture 1a.

Problem 1. (Level I):

The file `files/phpd-02.cnf` contains a DIMACS representation of the CNF encoding of PHP(2). It consists of the following lines

```
p cnf 6 9
1 4 0
2 5 0
3 6 0
-1 -2 0
-1 -3 0
-2 -3 0
-4 -5 0
-4 -6 0
-5 -6 0
```

A. Describe how each variable $p_{i,j}$ is mapped to a variable number in the file.

B. Describe how the following sets of clauses encode the problem:

Clauses 1–3 :

Clauses 4–6 :

Clauses 7–9 :

Problem 2. (Level II):

In this exercise, you will determine the number of clauses in the encoding of PHP(n) according to a *direct* encoding of the at-most-one constraints, as is described on Slide #18 of Lecture 1a.

A. How many at-least-one constraints are required? How many clauses does each require?

B. How many at-most-one constraints are required? How many clauses does each require?

C. What is the total number of clauses required to encode PHP(n)? You can write this as an expression of the form $\approx a \cdot n^b$, meaning that the exact count is an expression of the form $a \cdot n^b + o(n^b)$.

Problem 3. (Level III):

The direct encoding of at-most-one (AMO) constraints shown on Slide #18 of Lecture 1a scales quadratically with the number of variables. While polynomial, this can become unwieldy for large values of n. Here we will consider a method to reduce the size of the encoding to $O(n)$ by using auxilliary variables, analogous to their use in encoding parity constraints, as is described on Slides #16–17 of Lecture 1a. Our goal is to derive a method described in a paper by Carsten Sinz in 2005.

For $n > 2$, let us encode $AMO(x_1, x_2, \ldots, x_n)$ by the following process:

1. Introduce a new variable z
2. Encode some set of constraints $LCON(x_1, x_2, z)$
3. Recursively encode $AMO(z, x_3, \ldots, x_n)$.
4. The recursion terminates with a constraint of the form $RCON(z', x_n)$, where z' was the final variable added.

A. What constraints should be encoded as $LCON$? How would these be expressed as clauses?

B. What constraints should be encoded as $RCON$? How would these be expressed as clauses?

C. Show the set of clauses that this process would generate for $AMO(x_1, x_2, x_3, x_4, x_5)$

D. For $n \geq 3$, how many additional variables and how many clauses does this encoding require?

E. How many total variables and how many total clauses would this method require to encode PHP(n)? As before, you can write these as expressions of the form $\approx a \cdot n^b$.

CDCL Operation

In the following problems, you will simulate the behavior of a CDCL solver by hand. Pseudocode for the algorithm is given on Slide #6 of Lecture 1b. Rather than diagramming the execution as was done on Slide #7, you can simply write a sequence of literals describing one execution of the inner loop of the algorithm. Use some method (e.g., colors, underlining) to distinguish literals that are set by unit propagation versus those assigned by choice. A conflict occurs when the same variable has been assigned both 1 and 0. Then finish the line with the generated conflict clause.

For example, we would use the following notation to describe the execution shown on Slide#7, where assigned literals are indicated in red.

Sequence	Conflict Clause
$\overline{b}\ \overline{d}\ d$	b
$b\ c\ \overline{a}\ \overline{d}\ d$	$\overline{c}$
$b\ \overline{c}\ \overline{a}\ \overline{d}\ d$	$\bot$

To make the process deterministic, follow these conventions:

1. When choosing a variable and an assignment, choose the least numbered unassigned variable and assign it value 1.
2. Perform unit propagations in breadth-first order, and for each of these in the order of the clauses. That is, when processing a literal in the sequence, do a pass over the clauses, adding any unit propagations to the end of your sequence.

Problem 4. (Level I):

Show how CDCL would execute when following these conventions on the following DIMACS file, encoding PHP(2). This file is available as `files/phpd-02.cnf`

```
p cnf 6 9
1 4 0
2 5 0
3 6 0
-1 -2 0
-1 -3 0
-2 -3 0
-4 -5 0
-4 -6 0
-5 -6 0
```

You can use the DIMACS conventions for writing literals and clauses.

Problem 5. (Level I):

Show how CDCL would execute when following these conventions on the following DIMACS file, encoding the example formula from Lectures 2a and 2b.

This file is available as `files/eg-1.cnf`

```
p cnf 4 6
-1 -2 -3    0
-1 -2  3    0
 1       -4 0
 1        4 0
    2    -4 0
cd     2     4 0
```

You can use the DIMACS conventions for writing literals and clauses.

Problem 6. (Level II):

Show how CDCL would execute when following these conventions on the following DIMACS file, encoding PHP(3).

This file is available as `files/phpd-03.cnf`

```
p cnf 12 22
1 5 9 0
2 6 10 0
3 7 11 0
4 8 12 0
-1 -2 0
-1 -3 0
-1 -4 0
-2 -3 0
-2 -4 0
-3 -4 0
-5 -6 0
-5 -7 0
-5 -8 0
-6 -7 0
-6 -8 0
-7 -8 0
-9 -10 0
-9 -11 0
-9 -12 0
-10 -11 0
-10 -12 0
-11 -12 0
```

You can use the DIMACS conventions for writing literals and clauses.

CDCL Proof Generation

Problem 7. (Level I):

Create a DRAT file containing the conflict clauses you generated in your solution to Problem 4.

A. Check that it is a valid proof using DRAT-TRIM.

B. Use DRAT-TRIM to generate an LRAT proof. How do the steps in the LRAT proof correspond to those in the DRAT proof? **Note**: You can ignore the LRAT steps that contain the character '`d`'. These are deletion steps.

Problem 8. (Level I):

Create a DRAT file containing the conflict clauses you generated in your solution to Problem 5.

A. Check that it is a valid proof using DRAT-TRIM.

B. Use DRAT-TRIM to generate an LRAT proof. How do the steps in the LRAT proof correspond to those in the DRAT proof?

Problem 9. (Level II):

Create a DRAT file containing the conflict clauses you generated in your solution to Problem 6.

A. Check that it is a valid proof using DRAT-TRIM.

B. Use DRAT-TRIM to generate an LRAT proof. How do the steps in the LRAT proof correspond to those in the DRAT proof?

Experimenting with PHP(n)

In the following problems, you will explore how proofs of PHP(n) scale with n when running existing SAT solvers. You will evalute different solvers and different encodings of the formula.

The file `generators/gen_pigeon.py` can be used to generate instances of pigeonhole formulas. Here are its command-line options:

`-h`: Print documentation

`-v`: Verbose mode. The generator will put comments in the CNF file describing how the problem is encoded. You may find these instructive.

`-L`: Generate a linear encoding of the at-most-one constraints according to the solution you devised for Problem 3. **WARNING:** You must first add code to the generator before this will work.

`-r` *`ROOT`*: Specify root name of output file. For example, if you specify the root "`PHP`," the generated file will be named `PHP.cnf`.

`-n` N**:** Specify the number of holes

`-p` P**:** Specify the number of pigeons. The default is to have $P = N + 1$,

Problem 10. (Level II):

Generate direct encodings of PHP(n) and run KISSAT to fill in the following table. Use DRAT-TRIM to check the proofs generated by KISSAT. You can get the number of input variables and clauses from the header of the CNF file. The output line labeled "`proof_added`" shows the number of proof clauses. What can you infer about how the proof size scales?

n	Input Variables	Input Clauses	Proof Clauses
4			
6			
8			
10			
11			

Problem 11. (Level II):

Although the discussion of the BDD-based solver TBSAT will be in the second set of lectures, for now it be treated as a "black box" solver that supports two operating modes: "direct" and "bucket." With your direct encodings of PHP(n), run TBSAT in both direct and bucket mode to fill in the following table. Use LRAT-CHECK to check the proofs generated by TBSAT. The output line labeled "`Total clauses`" shows the number of proof clauses. What can you infer about how the proof size scales? How do these compare to KISSAT? How do these compare to each other? How does its runtimes compare to those of KISSAT?

n	Input Variables	Input Clauses	Direct Clauses	Bucket Clauses
4				
6				
8				
10				
11				

Problem 12. (Level III):

The file `generators/cnf_utilities.py` contains code to generate encodings of various formulas as clauses. Implement the linear at-most-one encoding scheme you devised in Problem 3. To do so, fill in code for the functions `lconEncode` and `rconEncode` in this file. Within these functions, call `writer.doClause` to generate the clauses.

Test your code by the following methods:

1. Generate small instances of PHP with the '`-L`' and '`-v`' flags set. Convince yourself that the correct clauses are listed
2. Run these with KISSAT. Make sure they're unsatisfiable.
3. Try generating instances where the number of holes and the number of pigeons are the same. These should be satisfiable. Check the solutions generated by KISSAT and make sure they're valid.
4. On the satisfiable instances, try running TBSAT in mode to generate multiple solutions Check that these solutions are all valid.

Problem 13. (Level III):

Generate linear encodings of PHP(n) and run KISSAT to fill in the following table. How does the performance compare to the direct encoding?

n	Input Variables	Input Clauses	Proof Clauses
4			
6			
8			
10			
11			
12			