Summer School on Formal Techniques

Boolean (Un)Satisfiability

Lab 2

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.

BDD-Based SAT Solving

Consider the following unsatisfiable set of parity constraints

$$\begin{array}{ccccccc} a & \oplus & b & & & = & 1 \\ & & b & \oplus & c & = & 1 \\ a & \oplus & & & c & = & 1 \end{array}$$

We refer to this formula as $CYC(3)$, where $CYC(n)$ represents a cyclic chain of odd parity constraints for n variables. It's easy to see that $CYC(n)$ is unsatisfiable for odd n.

File `cyc3.cnf` encodes these constraints in clausal form numbering variables a, b, and c as 1 through 3:

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

Problem 1. (Level I):

When running TBSAT, it can be directed to operate in "verbose" mode, adding comments to the generated proof file. Here is an excerpt of the comments generated when running in direct mode on `cyc3.cnf` (See Slide #14 in Lecture 2a). The full proof is in file `files/cyc3-dct.lrat`.

```
c Input Clause #1: 2 1 0
c Input Clause #2: -2 -1 0
c Input Clause #3: 3 2 0
c Input Clause #4: -3 -2 0
c Input Clause #5: 3 1 0
c Input Clause #6: -3 -1 0
c Defining clauses for node N4 = ITE(V1 (level=1), N1, N0)
c Defining clauses for node N5 = ITE(V1 (level=1), N0, N1)
c Defining clauses for node N6 = ITE(V2 (level=2), N1, N0)
c Defining clauses for node N7 = ITE(V2 (level=2), N0, N1)
c Defining clauses for node N8 = ITE(V3 (level=3), N1, N0)
c Defining clauses for node N9 = ITE(V3 (level=3), N0, N1)
c Defining clauses for node N10 = ITE(V1 (level=1), N1, N6)
c Validate BDD representation of Clause #1.  Node = N10.
c Defining clauses for node N11 = ITE(V1 (level=1), N7, N1)
c Validate BDD representation of Clause #2.  Node = N11.
c Defining clauses for node N12 = ITE(V2 (level=2), N1, N8)
c Validate BDD representation of Clause #3.  Node = N12.
c Defining clauses for node N13 = ITE(V2 (level=2), N9, N1)
c Validate BDD representation of Clause #4.  Node = N13.
c Defining clauses for node N14 = ITE(V1 (level=1), N1, N8)
c Validate BDD representation of Clause #5.  Node = N14.
c Defining clauses for node N15 = ITE(V1 (level=1), N9, N1)
c Validate BDD representation of Clause #6.  Node = N15.
c Defining clauses for node N16 = ITE(V1 (level=1), N7, N6)
c Generating proof that N10 & N11 --> N16
c Defining clauses for node N17 = ITE(V2 (level=2), N9, N8)
c Generating proof that N12 & N13 --> N17
c Defining clauses for node N18 = ITE(V1 (level=1), N9, N8)
c Generating proof that N14 & N15 --> N18
c Defining clauses for node N19 = ITE(V2 (level=2), N9, N0)
c Generating proof that N6 & N17 --> N19
c Defining clauses for node N20 = ITE(V2 (level=2), N0, N8)
c Generating proof that N7 & N17 --> N20
c Defining clauses for node N21 = ITE(V1 (level=1), N20, N19)
c Generating proof that N16 & N17 --> N21
c Validate empty clause for node N0 = N18 & N21
```

From these comments it is possible to trace how the program converted the clauses into BDDs, formed their conjunctions, and detected that this yielded BDD leaf $\perp$. Understanding these steps will help you better understand how the program operates.

Here are some guidelines on the notation:

- Nodes `N1` and `N0` denote the two leaf nodes, with values 1 and 0, respectively
- Nonterminal BDD nodes are given names of the form Nz, where z is the integer extension variable associated with the node.
- Nonterminal nodes are written as ITE(Vx, Nh, Nl), where x indicates the variable, and Nh and Nl indicate the two children.

A. The following nodes each correspond to conjunctions of some set of the input clauses. Fill in the following table with that information, as is illustrated with the first entry.

Node	Clauses
N16	1,2
N17	
N18	
N21	

B. Draw the BDD having node `N21` as root.

C. Looking at the paths from the root to Leaf 1 in this BDD, what constraint do these place on the relation between a and b?

D. What does that constraint imply when the conjunction of the BDDs with roots `N21` and `N18` is formed?

Problem 2. (Level I):

When running TBSAT on `cyc3.cnf` in bucket mode (see Slide #16–17 of Lecture 2b), the generated proof contains these comments. The full proof is in file `files/cyc3-bkt.lrat`.

```
c Input Clause #1: 2 1 0
c Input Clause #2: -2 -1 0
c Input Clause #3: 3 2 0
c Input Clause #4: -3 -2 0
c Input Clause #5: 3 1 0
c Input Clause #6: -3 -1 0
c Defining clauses for node N4 = ITE(V1 (level=1), N1, N0)
c Defining clauses for node N5 = ITE(V1 (level=1), N0, N1)
c Defining clauses for node N6 = ITE(V2 (level=2), N1, N0)
c Defining clauses for node N7 = ITE(V2 (level=2), N0, N1)
c Defining clauses for node N8 = ITE(V3 (level=3), N1, N0)
c Defining clauses for node N9 = ITE(V3 (level=3), N0, N1)
c Defining clauses for node N10 = ITE(V1 (level=1), N1, N6)
c Validate BDD representation of Clause #1.  Node = N10.
c Defining clauses for node N11 = ITE(V1 (level=1), N7, N1)
c Validate BDD representation of Clause #2.  Node = N11.
c Defining clauses for node N12 = ITE(V2 (level=2), N1, N8)
c Validate BDD representation of Clause #3.  Node = N12.
c Defining clauses for node N13 = ITE(V2 (level=2), N9, N1)
c Validate BDD representation of Clause #4.  Node = N13.
c Defining clauses for node N14 = ITE(V1 (level=1), N1, N8)
c Validate BDD representation of Clause #5.  Node = N14.
c Defining clauses for node N15 = ITE(V1 (level=1), N9, N1)
c Validate BDD representation of Clause #6.  Node = N15.
c Defining clauses for node N16 = ITE(V1 (level=1), N7, N6)
c Generating proof that N10 & N11 --> N16
c Defining clauses for node N17 = ITE(V1 (level=1), N9, N8)
c Generating proof that N14 & N15 --> N17
c Defining clauses for node N18 = ITE(V2 (level=2), N8, N0)
c Defining clauses for node N19 = ITE(V2 (level=2), N0, N9)
c Defining clauses for node N20 = ITE(V1 (level=1), N19, N18)
c Generating proof that N16 & N17 --> N20
c Defining clauses for node N21 = ITE(V2 (level=2), N8, N9)
c Generating proof that N20 --> N21
c Defining clauses for node N22 = ITE(V2 (level=2), N9, N8)
c Generating proof that N12 & N13 --> N22
c Validate empty clause for node N0 = N21 & N22
```

A. Which clauses are conjuncted to form the BDD with root node `N20`? What does this BDD represent?

B. Draw a diagram of the BDD with root node `N20`.

C. The BDD with root node `N21` is the result of existentially quantifying a from `N20`, and the BDD with root node `N22` is the result of conjuncting clauses C_3 and C_4. Draw a diagram showing both of these BDDs together.

D. What happens when the conjunction of the BDDs with root nodes `N21` and `N22` is formed? Explain

Proof Generation with BDDs

The following shows a portion of the LRAT proof file generated when TBSAT is applied to `cyc3.cnf`. The full proof is in file `files/cyc3-dct.lrat`.

```
c Input Clause #1: 2 1 0

c Defining clauses for node N6 = ITE(V2 (level=2), N1, N0)
15 6 -2 0  0
18 -6 2 0 -15 0

c Defining clauses for node N10 = ITE(V1 (level=1), N1, N6)
31 10 -1 0  0
32 10 -6 1 0  0
34 -10 6 1 0 -31 -32 0
c Validate BDD representation of Clause #1.  Node = N10.
35 10 0 31 32 15 1 0
```

Problem 3. (Level I):

As the comments indicate, clauses `15`, `18`, and `31`–`34` are defining clauses for two of the nodes. With LRAT, extension variables are introduced via such clauses, where the literal for the extension variable is listed first. Rather than the antecedents that we saw in the RUP clause examples of the lectures, the second portion of a defining clause lists any previous clauses containing the opposite literal of that of the extension variable, but these are also negated. So for example, Clause `18` lists `-15`, while clause `34` lists `-31` and `-32`. (Understanding this part of the syntax is not critical here.)

A. In general, there can be up to four defining clauses per BDD node, as was shown on Slide #4 of Lecture 2b. But, when a clause degenerates to a tautology, it is not included in the proof. Show how clauses `15` and `16` match up to the defining clauses for node `N6`.

B. Show how clauses `31`, `32`, and `34` match up to the defining clauses for node `N10`.

C. Simulate the RUP proof steps to provide a justification of unit clause `35`, indicating that node `N10` is the TBDD representation of input clause `1`.

The following shows a portion of the LRAT proof file generated when TBSAT is applied to `cyc3.cnf` in direct mode. The full proof is in file `files/cyc3-dct.lrat`.

```
c Input Clause #1: 2 1 0
c Input Clause #2: -2 -1 0

c Defining clauses for node N6 = ITE(V2 (level=2), N1, N0)
15 6 -2 0  0
18 -6 2 0 -15 0
c Defining clauses for node N7 = ITE(V2 (level=2), N0, N1)
20 7 2 0  0
21 -7 -2 0 -20 0

c Defining clauses for node N10 = ITE(V1 (level=1), N1, N6)
31 10 -1 0  0
32 10 -6 1 0  0
34 -10 6 1 0 -31 -32 0
c Validate BDD representation of Clause #1.  Node = N10.
35 10 0 31 32 15 1 0

c Defining clauses for node N11 = ITE(V1 (level=1), N7, N1)
36 11 -7 -1 0  0
37 11 1 0  0
38 -11 7 -1 0 -36 -37 0
c Validate BDD representation of Clause #2.  Node = N11.
40 11 0 37 36 20 2 0

c Defining clauses for node N16 = ITE(V1 (level=1), N7, N6)
61 16 -7 -1 0  0
62 16 -6 1 0  0
63 -16 7 -1 0 -61 -62 0
64 -16 6 1 0 -61 -62 0
c Generating proof that N10 & N11 --> N16
65 16 -11 -10 -1 0 61 38 0
66 16 -11 -10 0 65 62 34 0

c Validate unit clause for node N16 = N10 & N11
67 16 0 35 40 66 0
```

A. Proof clauses `65` and `66` justify that node `N16` is the conjunction of BDDs representing input clauses `1` and `2`. It is a special case of the general form shown on Slide #7 of Lecture 2b, where the justifications for the two recursive calls are the tautologies $b \wedge 1 \rightarrow b$ and $\overline{b} \wedge 1 \rightarrow \overline{b}$. Match up the other clauses to those shown on Slide #7.

B. Simulate the RUP proof steps to justify proof clauses `65` and `66`.

C. What is the significance of proof clause `67`? What is its justification?

Minimal Disagreement Parity

For the remainder of the lab, you will test SAT solvers on a problem involving a combination of parity reasoning and cardinality constraints.

Crawford, Kearns, and Shapire proposed the Minimum Disagreement Parity (MDP) Problem as a challenging SAT benchmark in an unpublished report from AT&T Bell Laboratories in 1994. The report is available at:

http://www.cs.cornell.edu/selman/docs/crawford-parity.pdf.

MDP is closely related to the "Learning Parity with Noise" (LPN) problem. LPN has been proposed as the basis for public key crytographic systems. Unlike the widely used RSA cryptosystem, it is resistant to all known quantum algorithms. The capabilities of SAT solvers on MDP is therefore of interest to the cryptology community.

The program `generators/mdparity.py` generates CNF formulas encoding instances of the MDP problem.

In the following, let $\mathcal{B} = \{0, 1\}$ and $\mathcal{N}_p = \{1, 2, \ldots, p\}$. Assume all arithmetic is performed modulo 2. Thus, if $a, b \in \mathcal{B}$, then $a + b \equiv a \oplus b$.

The problem is parameterized by a number of solution bits n, a number of samples m, and an error tolerance k, as follows. Let $\boldsymbol{s} = s_1, s_2, \ldots, s_n$ be a set of *solution* bits. For $1 \leq j \leq m$, let $X_j \subseteq \mathcal{N}_n$ be a *sample set*, created by generating n random bits $x_{1,j}, x_{2,j}, \ldots x_{n,j}$ and letting $X_j = \{i | x_{i,j} = 1\}$. Let $\boldsymbol{y} = y_1, y_2, \ldots, y_m$ be the parities of the solution bits for each of the m samples:

$$y_j \quad = \quad \sum_{i \in X_j} s_i \tag{1}$$

Given sufficiently many samples m for there to be at least n linearly independent sample sets, the values of the solution bits $\boldsymbol{s}$ can be uniquely determined from $\boldsymbol{y}$ and the sample sets S_j for $1 \leq j \leq m$ by Gaussian elimination. To make this problem challenging, we introduce "noise," allowing up to k of these samples to be "corrupted" by flipping the values of their parity. That is, let $T \subseteq \mathcal{N}_m$ be created by randomly choosing k values from $\mathcal{N}_m$ without replacement, and define m "corruption" bits $\boldsymbol{r} = r_1, r_2, \ldots, r_m$, with r_j equal to 1 if $j \in T$ and equal to 0 otherwise. We then provide noisy samples $\boldsymbol{y}'$, defined as:

$$y'_j \quad = \quad r_j + \sum_{i \in X_j} s_i \tag{2}$$

and require the correct solution bits $\boldsymbol{s}$ to be determined despite this noise. That is, the generated solution $\boldsymbol{s}$ must satisfy at least $m - k$ of equations (1). For larger values of k, the problem becomes NP-hard.

We can see the potential of this problem for cryptographic applications. A set of m parity constraints over n variables serves as a "lock", where the "key" is a set of n solution bits. Checking that the key fits the lock requires simply checking that it satisfies at least $m - k$ of the parity constraints. The lock can be encoded with $O(m \cdot n)$ bits, and the key is just m bits.

This problem can readily be encoded in CNF with variables for unknown values $\boldsymbol{s}$ and $\boldsymbol{r}$, along with some auxilliary variables. Each of the m equations (2) is encoded using auxilliary variables to avoid exponential expansion. An at-most-k constraint is imposed on the corruption bits $\boldsymbol{r}$.

Crawford and his colleagues suggests choosing n to be a multiple of 4 and letting $m = 2n$ and $k = m/8 = n/4$, but the tools they had to investigate these parameters didn't include today's powerful SAT solvers.

From the perspective of SAT solving, MDP has an interesting form. Structurally, it consists of a set of m parity constraints over a set of $n + m$ variables—the n solution bit $\boldsymbol{s}$, plus the m corruption bits $\boldsymbol{r}$, with each constraint j having the form shown in (2). Additionally, there is an at-most-k constraint imposed on the corruption bits. Performing Gaussian elimination on the parity constraints reduces them to a smaller set of constraints over just the corruption bits. The problem to be solved then becomes: "Given a set of parity constraints over m variables, is there a solution for which at most k of these variables are set to 1?" That is not a problem that can be solved by algebraic methods.

Thus, Gaussian elimination can greatly simplify the constraints, but unlike other parity constraint benchmarks (e.g., the Chew-Heule formulas), it must be combined with other SAT solving techniques. KISSAT does not do anything special with parity constraints—it will apply CDCL to the original problem. TBSAT, on the other hand, can be directed to first apply Gaussian elimination and then perform bucket elimination.

Experiments to Perform

An instance of the problem is guaranteed to have one solution, but it might also have multiple solutions, possibly weakening its cryptographic properties. The chances of the solution being unique can be improved by increasing m relative to n, but it would be good to avoid taking this to an extreme.

To enable checking for uniqueness, the benchmark generator has an option '`-x`' that causes the generator to insert a clause that excludes the nominal solution. The resulting formula will then be unsatisfiable if there are no other solutions.

The directory mdp-run is set up for generating instances of the benchmark (with the nominal solution excluded) and running both KISSAT and TBSAT, with and without proof generation via different options in the Makefile. The key parameters to be set are n, m, k, and the random seed. When running without proof generation, the Makefile also invokes a solution checker. This program reads the comments in the CNF file to determine the parameters of the problem instance and then checks the generated SAT solution to ensure that it is a valid solution to the problem. The Makefile is also configured to have the solvers time out if they run for more than 600 seconds.

Problem 4. (Level II):

The bash script g08.sh executes runs without proof generation for $n = 8$.

Try running this script. It will produce a number of data files. You can then use grep to check results for the runs. Here are some useful patterns to search:

Program	Search Pattern	Result
Either	"SATISFIABLE"	Whether or not the formula is satisfiable
Either	"SOLUTION VERIFIED"	Whether the generated solution is valid
TBSAT	"Performing Gauss-Jordan"	Information from Gaussian elimination phase
TBSAT	"Gauss-Jordan completed"	Information from Gaussian elimination phase
TBSAT	"Total clauses"	Size of the proof (if generated)
TBSAT	"Elapsed seconds"	TBSAT runtime
TBSAT	"Timeout"	Program timed out
KISSAT	"proof_added"	Size of the proof (if generated)
KISSAT	"process-time"	KISSAT runtime

Using those search terms, determine the following:

A. How did the number of satisfiable instances change as m increased?

B. Were all of the solutions for the satisfiable instances valid?

C. With TBSAT, how effective was Gaussian elimination in reducing the number of parity consraints?

D. How did the runtimes of KISSAT and TBSAT compare?

Problem 5. (Level II):

The bash script `g24.sh` executes runs without proof generation for $n = 24$.

Run this script an perform the same evaluations as you did for $n = 8$ in Problem 4.

Problem 6. (Level III):

Consider other experiments on the MDP problem that you can perform using the availabile tools. Some possible parameters to explore are as follows

n	m	k
8	16	2
8	24	3
16	32	4
16	40	5
16	48	6
24	48	6
24	56	7
24	64	8
28	56	7
28	64	8
28	72	9
32	64	8
32	72	9
32	80	10

Some things you could evaluate include:

A. How does the number of satisfiable instances vary with n and m?

B. How do the two SAT solvers perform on these problems?

C. For the unsatisfiable instances, how large are the proofs when proof generation is enabled?

D. How does the runtime change for the two solvers when proof generation is enabled?