

Algorithms and Data Structures III (course 1DL481)

Uppsala University – Spring 2023

Report for Assignment 1 by Team 7

Anderson LEONG

Louis HENKEL

8th February 2023

Problem 3: Boolean Satisfiability (SAT)

Task a: Ordered Resolution. Consider the following formula in conjunctive normal form (CNF):

$$\begin{aligned}\varphi \equiv & (x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge (x_5 \vee x_6) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_5) \\ & \wedge (\neg x_3 \vee \neg x_5) \wedge (\neg x_2 \vee \neg x_4) \wedge (\neg x_2 \vee \neg x_6) \wedge (\neg x_4 \vee \neg x_6)\end{aligned}$$

We apply resolution by selecting the variables in their own given order. In blue are the new clauses from the resolution.

$$\begin{aligned}\varphi \equiv & (x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge (x_5 \vee x_6) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_5) \\ & \wedge (\neg x_3 \vee \neg x_5) \wedge (\neg x_2 \vee \neg x_4) \wedge (\neg x_2 \vee \neg x_6) \wedge (\neg x_4 \vee \neg x_6) \\ \equiv & (x_2 \vee \neg x_3) \wedge (x_2 \vee \neg x_5) \wedge (x_3 \vee x_4) \wedge (x_5 \vee x_6) \wedge (\neg x_3 \vee \neg x_5) \\ & \wedge (\neg x_2 \vee \neg x_4) \wedge (\neg x_2 \vee \neg x_6) \wedge (\neg x_4 \vee \neg x_6) \\ \equiv & (\neg x_3 \vee \neg x_4) \wedge (\neg x_3 \vee \neg x_6) \wedge (\neg x_5 \vee \neg x_4) \wedge (\neg x_5 \vee \neg x_6) \\ & \wedge (x_3 \vee x_4) \wedge (x_5 \vee x_6) \wedge (\neg x_3 \vee \neg x_5) \wedge (\neg x_4 \vee \neg x_6) \\ \equiv & (x_4 \vee \neg x_6) \wedge (x_4 \vee \neg x_5) \wedge (\neg x_4 \vee \neg x_5) \wedge (\neg x_5 \vee \neg x_6) \\ & \wedge (x_5 \vee x_6) \wedge (\neg x_4 \vee \neg x_6) \\ \equiv & (\neg x_5 \vee \neg x_6) \wedge (\neg x_6) \wedge (\neg x_5) \wedge (x_5 \vee x_6) \\ \equiv & (\neg x_6) \wedge (x_6) \\ \equiv & ()\end{aligned}$$

As we reach the empty clause, the formula is unsatisfiable.

Task b: DPLL. Consider again the formula φ given in Task a. On the initial formula φ we have no unit clause and no pure literal, so we select the variable x_1 and run DPLL($\varphi \wedge x_1$): The formula we now apply DPLL is now:

$$\begin{aligned}\varphi_1 \equiv \varphi \wedge x_1 \equiv & (x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge (x_5 \vee x_6) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_5) \\ & \wedge (\neg x_3 \vee \neg x_5) \wedge (\neg x_2 \vee \neg x_4) \wedge (\neg x_2 \vee \neg x_6) \wedge (\neg x_4 \vee \neg x_6) \wedge (x_1)\end{aligned}$$

We have a unit clause now and make unit propagation with x_1

$$\begin{aligned}\varphi_2 \equiv & (x_3 \vee x_4) \wedge (x_5 \vee x_6) \wedge (\neg x_3) \wedge (\neg x_5) \\ & \wedge (\neg x_3 \vee \neg x_5) \wedge (\neg x_2 \vee \neg x_4) \wedge (\neg x_2 \vee \neg x_6) \wedge (\neg x_4 \vee \neg x_6)\end{aligned}$$

We have new unit clauses: First we apply the unit propagation on $\neg x_3$ and get:

$$\begin{aligned}\varphi_3 \equiv & (x_5 \vee x_6) \wedge (\neg x_5) \\ & \wedge (\neg x_2) \wedge (\neg x_2 \vee \neg x_6) \wedge (\neg x_4 \vee \neg x_6)\end{aligned}$$

We apply unit propagation on x_4 and get:

$$\begin{aligned}\varphi_4 \equiv & (x_5 \vee x_6) \wedge (\neg x_5) \\ & \wedge (\neg x_2) \wedge (\neg x_2 \vee \neg x_6) \wedge (\neg x_6)\end{aligned}$$

We apply unit propagation $\neg x_5$ and get:

$$\begin{aligned}\varphi_5 \equiv & (x_6) \wedge (\neg x_2) \\ & \wedge (\neg x_2 \vee \neg x_6) \wedge (\neg x_6)\end{aligned}$$

We apply unit propagation to x_6 and get:

$$\varphi_6 \equiv (\neg x_2) \wedge (\neg x_2) \wedge ()$$

and get an empty clause. We make backtracking and run $\text{DPLL}(\varphi \wedge \neg x_1)$. We make unit propagation with $\neg x_1$:

$$\begin{aligned}\varphi_7 \equiv \varphi \wedge \neg x_1 \equiv & (x_2) \wedge (x_3 \vee x_4) \wedge (x_5 \vee x_6) \wedge (\neg x_3 \vee \neg x_5) \\ & \wedge (\neg x_2 \vee \neg x_4) \wedge (\neg x_2 \vee \neg x_6) \wedge (\neg x_4 \vee \neg x_6)\end{aligned}$$

We make unit propagation with x_2 and get:

$$\begin{aligned}\varphi_8 \equiv & (x_3 \vee x_4) \wedge (x_5 \vee x_6) \wedge (\neg x_3 \vee \neg x_5) \\ & \wedge (\neg x_4) \wedge (\neg x_6) \wedge (\neg x_4 \vee \neg x_6)\end{aligned}$$

We make unit propagation with $\neg x_4$ and get:

$$\begin{aligned}\varphi_9 \equiv & (x_3) \wedge (x_5 \vee x_6) \\ & \wedge (\neg x_3 \vee \neg x_5) \wedge (\neg x_6)\end{aligned}$$

We we make unit propagation with x_3 and get:

$$\varphi_{10} \equiv (x_5 \vee x_6) \wedge (\neg x_5) \wedge (\neg x_6)$$

We make unit propagation with $\neg x_5$ and get:

$$\varphi_{11} \equiv (x_6) \wedge (\neg x_6)$$

We make unit propagation with x_6 and get:

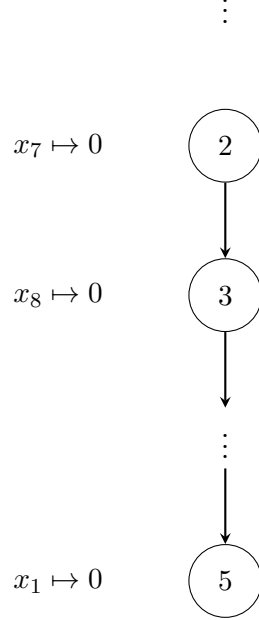
$$\varphi_{12} \equiv ()$$

Hence we get $\varphi = \text{UNSAT}$.

Task c: CDCL. Consider the following CNF formula:

$$(x_1 \vee x_8 \vee \neg x_2) \wedge (x_1 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_4 \vee \neg x_5) \wedge (x_7 \vee \neg x_4 \vee \neg x_6) \wedge (x_5 \vee x_6)$$

Assume that x_7 has been assigned 0 at decision level 2, and that x_8 has been assigned 0 at decision level 3. Moreover, assume that the current decision assignment is $x_1 = 0$ at decision level 5.



Using **green** text for trusted literals and gray for false ones we get the formula:

$$(x_1 \vee x_8 \vee \neg x_2) \wedge (x_1 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_4 \vee \neg x_5) \wedge (x_7 \vee \neg x_4 \vee \neg x_6) \wedge (x_5 \vee x_6)$$

Task d: Encoding. Describe your encoding, citing either [?], or [?, Section 2.2.2], or both, if you use their ideas: first explain the meaning of the Boolean variables that you use in your formula $\varphi_{d,c,e}$; then explain the encodings by the help functions of the hint that you actually use (there is no need to explain $\text{ATMOST}(k, x_1, \dots, x_n)$ if you use [?]); and finally explain how the constraints of the problem are encoded using those variables and help functions.

We chose the programming language $\langle \dots \rangle$, for which a compiler or interpreter is available on the Linux computers of the IT department. All source code is **uploaded** with this report. The compilation and running instructions are $\langle \dots \rangle$.

We validated the correctness of our encoding and implementation by **checking its outputs on many instances via the provided polynomial-time solution checker**.

Task e: Experiments. We chose the SAT solver **MiniSat** for our experiments. We **used or did not use** the provided script for running the experiments and tabling their results **under Linux Ubuntu 18.04 (64 bit) on an Intel Xeon E5520 of 2.27 GHz, with 4 processors of 4 cores each, with a 70 GB RAM and an 8 MB L3 cache (a ThinLinc computer of the IT department)**.

The results are in Table 1. The trivially unsatisfiable instances (which are the ones that violate the inequality $e \leq \left\lfloor \frac{d-1}{c-1} \right\rfloor$) that **were actually attempted in our experiments** are $\langle 8, 2, 8 \rangle$, $\langle 10, 2, 10 \rangle$, $\langle 12, 2, 12 \rangle$, $\langle 14, 2, 14 \rangle$, $\langle 16, 2, 16 \rangle$, $\langle 15, 3, 8 \rangle$, $\langle \dots \rangle$, $\langle 16, 4, 6 \rangle$, $\langle \dots \rangle$, and $\langle \dots \rangle$. We observe that our encoding detects their trivial unsatisfiability in $\langle \dots \rangle$ time.

d	c	e	status	time	d	c	e	status	time	d	c	e	status	time
8	2	7	sat		12	3	4	sat		16	4	5	sat	
8	2	8	unsat		12	3	5	unsat		16	4	6	unsat	
10	2	9	sat		15	3	6	sat		20	4	4	sat	
10	2	10	unsat		15	3	7	sat		20	4	5	sat	
12	2	11	sat		15	3	8	unsat		20	4	6	?	
12	2	12	unsat		18	3	6	sat		24	4	4	sat	
14	2	12	sat		18	3	7	sat		24	4	5	sat	
14	2	13	sat		18	3	8	?		24	4	6	?	
14	2	14	unsat		21	3	6	sat		28	4	4	sat	
16	2	10	sat		21	3	7	sat		28	4	5	sat	
16	2	11	sat		21	3	8	sat		28	4	6	sat	
16	2	12	sat		21	3	9	?		28	4	7	?	
16	2	13	sat		24	3	6	sat		32	4	3	sat	
16	2	14	sat		24	3	...	sat		32	4	...	sat	
16	2	15	sat		24	3	9	sat		32	4	9	sat	
16	2	16	unsat		24	3	10	?		32	4	10	sat	

Table 1: Cruise design: satisfiability and runtime (in seconds) using [MiniSat](#), with a timeout of 60.0 CPU seconds; a timeout is denoted by ‘t/o’; if no timeout occurred, then proven satisfiability is denoted by ‘sat’ and proven unsatisfiability by ‘unsat’, else trivial unsatisfiability is denoted by ‘unsat’ and the unknown status is denoted by ‘?’.

Problem 4: SAT Modulo Theories (SMT)

Task a: Encoding of instructions. We encode the transition between program states as follows for each MiniASM instruction:

- `pushj`:
 - In English: $\langle \dots \rangle$.
 - In SMT syntax: $\langle \dots \rangle$.
- `pop`:
 - In English: $\langle \dots \rangle$.
 - In SMT syntax: $\langle \dots \rangle$.
- `dup`:
 - In English: $\langle \dots \rangle$.
 - In SMT syntax: $\langle \dots \rangle$.
- `plus`:
 - In English: $\langle \dots \rangle$.
 - In SMT syntax: $\langle \dots \rangle$.
- `neg`:
 - In English: $\langle \dots \rangle$.
 - In SMT syntax: $\langle \dots \rangle$.
- `read`:
 - In English: $\langle \dots \rangle$.
 - In SMT syntax: $\langle \dots \rangle$.
- `write`:
 - In English: $\langle \dots \rangle$.
 - In SMT syntax: $\langle \dots \rangle$.

Task b: Partial-correctness checker. Describe your encoding. The final `assert` call, which ensures that the SMT solver produces `unsat` when needed, is $\langle \dots \rangle$.

We chose the SMT solver **Z3** for our experiments. We chose the programming language $\langle \dots \rangle$, for which a compiler or interpreter is available on the Linux computers of the IT department. All source code is **uploaded** with this report. The compilation and running instructions are $\langle \dots \rangle$.

Let `swap` be the abbreviation of `push0; write; push1; write; push0; read; push1; read`, that is changing the order of the two top-most numbers on the stack:

1. The program `push10; read` is or is not reported by **Z3** to be partially correct, because $\langle \dots \rangle$.
2. The program `push1; dup; dup; write; read` $\langle \dots \rangle$

3. The program `push1; dup; read; dup; neg; plus; plus` $\langle \dots \rangle$
4. The program `push1; push0; read; write; push0; read; read` $\langle \dots \rangle$
5. The program `push10; push0; swap` $\langle \dots \rangle$
6. The program `push10; dup; read; swap; push1; plus; read; plus` $\langle \dots \rangle$
7. The program `push10; dup; push1; plus; dup; push1; plus; plus; plus` $\langle \dots \rangle$

When **Z3** produces `sat` for a partial-correctness check, the output of `get-model` means $\langle \dots \rangle$.

Task c: Partial-equivalence checker. Describe your encoding. The final `assert` call, which ensures that the SMT solver produces `unsat` when needed, is $\langle \dots \rangle$.

We chose the SMT solver **Z3** for our experiments. We chose the programming language $\langle \dots \rangle$, for which a compiler or interpreter is available on the Linux computers of the IT department. All source code is **uploaded** with this report. The compilation and running instructions are $\langle \dots \rangle$.

Assuming that variable x is stored at heap address 0 and variable y is stored at heap address 1, the program $t := x; x := y; y := t$, translated into

`push0; read; push1; read; push0; write; push1; write`

and the program $x := x + y; y := x - y; x := x - y$, translated into

`push0; read; push1; read; plus; dup; push1; read; neg; plus; dup; push1; write; neg; plus; push0; write`

are or are not reported by **Z3** to be partially equivalent, because $\langle \dots \rangle$. When **Z3** produces `sat` for a partial-equivalence check, the output of `get-model` means $\langle \dots \rangle$.

Task d: Extended language. We encode the transition between program states for the `cmp0` instruction as follows:

- In English: $\langle \dots \rangle$.
- In SMT syntax: $\langle \dots \rangle$.

For encoding the `jmpj` instruction (where $j \geq 0$), we propose $\langle \dots \rangle$. The difficulty of encoding one or more `jmpj` instructions lies in $\langle \dots \rangle$. With our approach for encoding `jmpj`, the partial correctness of programs is determined by $\langle \dots \rangle$.

Feedback to the Teachers