Algorithms and Data Structures 3 (course 1DL481)
Uppsala University – Spring 2025
Assignment 2

Prepared by Pierre Flener, Tjark Weber, Philipp Rümmer, and Gustav Björdal

— Deadline: 13:00 on Friday 28 February 2025—

The assignments exercise the lecture material that is ***not*** covered by the exam. You ***must*** fill in the skeleton report in the provided materials, by following its instructions. Read the **Submission Instructions** and **Grading Rules** at the end of this document ***before*** tackling the following problems upon reading them a first time. Note that a problem *weight* indicated below might ***not*** reflect the relative *time* effort (within the assignment) that you actually spend on that problem: some points might be easier to earn. It is strongly recommended to prepare and attend the help sessions, as huge time savings may ensue.

Problem 3: Boolean Satisfiability (SAT) (60% weight)

The aim of solving this problem is first to understand the algorithms in SAT solving technology on three small warm-up tasks and then to encode declaratively a problem (whose high-level description has no Boolean unknowns) into a formula in conjunctive normal form (CNF) over Boolean variables, towards potentially impressive performance.

Tasks: You ***must*** fill in the placeholders in the skeleton report in the provided materials, by following its instructions:[1]

A. **Ordered Resolution:** Consider the following CNF formula:

$$\varphi \equiv (x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge (x_5 \vee x_6) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_5) \\ \wedge (\neg x_3 \vee \neg x_5) \wedge (\neg x_2 \vee \neg x_4) \wedge (\neg x_2 \vee \neg x_6) \wedge (\neg x_4 \vee \neg x_6)$$

Perform ordered resolution (without reporting on the steps) on this formula. Select the variables in the order given by their index (i.e., x_1 before x_2 before ...). Based on your ordered resolution, is φ satisfiable? Why?

B. **DPLL:** Apply the DPLL algorithm (without reporting on the steps) to the formula φ in Task A. Select the variables in the order given by their index (i.e., x_1 before x_2 before ...) and assign value 1 (i.e., True) before value 0 (i.e., False). Perform unit propagation and apply the pure-literal rule where possible, in order to prune parts of the search space. Based on your DPLL run, is φ satisfiable? Why?

[1] Solo teams (except PhD students) may omit Task A, but they are encouraged to perform it nevertheless.

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 was assigned 0 at decision level 2, and that x_8 was assigned 0 at decision level 3. Moreover, assume that the current decision assignment is $x_1 = 0$ at decision level 5. Draw a possible resulting implication graph. Does the graph contain any conflicts? If so, then mark these clearly, and provide a conflict clause.

D. **Encoding:** A *cruise design* $\langle d, c, e\rangle$, with $d \geq c \geq 2$ and c dividing d, is an assignment over e evenings of d diners to $\frac{d}{c}$ tables of c chairs each in a restaurant of a cruise ship such that all diners sit every evening at a table, with only people they have not dined with yet.

For example, the following is an $\langle 8, 2, 7\rangle$ cruise design, the diners being named 1 to 8:

	table 1	table 2	table 3	table 4
evening 1	$\{1,5\}$	$\{2,6\}$	$\{3,7\}$	$\{4,8\}$
evening 2	$\{1,4\}$	$\{2,5\}$	$\{3,6\}$	$\{7,8\}$
evening 3	$\{1,3\}$	$\{2,4\}$	$\{5,7\}$	$\{6,8\}$
evening 4	$\{1,2\}$	$\{3,4\}$	$\{5,8\}$	$\{6,7\}$
evening 5	$\{1,6\}$	$\{2,7\}$	$\{3,8\}$	$\{4,5\}$
evening 6	$\{1,7\}$	$\{2,8\}$	$\{3,5\}$	$\{4,6\}$
evening 7	$\{1,8\}$	$\{2,3\}$	$\{4,7\}$	$\{5,6\}$

Write a toolchain, in a programming language for which a compiler or interpreter is available on the Linux computers of the IT department:

1. An executable called `cruDes` reads the problem parameters d, c, e as command-line arguments and writes to standard output a propositional formula $\varphi_{d,c,e}$ in DIMACS CNF that is satisfiable if and only if a cruise design $\langle d, c, e\rangle$ exists, so that it can be fed to a SAT solver (say MiniSat: see the AD3 Resources), say `./cruDes 8 2 7 > in; minisat in out`.

   Explain the meaning of the Boolean variables that you use in $\varphi_{d,c,e}$, and explain how the constraints of the problem are encoded using those variables. Do not worry too much about symmetric candidate solutions in the search space.

   Hints: See [1, Section 2.2.2] for the basics of CNF encodings. For example, based on [2], write a help function $\textsc{AtMost}(k, x_1, \ldots, x_n)$ that generates a set of $2nk + n - 3k - 1$ clauses that is satisfiable if and only if at most k of the n Boolean variables x_i are True, with $0 \leq k \leq n$. If pressed for time, then note that you do *not* need to understand this encoding fully. Optionally define other help functions that make your explanation of the encoding more modular. Remember De Morgan's laws, namely that $\neg(\alpha \wedge \beta)$ is equivalent to $\neg\alpha \vee \neg\beta$, which is a clause if α and β are (conjunctions of) Boolean literals, and that $\neg(\alpha \vee \beta)$ is equivalent to $\neg\alpha \wedge \neg\beta$. Remember that $\alpha \Rightarrow \beta$ is equivalent to $\neg\alpha \vee \beta$, which is a clause if α is a (conjunction of) Boolean literal(s) and β is a (disjunction of) Boolean literal(s).

2. For satisfiable instances, an executable called `cruDesPrint` reads the problem parameters d, c, e as command-line arguments and the solver output from standard input, say `./cruDes 8 2 7 > in; minisat in out; cat out | ./cruDesPrint 8 2 7`,

and writes to standard output a line with the space-separated values of d, c, e, followed by one line per row of an $e \times d$ matrix representing the solution, the diner names of each table being separated by *one* space and the tables of each evening being separated by *three* spaces. For example, the $\langle 8, 2, 7 \rangle$ cruise design above is represented by the provided file `cruDes-8-2-7.txt`. Inspect a few outputs visually in order to make sure that *all* tables have c chairs.

For satisfiable instances, you ***must*** at least experimentally pipe the `cruDesPrint` output into the provided polynomial-time solution checker, which is a Python script that reads a solution from standard input, in order to gain confidence in the correctness of your encoding, for example by appending "`| cruDesChecker`" to the command above. Note that our checker assumes that all tables have c chairs, hence for example $\{1\}\,\{4, 2, 3\}$ with $\{2, 4, 1\}\,\{3\}$ will wrongly pass the checker, because it is parsed as $\{1, 4\}\,\{2, 3\}$ with $\{2, 4\}\,\{1, 3\}$.

Experimentally make sure that the times of producing a correct encoding are short.

E. **Experiments.** Indicate the SAT solver and hardware that you chose for your experiments. Consider the following instances:

d	c	e
8	2	7..8
12	3	4..5

d	c	e
15	3	6..8
20	4	4..7

d	c	e
28	4	4..10
32	4	3..11

Using a timeout of at least 60 seconds per run, report the performance (satisfiability status and runtime, in seconds, as the sum of the encoding time and the solving time) over a ***single*** run per instance, as the recommended MiniSat solver is deterministic by default; a precision of one decimal place suffices.

For full automation, do ***not*** use a web interface to MiniSat, but rather install MiniSat, make it executable via `chmod +x minisat`, and configure our provided Python script `cruDesTabler` in order to run the experiments requested above (with as defaults MiniSat, a timeout of 60 seconds, and one decimal place): it runs `cruDesChecker` on each solution and generates a results table in LaTeX that plugs into the skeleton report and looks like the one there; a failed check yields no table and an indication of the failed instance.

Trivial unsatisfiability. A necessary, but ***not sufficient***, condition for satisfiability is $e \leq \left\lfloor \frac{d-1}{c-1} \right\rfloor$, as every diner meets every evening $c - 1$ new diners among the $d - 1$ other diners. Discuss, *only* by observation in your results table and *not* by modifying `cruDes`, how slowly your encoding proves the trivial unsatisfiability of instances that violate this inequality. Note that `cruDesTabler` indicates trivial unsatisfiability by 'unsat+' within the results table.

References

[1] S. Prestwich. CNF encodings. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 2, pages 75–97. IOS Press, 2009. Available at https://dx.doi.org/10.3233/978-1-58603-929-5-75 and https://www.researchgate.net/publication/242029085_CNF_encodings.

[2] C. Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In P. van Beek, editor, *CP 2005*, volume 3709 of *LNCS*, pages 827–831. Springer, 2005. Available at https://dx.doi.org/10.1007/11564751_73; extended and corrected ($\mathrm{LT}^{n,k}_{\mathrm{SEQ}}$ has $2nk + n - 3k - 1$ clauses) at http://www.carstensinz.de/techreports/CardConstraints.pdf.

Problem 4: SAT Modulo Theories (SMT) (40% weight)

The aim of solving this problem is to understand how program verification problems can be encoded into a constraint satisfaction problem that can be solved efficiently by SMT technology.

Background: Let us use the SMT-LIB language and an SMT solver to implement simple verification tools for software programs with a heap. For this, we consider a minimalist assembler language, MiniASM, with some similarities to Java bytecode. The *instructions* of the language are the following:

Instr ::=	push_j	push the number j onto the top of the operand stack
|	pop	pop the topmost number from the operand stack and discard it
|	dup	pop the topmost number, say a, from the operand stack, and push a two times onto the stack
|	plus	pop the two topmost numbers, say a and b, from the operand stack, and push the sum $a + b$ onto the stack
|	neg	pop the topmost number, say a, from the operand stack, and push the negated value $-a$ onto the stack
|	read	pop the topmost number, say a, from the operand stack, read the number, say b, at address a on the heap, and push b onto the stack
|	write	pop the two topmost numbers, say a (at the top) and b (below the top), from the operand stack, and write b at address a on the heap

A MiniASM *program* is a list of instructions. Every element stored on the heap or the operand stack is a 32-bit integer. We do not consider any form of alignment, that is values are stored on the heap at consecutive addresses.

For example, the following shows the execution of a MiniASM program that reads the values v_0 and v_1 stored at addresses 0 and 1 on the heap, and writes their sum back at address 1:

Instruction	**Heap**	**Stack**	**Stack Pointer**
	$@0 : v_0,\ @1 : v_1$	$\langle \dots \rangle$	x
push_0	$@0 : v_0,\ @1 : v_1$	$\langle \dots, 0 \rangle$	$x + 1$
read	$@0 : v_0,\ @1 : v_1$	$\langle \dots, v_0 \rangle$	$x + 1$
push_1	$@0 : v_0,\ @1 : v_1$	$\langle \dots, v_0, 1 \rangle$	$x + 2$
read	$@0 : v_0,\ @1 : v_1$	$\langle \dots, v_0, v_1 \rangle$	$x + 2$
plus	$@0 : v_0,\ @1 : v_1$	$\langle \dots, v_0 + v_1 \rangle$	$x + 1$
push_1	$@0 : v_0,\ @1 : v_1$	$\langle \dots, v_0 + v_1, 1 \rangle$	$x + 2$
write	$@0 : v_0,\ @1 : v_0 + v_1$	$\langle \dots \rangle$	x

Hints: One can encode *program state* in MiniASM using three SMT-LIB variables:

- the heap H of type `(Array (_ BitVec 32) (_ BitVec 32))`;

- the stack S of type `(Array (_ BitVec 32) (_ BitVec 32))`;
- an index variable SP of type `(_ BitVec 32)`, pointing to the topmost element of S.

To translate a program with n instructions into SMT-LIB, declare $n+1$ triplets (H_i, S_i, SP_i) and generate constraints that imply that each triplet (H_i, S_i, SP_i) represents the program state after executing the ith instruction.

Tasks: You ***must*** fill in the placeholders in the skeleton report in the provided materials, by following its instructions:[2]

A. For each MiniASM instruction, describe how to encode the transition between program states, first in plain English and then in SMT-LIB syntax, for which all you need is in our SMT slides. Use `bvneg` (arithmetic negation) and not `bvnot` (bit-wise negation) for this assignment. Use `store` instead of a quantified formula in order to encode array writes.

B. For each MiniASM program (1) to (7) below write an SMT-LIB script that verifies its partial correctness. We say that a program P is *partially correct* if P leaves a non-zero number as topmost element on the stack whenever P terminates, starting from an *arbitrary* program state. The script for each P should contain exactly one `check-sat` call, which produces `unsat` (note that this is ***not*** a typo) if and only if P is partially correct. State the final `assert` call, common for all programs, which ensures that an SMT solver produces `unsat` upon partial correctness. Explain what the output of `get-model` means in this context when an SMT solver produces `sat`.

$$\mathsf{push}_{10}; \mathsf{read} \tag{1}$$

$$\mathsf{push}_1; \mathsf{dup}; \mathsf{dup}; \mathsf{write}; \mathsf{read} \tag{2}$$

$$\mathsf{push}_1; \mathsf{dup}; \mathsf{read}; \mathsf{dup}; \mathsf{neg}; \mathsf{plus}; \mathsf{plus} \tag{3}$$

$$\mathsf{push}_1; \mathsf{push}_0; \mathsf{read}; \mathsf{write}; \mathsf{push}_0; \mathsf{read}; \mathsf{read} \tag{4}$$

$$\mathsf{push}_{10}; \mathsf{push}_0; \mathsf{swap} \tag{5}$$

$$\mathsf{push}_{10}; \mathsf{dup}; \mathsf{read}; \mathsf{swap}; \mathsf{push}_1; \mathsf{plus}; \mathsf{read}; \mathsf{plus} \tag{6}$$

$$\mathsf{push}_{10}; \mathsf{dup}; \mathsf{push}_1; \mathsf{plus}; \mathsf{dup}; \mathsf{push}_1; \mathsf{plus}; \mathsf{plus}; \mathsf{plus} \tag{7}$$

where swap is the abbreviation of $\mathsf{push}_0; \mathsf{write}; \mathsf{push}_1; \mathsf{write}; \mathsf{push}_0; \mathsf{read}; \mathsf{push}_1; \mathsf{read}$, that is for changing the order of the two top-most numbers on the stack.

Indicate the SMT solver that you chose for your experiments.

C. For the MiniASM programs (8) and (9) below write an SMT-LIB script that verifies their partial equivalence. We say that programs P and Q are *partially equivalent* if terminating runs of P and Q, starting from the *same* but *arbitrary* program state, lead to the same final heap. The script should contain exactly one `check-sat` call, which produces `unsat` (note that this is ***not*** a typo) if and only if the two programs are partially equivalent. State the final `assert` call, which ensures that an SMT solver produces `unsat` upon partial equivalence. Explain what the output of `get-model` means in this context when an SMT solver produces `sat`.

The considered programs aim at swapping the values of two integer variables x and y, assuming that x is stored at heap address 0 and y at heap address 1:

[2] Solo teams (except PhD students) may omit Tasks B.(6), B.(7), and D, but they are encouraged to perform them nevertheless.

- The program $t \coloneqq x;\ x \coloneqq y;\ y \coloneqq t$, where t is a local variable, can be translated as follows:

$$\mathsf{push}_0; \mathsf{read}; \mathsf{push}_1; \mathsf{read}; \mathsf{push}_0; \mathsf{write}; \mathsf{push}_1; \mathsf{write} \tag{8}$$

- The program $x \coloneqq x + y;\ y \coloneqq x - y;\ x \coloneqq x - y$ can be translated as follows:

$$\begin{aligned} &\mathsf{push}_0; \mathsf{read}; \mathsf{push}_1; \mathsf{read}; \mathsf{plus}; \mathsf{dup}; \mathsf{push}_1; \mathsf{read}; \\ &\mathsf{neg}; \mathsf{plus}; \mathsf{dup}; \mathsf{push}_1; \mathsf{write}; \mathsf{neg}; \mathsf{plus}; \mathsf{push}_0; \mathsf{write} \end{aligned} \tag{9}$$

Indicate the SMT solver that you chose for your experiments.

D. We now extend MiniASM by introducing three additional instructions:

Instr ::= ⋯

|	$\mathsf{cmp}_\circ$	pop the two topmost numbers, say a and b, from the operand stack, and push 1 onto the stack if $a \circ b$, and 0 otherwise; with $\circ \in \{=, \leq\}$
|	jmp_j	pop the topmost number, say a, from the operand stack; if a is zero, then continue with the next instruction, else skip the next j instructions; with $j \geq 0$

For example, the following code replaces the topmost stack element by its absolute value:

$$\mathsf{dup}; \mathsf{push}_0; \mathsf{cmp}_\leq; \mathsf{jmp}_1; \mathsf{neg}$$

Propose an approach to encode a program with $\mathsf{cmp}_\circ$ and jmp_j instructions (where $j \geq 0$) and to analyse the partial correctness of programs also in this extended language. In particular, identify and discuss the difficulty of encoding the jmp_j instruction, and propose an encoding based on your insights. The proposal should be detailed enough for another student to be able to implement the approach without any further reasoning, but you need not implement anything.

Note that the extended MiniASM language only provides forward jumps, so that programs always terminate, as it is not possible to encode loops.

Submit also the commented source code of your tools and the generated SMT-LIB scripts. For the experiments, either use the web interface of Z3, but then set the logic with `(set-logic QF_ABV)` so as to have both arrays (A) and bit vectors (BV), or install any SMT solver on your own computer. You need neither report the runtimes nor make multiple runs per verification (even though SMT solvers are usually randomised), as the solving times should be very short here, the main difficulty lying here in the encoding phase.

Submission Instructions

1. The report instructions are in comments that start with '%%' in the LaTeX source of the skeleton report and should be followed even when not using LaTeX. All running text should be black: comment away line 19 (which typesets the placeholders in blue) and uncomment line 20 (which typesets them in black).

2. Spellcheck the report and the comments in ***all*** code, in order to show both self-respect and respect for your readers. Thoroughly proofread the report, at least once per teammate, and ideally grammar-check it. In case you are curious about technical writing: see the English Style Guide of UU, the technical-writing Check List & Style Manual of the Optimisation research group, a list of common errors in English usage, and a list of common errors in English usage by native Swedish speakers.

3. Match ***exactly*** the uppercase, lowercase, and layout conventions of any filenames and I/O texts imposed by the tasks, as we will process submitted source code automatically. Do ***not*** rename any of the provided skeleton codes, for the same reason. However, do not worry when *Studium* appends a version number to the filenames when you make multiple submission attempts until the deadline.

4. Remember that when submitting you implicitly certify (a) that your report and all its uploaded attachments were produced solely by your team, except where explicitly stated otherwise and clearly referenced, (b) that each teammate can individually explain any part starting from the moment of submitting your report, and (c) that your report and attachments are not freely accessible on a public repository.

5. Submit (by only ***one*** of the teammates) by the given ***hard*** deadline three files via *Studium*: your report as a ***PDF*** file (all other formats will be rejected) and two compressed folders in ***ZIP*** format with all source code required to run your experiments of each problem.

Grading Rules

For each problem:
If the requested source code is uploaded, ***and*** it runs *without* compilation or runtime errors on a Linux computer of the IT department under the compiler, interpreter, or solver that you indicate, ***and*** it computes *correct* and (*near-*)*optimal* solutions in *reasonable* time to *some* of our tests on that hardware, ***then*** you get a score of at least 1 point (read on), ***else*** your *final* score is 0 points. Furthermore:

- ***If*** the code *passes most* of our tests ***and*** the report addresses *all* tasks and subtasks in a *good enough* way, ***then*** you get a *final* score of 3 or 4 or 5 points and are *not* invited to the grading session for this problem.

- ***If*** the code *fails too many* of our tests, ***or*** the report does *not* address all tasks and subtasks, ***or*** *some* task or subtask answers have *severe errors*, ***then*** you get an *initial* score of 1 or 2 points and *might* be invited to the grading session for this problem, at the end of which you informed whether your initial score is increased or not by 1 point into your *final* score. A non-invitation leads to your final score being the initial one, and the same holds for each invited student who makes a no-show.

Also, ***if*** an assistant figures out a minor fix that is needed to make your code run as per our instructions above, ***then***, instead of giving 0 points, the assistant may at their discretion deduct 1 point from the score earned upon the fix.

Let s_i be your final score on Problem i, whose stated weight is w_i. Your final score on the entire assignment is $\sum_i (w_i \cdot 2 \cdot s_i)$, rounded to the ***nearest*** (and hence possibly previous) integer, provided $s_i > 0$ for both problems.

Considering there are three help sessions for each assignment, you ***must*** earn at least 3 points (of 10) on each assignment until the end of its grading session, including at least 1 point (of 5) on each problem and at least 10 points (of 20) over both assignments, in order to pass the *Assignments* part (2 credits) of the course, if you attend the guest lecture from industry.