Problem 3: Boolean Satisfiability (SAT)

Tasks: You must follow the structure of the demo report!

a. Modelling: 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>
<thead>
<tr>
<th></th>
<th>table 1</th>
<th>table 2</th>
<th>table 3</th>
<th>table 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>evening 1</td>
<td>{1, 5}</td>
<td>{2, 6}</td>
<td>{3, 7}</td>
<td>{4, 8}</td>
</tr>
<tr>
<td>evening 2</td>
<td>{1, 4}</td>
<td>{2, 5}</td>
<td>{3, 6}</td>
<td>{7, 8}</td>
</tr>
<tr>
<td>evening 3</td>
<td>{1, 3}</td>
<td>{2, 4}</td>
<td>{5, 7}</td>
<td>{6, 8}</td>
</tr>
<tr>
<td>evening 4</td>
<td>{1, 2}</td>
<td>{3, 4}</td>
<td>{5, 8}</td>
<td>{6, 7}</td>
</tr>
<tr>
<td>evening 5</td>
<td>{1, 6}</td>
<td>{2, 7}</td>
<td>{3, 8}</td>
<td>{4, 5}</td>
</tr>
<tr>
<td>evening 6</td>
<td>{1, 7}</td>
<td>{2, 8}</td>
<td>{3, 5}</td>
<td>{4, 6}</td>
</tr>
<tr>
<td>evening 7</td>
<td>{1, 8}</td>
<td>{2, 3}</td>
<td>{4, 7}</td>
<td>{5, 6}</td>
</tr>
</tbody>
</table>

Describe a propositional formula \(\varphi_{d,c,e}\) in conjunctive normal form (CNF) that is satisfiable if and only if a cruise design \(\langle d, c, e \rangle\) exists. Explain the meaning of the Boolean variables that you use in \(\varphi_{d,c,e}\). Do not worry too much about symmetric candidate solutions in the search space.

Hints: Use the compact encoding in [1] of the \textsc{atmost}(\(k, x_1, \ldots, x_n\)) constraint, where at most \(k\) of the \(n\) Boolean variables \(x_i\) are true, with \(0 \leq k \leq n\). Other useful modelling abstractions are the \textsc{andReif}(\(x_1, \ldots, x_n, b\)) constraint, where the Boolean variable \(b\) is true if (but not only if) all the \(n\) Boolean variables \(x_i\) are true, and the \textsc{orReif}(\(x_1, \ldots, x_n, b\)) constraint, where the Boolean variable \(b\) is true if (but not only if) at least one of the \(n\) Boolean variables \(x_i\) is true; these are easy to encode compactly.

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` must read the space-separated instance parameters $d$, $c$, and $e$ from standard input, say `cruDes 8 2 7`, and write $\varphi_{d,c,e}$ in DIMACS CNF to standard output, so that it can be fed to a SAT solver.

(2) For satisfiable instances, an executable called `cruDesPrint` must read from standard input one line with the space-separated instance parameters $d$, $c$, and $e$, followed by the SAT solver output, and write to standard output a line with the space-separated values of $d$, $c$, and $e$, followed by one line per row of an $e \times d$ matrix representing the solution, the diner names being space-separated. For example, the $(8, 2, 7)$ cruise design on the previous page is represented by `http://user.it.uu.se/~pierref/courses/AD3/assignments/cruDes-8-2-7.txt`.

For satisfiable instances, you can pipe the output into the polynomial-time solution checker at `http://user.it.uu.se/~pierref/courses/AD3/assignments` (for Linux and macOS), which reads such a solution from standard input, in order to gain confidence in the correctness of your encoding.

b. Report, with plots or tables, the performance (satisfiability and runtime) of your encoding on at least the following 30 instances:

<table>
<thead>
<tr>
<th>$d$</th>
<th>$c$</th>
<th>$e$</th>
<th>$d$</th>
<th>$c$</th>
<th>$e$</th>
<th>$d$</th>
<th>$c$</th>
<th>$e$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2</td>
<td>7</td>
<td>12</td>
<td>3</td>
<td>4</td>
<td>12</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>8</td>
<td>12</td>
<td>3</td>
<td>5</td>
<td>12</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>9</td>
<td>15</td>
<td>3</td>
<td>7</td>
<td>16</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>10</td>
<td>15</td>
<td>3</td>
<td>8</td>
<td>16</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>11</td>
<td>18</td>
<td>3</td>
<td>4</td>
<td>20</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>12</td>
<td>18</td>
<td>3</td>
<td>5</td>
<td>20</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>12</td>
<td>21</td>
<td>3</td>
<td>2</td>
<td>24</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>13</td>
<td>21</td>
<td>3</td>
<td>3</td>
<td>28</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>9</td>
<td>24</td>
<td>3</td>
<td>1</td>
<td>32</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>10</td>
<td>24</td>
<td>3</td>
<td>2</td>
<td>36</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Use a time-out of at least 60 seconds per run and report the performance (satisfiability status and runtime) over a single run per instance, as the recommended MiniSAT solver is deterministic by default; a precision of one decimal place suffices here.

For every pair $(d, c)$ above, increase $e$ beyond the given values as long as your encoding yields no time-out or memory-out. 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: how fast does your encoding detect the trivial unsatisfiability of instances violating this inequality?

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

$$
\varphi \equiv (x_1 \lor x_2) \land (x_3 \lor x_4) \land (x_5 \lor x_6) \land (\neg x_1 \lor \neg x_3) \land (\neg x_1 \lor \neg x_5) \\
\land (\neg x_3 \lor \neg x_5) \land (\neg x_2 \lor \neg x_4) \land (\neg x_2 \lor \neg x_6) \land (\neg x_4 \lor \neg x_6)
$$

Perform ordered resolution on this formula, selecting the variables in the order given by their index (i.e., $x_1$ before $x_2$ before $\ldots$). Show the result after each iteration. Based on your resolution, is $\varphi$ satisfiable?
d. **DPLL:** Consider again the formula $\varphi$ given in Task c. Explain in detail how the DPLL algorithm, when applied to $\varphi$, determines whether the formula is satisfiable. Assume that the variables are selected in the order given by their index (i.e., $x_1$ before $x_2$ before $\ldots$), and that they are assigned 1 (i.e., True) before they are assigned 0 (i.e., False). Remember to perform unit propagation and to apply the pure-literal rule where possible, in order to prune parts of the search space.

e. **CDCL:** Consider the following CNF formula:

$$(x_1 \lor x_8 \lor \neg x_2) \land (x_1 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_4 \lor \neg x_5) \land (x_7 \lor \neg x_4 \lor \neg x_6) \land (x_5 \lor x_6)$$

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. Draw the resulting implication graph. Does the graph contain any conflicts? If so, then mark these clearly, and provide a conflict clause.

Solo teams may omit Task c.

**References**


**Problem 4: SAT Modulo Theories (SMT)**

**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:

$$Instr ::= push_j \quad \text{push the number } j \text{ onto the operand stack}$$

$$| \quad \text{pop} \quad \text{pop a number from the operand stack and discard it}$$

$$| \quad \text{dup} \quad \text{duplicate the topmost number on the operand stack}$$

$$| \quad \text{plus} \quad \text{pop the two topmost numbers, say } a \text{ and } b, \text{ from the operand stack, and push the sum } a + b \text{ onto the stack}$$

$$| \quad \text{neg} \quad \text{pop the topmost number, say } a, \text{ from the operand stack, and push the negated value } -a \text{ onto the stack}$$

$$| \quad \text{read} \quad \text{pop the topmost number, say } a, \text{ from the operand stack, read the number, say } b, \text{ at address } a \text{ on the heap, and push } b \text{ onto the stack}$$

$$| \quad \text{write} \quad \text{pop the two topmost numbers, say } a \text{ and } b, \text{ from the operand stack, and write } b \text{ at address } a \text{ on the heap}$$

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 simply at consecutive addresses.

3
For example, the following MiniASM program reads the values $v_0$ and $v_1$ stored at addresses 0 and 1 on the heap, and writes their sum back at address 0:

```
push 0  Stack: ⟨⟩
read   Stack: ⟨0⟩
push 1  Stack: ⟨v_0⟩
read   Stack: ⟨v_0, 1⟩
plus   Stack: ⟨v_0 + v_1⟩
push 0  Stack: ⟨v_0 + v_1, 0⟩
write  Stack: ⟨⟩
```

**Tasks:** Write two verification tools, in a programming language for which a compiler or interpreter is available on the Linux computers of the IT department, for the following purposes:

a. Take a MiniASM program $P$ and generate an SMT-LIB script that encodes the partial correctness of $P$. That is, the script should contain exactly one `check-sat` call, which produces `unsat` if and only if $P$ is partially correct: we say that $P$ is partially correct if $P$ leaves a non-zero number as topmost element on the stack whenever $P$ terminates.

Use your correctness checker to find out whether the following program is partially correct:

```
push_1; push_0; read; write; push_0; read; read
```

Explain the outcome of the verification process.

b. Take two MiniASM programs $P$ and $Q$ and generate an SMT-LIB script that encodes the equivalence of $P$ and $Q$. That is, the script should contain exactly one `check-sat` call, which produces `unsat` if and only if $P$ and $Q$ are partially equivalent: we say that $P$ and $Q$ are partially equivalent if terminating runs of $P$ and $Q$, starting from the same initial stack and heap, lead to the same final heap.

Use your equivalence checker to find out whether the following two programs for swapping the values of two integer variables $x$ and $y$ are partially equivalent:

- Assuming that $x$ is stored at heap address 0, and $y$ at heap address 1, the temporary variable $t$ of the program $t := x; x := y; y := t$ can be represented using the stack:
  
  ```
push_0; read; push_1; read; push_0; write; push_1; write
  ```

- Assuming that $x$ is stored at heap address 0, and $y$ at heap address 1, the program $x := x + y; y := x - y; x := x - y$ can be translated as follows:

  ```
push_0; read; push_1; read; plus; dup; push_1; read; neg; plus; dup; push_1; write; neg; plus; push_0; write
  ```

Explain the outcome of the verification process.

Solo teams may omit Task b. The tools need not include parsers and may hardcode the considered programs. Submit also the commented source codes of your tools and the generated SMT-LIB constraints. For the experiments, either use the web interface of Z3 at [http://rise4fun.com/z3](http://rise4fun.com/z3) 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.
Hints: One can encode program state in MiniASM using three SMT-LIB variables:

- the heap $H$ of type \( \text{Array} \ (\_ \ 	ext{BitVec} \ 32) \ (\_ \ 	ext{BitVec} \ 32) \);
- the stack $S$ of type \( \text{Array} \ (\_ \ 	ext{BitVec} \ 32) \ (\_ \ 	ext{BitVec} \ 32) \);
- an index variable $SP$ of type \( \_ \ 	ext{BitVec} \ 32 \), 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 $i$th instruction.

Submission Instructions

All task answers, other than source code, must be in a single report in PDF format; all other formats are rejected. Furthermore:

- Identify the team members and state the team number inside the report.
- Certify that your report and all its uploaded attachments were produced solely by your team, except where explicitly stated otherwise and clearly referenced, that each teammate can individually explain any part starting from the moment of submitting your report, and that your report and attachments are not and will not be freely accessible on a public repository.
- State the problem number and task identifier for each answer in the report.
- Take the instructions of the demo report at [http://user.it.uu.se/~pierref/courses/AD3/demoReport](http://user.it.uu.se/~pierref/courses/AD3/demoReport) as a strict guideline for document structure and content, as well as an indication of its expected quality.
- Write clear task answers, source code, and comments.
- Justify all task answers, except where explicitly not required.
- State any assumptions you make that are not in this document.
- Thoroughly proofread, spellcheck, and grammar-check your report.
- Upload all source code required to run your experiments, and include running instructions.
- Match exactly the uppercase, lowercase, and layout conventions of any filenames and I/O texts imposed by the tasks, as we may process your source code automatically.
- Write a paragraph, which will not be graded, describing your experience with this assignment. Which tasks were too difficult or too easy? Which tasks were interesting or boring? This will help us improve the course in the coming years.

Only one of the teammates submits the solution files (one PDF report with answers to all the tasks, plus all source code files), without folder structure and without compression, via the Student Portal, whose clock may differ from yours, by the given hard deadline.
Grading Rules

For each problem: *If* the requested source code exists, *and* runs without runtime errors on the Linux/Unix computers of the IT department under the compiler, interpreter, or solver you indicate, *and* computes (near-)optimal solutions to some of our chosen instance data in reasonable time on our chosen hardware, *and* has all the requested task answers in the report, *then* you get at least 1 point (of 5), *otherwise* you get 0 points. Furthermore:

- If your code *passes most* of our grading tests, then you get 4 or 5 points, depending also on the quality of the report part for this problem; you are not invited to the grading session for this problem.

- If your code *fails many* of our grading tests, then you get an initial mark of 1 or 2 points, depending also on the quality of the report part for this problem; you are invited to the grading session for this problem, where you can try and increase your initial mark by 1 point into your final mark.

However, if the assistant figures out a minor fix that is needed to make your source code run as per your and our instructions, then, instead of giving 0 points up front, the assistant may deduct 1 point at his discretion.

Considering that there are two help sessions for each assignment, you must get minimum 3 points (of 10) on each assignment, including minimum 1 point (of 5) on each problem, until the end of its grading session, and minimum 10 points (of 20) over both assignments; otherwise you fail the *Assignments* part of the course.