For this project, students must work in groups (of 3 people ideally, but there can be groups of 4 if need be), select one of the topics suggested here, or choose their own topic, which must be approved by the TD lecturer.

1 Planning

— This document is online from the very beginning of the course, students may ask for a printed version if they want one.
— You have to have chosen your group and topic at the end of the second week of TD.
— Delivery, by email to the TD lecturer, of the pre-report describing the modelization part of the project (max 5 pages) no later than March 1st 2017.
— Delivery, by email to the TD lecturer, of the final report no later than April 26th 2017.
— Group presentation of your project during the week of May 2nd-5th (20 minute presentations, question time included).

2 Description of the Project

The project consists in creating a tool that automatically solves problems formalized with clauses.

Part 1. For the first part of the project, you will have to write a pre-report in which:
1. you describe the topics you chose in English (rules, examples, constraints, etc.);
2. you translate the problems to (propositional or first-order) logic;
3. you model the problems in conjunctive normal form (product of clauses).
In addition:
4. you will program an interface that lets users manually enter various instances of the problems.
5. your interface shall be able to create a DIMACS file encoding any instance of a problem as a formula in conjunctive normal form.

Your interface should be able to take its entry at least from a file, and optionally, from the keyboard or a graphical interface.

1. if you are using JAVA, you can mimic the example found at www-verimag.imag.fr/~devismes/JAVA/Fichier.java.
Part 2. In the second phase of the project, once the problems have been modelled and validated by your TD lecturer, each group must transform the model of their problems into equivalent 3-SAT clauses (you can use the algorithm proposed in exercise 50 p. 10 of your exercise handout). You will then choose a SAT-solver\(^2\) that you will use to solve the instances of your problems. You will then have to produce:

1. a program in the language of your choice:
   - (a) which reads a DIMACS file and returns a set of 3-SAT clauses in DIMACS format,
   - (b) that you will test using the SAT clauses of the various modeled problems.
2. a program that, given a trace of execution from the SAT-solver you chose, will display an understandable solution to the instance of your problem;
3. a report in which
   - you explain the implementation of your programs,
   - you illustrate the use of your programs with relevant examples,
   - you give the source code of your programs;
4. a group presentation during which you will run your programs on your personal computer, or a university computer. At a minimum, your programs should:
   - allow a user to enter valid instances of your chosen problem,
   - display the DIMACS file for the problem instance entered,
   - display an understandable solution to the problem found by the SAT-solver.

Note: The content of both the pre-report and final report will be graded (30% de la note finale). The presentation is MANDATORY (unless you get an official exemption), any group member absent from the presentation will receive the grade 0 for his project. Each group member should contribute equally to the presentation. The presentation should cover the topic of your project, and a demonstration of your programs at work.

SAT and \(n\)-SAT. Knowing if a propositional logic formula in conjunctive normal form (CNF) is satisfiable is called "satisfiability problem" or "SAT problem". When a CNF formula only contains clauses of \(n\) literals, the problem is called " \(n\)-SAT problem".

\(^2\) For instance, MiniSat, Z3, SatTime, SAT4j, SATzilla, precosat, MXC, clasp, SAppeloT, TNM, gNovelty2+, Reat, Picosat, Minisat, Zchaff, Jerusat, Satzoo, Linniat, Berkmin, OKSolver, ManySAT 1.1, ... (http://www.satcompetition.org/).
**DIMACS format:** The format used as input to SAT-solvers is an international standard used for encoding formulas in conjunctive normal form. A file in the DIMACS format begins with a line specifying that the formula is in normal form, the number of variables in the formula, and how many clauses it contains. For example, \texttt{p cnf 5 3} says that the file contains a formula in conjunctive normal form with 5 variables and 3 clauses. Then, the following lines describe the clauses, one on each line. Each line contains positive or negative integers, and ends with a zero. A positive integer \(i\) indicates that the \(i^{th}\) variable appears in the clause, whereas a negative integer \(-i\) indicates that the negation of the \(i^{th}\) variable appears in the clause. For example, here's a formula, followed by its encoding using the DIMACS format:

\[
(x_1 \lor \neg x_5 \lor x_4) \land (\neg x_1 \lor x_5 \lor x_3 \lor x_4) \land (\neg x_3 \lor \neg x_4)
\]

```plaintext
c
```
c
```plaintext
c
```
p cnf 5 3
1 -5 4 0
-1 5 3 4 0
-3 -4 0
```

3 Examples of Possible Topics

Any logical game that consists in filling a grid with all the useful information available can be modeled. For instance, you could tackle:
- Sudoku
- Tetravex
- Master Mind
- Squaro
- Lights Out
- Nonogram
- Any other topic validated by your TD lecturer

Other topics will be listed on page http://inf242.forge.imag.fr together with links to demo applets.
4 Example with the Pigeon Problem

We illustrate our minimum expectations for this project with a simple problem.

4.1 Problem

A pigeon-fancier owns \( n \) nests and \( p \) pigeons. He wishes that:
- each pigeon lives in a nest,
- there be at most one pigeon per nest
How can we help him with logic?

4.2 Modeling the Problem Using First Order Logic

Predicate \( P(i, j) \) is true if and only if pigeon \( i \) is in nest \( j \). The constraints of the problem are modelled as follows:
- Each pigeon lives in a nest : \( \forall i, \exists j, P(i, j) \).
- There is at most one pigeon in each nest : \( \forall i, \forall k, i \neq k \implies \forall j, P(i, j) \lor P(k, j) \).

4.3 Modelization in Conjunctive Normal Form

The boolean variable \( x_{i,j} \) is true if and only if pigeon \( i \) lives in nest \( j \). For example, for a set of pigeons \( \{a, b, c\} \) and a set of nests \( \{1, 2, 3, 4\} \), the following constraints must be given to the SAT-solver:

\[
(x_{a,1} \lor x_{a,2} \lor x_{a,3} \lor x_{a,4}) \\
\land (x_{b,1} \lor x_{b,2} \lor x_{b,3} \lor x_{b,4}) \\
\land (x_{c,1} \lor x_{c,2} \lor x_{c,3} \lor x_{c,4}) \\
\land (x_{a,1} \lor \overline{x_{b,1}}) \land (x_{a,1} \lor \overline{x_{c,1}}) \land (x_{b,1} \lor \overline{x_{c,1}}) \\
\land (x_{a,2} \lor \overline{x_{b,2}}) \land (x_{a,2} \lor \overline{x_{c,2}}) \land (x_{b,2} \lor \overline{x_{c,2}}) \\
\land (x_{a,3} \lor \overline{x_{b,3}}) \land (x_{a,3} \lor \overline{x_{c,3}}) \land (x_{b,3} \lor \overline{x_{c,3}}) \\
\land (x_{a,4} \lor \overline{x_{b,4}}) \land (x_{a,4} \lor \overline{x_{c,4}}) \land (x_{b,4} \lor \overline{x_{c,4}})
\]

4.4 Resolution by a SAT-solver

After translating the cnf formula into the DIMACS format, a SAT-solver will give an assignment to each variable satisfying the system of constraints. Using this, we can deduce how to place each pigeon. We note that while this problem is very easy to solve manually, the computer will have to solve an exponential number of constraints to solve it using logic.

5 Optional part : SAT solver

For the most advanced groups, you have the possibility of programming your own SAT solver (in your favourite programming language) with several resolution strategies.

Your solver will be a WalkSat type solver (see paragraph below). It will take as input the set of clauses to resolve in the form of a DIMACS file. The aim is then to return an assignment satisfying all clauses in that set, whenever possible. Notice that WalkSat
solvers are \textit{incomplete}: if the set of clauses given as input is contradictory the solver will not be able to determine it.

The pseudo-code of the \textit{WalkSat} algorithm is given below as a reference.

1: Draw an assignment $v$ at random according to a uniform distribution
2: $i = 0$
3: \textbf{WHILE} ($v$ is not a model) and $i < N$ \textbf{DO}
4: \hspace{1em} Pick a clause $C$ amongst clauses $C'$ such that $v(C') = \text{false}$
5: \hspace{1em} Draw a real number $q$ in $[0,1]$ according to a uniform distribution
6: \hspace{1em} \text{IF} $q < P$ \text{THEN}
7: \hspace{2em} Pick a variable $x$ in $C$ according to a uniform distribution
8: \hspace{1em} \text{ELSE}
9: \hspace{2em} Pick a variable $x$ in $C$ deterministically
10: \hspace{1em} \text{END IF}
11: \hspace{1em} Flip the value of $v(x)$ in the assignment $v$
12: \hspace{1em} $i++$
13: \textbf{END WHILE}
14: \textbf{IF} $v$ is a model \textbf{THEN}
15: \hspace{1em} RETURN $v$
16: \textbf{ELSE}
17: \hspace{1em} RETURN "undecided"
18: \textbf{ENDIF}

The code above has gaps, and that is on purpose... Indeed it does not specify the values of $N$, $P$, or the procedure to deterministically choose variable $x$ at line 9. It is up to you to determine a reasonable value for such constants, and to choose variable $x$ so that it minimizes the number of unsatisfied clauses.

Optionally, you may experiment so as to find the best possible values of $N$ and $P$, and program other heuristics for the choice of $x$, for instance:

- Choose the least modified variable of $C$;
- Assign to each variable $y$ a score with computed as the difference between the number of positive occurences, and negative occurences across clauses. Then choose the variable with the best score;
- Heuristics inspired from MOMS and JW heuristics given in the course notes;
- A mix some of these strategies;
- Etc.

In that case it would be interesting to compare the different heuristics that you implemented.

Note that to simplify your implementation, you can assume that the set of clauses received has been preprocessed to get an equivalent 3-SAT problem (see beginning of Part 2 of this document).