Advanced algorithms

Exercise sheet #5 (Solutions) – SAT solvers

October 26, 2022

Exercice 1 (Lazy data structures)**.** DPLL algorithm (Davis, Putnam, Logemann, Loveland) is a backtracking algorithm to solve SAT problems. It uses a procedure called ASSIGN(P, v, b) which, given a CNF SAT formula P, a variable v and a boolean b, returns a CNF SAT formula P', equivalent to $P \wedge v$ (if b is TRUE) or $P \wedge \neg v$ (if b is FALSE). The trivial way to implement ASSIGN is to simplify P as much as possible after specializing v. But this is costly, so it is worth trying to do the least possible work. We assume the following properties hold, where $P' = $ ASSIGN(P, v, b):

- ◯ conflicts are detected: if some clause of P evaluates to FALSE then $P' =$ FALSE (This yields to either a backtracking or returning UNSAT if no backtracking is possible);
- □ full propagation: if P is satisfied by $v \leftarrow b$, then $P' = \varnothing$.

(a) Write formally a backtracking algorithm for solving SAT problems using ASSIGN.

Solution —

Intput: *a CNF SAT instance P*
Output: TRUE *if P has a solution,* FALSE *otherwise*

```
procedure SOLVE($P$)
    if $P =$ FALSE then
        return FALSE
    else if $P = \varnothing$ then
        return TRUE
    else
        $v \leftarrow$ a free variable in $P$
        if SOLVE(ASSIGN($P, v$, TRUE)) then
            return TRUE
        else if SOLVE(ASSIGN($P, v$, FALSE)) then
            return TRUE
        else
            return FALSE
        end if
    end if
end procedure
```

We now assume that we have a procedure LAZYASSIGN which satisfies ◯ but not □.

(b) Show that DPLL is still valid if we replace ASSIGN with LAZYASSIGN.

Solution — It the exact same algorithm!

The only difference is that the algorithm may not return TRUE *as soon as a solution is found, only when there are no more variables to assign. The invariant ◯ ensures correctness: if there is no* FALSE *clause in P after the lazy unit propagation, then there is no conflict; and if, in addition, there are no free variables, then P is just* TRUE.

Note that the performance penalty is negligible: once P is satisfied (even if is not detected), we just have to assign all the remaining variables, we cannot do wrong choices anymore.

To design such a procedure LazyAssign, we keep for each clause C of the input formula two pointers to distinct literals of C, called *watched literals*, satisfying the following property:

$\triangle$ if one watched literal is assign to FALSE, then the other is assigned to TRUE.

We also maintain for each variable x the list of clauses in which x appears in a watched literal.

(c) Using this data structure, describe a procedure LazyAssign which maintains $\triangle$ and satisfies ◯.

Solution — When we assign a variable v, we look at each clause C in which v is watched. If the corresponding literal becomes TRUE, *there is nothing to do to ensure $\triangle$. Otherwise, we try to watch another literal to satisfy $\triangle$. If this is impossible, it means that every literal but one in C is* FALSE: *we have a unit clause that we can propagate. This gives the following algorithm.*

```
procedure LazyAssign($P$, $v$, $b$)
    for each clause $C$ of $P$ in which $v$ is watched do
        $L \leftarrow$ the watched literal of $C$ in which $v$ appears
        if $L|_{v\leftarrow b}$ = FALSE then
            if the other watched literal is unassigned then
                instead of $L$ watch another literal that is TRUE or not assigned
                if impossible then
                    propagate the unit clause
                end if
            end if
        end if
        return $P$
    end for
end procedure
```

(d) When backtracking, we need to unassign a variable. Here, we can do it for free. Why?

Solution — When backtracking, we don't need to undo the moves that we did. Indeed, no watched literal can have been implied in the conflict, and the property $\triangle$ is always maintained when a variable gets unassigned.

Exercice 2 (SAT modeling of 3-coloring)**.** Since SAT solvers are so optimized, in order to solve NP-hard problems it can be convenient in practice to encode them into a SAT formula (Notice that it is the reverse way of proving that a problem is NP-hard !).

Given a graph $G = (V, E)$, find a formula Φ in CNF such that Φ is satisfiable if and only if G has a proper 3-coloring.

Solution — For each vertex $v_i \in V$, we introduce 3 variables $x_{i,1}, x_{i,2}, x_{i,3}$ that encode in which color is v_i: $x_{i,j} = \mathbb{1}_{v_i \text{ is colored with color } k}$.

Now, we design 3 subformulas Φ_1, Φ_2, Φ_3 such that $\Phi = \Phi_1 \wedge \Phi_2 \wedge \Phi_3$.

- *Every vertex is given* at least *one color:*

$$\Phi_1 \doteq \bigwedge_{i=1}^{n} \bigvee_{j=1}^{k} x_{i,j}.$$

- *Every vertex is given* at most *one color:*

$$\Phi_2 \doteq \bigwedge_{i=1}^{n} \bigwedge_{1 \le c < d \le 3} \neg x_{i,c} \vee \neg x_{i,d}.$$

- *Two adjacent vertices* must *have different colors:*

$$\Phi_3 \doteq \bigwedge_{(i,j)\in E} \bigwedge_{c=1}^{k} \neg x_{i,c} \vee \neg x_{j,c}.$$

Exercice 3 (Optimization of SAT modeling)**.** Let $x_1, \dots, x_n$ be booleans and consider the following property:

$(*)$ At most one of the x_i is TRUE.

(a) Give a CNF SAT formula in $x_1, \dots, x_n$ with $O(n^2)$ clauses that is equivalent to $(*)$.

Solution — It was already done in the previous exercise, $\bigwedge_{i \neq j} (\neg x_i \vee \neg x_j)$

(b) Give a CNF SAT formula in $x_1, \dots, x_n$ and n auxiliary variables and size $O(n)$ whose satisfiability (with fixed $x_1, \dots, x_n$) is equivalent to $(*)$.

Solution — Let $a \Rightarrow b$ denote $\neg a \wedge b$ and consider the following formula, with auxiliary variables $s_1, \dots, s_n$:

$$\Phi = \bigwedge_{i=1}^{n} (x_i \Rightarrow \neg s_i) \ \wedge \ \bigwedge_{i=1}^{n-1} (s_{i+1} \Rightarrow s_i) \ \wedge \ \bigwedge_{i=1}^{n-1} (x_{i+1} \Rightarrow s_i).$$

If $(*)$ holds, then Φ is satisfiable with $s_i = \neg x_1 \wedge \cdots \wedge \neg x_i$. Conversely, if $(*)$ does not hold, then let $i < j$ be the two smallest indices such that x_i and x_j are TRUE*. If $s_1, \dots, s_n$ satisfies Φ, then $s_{j-1} =$* TRUE *(using the third group of clauses), it follows that $s_i =$* TRUE *(using the second group), but $s_i =$* FALSE *using the first group. Contradiction.*

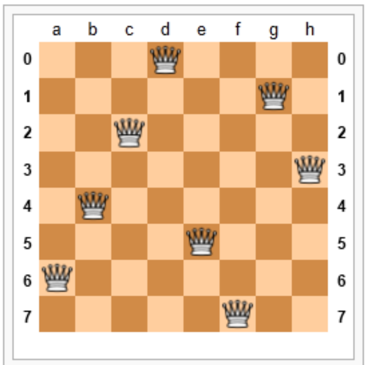
(c) Give a SAT formula in $x_1, \dots, x_n$ and $O(\log n)$ auxiliary variables and size $O(n \log n)$ whose satisfiability (with fixed $x_1, \dots, x_n$) is equivalent to $(*)$.

Solution — We use a binary encoding. Let $r = \lceil \log_2 n \rceil$ and consider r auxiliary variables $a_0, \dots, a_{r-1}$. Let $\epsilon_{ik} = a_k$ (resp. $\epsilon_{ik} = \neg a_k$) if the kth binary digits of i is 1 (resp. 0). Let

$$\Phi = \bigwedge_{i=1}^{n} \bigwedge_{k=1}^{r} \neg x_i \vee \epsilon_{ik}.$$

If $x_j = \text{TRUE}$, this clauses implies that $a_{r-1} \cdots a_0$ is the binary representation of j. So if Φ is satisfiable, then at most one x_j can be TRUE since j is determined by the a_k.

Exercice 4 (n queens puzzle)**.** Let's consider an $n \times n$ chessboard. The goal of the n queens problem is to put n queens on this chessboard such that no queen can attack another one according to the chess rules. It means that two queens can't share the same row, the same column (or file) or the same diagonal. Therefore, on each row and on each file, there is exactly one queen.



One solution to the eight queens puzzle

source: Wikipedia

We represent a chessboard by an $n \times n$ grid indexed by $\{0, \dots, n-1\}^2$ where $(0,0)$ is the bottom-left cell. Give a formula in CNF whose satisfiability yields a valid solution of the n queens problem.

Hint: In order to handle the diagonals, find a characterization on the cell (i,j) to belong to a specific diagonal.

Solution — We introduce n^2 fresh variables $s_{i,j}$ which will encode whether or not a queen is assigned to the cell (i,j). The different constraints on the rows and the columns of the puzzle can be expressed in the following way:

- *There is at least one queen on each row:*

$$\Phi_1 \doteq \bigwedge_{i=1}^{n} \bigvee_{j=1}^{n} s_{i,j}.$$

- *There is at least one queen on each column:*

$$\Phi_2 \doteq \bigwedge_{j=1}^{n} \bigvee_{i=1}^{n} s_{i,j}.$$

- *There is at most one queen on each row:*

$$\Phi_3 \doteq \bigwedge_{i=1}^{n} \bigwedge_{1 \leq j < k \leq n} \neg s_{i,j} \vee \neg s_{i,k}.$$

- *There is at most one queen on each column:*

$$\Phi_4 \doteq \bigwedge_{j=1}^{n} \bigwedge_{1 \leq k < l \leq n} \neg s_{k,j} \vee \neg s_{l,j}.$$

For the diagonals it is a little bit more complicated. Let D_- be the set of lines parallel to the $(0,0)-(n-1,n-1)$ big diagonal, and let D_+ be the set of lines parallel to the $(0,n-1)-(n-1,0)$ big diagonal.

Each line of D_- is characterized by an integer in $\{-(n-1),-(n-2),\ldots,n-2,n-1\}$ which represents the difference $i-j$ in the cell (i,j). Respectively, any line in D_+ is characterized by an integer in $\{0,1,\ldots,2n-2\}$ which represents the sum $i+j$ in the cell (i,j).

- *There is at most one queen on each diagonal in D_- where $d \geq 0$:*

$$\Phi_4 \doteq \bigwedge_{d=0}^{n-1} \bigwedge_{j=0}^{n-1-d} \bigwedge_{k=j+1}^{n-1} (\neg s_{d+j,j} \vee \neg s_{d+k,k})$$

- *There is at most one queen on each diagonal in D_- where $d < 0$:*

$$\Phi_5 \doteq \bigwedge_{-n+1 \leq d \leq -1} \left(\bigwedge_{0 \leq j < k \leq n-1+d} (\neg s_{d+j,j} \vee \neg s_{d+k,k}) \right)$$

- *There is at most one queen on each diagonal in D_+ where $0 \leq d \leq n-1$ (Bottom semi-square):*

$$\Phi_6 \doteq \bigwedge_{d=0}^{n-1} \bigwedge_{i=0}^{d} \bigwedge_{k=i+1}^{d} (\neg s_{i,d-i} \vee \neg s_{k,d-k})$$

- *There is at most one queen on each diagonal in D_+ where $n \leq d \leq 2n-2$ (Upper semi-square):*

$$\Phi_7 \doteq \bigwedge_{d=n}^{2n-2} \bigwedge_{i=1}^{n-1} \bigwedge_{k=i+1}^{n-1} (\neg s_{i,d-i} \vee \neg s_{k,d-k})$$

Finally, the formula in CNF that encodes the n queens puzzle is the conjonction of all of the subformulas:

$$\Phi \doteq \Phi_1 \wedge \Phi_2 \wedge \Phi_3 \wedge \Phi_4 \wedge \Phi_5 \wedge \Phi_6 \wedge \Phi_7.$$

Exercice 5 (Bounded model checking)**.** We model a system in the following way: the state of the system is a boolean vector $s \in \{\text{TRUE}, \text{FALSE}\}^d$ and there is a boolean formula $T(s,s')$ that holds if and only if the system can transition from state s to state s'. Some states are *invalid* and we want to check that the system avoid them. There is a boolean formula $P(s)$ which holds if and only if the state s is valid. Lastly, there is a boolean formula $I(s)$ which holds if and only if s is a possible initial state.

Design an algorithm, using a SAT solver, that decide whether or not the system can reach an invalid state.

Solution — The first naive try is to check that $I(s) \wedge \neg P(s)$ and $T(s,s') \wedge P(s) \wedge \neg P(s)$ have no solution. The first formula (if not satisfiable) implies that all initial states are valid, the second implies that a valid state may only transition to a valid state.

But the algorithm is incomplete: it is possible that the the second formula has a solution but concerning a state that is not reachable from possible initial states.

We consider three families of formulas:

$$\begin{aligned}
\Phi_n(s_0,\dots,s_n) &= I(s_0) \land T(s_0,s_1) \land \cdots \land T(s_{n-1},s_n) \land \neg P(s_n),\\
\Psi_n(s_0,\dots,s_n) &= T(s_0,s_1) \land \cdots \land T(s_{n-1},s_n) \land P(s_0) \land \cdots \land P(s_{n-1}) \land \neg P(s_n),\\
E_n(s_0,\dots,s_n) &= \bigwedge_{0 \leqslant i < j \leqslant n} s_i \neq s_j.
\end{aligned}$$

For increasing values of n we compute the satisfiability of Φ_n and $\Psi_n \land E_n$. If Φ_n is not satisfiable, then the system may not reach an invalid state in at most n transitions. If $\Psi_n \land E_n$ is not satisfiable, it means that any sequence of n distinct valid states can only lead to a valid state. In particular, if both are not satisfiable, then the system cannot reach an invalid state. So we stop the algorithm as soon as both are not satisfiable, or Ψ_n is satisfiable (the system is buggy).

Now, due to the addition of E_n, it is clear that $\Psi_n \land E_n$ is not satisfiable when n is bigger than the number of possible states. So the algorithm terminates.