École Polytechnique — CSE202 Design and Analysis of Algorithms

SAT Solving for Latin Squares (Sudokus)

In this tutorial we will see how to solve efficiently Latin square puzzles, a simplified form of Sudoku puzzles, by a reduction to SAT solving. In the first part we work with the class `Sat`, where we will implement the WalkSat algorithm. In the second part we will work with the class `LatinSquarePuzzle` where we will formulate the problem of completing an incomplete grid into a Latin square as a CNF formula to be solved by calling the WalkSat algorithm. We will *only manipulate CNF formulas that are satisfiable* (which is the case for those arising from Latin square puzzles). Finally, in the third part, we will implement (in the class `Sat`) a propagation technique that is very useful to simplify the CNF formulas arising from Latin square, and make the computation of the solution much faster.

Download the files `Sat.py`, `LatinSquarePuzzle.py`, and `test.py`.

1 The class `Sat` and the WalkSat algorithm

An instance of this class contains a CNF formula stored in `self.clauses` as a list of lists, e.g. the formula $(x_1 \vee x_3) \wedge (x_2 \vee \bar{x}_4 \vee \bar{x}_6) \wedge (\bar{x}_5 \vee \bar{x}_1 \vee x_4)$ is stored as `[[1,3],[2,-4,-6],[-5,-1,4]]`. The attribute `self.nr_var` gives the total number n of variables, i.e., the variables are $x_1, \ldots, x_n$. The attribute `self.values` is a Boolean array that gives access to the current values of the variables, i.e., `self.values[i]` gives access to the current value of x_i, for $i \in [1..n]$. (We will explain and use the last attribute, `fixed`, in the last part of the tutorial).

Question 1. Complete the method `is_clause_satisfied(self,c)` that returns the Boolean indicating if *one single* clause `c` is satisfied by the current assignment of the variables. Complete the method `satisfied(self)` that returns the Boolean giving the evaluation of the CNF formula under the current assignment of the variables. To test your code call the function `test_satisfied()` in `test.py`.

Question 2. Complete the method `initialize(self)` that assigns random Boolean values to the variables (use `random.choice([True, False])` to obtain a random Boolean value). Then, complete the method `walk_sat(self,N)` that runs the WalkSat algorithm, initializing the variables every N steps. When running the algorithm, the clauses should be ordered by increasing length (this can be done by putting `self.clauses.sort(key=len)` as the first line of the method). At every iteration you should run over the clauses and choose the first one (if it exists) not satisfied as the clause where you flip a random variable (this way, the chosen clause is of minimal size among all the non-satisfied clauses, which increases the chances to flip a variable in the good direction). If all clauses are satisfied, then the algorithm should stop (the formula is satisfied by the current variable assignment). To test your code, call the function `test_walk_sat()` in `test.py`.

2 Solving Latin squares

A Latin square of order k is a $k \times k$ matrix of entries in $[0..k-1]$ such that, for every row, every value $i \in [0..k-1]$ appears exactly once in the row, and, for every column, every value $i \in [0..k-1]$ appears exactly once in the column. For instance a Latin square of order 5 is shown below:

$$\begin{bmatrix} 1 & 4 & 3 & 2 & 0 \\ 2 & 3 & 0 & 4 & 1 \\ 3 & 1 & 4 & 0 & 2 \\ 4 & 0 & 2 & 1 & 3 \\ 0 & 2 & 1 & 3 & 4 \end{bmatrix}$$

We consider here the problem of completing a partially filled Latin square as shown in Figure 1.

We encode this as a CNF formula as follows: there are k^3 variables $x_{v,i,j}$ (where v, i, j are in $[0..k-1]$), and $x_{v,i,j} = \text{True}$ indicates that the matrix entry at position (i, j) contains the value v. We

$$\begin{bmatrix} * & 1 & * & 2 \\ * & * & 3 & 1 \\ 0 & * & * & * \\ 1 & * & 2 & * \end{bmatrix} \Longrightarrow \begin{bmatrix} 3 & 1 & 0 & 2 \\ 2 & 0 & 3 & 1 \\ 0 & 2 & 1 & 3 \\ 1 & 3 & 2 & 0 \end{bmatrix}$$

Figure 1: Completing a partially filled Latin square.

have the following constraints on Latin squares (some of these are redundant, but it is actually good to not be economic on clauses, as it favors the possibility of propagation of fixed variables as we will see later):

- for every $(i,j) \in [0..k-1]^2$, the matrix entry at position (i,j) contains at least one value, hence the clause $\bigvee_{v=0}^{k-1} x_{v,i,j}$, and it contains at most one value, hence, for every pair $0 \leq v_1 < v_2 \leq k-1$, there is a clause $\bar{x}_{v_1,i,j} \vee \bar{x}_{v_2,i,j}$ (meaning that it is not possible to have both values v_1 and v_2 at position (i,j)).
- for every $(v,i) \in [0..k-1]^2$, the ith row contains the value v at least once, hence the clause $\bigvee_{j=0}^{k-1} x_{v,i,j}$, and it contains the value v at most once, hence, for every pair, $0 \leq j_1 < j_2 \leq k-1$ there is a clause $\bar{x}_{v,i,j_1} \vee \bar{x}_{v,i,j_2}$ (meaning that, in the ith row, it is not possible to have the value v at both columns j_1 and j_2).
- for every $(v,j) \in [0..k-1]^2$, the jth column contains the value v at least once, hence the clause $\bigvee_{i=0}^{k-1} x_{v,i,j}$, and it contains the value v at most once, hence, for every pair, $0 \leq i_1 < i_2 \leq k-1$ there is a clause $\bar{x}_{v,i_1,j} \vee \bar{x}_{v,i_2,j}$ (meaning that, in the jth column, it is not possible to have the value v at both rows i_1 and i_2).

Question 3. The attribute `sat` in the class `LatinSquarePuzzle` is to contain the CNF formula for the problem. Complete the method `build_generic_clauses(self)` that adds to `sat` the above constraints (there are $3k^2$ clauses of size k and $3k^2\binom{k}{2}$ clauses of size 2). Note that the variables in the class `Sat` have to be named as $x_1, \ldots, x_n$, with $n = k^3$ the total number of variables. We have included two methods `triple_to_int(self,v,i,j)` and `int_to_triple(self,r)` to translate a triple (i,j,k) to an integer in $[1..n]$ as used in the `Sat` class, and vice-versa. To test your code call the function `test_build_clauses()` in `test.py`.

Question 4. The attribute `initial` contains a matrix (as a list of lists) that gives the partially filled Latin square, so that `self.initial[i][j]` either contains the char `'*'` if the entry at position (i,j) is undetermined, or contains a prescribed integer in $[0..k-1]$. Complete the method `add_fixed_value_clauses(self)` that adds clauses of size 1 to `self.sat` according to the prescribed values in `initial`. For instance, if `initial` is the partially filled Latin square on the left of Figure 1, the position $(0,1)$ is filled with the integer 1, and therefore, we add the clause $x_{1,0,1}$ (which is of size 1). To test your code call the function `test_1clauses()`.

Question 5. Complete the method `solve(self)` to solve the puzzle. This method should build the clauses and call the WalkSat solver (since most of the clauses are of size 2, we can use $N = 4n^2$ as the number of iterations between two reinitializations, with n the number of variables). Finally the method has to fill the attribute `self.final` from the obtained solution (for every $x_{v,i,j}$ that is true, do `self.final[i][j]=v`). To test your code call the function `test_solve()`.

3 Propagation of fixed variables in a CNF formula

On the classical 9×9 sudokus, the number of variables is $9^3 = 729$, which is too high to have WalkSat find a solution in reasonable time. However we can take advantage of another simple yet powerful idea, called *propagation of fixed variables*.

Let us take an example to illustrate it. Consider the formula

$$x_3 \wedge \bar{x}_4 \wedge \big(x_1 \vee x_3 \vee x_2\big) \wedge \big(\bar{x}_5 \vee \bar{x}_3 \vee x_4\big) \wedge \big(x_2 \vee \bar{x}_6 \vee x_4\big) \wedge \big(x_1 \vee x_4 \vee x_5\big).$$

For this formula to be satisfied, x_3 has to be fixed to `True` and x_4 to `False`. Once this is done, the clause $(x_1 \vee x_3 \vee x_2)$ becomes satisfied (hence can be removed), the clause $\big(\bar{x}_5 \vee \bar{x}_3 \vee x_4\big)$ becomes equivalent to $(\bar{x}_5)$, the clause $(x_2 \vee \bar{x}_6 \vee x_4)$ becomes equivalent to $(x_2 \vee \bar{x}_6)$, and the clause $(x_1 \vee x_4 \vee x_5)$ becomes equivalent to $(x_1 \vee x_5)$. Hence the formula becomes

$$(\bar{x}_5) \wedge (x_2 \vee \bar{x}_6) \wedge (x_1 \vee x_5).$$

But now there is a new clause of size 1, which fixes x_5 to `False`. Once this is fixed, the clause $(x_2 \vee \bar{x}_6)$ is unchanged, and the clause $(x_1 \vee x_5)$ becomes (x_1). Hence the formula becomes

$$(x_2 \vee \bar{x}_6) \wedge (x_1).$$

We again have a clause of size 1, which fixes x_1 as `True`. Finally we obtain the formula

$$(x_2 \vee \bar{x}_6).$$

Hence the solutions to the initial formula are those assignements such that x_1 is `True`, x_5 is `False`, x_3 is `True`, x_4 is `False`, and $x_2 \vee \bar{x}_6$ is `True`. Note that this technique, applied to a sudoku formula, corresponds to the intuitive easy first steps (e.g., if 8 entries in a line are known, this forces the value of the 9th entry, etc.).

We will again program in the class `Sat`. Fixed variables will be stored in the dictionary `self.fixed`. For instance if x_2 is fixed as `False`, then one can test that 2 is a key using an instruction such as `if 2 in self.fixed`, and `self.fixed[2]` contains `False` (and `self.values[2]` also contains `False`). We will now write methods to propagate the fixed values, while updating the CNF formula.

Question 6. Complete the method `fix_values_from_1clauses(self)` that updates the dictionary `self.fixed` from the clauses of size 1 in the current CNF formula (stored in `self.clauses`), for instance if the current CNF formula contains the clause $(\bar{x}_4)$, then one has to add 4 to `self.fixed` assigning it to `False` (`self.fixed[4]=False`) and assign `self.values[4]=False`. The method also has to return a Boolean that indicates if at least one clause of size 1 has been found.

Question 7. Complete the method `simplify_formula_by_propagation(self)` that propagates fixed variables from clauses of size 1, as in the example above. In order to update the attribute `self.clauses` at each round of the propagation, we give to you the method `simplify_clauses(self)` that returns the list of clauses corresponding to applying the simplification (using the current knowledge of fixed variables) to the list of clauses currently in `self.clauses`. To test your code call `test_propagate()`.

Question 8. Update the method `initialize(self)` so that only the non-fixed variables are reassigned. And, in the class `LatinSquarePuzzle`, update the method `solve` so that the propagation of fixed variables is applied to the formula before running WalkSat (if the number f of nonfixed variables after propagation is zero there is no need to run WalkSat since the formula is already solved; if not, call WalkSat with parameter $N = 4 * f^2$). It is useful to call `self.sat.display_statistics()` before and after propagation to observe the difference.

Once this is done, you can run the sudoku test method for a 9×9 grid that is at the end of the file `test.py`. This may take some number of seconds to finish. If you try to use WalkSat without propagation on this grid, you can see that it will take much more time (unless you are extremely lucky).