1 Introduction

We implement in OCaml a CDCL SAT solver and test it by encoding sudoku problems into SAT formulas.

2 Sudoku encoding

Following [1], we encode a sudoku problem as a SAT problem with $729 = 9^3$ variables $x^d_{i,j}$ (which we take to be true precisely when the (i, j) entry has value d), where we define the validity function as follows

$$\texttt{valid}((i_1, j_1), \ldots, (i_9, j_9)) = \bigwedge_{1 \leq k < l \leq 9} \bigwedge_{1 \leq d \leq 9} \neg(x^d_{i_k,j_k}) \vee \neg(x^d_{i_l,j_l}),$$

which we use to define a function `encode`, which precisely gives the sudoku constraints for the variables $x^d_{i,j}$. In short, a sudoku problem is the conjunction of clauses describing

1. each row, column, and square have distinct variables (each clause of length 2)
2. each entry can take at most one value (each clause of length 2)
3. each entry has at least one value (each clause of length 9)

In order to encode a given specific sudoku problem $p_{i,j} \in \{1, \ldots, 9\}$, we add 9 unit clauses for each nonzero entry.

3 Data structures

The main data structures are formulas and the dependency graph. A literal is a positive or negative variable, a clause is a list of literals, and a formula is a list of clauses. I define a module `Formulas` in order to encapsulate the operations on literals, clauses, and formulas (for example, disjunction of clauses avoiding repetitions, conjunction of formulas, resolution of a pair of clauses, etc.). The dependency graph has as vertices pairs of literals and the levels at which they were assigned.

4 CDCL implementation

We describe in detail the CDCL algorithm used and its implementation. The core of CDCL is given by an extension of the DPLL algorithm with a function `AnalyzeConflict`, which allows us to decide what decision level to backtrack to, and gives us a clause to learn. Each function below except for `AnalyzeConflict` and `FindUnit`, can be seen as procedurally modifying the decision level, model, dependency graph, and formula.

We define the following functions:

1. `AnalyzeConflict`(M, C, G, l): The purpose of this function is to compute a cut of the dependency graph G (a backjump clause). The conflict clause C gives an initial cut in the original formula. To compute a new conflict clause, we find a non-decision literal L in the clause of the current level l, compute its implication clause $C(L)$ (the clause from which its value was implied in the dependency graph), and resolve it with the conflict clause to get a new conflict clause $C(L) * C$. This represents a new cut which swaps the literal L. The idea is to iterate this procedure until a good cut has been found.

   We use the heuristic of the First Unique Implication Point to determine when to stop. This means we stop when precisely one literal of the current decision level l lies in the conflict

clause $C = C' \vee L'$ (this will imply that L' must be a decision literal). This procedure is precisely detailed in Chapter 4 of [2].

The level to which we backjump is determined by the maximum of the levels of the literals in the conflict clause which is not the decision literal. The resulting conflict clause is added to the problem. Since it has exactly one literal of the current level, the learnt clause is guarenteed to be eliminated with `BCP` at a lower decision level.

2. `FindUnit`(M, F, G, l): This first looks for an empty clause in the formula F, in which case a conflict has been found. Otherwise it looks for a clause $C \in F$ such that its instantiation in the current model $C[M]$ is a unit clause, and it adds the literal to both the model M and dependency graph G with incomming edges from all literals in C.

3. `Decide`(M, F, G, l): Picks a free variable in the formula F not already bound in the model M, increase the decision level to $l + 1$ and continue with `BCP`.

4. `Backtrack`(M, F, G, l): Remove all literals in both the model M and dependency graph G which lie at a decision level above l, then proceed with `BCP`.

5. `BCP`(M, F, G, l): This calls `FindUnit`. If a conflict was found, call `AnalyzeConflict`, and then `Backtrack` to that decision level (unless already at decision level 0, in which case F is unsat). Otherwise, if a unit clause is found, recur with `BCP`. If neither is found and free variables in F not bound by M are still available, `Decide` is called. Otherwise, check if $M \models F$, if so return SAT, otherwise F must be UNSAT.

6. `CDCL`(F): This initializes the model, dependency graph, and level with a call to `BCP`.

5 Tests and optimizations

The sudoku tests are slow, but not interminable. Even after some optimizations of the dependency graph structure, and `FindUnit` (which takes up most of the time), the functional style of the program leads to a lot of redundancy. Some more heuristics to the main structure of the CDCL algorithm should be implemented in order to make this significantly quicker. Since the running time for each computation ranged from 20 to 200 seconds, with several of the problems taking up to 1500 seconds, the output and running time of of each problem is saved in a data file `output.txt`.

References

[1] Tjark Weber. A SAT based sudoku solver. 2005.

[2] J. Marques-Silva, I. Lynce, S. Marik (Auth.), A. Biere, M. Heule, H. vaan Maren, T. Walsh (Eds.). Handbook of satisfiability. Chapter 4: Conflict-driven clause learning SAT solvers. 2008.