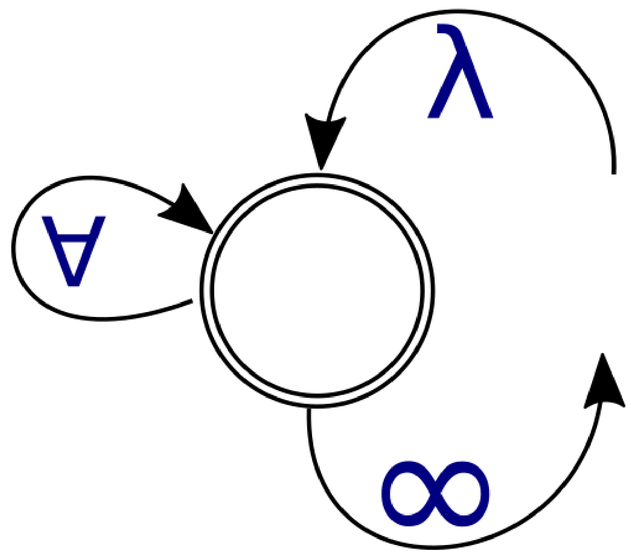
# Tractability in Many-Valued SAT Solving:
*Implementation and Theoretical Exploration*




**Nika Pona**

Theory and Logic Group, Faculty of Computer Science
Vienna University of Technology

nika@logic.at

**Abstract**

*Here I describe the progress, preliminary results and future work directions of a project of implementing a many-valued SAT solver based on a generalization of algorithms used in modern Boolean SAT solvers. Mimicking Boolean SAT solvers minimizes the algorithm-design and implementation challenges related to such a task, since many ideas can be easily adapted to the many-valued setting. Experimental results show that even on the early stages of the development a many-valued solver can perform better on some problems than modern Boolean SAT solvers.*

## Problem

The most common approach to test a many-valued formula for satisfiability is to translate it to a classical propositional formula and then use a Boolean SAT solver on the resulting translation. Previously some complete many-valued SAT solvers were implemented, and the results suggested that solving a many-valued formulation may be more efficient [1] [2]. However, now there are no complete many-valued SAT solvers available.

**Task:** Implement a many-valued SAT solver that would use advantages of many-valued modelling. See `github.com/akinanop/mvl-solver` for the current version of the solver.

## Why Many-Valued SAT?

**Boolean SAT easily generalizes to Many-Valued SAT.** The basic structure of CDPLL algorithms for Boolean SAT and many-valued SAT is the same. Decision and propagations are made until a falsified clause is found. Each decision literal increases the decision level. Every time a conflict is reached, a so-called no-good (clause representing an impossible assignment, derived from the "reason clauses" that lead to the conflict) is learned. The main difference is that the decisions and propagations are now made simultaneously on several literals.

```
Data: Problem in extended DIMACS format
Result: SAT / UNSAT
while checkSat() ≠ sat do
    if checkSat() = conflict then
        if level = 0 then
            return UNSAT
        end
        else
            level = analyzeConflict(); /* Learn a clause via extended resolution      */
            backtrack(level);
        end
    end
    else if checkUnit() then
        propagate(unitLiteral); /* Propagate positive and negative value
        assignments; check for entailed values                                        */
    end
    else
        chooseLiteral(); /* Eliminate one or several values from the domain          */
        propagate(decisionLiteral);
    end
end
return SAT;
```

**Algorithm 1:** Conflict-Driven DPLL

**Many-Valued SAT preserves structure of Constraint Satisfaction Problems.** One can translate CSP into a many-valued CNF formula by representing no-goods of every constraint as a clause. Such translation preserves the structure (domains) of the original problem, thus it may be more efficient to use a many-valued SAT Solver as a the back-end of a generic CSP solver. Below I provide an example of one such problem.

## Modelling Advantage of Many-Valued SAT

Solving some problems directly as many-valued problems can have a significant advantage. In the case of pigeonhole problem, despite the implementation advances in Boolean SAT, the many-valued solver still performs better. Moreover, encoding a problem into Boolean SAT via many-valued formulation already gives an advantage in the search.

### Pigeonhole problem

Pigeonhole problem (PHP) is a famous unsatisfiable problem, since despite it's easy formulation: "it is impossible to fit $n$ pigeons into $n-1$ holes, such that each hole contains exactly one pigeon", its unsatisfiability is known to be difficult to prove via automatic means. I consider the following encodings of the PHP:

**SAT** The Boolean SAT PHP is a CNF formula with variables $x_{ij}$ for each pair $i \in [n]$ and $j \in [n-1]$ and with two types of clauses for all $m \in [n-1]$:

1. $\bigvee_i x_{im}$ for $i \in [n]$;
2. $\neg x_{km} \vee \neg x_{lm}$ for $k \neq l \in [n]$

**MVL** The many-valued SAT PHP consists of $n$ variables of domain $n-1$. Domain declaration express the condition that each pigeon should be placed in some hole, and the clauses $k \neq j \vee l \neq j$ for $k \neq l \in [n]$ and $j \in [n-1]$ express the condition that no two pigeons should be placed in the same hole.

*Example of an input file in DIMACS format*

```
c This is a pigeonhole problem with 3 pigeons and 2 holes
p cnf 3 5
d 1 2  c domain of each variable is # of holes: 2
d 2 2
d 3 2
2!=0 1!=0 0  c pgn 2 and 1 cannot be in the hole 0 together
3!=0 1!=0 0
3!=0 2!=0 0
2!=1 1!=1 0
3!=1 1!=1 0
3!=1 2!=1 0
```

**MVL-SAT** Another Boolean SAT formulation of the PHP is created by translating a many-valued PHP into a Boolean formula using *linear encoding*. Replace each (negative) literal of a many-valued PHP with a (negated) boolean variable. As in SAT encoding add clauses of type 1. Furthermore, for each many-valued variable $v$, introduce $|dom(v)|-1$ new Boolean variables $v_i$ which will be used to enforce the property that at most one value has to be assigned to the variable. This will introduce only linear increase in the size of the original problem. For $i \in \{2, \ldots, |dom(v)|-1\}$ add:

1. $\neg v_{i-1} \vee v \neq i$;
2. $v \neq i \vee v_i$;
3. $\neg v_i \vee v_{i-1} \vee v = i$;
4. $\neg v_1 \vee v = 1$.

### Experiments

Below you can see that `mvl-solver` needs less time then `minisat` on both Boolean formulations of PHP. On $n = 15$ neither `minisat`, nor modern 2014-2015 winner solvers `glucose` and `COMiniSatPS` terminated within 24 hours, whereas `mvl-solver` with both heuristics was finished within 10-17 hours. Since the architecture of the `mvl-solver` is quite basic and not quite efficient yet (in particular, the propagation is very slow – on big satisfiable graph coloring instances where only extensive propagation is needed `mvl-solver` performs slowly compared to `minisat` that finishes instantly), one can make the conclusion the difference lies in the modelling advantage of the many-valued SAT.

**Table 1:** Times (s) on PHP with $n = 10 \ldots 15$

| | `minisat` | | `mvl-solver` | | `COMiniSatPS` |
|---|---|---|---|---|---|
| $n$ | MVL-SAT | SAT | BK | VSIDS | SAT |
| 15 | t/o | t/o | 17hrs | 10hrs | t/o |
| 13 | 19hrs | t/o | 3239 | 999 | 13hrs |
| 12 | 1061 | 1624 | 414 | 170 | 450 |
| 11 | 49 | 82 | 44 | 26 | 24 |
| 10 | 3 | 6 | 4 | 3 | 3 |

## Conclusion and Further Research

In this project I developed a many-valued solver with several basic conflict-driven algorithms generalized from Boolean SAT. Adapting the SAT algorithms to the many-valued setting can be worthwhile, given that the generalizations come naturally and don't require special theoretical effort. Provided the benefits of many-valued SAT solving mentioned in the literature and exemplified by the case study here, it seems like a fruitful direction of research. Further one should work on improving data structures, develop benchmarks and try adapting further techniques from SAT and CSP, such as random restarts and impact-based decision heuristics.

## References

[1] Siddhartha Jain, Eoin O'Mahony, and Meinolf Sellmann. A complete multi-valued SAT solver. In *Principles and Practice of Constraint Programming - CP 2010 - 16th International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings*, pages 281–296, 2010.

[2] Cong Liu, Andreas Kuehlmann, and Matthew W. Moskewicz. CAMA: A multi-valued satisfiability solver. In *2003 International Conference on Computer-Aided Design, ICCAD 2003, San Jose, CA, USA, November 9-13, 2003*, pages 326–333, 2003.