Logical reasoning and programming

SAT solving (cont'd)

Karel Chvalovský

CIIRC CTU

The Nobel Prize in Physics 2021

Giorgio Parisi (Prize share: $1/2$)

For the discovery of the interplay of disorder and fluctuations in physical systems from atomic to planetary scales.

The Nobel Prize in Physics 2021 and SAT

Giorgio Parisi (Prize share: $1/2$)

For the discovery of the interplay of disorder and fluctuations in physical systems from atomic to planetary scales.

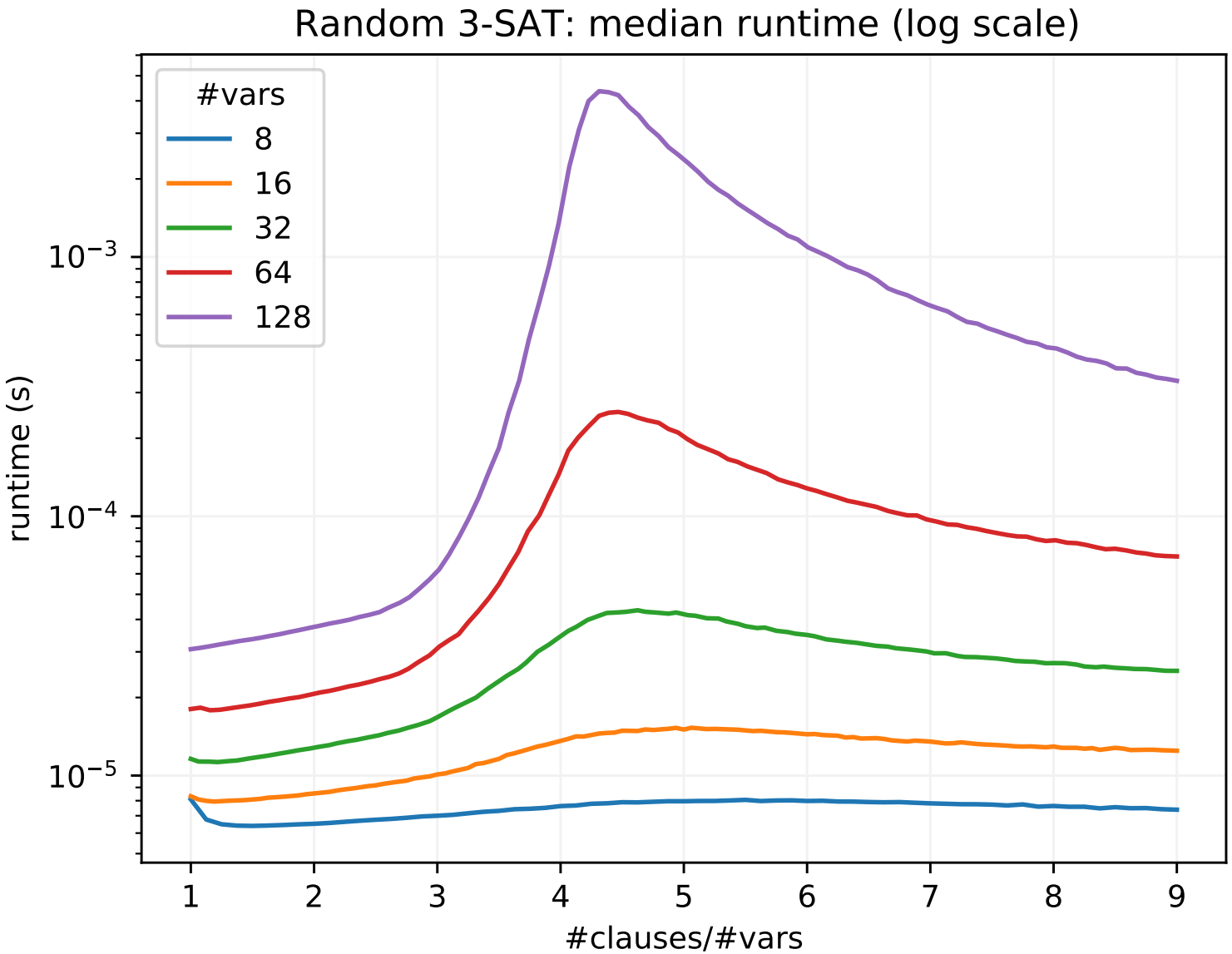
Analytic and Algorithmic Solution of Random Satisfiability Problems

M. Mézard,[1] G. Parisi,[1,2] R. Zecchina[1,3]*

We study the satisfiability of random Boolean expressions built from many clauses with *K* variables per clause (K-satisfiability). Expressions with a ratio α of clauses to variables less than a threshold α_c are almost always satisfiable, whereas those with a ratio above this threshold are almost always unsatisfiable. We show the existence of an intermediate phase below α_c, where the proliferation of metastable states is responsible for the onset of complexity in search algorithms. We introduce a class of optimization algorithms that can deal with these metastable states; one such algorithm has been tested successfully on the largest existing benchmark of K-satisfiability.

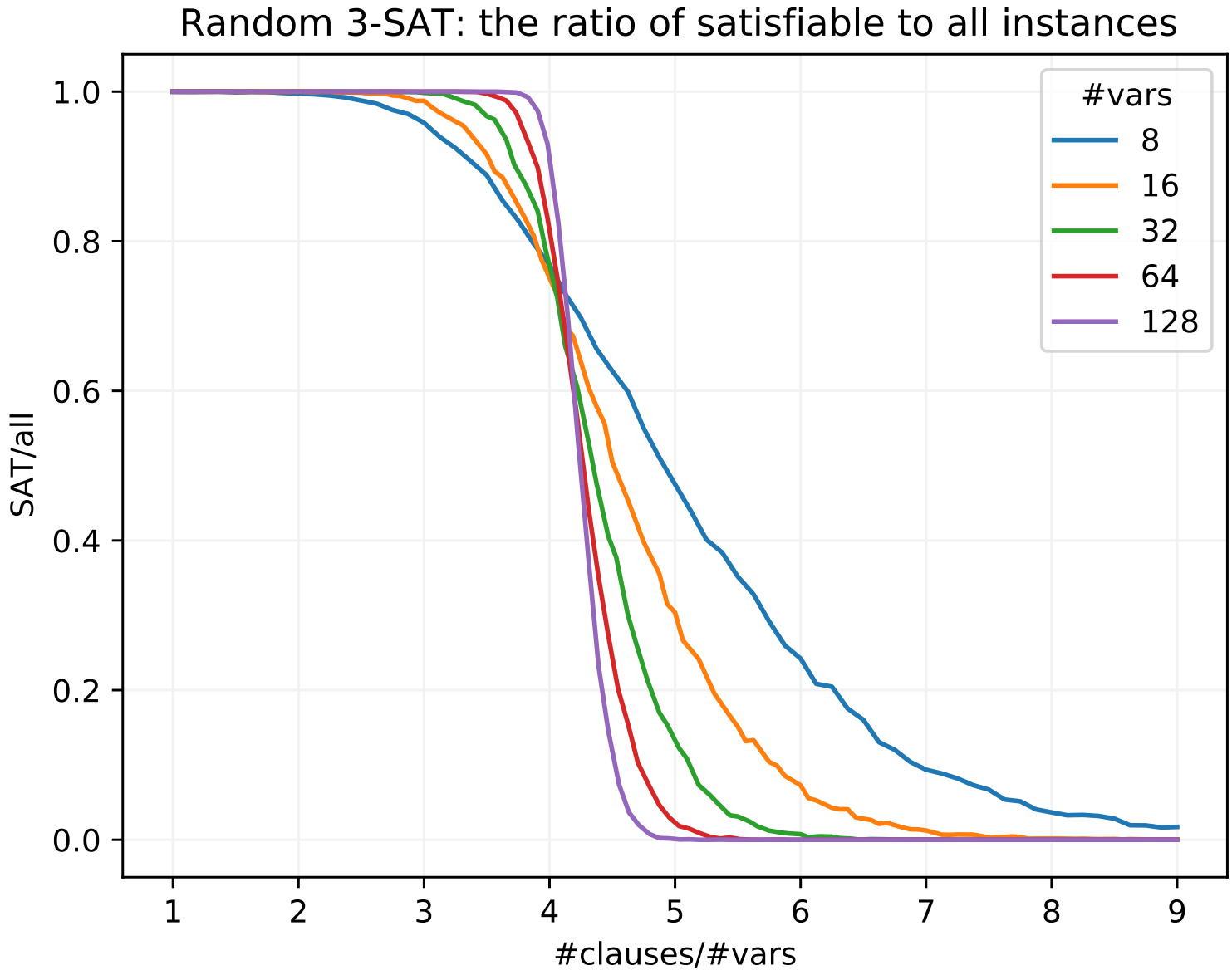
Phase transition for random k-SAT

(literals are selected randomly, each clause has length exactly k)



Phase transition for random k-SAT

(literals are selected randomly, each clause has length exactly k)



Probabilistic algorithms — stochastic local search

We start with a random complete valuation and try to minimize the number of unsatisfied clauses by flipping variables.

These methods are incomplete and it is an open problem how to use these techniques for showing unsatisfiability.

GSAT

Require: A set of clauses φ
  function $\mathrm{GSAT}(\varphi)$
    for $i \in (1, \mathrm{MAXITERS})$ **do**
      $v \leftarrow$ a random valuation on φ
      for $j \in (1, \mathrm{MAXFLIPS})$ **do**
        if $v \models \varphi$ **then return** v
        else minimize #unsat clauses by flipping a variable
    return None

Many extensions and variants, the most famous one is WalkSAT. You can try some of them in UBCSAT.

WalkSAT

We try to avoid local minima by combining the greedy moves of GSAT with random walk moves.

- Select randomly an unsatisfied clause c.
  - If by flipping a variable x occurring in c no satisfied clause becomes unsatisfied, then flip x. ("freebie" move)
  - Otherwise with a probability
    - p flip a random variable x in c ("random walk" move),
    - $(1-p)$ perform a GSAT step ("greedy" move) on variables from c; flip the best variable $x \in c$.

For details see Walksat Home Page. It is efficient on random k-SAT. Also historically good for planning and circuit design problems.

probSAT

A generalization of WalkSAT that calculates a probability distribution; p has weight $k^{-f(p)}$, where k is a constant and $f(p)$ is the number of clauses that become false after flipping p. It works also on some hard non-random problems. For details, see here.

CDCL or/and stochastic local search

On some instances stochastic local search methods work very well, you can try UBCSAT.

Moreover, it is even possible to combine CDCL with a local search and many modern solvers take advantage of that,

- for example, we can use a local search to produce a long partial assignment (trail) and then take advantage of this knowledge when we decide the values of variables (phase picking).

How to select a SAT solver?

Try different solvers (based on CDCL), they use the same input format and hence it is easy to experiment. However, the good encoding of your problem is usually at least as important as a good solver.

MiniSat is free, fast, and very popular implementation in C. It won all three industrial categories in the SAT Competition 2005. A new version is called MiniSat 2. However, it is not the state of the art. A good choice if you want to use a SAT solver in your software.

For playing in Python you can use pycosat, a package that provides bindings to PicoSAT on the C level. A rapidly developing toolkit is PySAT (includes CaDiCaL and Glucose).

Check results of SAT Competition 2021 and from previous years for the state of the art.

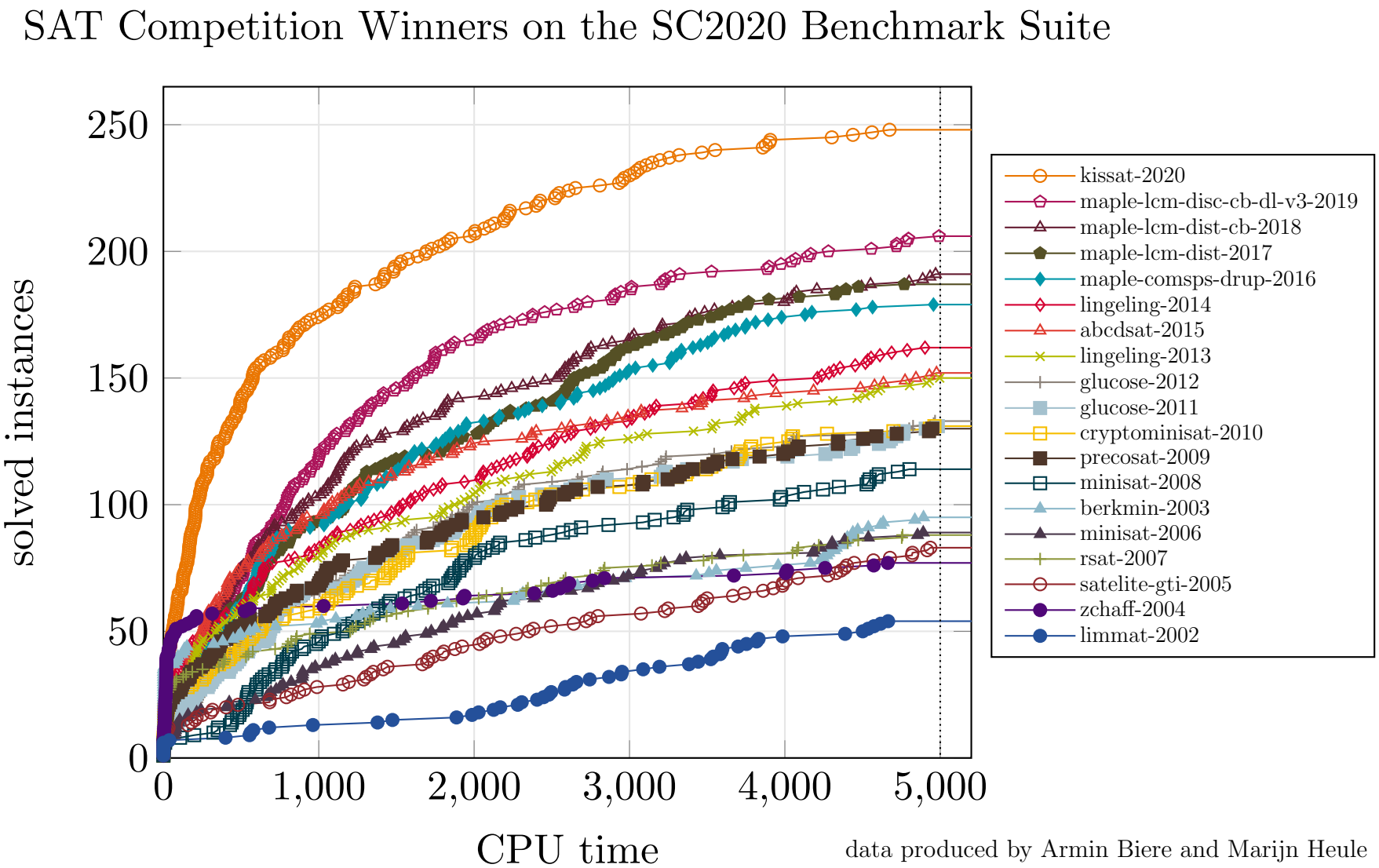
Some modern solvers

There are many modern solvers available, e.g.,

- solvers developed by Armin Biere, for example,
  - PicoSAT,
  - Lingeling,
  - CaDiCaL,
  - Kissat,
  - SATCH—a simple and clean code base for explaining and experimenting with SAT solvers,
- Glucose
  - based on MiniSat,
- MapleSAT
  - based on Glucose,
- CryptoMiniSat,
- Sat4j—a java library; good for Windows.

Progress over years. . .

source



data produced by Armin Biere and Marijn Heule

Planning

In classical planning we want to produce a sequence of actions that translate an initial state into a goal state.

It is well-known that the plan existence problem is PSPACE-complete. Hence it is not (assuming $\mathrm{NP} \neq \mathrm{PSPACE}$) easily solvable using SAT. However, if we consider only plans up to some length, then it is solvable by SAT, because the lengths of plans are usually polynomially bounded.

Planning as a SAT problem

We encode as a CNF formula "there exists a plan of length k", denoted φ_k, and search iterativelly.

- If $\varphi_k \in \mathtt{SAT}$, then we extract a plan from a satisfying assignment.
- If $\varphi_k \notin \mathtt{SAT}$, then we continue with φ_{k+1}.

Classical planning (recap)

We have a set of state variables $X = \{x_1, \ldots, x_n\}$ that are assigned values from a finite set. A state s is such an assignment for X, we write $\{x_1 = v_1, \ldots, x_n = v_n\}$. A set of conditions is a subset of a state.

We have

- an initial state,
- a set of goal conditions—a goal state is such a state that satisfies all the goal conditions.

Moreover, we have a set of actions A where every $a \in A$ has preconditions and effects which are both sets of conditions.

Example

We have a chessboard and $X = \{x_1, \ldots, x_{64}\}$ are the squares of the chessboard. An assignment says how pawns, pieces, and the empty square are distributed. A goal condition can be that the white king and queen are at x_{10} and x_{28}, respectively.

There exists a plan of length k in SAT

We introduce propositional variables for

- actions—meaning the action a is used in the step t,
- assignments—meaning $x = v$ holds before an action in the step t is applied,

for every $a \in A$, $x \in X$, possible value v of x, and step $t \leq k$.

Then we describe all the required properties of a valid plan by a conjunction of clauses:

- the initial state,
- the goal conditions are satisfied after k steps,
- state variables are assigned exactly one value,
- exactly one action is performed in one step,
- the values of state variables change only by actions,
- an applied action must satisfy preconditions and effects.

Planning using SAT

We can do various improvements, e.g.,

- perform more actions in a step if they are non-conflicting,
- introduce variables for transitions instead of assignments,
- symmetry breaking.

Incremental SAT solving

Instead of solving a new problem for every k, we can observe that many parts remain the same—we solve a sequence of similar SAT problems. We want to add and remove clauses, but keep learned clauses and variable scores.

Note that in our problem we only add clauses and change the goal conditions, which are described by unit clauses, when we go from φ_k to φ_{k+1}.

Assumptions

Clearly, adding clauses is possible in CDCL, but removing clauses can lead to various problems. However, we have

- a formula φ and
- assumptions $l_1, \ldots, l_n$, where l_i are literals.

The question is whether $\varphi \wedge l_1 \wedge \cdots \wedge l_n \in \texttt{SAT}$. It is incremental, because we can change assumptions and add new clauses.

We can select all the assumptions as decision variables and continue as always. Hence we can keep all learned clauses from CDCL!

Bounded model checking

It is very similar to planning. We want to verify a property of an automaton with transition states, an initial state, and a given property P that has to be valid at each step.

Bounded model checking as a SAT problem

We bound the number of steps to k and try to reach in k steps a state where P fails. Hence φ_k means "there is a state reachable in k steps where P fails".

- If $\varphi_k \in \texttt{SAT}$, then we extract a bug from a satisfying assignment.
- If $\varphi_k \notin \texttt{SAT}$, then we continue with φ_{k+1}.

How to encode typical constraints

We want to express

$$p_1 + p_2 + \cdots + p_n \bowtie k,$$

where $\bowtie \in \{\leq, \geq, =\}$, k is a positive integer, and $\sum_{1 \leq i \leq n} p_i$ is equal to the number of true p_is.

- $=$ is expressed as both $\leq$ and $\geq$,
- ≥ 1 is $\{p_1, \ldots, p_n\}$,
- ≤ 1 is
  - pairwise—$\mathcal{O}(n^2)$ clauses by $\{\{\overline{p_i}, \overline{p_j}\} : 1 \leq i < j \leq n\}$,
  - sequential counter—$\mathcal{O}(n)$ clauses and $\mathcal{O}(n)$ new variables,
  - bitwise encoding—$\mathcal{O}(n \log n)$ clauses and $\mathcal{O}(\log n)$ new variables,
- $\geq k$ is no more than $n - k$ literals can be false,
- $\leq k$ use generalized pairwise, sequential counters, BDDs, sorting networks, (pairwise) cardinality networks, ...

Or use a pseudo-Boolean (PB) solver for $\sum a_i p_i \bowtie k$.

Consistency and arc-consistency

A very nice property of encodings, e.g., for an encoding of constraints. We say that an encoding is

consistent if any partial assignment that cannot be extended to a satisfying assignment (is inconsistent) leads to a conflict by unit propagation,

arc-consistent if consistent and unit propagations eliminate values that are inconsistent.

Example

For ≤ 1 we have

consistency if two variables are true, then unit propagation produces a conflict,

arc-consistency if a variable is true, then unit propagation assigns false to all other variables. (+consistency)

Finite-domain encoding

We encode that a variable x takes one of the values $\{1, \ldots, n\}$.

One-hot encoding

- we use x_i for x takes value i (n variables),
- we need x has exactly one value,
- easy to use constraints and other rules

Unary encoding (order encoding)

- $\underbrace{1 \ldots 1}_{i-1} \underbrace{0 \ldots 0}_{n-i}$ for x takes value i ($n-1$ variables),
- we need $\{\overline{x_{j+1}}, x_j\}$ for $1 \leq j < n-1$

Binary encoding

- we encode i as a binary number ($\lceil \log n \rceil$ variables),
- if $n \neq 2^k$ some values are not valid,
- using constraints and other rules can be non-trivial

MaxSAT

There are various variants of SAT. For example, many problems in computer science are expressible as the maximum satisfiability problem—what is the maximum number of clauses that can be satisfied simultaneously.

We usually have (weighted) partial MaxSAT with two types of clauses:

- hard—must be satisfied,
- soft—desirable to be satisfies (possibly with weights)

and we want to maximize the sum of the weights of satisfied soft clauses.

You can check benchmark results at MaxSAT Evaluation 2021. For example RC2 (Python), MaxHS (also MIP solvers), Open-WBO (incomplete solver) and its extensions. There exists the standard WCNF format so we can experiment.

MaxSAT via SAT

We can replace all soft clauses

$$c_1, \ldots, c_n$$

by

$$c_1 \vee r_1, \ldots, c_n \vee r_n,$$

where $r_1, \ldots, r_n$ are fresh variables, called relaxation variables.

And we express that at most m clauses are not satisfied by adding

$$r_1 + \cdots + r_n \leq m.$$

By an iterative calling of a SAT solver, we can solve the MaxSAT problem (minimize m).

For more details on MaxSAT, check this tutorial and this chapter from Biere, M. Heule, et al. 2021.

Unsatisfiable cores

Let φ and ψ be unsatisfiable formulae in CNF such that $\varphi \subseteq \psi$. We say that

- φ is an *unsatisfiable core* of ψ,
- φ is a *minimal unsatisfiable core* of ψ, if every proper subset of φ is satisfiable.

A very important (and hard) practical problem is to extract minimal unsatisfiable cores.

Used, for example, in MaxSAT and formal verification.

What should you use to solve problems in NP?

You have seen many approaches how to solve such problems in Combinatorial Optimization, for example,

- Integer Linear Programming and
- Constraint Programming (MiniZinc Challenge 2020 Results).

Here we have discussed (Max)SAT and we will discuss other approaches later on.

What should you use?

A useful rule of thumb is to select a method based on the language suitable for your problem, see, for example, Which solver should I use? at Google OR-Tools.

See also, for example, LP/CP Programming Contest 2021.

SAT solving summary

SAT solvers are very powerful, among other things, thanks to

- small representations in CNFs,
- preprocessing, (inprocessing),
  - subsumption,
  - variable elimination, (variable addition),
  - symmetry breaking,
- unit propagations,
  - good data structures for backtracking,
- clause learning and back-jumping,
  - restarts,
  - deletion of learned clauses,
  - learned clause minimization,
- fast decision heuristics,
- local search,
- and much much more techniques we have not mentioned.

We do clever tricks, but first and foremost they have to be fast!

Bibliography I


Biere, Armin, Marijn Heule, et al., eds. (2021). *Handbook of Satisfiability*. 2nd. Vol. 336. Frontiers in Artificial Intelligence and Applications. Washington: IOS Press. ISBN: 978-1-64368-161-0.


Biere, Armin, Marijn J. H. Heule, et al., eds. (Feb. 2009). *Handbook of Satisfiability*. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, p. 980. ISBN: 978-1-58603-929-5.


Mézard, M., G. Parisi, and R. Zecchina (2002). "Analytic and Algorithmic Solution of Random Satisfiability Problems". In: *Science* 297.5582, pp. 812–815. DOI: 10.1126/science.1073287. eprint: https://www.science.org/doi/pdf/10.1126/science.1073287.