2 - SAT

SAT

@March 16, 2023

It’s short for SATisfiability. Also in CP we talk about satisfiability, but here, the given input is a propositional formula. For a well-known decision problem, given a propositional formula, is it SATisfiable?

A formula in propositional logic is composed of Boolean (decision) variables (T or F) and are constrained using logical operators only: negation, disjunction, conjunction, implication and bi-implication (we have a lot of disjunctions around).

A propositional formula is satisfiable if and only if it is possible to find a truth assignment to the variables that the whole formula yields true.

The Cook-Levin theorem (1971) first proves SAT to be NP-complete decision problem and the starting point of NP-completeness research. This gives us a lot of intuitions to use to solve problems in an efficient way. It is a theoretical standard for hard problems: all NP-Complex problems, including a wide range of decision and optimization problems, are reducible to SAT in polynomial time (i.e. at most as difficult to solve as SAT). SAT can be used to prove that a problem is NP-Complete (show that a problem is in NP and show that SAT can be reduced to the problem in polynomial time). It is a widely used modeling framework for solving combinatorial decision problems.

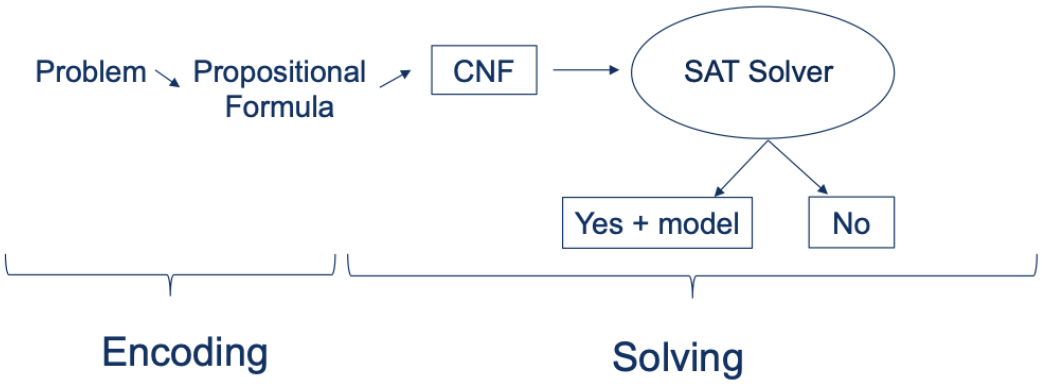
Applications:

- Artificial Intelligence: automated theorem proving, non-monotonic reasoning, planning, ML (verification and explanation of Models) …
- Model Checking in HW and SW verification, an automated method used to find design flaws in computer hardware and software systems. Specifications about the system are expressed as temporal logic (i.e. a logic for specifying properties over time) formulas. “Counter examples” are produced, telling developers where they have potential errors in their designs (2007 Turing Award - Clarke, Emerson and Sifakis)
- Equivalence check in HW Optimization → checking the functional equivalence of two combinatorial circuits.
- Equivalence check in SW Optimization → maintenance of the functionality of generated code in compilers

How do we solve the SAT Problem? **Brute Force Approach (**Truth Tables**)**: given a formula with n variables, compute the values of the formula for all 2^n ways to choose values T and F for the variables. The formula is SAT if and only if at least one of the 2^n cases yields true. It is simple, always works. But the drawback is that it is exponential in size of the number of variables, meaning that it can become impractical for high values of n. We need dedicated SAT solvers for checking the satisfiability of a formula and finding a satisfying assignment.

Issue: being an NP-complete problem, it is generally believed that there's no such algorithm that can efficiently solve SAT exists (resolving the question of whether SAT has a polynomial-time solving algorithm is equivalent to the P vs NP problem, which is a famous open problem in Theory of Computing). As a good news, though, current SAT solvers are successful for big formulas (they can solve formulas with millions of variables and clauses).

Problem Solving with SAT



The input I give to my solver must be expressed as a propositional formula, expressed in CNF. The transformation is done automatically by the solver.

We can use SAT to solve:

- Validation: Logic Puzzles → to structure as a propositional formula, we give a name for every notion to be formalized, and that name's gonna be a decision variable for us. We can use the variables to express the sentences in terms of those names using logical operators: express as a formula. We give to the SAT solver the conjunction of all the sentences, plus the negated result and prove it is UNSAT.
- Find Solutions: Graph Coloring → we now can use only boolean (language restriction) language. The difficulty lies in the elaboration of a suitable encoding. We will use two dimensional boolean variable x_{ic} for each possible color $c \in \{1, ..., k\}$ and region $v_i \in V$. Saying that $x_{ic} = T$ means that the region i is assigned to the color c. And

then you explicitly express all the constraints (each region gets at least one color, neighboring regions have different colors…)

- N-Queens, Sudoku…

Default: one solution. If you want to find more, you add the negation of the first solution to the formula and repeat the process until the formula is UNSAT.

- Arithmetics → How can we solve a formula like $7 + 21$ in SAT?. We represent each number in base 2 via Binary variables taking values $T = 1$ and $F = 0$. The binary representation $a_1 a_2 ... a_n$ of a number a:

$$a = \sum_{i=1}^{n} a_i * 2^{n-i} \quad a_i \in \{0, 1\}$$

  An example of decision problem would be: given a and b (represented in binary), find d (represented in binary) satisfying $a + b = d$.

Formula Validation and Satisfiability

A formula ϕ is valid if ϕ always evaluates to T for any assignment of values to the variables.

A formula ϕ is satisfiable if there is an assignment of values to the variables under which ϕ evaluates to T.

Validity is about finding a proof of a statement; satisfiability is about finding a solution to a set of constraints.

A formula ϕ is valid $\iff$ $\neg\phi$ is UNSAT.

Problem Solving in SAT

@March 17, 2023

Remember, the formula itself is not First Order Logic. We will concentrate on how the propositional formula is transformed to CNF.

- Literal → refers either to a Boolean variable p or to its negation $\neg p$
- Clause → Disjunction of literals. Can be falsified with only one assignment to its literals, where all literals are assigned to F. It is satisfied with $2^k - 1$ assignments to its k literals. As long as one literal is true, the whole clause is true. The empty clause, denoted by $\bot$, is always falsified (it's the equivalent of a variable with empty domain in CP).

- Propositional formula ϕ in CNF → conjunction of disjunction of literals. $\bigwedge_i \bigvee_j l_{ij}$. Conjunction of clauses. It is satisfiable if there exists an assignment satisfying all clauses, otherwise it is unsatisfiable. An arbitrary formula can be transformed into CNF preserving satisfiability.

Tseitin Transformation

Linear transformation of arbitrary formula ϕ to CNF, preserving satisfiability (1966).
The key idea is: do not find a CNF that is logically equivalent to ϕ (wouldn't be feasible and would blow up the size of the formula). Give a name to every sub-formula, except literals, and use these names as new variables. Define the formula on the new and original variables. It is automatically applied by many SAT solvers like Z3.

A formula ϕ on ≤ 3 variables → There is a small CNF that is logically equivalent to ϕ (there's no computational explosion).
Big formula ϕ (more than 3 variables) → Transform it to the conjunction of $cnf(\phi_i)$ of small formulas ϕ_i, obtained from ϕ for each sub-formula, using the extra variables s.t. the satisfiability is preserved.

For every sub-formula ψ, we define a name n_ψ: if ψ is a literal, $n_\psi = \psi$; otherwise we introduce a name n_ψ for ψ, which is a new variable.

$T(\phi)$ consists of:

- n_ϕ → the new name of the entire formula
- $cnf(n_\psi \leftrightarrow \psi)$ for every sub-formula ψ of the shape:
  - $\psi = \neg\psi'$ where ψ' is a non-literal sub-formula
  - $\psi = \psi_1 \diamond \psi_2$ where ψ_1 and ψ_2 are sub-formula with $\diamond \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$

We know that for every sub-formula ψ on at most 3 variables, there is an equivalent CNF $cnf(\psi)$ such that $cnf(\psi) \equiv \psi$ and $cnf(\psi)$ contains at most 4 clauses.

Resolution is the basic method of satisfiability of propositional formulas, and the basis of current SAT solvers (state-of-the-art still relies on Resolution). It is applicable to formulas in CNF. The idea is: from the given clauses, derive new clauses, with the aim of deriving the empty clause $\perp$ (the contradiction). So proves UNSAT.

Given the clauses of the shape $p \vee V$ and $\neg p \vee W$, we can derive $V \vee W$

$$\frac{p \vee V,\ \neg p \vee W}{V \vee W}$$

If a clause consists of a single literal l (a unit clause), then the resolution rule allows to remove the literal $\neg l$ from a clause containing $\neg l$.

$$\text{When } V \text{ or } W \text{ in } \frac{p \vee V, \neg p \vee W}{V \vee W} \text{ is empty, we have:}$$

$$\frac{p, \neg p \vee W}{W} \text{ or } \frac{p \vee V, \neg p}{V}$$

A CNF is UNSAT $\iff \bot$ can be derived by only using the resolution rule. The Resolution rule can be used for formula validation. Beyond UNSAT, a formula ϕ is a tautology (every I is a model) $\iff \neg\phi$ is UNSAT. $\phi \to \psi \iff (\phi \wedge \neg\psi)$ is UNSAT. $\phi \leftrightarrow \psi \iff \neg(\phi \leftrightarrow \psi)$ is UNSAT.

DPLL

Resolution is good if the formula is UNSAT. It is straightforward to give a refutation, but not direct to obtain a model (can't be used if we want to find a solution). We want an equivalent idea to the Propagation&Search in CP. DPLL is an algorithm to establish the SAT/UNSAT of a CNF. It is based on unit resolution and it is due to Davis, Putnam, Logemann and Loveland in 1962.

The basic idea is:

- Apply unit resolution as long as possible
- Then, choose a variable p
- Introduce the cases p and $\neg p$ and go on recursively. At some point you will find a solution, or fail and backtrack.

`unit-resol` → while there exists a clause consisting of one literal l (unit clause):

- remove $\neg l$ from all clauses containing $\neg l$
- remove all clauses containing l (since they are now redundant)

$\text{DPLL}(X)$:
$X := \text{unit-resol}(X)$
if $X = \emptyset$ then return(sat)
if $\bot \notin X$ then
    choose variable p in X
    $\text{DPLL}(X \cup \{p\})$
    $\text{DPLL}(X \cup \{\neg p\})$

Unit resolution and case analysis similar to constraint propagation and search in CP. As CP, it is a complete method and its efficiency strongly depends on the choice of the variable.

A direct implementation would make a copy of the CNF X at every recursive call, which is inefficient. We need to work on the original CNF X and mimic the DPLL algorithm which consists of a series of unit resolution, case analysis, backtrack and fail.

Efficient implementation of DPLL

The basic idea behind the efficient implementation is to keep track of a list M of literals that have been decided and derived during the execution of DPLL. M is originally empty and is extended when a literal is derived by unit resolution (`UnitPropagate`) or a case analysis start (`Decide`)

Notation:

- A literal l holds in M ($M \models l$) if and only if l occurs in M
- A clause C yields contradiction ($M \models \neg C$) after a number of unit resolution steps in the DPLL algorithm if and only if for every literal l in C, $M \models \neg l$.
- l is undefined in M if and only if neither $M \models l$ nor $M \models \neg l$.
- A decision literal l^d originates from a decision in the DPLL algorithm.

The algorithm can be mimicked by starting with an empty M and applying four rules (backtrack, unit propagate, decide, fail) as long as possible. At any moment the current CNF of the DPLL algorithm corresponds to M (what's happening) + the original CNF from which all negations of literals from M have been stripped away. At the end, we have either:

- fail, proving that the CNF is unsatisfiable or,

- a list M containing p or $\neg p$ for every variable p, yielding a satisfying assignment.

`UnitPropagate`

Mimics the generation of a new unit clause DPLL

$$M \implies Ml$$

if l is undefined in M and the CNF contains a clause $C \vee l$ satisfying $M \models \neg C$

`Decide`

Mimics the choice p in DPLL, when no UnitPropagate is possible

$$M \implies Ml^d$$

if l is undefined in M

`Backtrack`

Mimics backtracking to the negation of the last decision in case a branch is unsatisfiable

$$Ml^dN \implies M\neg l$$

if $Ml^dN \models \neg C$ for a clause C in the CNF and N does not contain decision literals

`Fail`

Mimics the end of DPLL when every branch, and hence the CNF, is unsatisfiable.

$$M \implies \text{fail}$$

if $M \models \neg C$ for a clause C in the CNF and M does not contain decision literals.

Conflict-Driven Clause Learning SAT Solvers

@March 28, 2023

Shortcomings of DPLL: first of all, you choose a random variable, we have chronological backtracking each time you fail and you backtrack just one level, even if maybe the current PA became doomed at another level. No learning: from p to $\neg p$ you throw away some of the work performed to conclude the current partial assignment is bad, and therefore risk re-visiting bad PAs, because you don't know which decision led to conflict.

In Conflict-Driven Clause Learning, each time a contradiction occurs:

- Clause learning mechanism augments the original CNF formula with a conflict clause that summarizes the reason of the contradiction that occurred, that can be used to avoid a new occurring of the same error(learn from mistakes)
- Back-jumping once we learn the cause of the contradiction. Skip the decisions that didn't lead to conflict and backtracks to an earlier decision, instead of the last

`Backjumping`

Mimics backtracking to an earlier decision literal than the last one

$$Ml^dN \implies Ml'$$

If $Ml^dN \models \neg C$ (condition for contradiction) for a clause C in the CNF and there is a clause $C' \vee l'$ derivable from the CNF such that $M \models \neg C'$ and l' is not defined in M; you backtrack back to M, discarding everything you did afterwards and then add l'.

Back-jumping is unit propagation on the derived clause which is added to the original CNF. The SAT solver stores every decision and at every `UnitPropagation` step, the corresponding clause and derived literal. At every contradiction, we use the information we stored to obtain which literals/clauses played a role. I can obtain a backjumping clause by resolution from the original CNF.

The Implication graph $G = (V, E)$ is a Directed Acyclic Graph that records the history of decisions and resulting deductions derived with `UnitPropagate`. Each node $v \in V$ represents a decision, $l@d$ (derived literal) or $\kappa@d$ (conflict). An edge $v \xrightarrow{c_i} w \in E$ indicates that w is derived with unit propagation from the clause c_i with one of its literals being v.

I can already derive the back-jumping clause from the decision assignments, negating them, but it might not necessarily be a good clause. In general, every cut that separates sources from the sink define a valid conflict clause.

Where do I stop?

We use the idea of Unit Implication Point (UIP): any node in the implication graph other than the conflict that is on all paths from the current decision literal to the conflict. We call 1UIP (First UIP) the UIP closest to the conflict. Using the UIP reduces the size of learnt clause and guarantees the highest backtrack jump.

Activity-based Search

We don't want to choose naively. Activity-based search concentrates on the variables causing conflict (can also be found in some CP solvers). Each time a variable is involved in clause learning (i.e. appears in the learnt clause or is eliminated during the resolution process), with

a special focus on recent conflicts. Selects the variable with the highest activity (one of possible heuristics). Most of the SAT solvers integrate this idea.

Unrestricted clause learning maybe impractical. The number of clauses grows with the number of conflicts. Large clauses are not useful for pruning the search space. Therefore you have a spike in computational complexity. The length/size of the clause is important (the smaller it is, the easier it is to do resolution on it). Possible clause learning restriction/deletion heuristics:

- n-order learning → learn only clauses up to n literals
- m-size relevance-based learning → learn a clause while it is either a unit clause or implies variable assignments, and delete a clause when the number of unassigned literals is greater than m
- k-bounded learning → learns a clause up to k literals, and delete it when the number of unassigned literals is greater than 1
- Clause deletion based on its activity in contributing to conflict and on the number of decisions taken since its creation

SAT solvers vary a lot between them in terms of strategies used: they include many heuristics ideas.

Search Restarts

The idea we've seen in CP actually comes from SAT: restart, if the search for a solution does not progress while many clauses have been learned. Can be Geometric, Luby… It restart with an empty M and the new CNF. Randomization in variable selection heuristics. Previously learned clauses and the activities influence the variable ordering heuristics.

Optimizing Unit Propagation

Double-edged sword as the wrong optimization could slow the search.Two rules:

1. If all but one of a clause's literals are assigned F and the remaining literal is unassigned, assign it T
2. If all of a clause's literals are assigned F, return `UNSAT`.

Naive algorithm: inspect each clause and apply the rule; repeat until no new assignments are made.

Watched Literals

Ideal algorithm: each clause is inspected only after all but one literal is assigned F. Nothing is accomplished by inspecting a clause when it is satisfied or when multiple literals are unassigned. The best known way to approximate this ideal would be:

1. watch two unassigned literals in every clause in preparation of unit propagation
2. examine the clause only when one of them is assigned F
3. for each literal, maintain a watch list containing the clauses it is currently watched by

The algorithm goes like:

When a literal a is assigned to T:

For each clause k in the watch list of $\bar{a}$, do:

- if all but one literal b is assigned to F, assign b to T
- if all literals are assigned to F, return `UNSAT`
- if any literal is assigned to T, continue
- otherwise, remove k from the watch list of $\bar{a}$ and add it to the watch list of one of its remaining unassigned literals

Benefits: low overhead, large reduction in number of clause inspections relative to naive algorithms.

SAT Encodings

@April 13, 2023

Encodings in SAT can be challenging, and need to be careful with size (in terms of number of clauses and size of clauses), as SAT solving efficiency may significantly degrade.

Cardinality Constraints

$$\sum_{1\le i\le n} x_i \bowtie k,\ k\in\{<,\le,=,\neq,>,\ge\}$$

A special case is $\sum_{1\le i\le n} x_i = 1$. We have:

- Exactly One if it's At Most One $\wedge$ At Least One
- At Most One if $\sum_{1\le i\le n} x_i \le 1$
- At Least One if $\sum_{1\le i\le n} x_i \ge 1$

It frequently occurs in SAT models (e.g. the N-Queens and Sudoku).

How do we encode this?

- At Least One → a sequence of ORs
- At Most One → many possibilities

AtMostOne: Pairwise Encoding

Any combination of 2 variables cannot be true at the same time.

$$\bigwedge_{i \leq i < j \leq n} \neg(x_i \wedge x_j)$$

We have no additional variables and we have $O(n^2)$ clauses: this might be heavy on the computational size. Idea to add new variables to reduce the number of clauses

AtMostOne: Sequential Encoding

We introduce n new variables s_i to indicate that the sum has reached 1 by the i-th variable (from that point on, everything else must be 0).

$$x_1 \rightarrow s_1 \wedge \bigwedge_{1<i<n} [(x_i \vee s_{i-1}) \rightarrow s_i) \wedge (s_{i-1} \rightarrow \bar{x_i})] \wedge (s_{n-1} \rightarrow \bar{x_n})$$

Which can be converted (by substituting the implications with ORs) to:

$$(\bar{x_1} \vee s_1) \wedge \bigwedge_{1<i<n} [(x_i \vee s_i) \wedge (\bar{s_{i-1}} \vee s_i) \wedge (\bar{s_{i-1}} \vee \bar{x_i})] \wedge (\bar{s_{n-1}} \vee \bar{x_n})$$

We now have $O(n)$ clauses and $O(n)$ new variables (remember, complexity doesn't care about the constants).

AtMostOne: Bitwise Encoding

The purpose is the same as Sequential Encoding. We introduce m new variables r_i where $m = \log_2 n$. For $1 \leq i \leq n$, let $b_{i,1}, ..., b_{i,m}$ be the binary encoding of $i - 1$

$$\bigwedge_{1 \leq i \leq n} x_i \rightarrow (r_1 = b_{i,1} \wedge r_2 = b_{i,2} \wedge ... \wedge r_m = b_{i,m})$$

Can also be written as:

$$\bigwedge_{1\leq i\leq n}\bigwedge_{1\leq j\leq n} \bar{x_i} \vee r_j [\vee \bar{r_j}]$$

if bit j of the binary encoding of $i-1$ is 1 [or 0]. We have a total of $O(n\log_2 n)$ (more than sequential) clauses and $O(\log_2 n)$(less than sequential) new variables.

AtMostOne: Heule Encoding

We still split the constraint using additional variables. When $n \leq 4$, apply pairwise encoding, using at most 6 clauses. When $n > 4$, introduce a new Boolean variable y and then:

$$\text{atMostOne}([x_1, x_2, x_3, y]) \wedge \text{atMostOne}([\bar{y}, x_4, ..., x_n])$$

and encode the second one recursively. $O(n)$ clauses and $O(n)$ new variables.

Cardinality Constraints of the type $\sum_{1\leq i\leq n} x_i = k$

They also frequently occur in SAT models, e.g. the Nurse Scheduling exercise.

$\sum_{1\leq i\leq n} x_i \bowtie k$ where $k \in \{<, \leq, =, \neq, >, \geq\}$

- $\sum_{1\leq i\leq n} x_i = k$ iff $(\sum_{1\leq i\leq n} x_i \leq k) \wedge (\sum_{1\leq i\leq n} x_i \geq k)$
- $\sum_{1\leq i\leq n} x_i \neq k$ iff $(\sum_{1\leq i\leq n} x_i > k) \vee (\sum_{1\leq i\leq n} x_i < k)$
- $\sum_{1\leq i\leq n} x_i \geq k$ iff $\sum_{1\leq i\leq n} \bar{x_i} \leq n - k$
- $\sum_{1\leq i\leq n} x_i > k$ iff $\sum_{1\leq i\leq n} \bar{x_i} \leq n - k - 1$
- $\sum_{1\leq i\leq n} x_i < k$ iff $\sum_{1\leq i\leq n} x_i \leq k - 1$

It is, for the first cases, enough to know one encoding to obtain all the others (by knowing `AtMostK` I can obtain all).

AtMostK: Generalized Pairwise Encoding

This is the naive way (generalization of the pairwise encoding). Any combination of $k+1$ variables cannot be true at the same time

$$\bigwedge_{M \subseteq \{1,\dots,n\}, |M| = k+1} \bigvee_{i \in M} \bar{x}_i$$

The result is I have $O(n^{k+1})$ clauses

AtMostK: Sequential Encoding

Introduce $n * k$ new variables s_{ij} to indicate that the sum has reached to j by i, for a total of $O(nk)$ clauses and $O(nk)$ new variables.

$$\begin{array}{l}
(\neg x_1 \vee s_{1,1}) \\
(\neg s_{1,j}) \qquad \text{for } 1 < j \leq k \\
(\neg x_i \vee s_{i,1}) \\
(\neg s_{i-1,1} \vee s_{i,1}) \\
\left.\begin{array}{l}(\neg x_i \vee \neg s_{i-1,j-1} \vee s_{i,j}) \\ (\neg s_{i-1,j} \vee s_{i,j})\end{array}\right\} \quad \text{for } 1 < j \leq k \quad \left.\begin{array}{l}\\ \\ \\ \\ \\ \end{array}\right\} \text{for } 1 < i < n \\
(\neg x_i \vee \neg s_{i-1,k}) \\
(\neg x_n \vee \neg s_{n-1,k})
\end{array}$$

Directly the version with just ORs, on the slides there’s the version with Implications

Arc-Consistency

Let us consider an encoding E of a constraint C such that there is a correspondence between the assignments of the variables in C with Boolean assignments of the variables in E. The encoding E is arc-consistent if whenever a partial assignment is inconsistent w.r.t. C (i.e. cannot be extended to a solution of C), unit propagation in E causes conflict (fail whenever the constraint is supposed to fail), otherwise unit propagation in E discards arc-inconsistent values (values that cannot be assigned).