Propositional Logic & SAT Solvers

Maria João Frade

HASLab - INESC TEC
Departamento de Informática, Universidade do Minho

2023/2024

Roadmap

- Introduction
- Review of Propositional Logic
- SAT solvers
- Modeling with Propositional Logic

Introduction

Formal modeling

- For us to be able to use an automated tool (to test or verify a property, or to analyze the behavior of a system), we first have to formally represent the system and its properties in the syntactic conventions that the tool understands and can process. This task is called **formal modeling**.
- The difficulty of the modeling task naturally varies; and, most typically, there are several ways of doing it.

What is a (formal) logic?

A formal logic consists of

- A *logical language* in which (well-formed) sentences are expressed. It consists of
  - *logical symbols* whose interpretations are fixed
  - *non-logical symbols* whose interpretations vary
- A *semantics* that defines the intended interpretation of the symbols and expressions of the logical language.
- A *proof system* that is a framework of rules for deriving valid judgments.

Logic and computer science

- Logic and computer science share a symbiotic relationship.
  - Logic provides language and methods for the study of theoretical computer science.
  - Computers provide a concrete setting for the implementation of logic.
  - Moreover, logic can be used to model the situations we encounter as computer science professionals, in such a way that we can reason about them formally.
- Logic is a fundamental part of computer science.
  - Program analysis: static analysis, software verification, test case generation, program understanding, ...
  - Artificial intelligence: constraint satisfaction, automated game playing, planning, ...
  - Hardware verification: correctness of circuits, ATPG, ...
  - Programming Languages: logic programming, type systems, programming language theory, ...

What is SAT?

- The Boolean satisfiability (SAT) problem:
  - *Find an assignment to the propositional variables of the formula such that the formula evaluates to TRUE, or prove that no such assignment exists.*
- SAT is an NP-complete decision problem.
  - SAT was the first problem to be shown NP-complete.
  - There are no known polynomial time algorithms for SAT.

What is SAT?

- Usually SAT solvers deal with formulas in conjunctive normal form (CNF)
  - literal: propositional variable or its negation. $A, \neg A, B, \neg B, C, \neg C$
  - clause: disjuntion of literals. $(A \vee \neg B \vee C)$
  - conjunctive normal form: conjuction of clauses. $(A \vee \neg B \vee C) \wedge (B \vee \neg A) \wedge \neg C$
- SAT is a success story of computer science
  - Modern SAT solvers can check formulas with hundreds of thousands variables and millions of clauses in a reasonable amount of time.
  - A huge number of practical applications.

Why should we care?

- No matter what your research area or interest is, SAT solving is likely to be relevant.
- Very good toolkit because many difficult problems can be reduced deciding satisfiabilty of formulas in logic.

(Classical) Propositional Logic

Propositional logic

- The language of propositional logic is based on propositions, or declarative sentences which one can, in principle, argue as being "true" or "false".
- Propositional symbols are the atomic formulas of the language. More complex sentences are constructed using logical connectives.
- In classical propositional logic (PL) each sentence is either true or false.

Syntax

The alphabet of the propositional language is organised into the following categories.

- *Propositional variables*: $P, Q, R, \ldots \in \mathcal{V}_{\mathsf{Prop}}$ (a countably infinite set)
- *Logical connectives*: $\bot$ (*false*) ,$\top$ (*true*), $\neg$ (*not*), $\wedge$ (*and*), $\vee$ (*or*), $\rightarrow$ (*implies*), $\leftrightarrow$ (*equivalent*)
- *Auxiliary symbols*: "(" and ")".

The set **Form** of *formulas* of propositional logic is given by the abstract syntax

$$\textbf{Form} \ni A, B \ ::= \ P \mid \bot \mid \top \mid (\neg A) \mid (A \wedge B) \mid (A \vee B) \mid (A \rightarrow B) \mid (A \leftrightarrow B)$$

We let $A, B, C, F, G, H, \ldots$ range over **Form**.

Outermost parenthesis are usually dropped. In absence of parentheses, we adopt the following convention about precedence. Ranging from the highest precedence to the lowest, we have respectively: $\neg$, $\wedge$, $\vee$, $\rightarrow$ and $\leftrightarrow$. All binary connectives are right-associative.

Semantics

The meaning of PL is given by the truth values true and false, where true $\neq$ false. We will represent true by 1 and false by 0.

An *assignment* is a function $\mathcal{A} : \mathcal{V}_{\mathsf{Prop}} \to \{0,1\}$, that assigns to every propositional variable a truth value.
An assignment $\mathcal{A}$ naturally extends to all formulas, $\mathcal{A}$: **Form** $\to \{0,1\}$.
The truth value of a formula is computed using *truth tables*:

F	A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \to B$	$A \leftrightarrow B$	$\bot$	$\top$
$\mathcal{A}_1(F)$	0	1	1	0	1	1	0	0	1
$\mathcal{A}_2(F)$	0	0	1	0	0	1	1	0	1
$\mathcal{A}_3(F)$	1	1	0	1	1	1	1	0	1
$\mathcal{A}_4(F)$	1	0	0	0	1	0	0	0	1

This way the meaning of each connective is established.

Semantics

Let $\mathcal{A}$ be an assignment and let F be a formula.
If $\mathcal{A}(F) = 1$, then we say F *holds* under assignment $\mathcal{A}$, or $\mathcal{A}$ *models* F.

We write $\mathcal{A} \models F$ iff $\mathcal{A}(F) = 1$, and $\mathcal{A} \not\models F$ iff $\mathcal{A}(F) = 0$.

An alternative (inductive) definition of $\mathcal{A} \models F$ is

$$
\begin{array}{lll}
\mathcal{A} \models \top & & \\
\mathcal{A} \not\models \bot & & \\
\mathcal{A} \models P & \text{iff} & \mathcal{A}(P) = 1 \\
\mathcal{A} \models \neg A & \text{iff} & \mathcal{A} \not\models A \\
\mathcal{A} \models A \wedge B & \text{iff} & \mathcal{A} \models A \text{ and } \mathcal{A} \models B \\
\mathcal{A} \models A \vee B & \text{iff} & \mathcal{A} \models A \text{ or } \mathcal{A} \models B \\
\mathcal{A} \models A \to B & \text{iff} & \mathcal{A} \not\models A \text{ or } \mathcal{A} \models B \\
\mathcal{A} \models A \leftrightarrow B & \text{iff} & \mathcal{A} \models A \text{ iff } \mathcal{A} \models B
\end{array}
$$

Validity, satisfiability, and contradiction

A formula F is

valid iff it holds under every assignment. We write $\models F$.
A valid formula is called a *tautology*.

satisfiable iff it holds under some assignment.

unsatisfiable iff it holds under no assignment.
An unsatisfiable formula is called a *contradiction*.

refutable iff it is not valid.

Proposition

F is **valid** iff $\neg F$ is **unsatisfiable**

Validity, satisfiability, and contradiction

Classify the following formulas:

- $A \to B$ is satisfiable and refutable.
- $P \vee \neg P$ is valid.
- $A \wedge \neg A$ is a contradiction.
- $B \vee (A \to \neg B)$ is valid.
- $B \wedge (A \vee \neg B)$ is satisfiable and refutable.
- $A \to \neg A \vee B$ is satisfiable and refutable.
- $(A \wedge (A \to B)) \to B$ is valid.

Consequence and equivalence

- $F \models G$ iff for every assignment $\mathcal{A}$, if $\mathcal{A} \models F$ then $\mathcal{A} \models G$. We say G is a *consequence* of F.
- $F \equiv G$ iff $F \models G$ and $G \models F$. We say F and G are *equivalent*.
- Let $\Gamma = \{F_1, F_2, F_3, \dots\}$ be a set of formulas.
  $\mathcal{A} \models \Gamma$ iff $\mathcal{A} \models F_i$ for each formula F_i in Γ. We say $\mathcal{A}$ *models* Γ.
  $\Gamma \models G$ iff $\mathcal{A} \models \Gamma$ implies $\mathcal{A} \models G$ for every assignment $\mathcal{A}$. We say G is a *consequence* of Γ.

Proposition

- $F \models G$ iff $\models F \to G$
- $\Gamma \models G$ and Γ finite iff $\models \bigwedge \Gamma \to G$

Some basic equivalences

$$
\begin{array}{lcl}
A \vee A & \equiv & A \\
A \wedge A & \equiv & A \\
\\
A \vee B & \equiv & B \vee A \\
A \wedge B & \equiv & B \wedge A \\
\\
A \wedge (A \vee B) & \equiv & A \\
\\
A \wedge (B \vee C) & \equiv & (A \wedge B) \vee (A \wedge C) \\
A \vee (B \wedge C) & \equiv & (A \vee B) \wedge (A \vee C) \\
\\
\neg(A \vee B) & \equiv & \neg A \wedge \neg B \\
\neg(A \wedge B) & \equiv & \neg A \vee \neg B \\
\\
A \leftrightarrow B & \equiv & (A \to B) \wedge (B \to A)
\end{array}
$$

$$
\begin{array}{lcl}
A \wedge \neg A & \equiv & \bot \\
A \vee \neg A & \equiv & \top \\
\\
A \wedge \top & \equiv & A \\
A \vee \top & \equiv & \top \\
\\
A \wedge \bot & \equiv & \bot \\
A \vee \bot & \equiv & A \\
\\
\neg\neg A & \equiv & A \\
\\
A \to B & \equiv & \neg A \vee B
\end{array}
$$

Consistency

Let $\Gamma = \{F_1, F_2, F_3, \dots\}$ be a set of formulas.

- Γ is *consistent* or *satisfiable* iff there is an assignment that models Γ.
- We say that Γ is *inconsistent* or *unsatisfiable* iff it is not consistent and denote this by $\Gamma \models \bot$.

Proposition

- $\{F, \neg F\} \models \bot$
- If $\Gamma \models \bot$ and $\Gamma \subseteq \Gamma'$, then $\Gamma' \models \bot$.
- $\Gamma \models F$ iff $\Gamma, \neg F \models \bot$

Substitution

- Formula G is a *subformula* of formula F if it occurs syntactically within F.
- Formula G is a *strict subformula* of F if G is a subformula of F and $G \neq F$

Substitution theorem

Suppose $F \equiv G$. Let H be a formula that contains F as a subformula. Let H' be the formula obtained by replacing some occurrence of F in H with G. Then $H \equiv H'$.

Decidability

Given formulas F and G as input, we may ask:

Decision problems

Validity problem:	"Is F valid ?"
Satisfiability problem:	"Is F satisfiable ?"
Consequence problem:	"Is G a consequence of F ?"
Equivalence problem:	"Are F and G equivalent ?"

All these problems are **decidable!**

Decidability

Any algorithm that works for one of these problems also works for all of these problems!

F is satisfiable	iff	$\neg F$ is not valid
$F \models G$	iff	$\neg(F \to G)$ is not satisfiable
$F \equiv G$	iff	$F \models G$ and $G \models F$
F is valid	iff	$F \equiv \top$

Truth-table method

For the satisfiability problem, we first compute a truth table for F and then check to see if its truth value is ever one.

This algorithm certainly works, but is very inefficient.
It's exponential-time! $\mathcal{O}(2^n)$

If F has n atomic formulas, then the truth table for F has 2^n rows.

Complexity

- Computing a truth table for a formula is exponential-time in order to the number of propositional variables.
- There are several techniques and algorithms for SAT solving that perform better in average.
- There are no known polynomial time algorithms for SAT.
  - If it exists, then **P** = **NP**, because the SAT problem for PL is NP-complete (it was the first one to be shown NP-complete).

Cook's theorem (1971)

SAT is NP-complete.

- **Conjecture:** Any algorithm that solves SAT is exponential in the number of variables, in the worst-case.

An example

The unicorn puzzle

- If the unicorn is mythical, then it is immortal.
- If the unicorn is not mythical, then it is a mortal mammal.
- If the unicorn is either immortal or a mammal, then it is horned.
- The unicorn is magical if it is horned.
- **Questions:**
  - Is the unicorn magical?
  - Is it horned?
  - Is it mythical?
  - Is it possible for the unicorn to be simultaneously mythical and immortal?

An example

- Consider the following propositional variables:
  - M - the unicorn is mythical
  - I - the unicorn is immortal
  - A - the unicorn is mammal
  - H - the unicorn is horned
  - G - the unicorn is magical
- If the unicorn is mythical, then it is immortal.
  $M \to I$
- If the unicorn is not mythical, then it is a mortal mammal.
  $\neg M \to (\neg I \wedge A)$
- If the unicorn is either immortal or a mammal, then it is horned.
  $(I \vee A) \to H$
- The unicorn is magical if it is horned.
  $H \to G$

An example

Let Γ be $\{ M \to I, \neg M \to (\neg I \wedge A), (I \vee A) \to H, H \to G \}$

If the puzzle is consistent, Γ should be satisfiable.

Questions:

- Is the unicorn magical? $\Gamma \models G$?
- Is it horned? $\Gamma \models H$?
- Is it mythical? $\Gamma \models M$?

Recall that: $\Gamma \models F$ iff $\Gamma, \neg F$ UNSAT

- Is it possible for a unicorn to be simultaneously mythical and immortal? Is $\Gamma, M \wedge \neg I$ SAT?

SAT solvers

SAT solving algorithms

- There are several techniques and algorithms for SAT solving.
- Usually SAT solvers receive as input a formula in a specific syntactical format.
- SAT solvers deal with formulas in **conjunctive normal form (CNF)**
  - literal: propositional variable or its negation. $A, \neg A, B, \neg B, C, \neg C$
  - clause: disjuntion of literals. $(A \vee \neg B \vee C)$
  - conjunctive normal form: conjuction of clauses.
    $(A \vee \neg B \vee C) \wedge (B \vee \neg A) \wedge \neg C$
- So, one has first to transform the input formula to this specific format preserving satisfiability.
- Most current state-of-the-art SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) framework.

DPLL framework

- The ideia is to incrementally construct an assignment compatible with a CNF, propagating the implications of the decisions made that are easy to detect and simplifying the clauses.
- A CNF is satisfied by an assignment if all its clauses are satisfied. And a clause is satisfied if at least one of its literals is satisfied.
- When a literal is true, the formula can be simplified by removing the clauses where the literal occurs and removing the opposite literal from the remaning clauses.
  - $(\neg A \vee \neg B) \wedge A \wedge (\neg B \vee A) \wedge (\neg A \vee C \vee B)$ choose $\mathcal{A}(A) = 1$
    $(\neg A \vee \neg B) \wedge A \wedge (\neg B \vee A) \wedge (\neg A \vee C \vee B)$ simplification
    $\neg B \wedge (C \vee B)$
- Unit propagation is the iterated application of the unit clause rule (assign true to a literal that is isolated in a clause) and subsequent simplification of the formula.

DPLL algorithm

- The algorithm starts with an empty assignment and performs unit propagation, updating the assignment accordingly and simplifying the formula.
  - If the formula becomes empty, that means that the formula is SAT.
  - If the formula contains an empty clause, that indicates a contradiction (a conflict).
  - If no conclusion is attained, a decision about an unassigned variable is made, propagating the implications of this decision.
- In case a conflict is detected, the algorithm backtracks to the previous decision point and tries a different assignment for the last decision variable.
- DPLL is a complete algorithm for SAT. Unsatisfiability of the complete formula can only be detected after exhaustive search.

DPLL framework: heuristics & optimizations

Many different techniques are applied to achieve efficiency in DPLL-based SAT solvers.

- Decision heuristic: a very important feature in SAT solving is the strategy by which the literals are chosen.
- Look-ahead: exploit information about the remaining search space.
  - unit propagation
  - pure literal rule
- Look-back: exploit information about search which has already taken place.
  - clause learning (new clauses are learnt from conflicts that prune the search space)
  - non-chronological backtracking (a.k.a. backjumping)
- Other techniques:
  - preprocessing (detection of subsumed clauses, simplification, ...)
  - (random) restart (restarting the solver when it seams to be is a hopeless branch of the search tree)

Conflict-Driven Clause Learning (CDCL) solvers

- **Conflict-Driven Clause Learning (CDCL) solvers**
  - DPLL framework.
  - New clauses are learnt from conflicts.
  - Structure of conflicts exploited (using implication graphs).
  - Backtracking can be non-chronological.
  - Efficient data structures (compact and reduced maintenance overhead).
  - Backtrack search is periodically restarted.
  - Can deal with hundreds of thousand variables and tens of million clauses.
- The most successful modern SAT solvers use this technology.
- The satisfiability library **SAT Live!**[1] is an online resource that proposes, as a standard, a unified notation and a collection of benchmarks for performance evaluation and comparison of tools.

[1] http://www.satlive.org

DIMACS CNF format

- DIMACS CNF format is a standard format for CNF used by most SAT solvers.
- Plain text file with following structure:

```
c <comments>
...
p cnf <num.of variables> <num.of clauses>
<clause> 0
<clause> 0
...
```

  - Every number 1, 2, . . . corresponds to a variable (variable names have to be mapped to numbers).
  - A negative number denote the negation of the corresponding variable.
  - Every clause is a list of numbers, separated by spaces. (One or more lines per clause).

DIMACS CNF format

Example

$$A_1 \wedge (A_1 \vee P) \wedge (\neg A_1 \vee \neg P \vee A_2) \wedge (A_1 \vee \neg A_2)$$

- We have 3 variables and 4 clauses.
- Let $A_1 = 1$, $A_2 = 2$ and $P = 3$.
- CNF file:

```
p cnf 3 4
1 0
1 3 0
-1 -3 2 0
1 -2 0
```


Minisat demo

```
$ cat example.cnf
c A1 = 1
c A2 = 2
c P = 3

p cnf 3 4
1 0
1 3 0
-1 -3 2 0
1 -2 0

$ minisat example.cnf OUT
============================[ Problem Statistics ]=============================
|                                                                             |
|  Number of variables:             3                                         |
|  Number of clauses:               1                                         |
|  Parse time:                   0.00 s                                       |
|  Eliminated clauses:           0.00 Mb                                      |
|  Simplification time:          0.00 s                                       |
|                                                                             |
============================[ Search Statistics ]==============================
| Conflicts |          ORIGINAL         |          LEARNT          | Progress |
|           |    Vars  Clauses Literals |    Limit  Clauses Lit/Cl |          |
===============================================================================
===============================================================================
restarts              : 1
conflicts             : 0              (0 /sec)
decisions             : 1              (0.00 % random) (490 /sec)
propagations          : 1              (490 /sec)
conflict literals     : 0              ( nan % deleted)
Memory used           : 0.16 MB
CPU time              : 0.002041 s

SATISFIABLE

$ cat OUT
SAT
1 -2 -3 0
```


SAT solver API

- Several SAT solvers have API's for different programming languages that allow an incremental use of the solver.
- For instance, PySAT[2] is a Python toolkit which provides a simple and unified interface to a number of state-of-art SAT solvers, enabling to prototype with SAT oracles in Python while exploiting incrementally the power of the original low-level implementations of modern SAT solvers.

```python
from pysat.solvers import Minisat22

s = Minisat22()

s.add_clause([-1, 2])
s.add_clause([-1, -2, 3])

if s.solve():
    print("SAT")
    print(s.get_model())
else:
    print("UNSAT")
```

[2] https://pysathq.github.io

Looking for $A_1 \wedge (A_1 \vee P) \wedge (\neg A_1 \vee \neg P \vee A_2) \wedge (A_1 \vee \neg A_2)$ models

```
from pysat.solvers import Minisat22
s = Minisat22()          # cria o solver s

s.add_clause([1])        # acrescenta cláusulas
s.add_clause([1, 3])
s.add_clause([-1, -3, 2])
s.add_clause([1, -2])

if s.solve():            # testa a satisfatibilidade
    print("SAT")
    print(s.get_model()) # imprime o modelo
else:
    print("UNSAT")
```

```
SAT
[1, -2, -3]
```

- The model found by the solver indicates that $A_1 \wedge \neg A_2 \wedge \neg P$ is true.
- Is there other possible models? To check that we must add the clause that corresponds to the negation of the above formula and check again.

$$\neg(A_1 \wedge \neg A_2 \wedge \neg P) \equiv \neg A_1 \vee A_2 \vee P$$

Looking for $A_1 \wedge (A_1 \vee P) \wedge (\neg A_1 \vee \neg P \vee A_2) \wedge (A_1 \vee \neg A_2)$ models

```
s = Minisat22()

s.add_clause([1])
s.add_clause([1, 3])
s.add_clause([-1, -3, 2])
s.add_clause([1, -2])

if s.solve():
    print("SAT")
    print("Modelos:")
    m = s.get_model()
    print(m)
    s.add_clause([-l for l in m])
    while s.solve():
        m = s.get_model()
        print(m)
        s.add_clause([-l for l in m])
else:
    print("UNSAT")
```

```
SAT
Modelos:
[1, -2, -3]
[1, 2, -3]
[1, 2, 3]
```


Modeling with PL

Example: Placement of guests

Placement of guests

We have three chairs in a row and we need to place Anne, Susan and Peter.

- Anne does not want to sit next to Peter.
- Anne does not want to sit in the left chair.
- Susan does not want to sit to the left of Peter.

Can we satisfy these constrains? How can we sit the guests?

- Denote: Anne $= 1$, Susan $= 2$, Peter $= 3$
  left chair $= 1$, middle chair $= 2$, right chair $= 3$
- Introduce a propositional variable x_{ij} for each pair *(person i, place j)*
- x_{ij} is true iff person i is sited in place j; x_{ij} is false otherwise

Example: Placement of guests

- Anne does not want to sit next to Peter.

$$((x_{11} \vee x_{13}) \to \neg x_{32}) \wedge (x_{12} \to (\neg x_{31} \wedge \neg x_{33}))$$

- Anne does not want to sit in the left chair.

$$\neg x_{11}$$

- Susan does not want to sit to the left of Peter.

$$(x_{33} \to \neg x_{22}) \wedge (x_{32} \to \neg x_{21})$$

- Are these constraints enough to model the problem?
- Can you point out an unexpected trivial solution?

Example: Placement of guests

There are two constrains that are implict in the problem.

- Everyone must be seated in a chair.

$$(x_{11} \vee x_{12} \vee x_{13}) \wedge (x_{21} \vee x_{22} \vee x_{23}) \wedge (x_{31} \vee x_{32} \vee x_{33})$$

  Using a compact notation:

$$\bigwedge_{i=1}^{3} \bigvee_{j=1}^{3} x_{ij}$$

- No more than one person per chair.
  For each chair $c = 1, 2, 3$,

$$(x_{1c} \to \neg x_{2c} \wedge \neg x_{3c}) \wedge (x_{2c} \to \neg x_{1c} \wedge \neg x_{3c}) \wedge (x_{3c} \to \neg x_{1c} \wedge \neg x_{2c})$$

  This is correct, but there is some redundancy here...

Example: Placement of guests

- No more than one person per chair.

  Note that

$$\begin{array}{l} x_{1c} \to \neg x_{2c} \wedge \neg x_{3c} \equiv \neg x_{1c} \vee (\neg x_{2c} \wedge \neg x_{3c}) \equiv (\neg x_{1c} \vee \neg x_{2c}) \wedge (\neg x_{1c} \vee \neg x_{3c}) \\ x_{2c} \to \neg x_{1c} \wedge \neg x_{3c} \equiv \neg x_{2c} \vee (\neg x_{1c} \wedge \neg x_{3c}) \equiv (\neg x_{2c} \vee \neg x_{1c}) \wedge (\neg x_{2c} \vee \neg x_{3c}) \\ x_{3c} \to \neg x_{1c} \wedge \neg x_{2c} \equiv \neg x_{3c} \vee (\neg x_{1c} \wedge \neg x_{2c}) \equiv (\neg x_{3c} \vee \neg x_{1c}) \wedge (\neg x_{3c} \vee \neg x_{2c}) \end{array}$$

  Since $\vee$ and $\wedge$ are commutative and idempotent, we can remove the green clauses:

$$\bigwedge_{c=1}^{3} (\neg x_{1c} \vee \neg x_{2c}) \wedge (\neg x_{1c} \vee \neg x_{3c}) \wedge (\neg x_{2c} \vee \neg x_{1c})$$

  Using compact notation

$$\bigwedge_{c=1}^{3} \bigwedge_{a=1}^{2} \bigwedge_{b=a+1}^{3} (\neg x_{ac} \vee \neg x_{bc})$$

Example: Placement of guests

- The compact notation used is perhaps a bit more difficult to read, but is quit close to programming.
- For instance, using PySAT we can use 3 nested cicles to codify the formula

$$\bigwedge_{c=1}^{3} \bigwedge_{a=1}^{2} \bigwedge_{b=a+1}^{3} (\neg x_{ac} \vee \neg x_{bc})$$

- See the Colab Notebook with an implementation of this example.

Example: Graph coloring

Graph coloring

Can one assign one of K colors to each of the vertices of graph $G = (V, E)$ such that adjacent vertices are assigned different colors?

- Create $|V| \times K$ variables:

  x_{ij} is true iff vertex i is assigned color j

- For each edge (u, v), require different assigned colors to u and v:

$$\text{for each } \; 1 \leq j \leq K, \quad (x_{uj} \to \neg x_{vj})$$

- ...

Example: Graph coloring

- Each vertex is assigned exactly one color.
  - *At least* one color to each vertex:

$$\text{for each } 1 \leq i \leq |V|, \quad \bigvee_{j=1}^{K} x_{ij}$$

  - *At most* one color to each vertex:

$$\text{for each } 1 \leq i \leq |V|, \quad \bigwedge_{a=1}^{K} (x_{ia} \to \bigwedge_{b=1, b\neq a}^{K} \neg x_{ib})$$

  since $\vee$ and $\wedge$ are commutative and idempotent, a better encoding is

$$\text{for each } 1 \leq i \leq |V|, \quad \bigwedge_{a=1}^{K-1} (x_{ia} \to \bigwedge_{b=a+1}^{K} \neg x_{ib})$$

  or equivalently,

$$\text{for each } 1 \leq i \leq |V|, \quad \bigwedge_{a=1}^{K-1} \bigwedge_{b=a+1}^{K} (\neg x_{ia} \vee \neg x_{ib})$$

Example: Graph coloring

Summing up:

- Adjacent vertices must have different colors.

$$\bigwedge_{(u,v)\in E} \bigwedge_{j=1}^{K} (\neg x_{u,j} \vee \neg x_{vj})$$

- At least one color to each vertex.

$$\bigwedge_{i=1}^{|V|} \bigvee_{j=1}^{K} x_{ij}$$

- At most one color to each vertex.

$$\bigwedge_{i=1}^{|V|} \bigwedge_{a=1}^{K-1} \bigwedge_{b=a+1}^{K} (\neg x_{ia} \vee \neg x_{ib})$$

- See the Colab Notebook with an implementation of this problem.