Logic and Mechanized Reasoning
Using SAT Solvers

Marijn J.H. Heule

Carnegie Mellon University

Solving 2-SAT: Complexity

A k-SAT formula is a CNF formula such that each clause has a length of at most k.

Solving a k-SAT formula is NP-complete for $k \geq 3$

However, 2-SAT can be solved in polynomial time using

- Unit propagation; and
- Autarky reasoning.

Solving 2-SAT: Unit Propagation

Let Γ be a 2-SAT formula, p a propositional variable occurring in Γ, and τ the assignment with $\tau(p) = \top$.

Unit propagation on Γ using τ has two possible outcomes:

- Unit propagation results in a conflict: All satisfying assignments of Γ assign p to false.

Solving 2-SAT: Unit Propagation

Let Γ be a 2-SAT formula, p a propositional variable occurring in Γ, and τ the assignment with $\tau(p) = \top$.

Unit propagation on Γ using τ has two possible outcomes:

- Unit propagation results in a conflict: All satisfying assignments of Γ assign p to false.
- Unit propagation terminates without a conflict: Let τ' be the final assignment with unit propagation terminated. Now τ' is an autarky for Γ. Why?

Solving 2-SAT: Autarky

Given a 2-SAT formula Γ and a non-empty truth assignment. If unit propagation terminates without a conflict, then the extended assignment is an autarky for Γ.

- For a clause C and a non-conflicting assignment τ it holds that i) τ does not touch C, ii) τ satisfies C, or iii) τ reduces C to a unit clause (by falsifying the other literal);
- Unit clauses extend the assignment and maintain the above invariant;
- At the non-conflicting fixpoint, no touched clause is reduced in length; so
- All touched clauses are satisfied.

Solving 2-SAT: Decision Procedure

Given a 2-SAT formula Γ, the following procedure solves it in polynomial time:

- Pick an arbitrary variable p and let τ be $\tau(p) = \top$
- Let τ' be the extended assignment after applying unit propagation on Γ starting with τ
- If $[\![\Gamma]\!]_{\tau'}$ does not contain $\bot$, continue with $[\![\Gamma]\!]_{\tau'}$ (autarky)
- Otherwise continue with $[\![\Gamma]\!]_{\tau''}$ with $\tau''(p) = \bot$
- Stop if either $[\![\Gamma]\!]_{\tau'} = \top$ or $[\![\Gamma]\!]_{\tau'} = \bot$

Solving 2-SAT: Decision Procedure

Given a 2-SAT formula Γ, the following procedure solves it in polynomial time:

- Pick an arbitrary variable p and let τ be $\tau(p) = \top$
- Let τ' be the extended assignment after applying unit propagation on Γ starting with τ
- If $[\![\Gamma]\!]_{\tau'}$ does not contain $\bot$, continue with $[\![\Gamma]\!]_{\tau'}$ (autarky)
- Otherwise continue with $[\![\Gamma]\!]_{\tau''}$ with $\tau''(p) = \bot$
- Stop if either $[\![\Gamma]\!]_{\tau'} = \top$ or $[\![\Gamma]\!]_{\tau'} = \bot$

Tarjan's algorithm can be used to reduce it to linear runtime.

SAT Game

by Olivier Roussel

https://www.cs.utexas.edu/~marijn/game/2SAT/

Solving 2-SAT

SAT Solving First Steps

DPLL

Graph Coloring

SAT Solving: Introduction

Dozens of (open source) SAT solvers have been developed.

International competition have been organized since 2002

- Solvers are evaluated on a representative benchmark suite
- Practically every year clear progress is observed
- Arguably one of the drivers that advances the technology

CaDiCaL by Armin Biere is one of the strongest solvers

- Compiles easily on most operating systems
- Readable and understanable code and thus easy to modify
- Works normally from the command line, but also in Lean

SAT Solving: Demo in Lean

```
/-
Examples of use of Cadical.
-/

-- textbook: SAT example
def cadicalExample : IO Unit := do
  let (s, result) ← callCadical exCnf0
  IO.println "Output from CaDiCaL :\n"
  --IO.println s
  --IO.println "\n\n"
  IO.println (formatResult result)
  pure ()

#eval cadicalExample
-- end textbook: SAT example
```

SAT Solving: DIMACS Input Format

The DIMACS format for SAT solvers has three types of lines:

- header: `p cnf n m` in which `n` denotes the highest variable index and `m` the number of clauses
- clauses: a sequence of integers ending with "0"
- comments: any line starting with "`c` "

$$
\begin{array}{l}
(p \vee q \vee \neg r) \wedge \\
(\neg p \vee \neg q \vee r) \wedge \\
(q \vee r \vee \neg s) \wedge \\
(\neg q \vee \neg r \vee s) \wedge \\
(p \vee r \vee s) \wedge \\
(\neg p \vee \neg r \vee \neg s) \wedge \\
(\neg p \vee q \vee s)
\end{array}
$$

```
c example
p cnf 4 7
 1  2 -3 0
-1 -2  3 0
 2  3 -4 0
-2 -3  4 0
 1  3  4 0
-1 -3 -4 0
-1  2  4 0
```

SAT Solving: DIMACS Output Format

The solution line of a SAT solver starts with "`s `":

- `s SATISFIABLE`: The formula is satisfiable
- `s UNSATISFIABLE`: The formula is unsatisfiable
- `s UNKNOWN`: The solver cannot determine satisfiability

In case the formula is satisfiable, the solver emits a certificate:

- lines starting with "`v `"
- a list of integers ending with `0`
- e.g. `v -1 2 4 0`

In case the formula is unsatisfiable, then most solvers support emitting a proof of unsatisfiability to a separate file

Solving 2-SAT

SAT Solving First Steps

DPLL

Graph Coloring

DPLL: Introduction

Davis Putnam Logemann Loveland [DP60,DLL62]

Recursive procedure that in each recursive call:

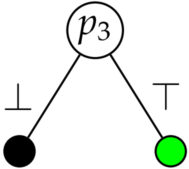
- Simplifies the formula (using unit propagation)
- Splits the formula into two subformulas
  - Variable selection heuristics (which variable to split on)
  - Direction heuristics (which subformula to explore first)

DPLL: Example

$$\Gamma_{\mathrm{DPLL}} := (p_1 \vee p_2 \vee \neg p_3) \wedge (\neg p_1 \vee p_2 \vee p_3) \wedge (\neg p_1 \vee \neg p_2 \vee p_3) \wedge (p_1 \vee p_3) \wedge (\neg p_1 \vee \neg p_3)$$

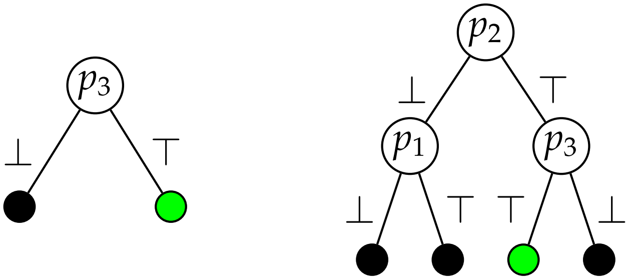
DPLL: Example

$$\Gamma_{\text{DPLL}} := (p_1 \vee p_2 \vee \neg p_3) \wedge (\neg p_1 \vee p_2 \vee p_3) \wedge (\neg p_1 \vee \neg p_2 \vee p_3) \wedge (p_1 \vee p_3) \wedge (\neg p_1 \vee \neg p_3)$$



DPLL: Example

$$\Gamma_{\text{DPLL}} := (p_1 \lor p_2 \lor \neg p_3) \land (\neg p_1 \lor p_2 \lor p_3) \land (\neg p_1 \lor \neg p_2 \lor p_3) \land (p_1 \lor p_3) \land (\neg p_1 \lor \neg p_3)$$



DPLL: Slightly Harder Example

Construct a DPLL tree for:

$$
\begin{aligned}
&(p \vee q \vee \neg r) \wedge (\neg p \vee \neg q \vee r) \wedge \\
&(q \vee r \vee \neg s) \wedge (\neg q \vee \neg r \vee s) \wedge \\
&(p \vee r \vee s) \wedge (\neg p \vee \neg r \vee \neg s) \wedge \\
&(\neg p \vee q \vee s)
\end{aligned}
$$

What is a good heuristic?

DPLL: Slightly Harder Example

Construct a DPLL tree for:

$$(p \vee q \vee \neg r) \wedge (\neg p \vee \neg q \vee r) \wedge$$
$$(q \vee r \vee \neg s) \wedge (\neg q \vee \neg r \vee s) \wedge$$
$$(p \vee r \vee s) \wedge (\neg p \vee \neg r \vee \neg s) \wedge$$
$$(\neg p \vee q \vee s)$$

What is a good heuristic?

A cheap and reasonably effective heuristic is MOMS:
Maximum Occurrence in clauses of Minimum Size

DPLL: Pseudocode

DPLL (τ, Γ)

$\tau' :=$ Simplify (τ, Γ)

if $[\![\Gamma]\!]_{\tau'} = \top$ **then return** `satisfiable`

if $[\![\Gamma]\!]_{\tau'} = \bot$ **then return** `unsatisfiable`

$l_{\text{decision}} :=$ Decide (τ', Γ)

if (DPLL($\tau' \cup l_{\text{decision}} := \top, \Gamma$) = `satisfiable`) **then**

    return `satisfiable`

return DPLL $(\tau' \cup l_{\text{decision}} := \bot, \Gamma)$

DPLL: Demo in Lean

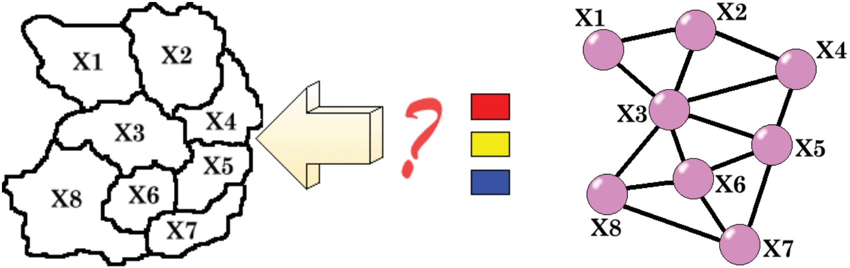
```
-- textbook: dpllSat
partial def dpllSatAux (τ : PropAssignment) (φ : CnfForm) : Option (PropAssignment × CnfForm) :=
  if φ.hasEmpty then none
  else match pickSplit? φ with
  -- No variables left to split on, we found a solution.
  | none => some (τ, φ)
  -- Split on `x`.
  -- `<|>` is the "or else" operator which tries one action, and if that failed tries the other.
  | some x => goWithNew x τ φ <|> goWithNew (-x) τ φ

where
  /-- Assigns `x` to true and continues out DPLL. -/
  goWithNew (x : Lit) (τ : PropAssignment) (φ : CnfForm) : Option (PropAssignment × CnfForm) :=
    let (τ', φ') := propagateWithNew x τ φ
    dpllSatAux τ' φ'

/-- Solve `φ` using DPLL. Return a satisfying assignment if found, otherwise `none`. -/
def dpllSat (φ : CnfForm) : Option PropAssignment :=
  let ⟨τ, φ⟩ := propagateUnits [] φ
  (dpllSatAux τ φ).map fun ⟨τ, _⟩ => τ
-- end textbook: dpllSat
```

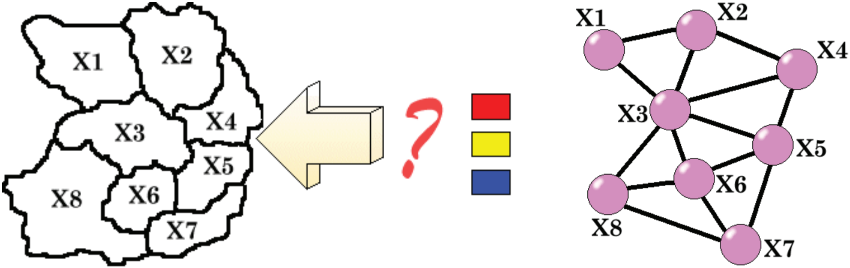
Graph Coloring: Introduction

Given a graph $G(V, E)$, can the vertices be colored with k colors such that for each edge $(v, w) \in E$, the vertices v and w are colored differently.



Graph Coloring: Introduction

Given a graph $G(V, E)$, can the vertices be colored with k colors such that for each edge $(v, w) \in E$, the vertices v and w are colored differently.

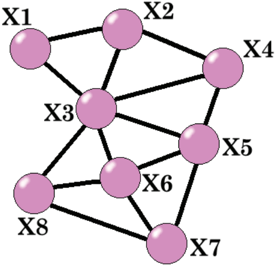


Possible problem: symmetries!

Graph Coloring: Format

- Header starts with `p edge`
- Followed by number of vertices and number of edges

```
p edge 8 13
e 1 2
e 1 3
e 2 3
e 2 4
e 3 5
e 3 6
e 3 8
e 4 5
e 5 6
e 5 7
e 6 7
e 6 8
e 7 8
```



Graph Coloring: Encoding

Variables	**Range**	**Meaning**
$p_{v,i}$	$i \in \{1, \ldots, c\}$ $v \in \{1, \ldots, \|V\|\}$	node v has color i

Clauses	**Range**	**Meaning**
$(p_{v,1} \vee p_{v,2} \vee \cdots \vee p_{v,c})$	$v \in \{1, \ldots, \|V\|\}$	v is colored
$(\neg p_{v,s} \vee \neg p_{v,t})$	$s \in \{1, \ldots, c-1\}$ $t \in \{s+1, \ldots, c\}$	v has at most one color
$(\neg p_{v,i} \vee \neg p_{w,i})$	$(v, w) \in E$	v and w have a different color
???	???	breaking symmetry

Graph Coloring: Lean Demo

```
def main (args : List String) : IO Unit := do
  let graphFname :: nColours :: _ ← args
    | do
      IO.println "Usage: <graph.edge> <colors>"
      return ()
  let some nColours ← nColours.toNat?
    | throwThe IO.Error s!"Invalid colour count: {nColours}"
  let g ← readEdgeFile graphFname
  match (← checkColourable g nColours) with
  | some vs =>
    IO.println s!"The graph is {nColours}-colourable! Satisfying assignment: {vs}"
  | none =>
    IO.println s!"The graph is not {nColours}-colourable."
```

Graph Coloring: Sudoku

Sudoku can be viewed as a graph coloring problem:

- Each cell is a vertex
- Vertices are connected if they occur in the same row / column / square
- There are 9 colors

The solution must be unique

- At least 17 givens

Who can solve this sudoku?

	4		3					
						7	9	
			6					
			1	4		5		
9							1	
2								6
				7	2			
	5					8		
				9				

Graph Coloring: Sudoku

Sudoku can be viewed as a graph coloring problem:

- Each cell is a vertex
- Vertices are connected if they occur in the same row / column / square
- There are 9 colors

The solution must be unique

- At least 17 givens

Who can solve this sudoku?

1	4	*7*	3	*8*	*9*	*2*	*6*	*5*
5	*8*	*6*	*2*	*1*	*4*	7	9	*3*
3	*9*	*2*	6	*5*	*7*	*1*	*8*	*4*
8	*7*	*3*	1	4	*6*	5	*2*	*9*
9	*6*	*4*	*7*	*2*	*5*	*3*	1	*8*
2	*1*	*5*	*9*	*3*	*8*	*4*	*7*	6
6	*3*	*8*	*5*	7	2	*9*	*4*	*1*
7	5	*9*	*4*	*6*	*1*	8	*3*	*2*
4	*2*	*1*	*8*	9	*3*	*6*	*5*	*7*

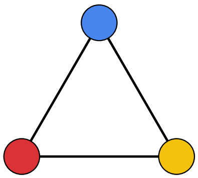
Graph Coloring: Sudoku in Lean

```
+-----------------------+
| 1 4 7 | 3 8 9 | 2 6 5 |
| 5 8 6 | 2 1 4 | 7 9 3 |
| 3 9 2 | 6 5 7 | 1 8 4 |
+-------+-------+-------+
| 8 7 3 | 1 4 6 | 5 2 9 |
| 9 6 4 | 7 2 5 | 3 1 8 |
| 2 1 5 | 9 3 8 | 4 7 6 |
+-------+-------+-------+
| 6 3 8 | 5 7 2 | 9 4 1 |
| 7 5 9 | 4 6 1 | 8 3 2 |
| 4 2 1 | 8 9 3 | 6 5 7 |
+-----------------------+
```

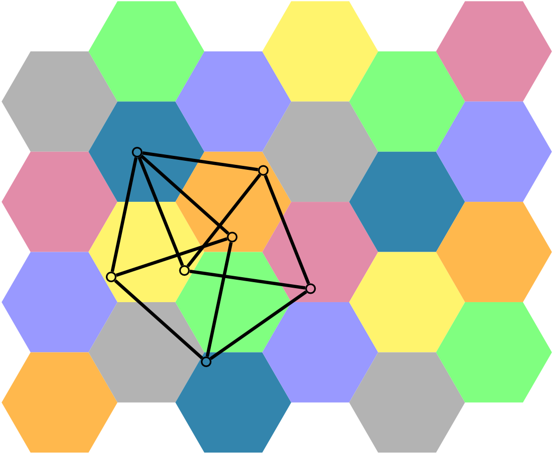
Graph Coloring: Chromatic Number of the Plane

The Hadwiger-Nelson problem:
How many colors are required to color the plane such that each pair of points that are exactly 1 apart are colored differently?

The answer must be three or more because three points can be mutually 1 apart—and thus must be colored differently.



Graph Coloring: Bounds since the 1950s



- The Moser Spindle graph shows the lower bound of 4
- A coloring of the plane showing the upper bound of 7

Graph Coloring: First progress in decades

Recently enormous progress:

- Lower bound of 5 [DeGrey '18] based on a 1581-vertex graph
- This breakthrough started a polymath project
- Improved bounds of the fractional chromatic number of the plane

Graph Coloring: First progress in decades

Recently enormous progress:

- Lower bound of 5 [DeGrey '18] based on a 1581-vertex graph
- This breakthrough started a polymath project
- Improved bounds of the fractional chromatic number of the plane

We found smaller graphs with SAT:

- 874 vertices on April 14, 2018
- 803 vertices on April 30, 2018
- 610 vertices on May 14, 2018

Record by Proof Minimization: 510 Vertices [Heule 2021]