Solver-aided reasoning

UW CSE P 504
Ways to solve a program analysis problem

- Abstract interpretation
- Type checking
- Model checking
  - Exhaustive exploration of a graph of possible executions
- SAT-solving
  - One approach to automated theorem proving
Reducing one problem to another

Examples:

- Want kth largest element in a set. Know how to sort.
- Want to flip a fair coin. Only have a biased coin.
- Want to sort. Know how to compute convex hull. Use points $\langle x, x^2 \rangle$.

Reductions are common in proofs about computational complexity.
Reducing a problem to SAT (boolean satisfiability)

The SAT problem: Given a boolean formula over variables \( V \), assign each variable to true or false to make the formula true.

Example SAT problem: \((a \land (b \lor \neg c)) \lor (\neg b \land c)\)

One solution: \(a=true, \ b=true, \ c=false\)

The output is also called a “model” of the formula.
Reducing a problem to SAT (boolean satisfiability)

The SAT problem: Given a boolean formula over variables $V$, assign each variable to true or false to make the formula true.

Example SAT problem: $(a \text{ and } (b \text{ or } \neg c)) \text{ or } (\neg b \text{ and } c)$

One solution: $a=$true, $b=$true, $c=$false

The output is also called a “model” of the formula.

Why SAT?
Reducing a problem to SAT (boolean satisfiability)

Idea: Kautz & Selman, 1996

First implementation:

Automatic SAT-Compilation of Planning Problems

Michael D. Ernst, Todd D. Millstein, and Daniel S. Weld*
Department of Computer Science and Engineering
University of Washington, Box 352350 Seattle WA 98195–2350 USA
{mernst, todd, weld}@cs.washington.edu

Abstract
Recent work by Kautz et al. provides tantalizing evidence that large, classical planning problems may be efficiently solved by translating them into propositional satisfiability problems, using stochas-

* We present an analytic framework that accounts for all previously reported non-causal encodings, including several novel possibilities. We parameterize the space of encodings along two major dimensions, action and frame representation. For twelve points in this two-dimensional space, we list the axioms necessary for a
Reducing a problem to SAT (boolean satisfiability)

This is just solving the problem with extra steps. Why would we do this?
Reducing a problem to SAT (boolean satisfiability)

This is just solving the problem with extra steps. Why would we do this?

Counter-arguments:

- Cannot be faster than solving the original problem directly
- SAT is NP-complete
Reducing a problem to SAT (boolean satisfiability)

This is just solving the problem with extra steps. Why would we do this?

Counter-arguments:

- Cannot be faster than solving the original problem directly (might be slower)
- SAT is NP-complete

Arguments in favor:

- Searching for a SAT solution is *highly* optimized
  The solver either returns a solution, or times out
Reducing a problem to SAT (boolean satisfiability)

This is just solving the problem with extra steps. Why would we do this?

Counter-arguments:

- Cannot be faster than solving the original problem directly (might be slower)
- SAT is NP-complete

Arguments in favor:

- Searching for a SAT solution is *highly* optimized
- The solver either returns a solution, or times out

1972: A problem reduces to SAT, therefore it is hard.
Today: A problem reduces to SAT, therefore it is easy.
SAT solver input must be in CNF form

Example SAT problem: (a and (b or ¬c)) or (¬b and c)

The same formula in CNF (conjunctive normal form):

\[(x_1 \lor x_2) \land (\neg x_1 \lor a) \land (\neg x_1 \lor b \lor \neg c) \land (\neg x_2 \lor \neg b) \land (\neg x_2 \lor c)\]

CNF = conjunction of disjunctions of possibly-negated variables
Satisfiability modulo theories

Extend a SAT solver:

- Input format permits equations in a “theory”
  - Example theories: real arithmetic, modular arithmetic, arrays, bit vectors
- The SAT solver calls a solver or checker for the theory
SAT solving

Brute force checking of all possible assignments, with some smarts
The n-queens problem [Bezzel, 1848]

Place $n$ queens on a $n \times n$ chessboard so that none is attacking any other.

Simple approach: brute-force search
The n-queens problem
The n-queens problem
The n-queens problem
The n-queens problem
The n-queens problem
The n-queens problem
The n-queens problem
The n-queens problem
Exercise: Write a SAT formula for the 8-queens problem

1. Write out the constraints in math or in English
2. Decide encoding as boolean variables
3. Translate the constraints to boolean formulas
   Does not have to be in CNF
4. Translate the variable assignment into a chess board
N-queens constraints

Exactly one queen per row
Exactly one queen per column
At most one queen per diagonal
N-queens encoding

One boolean variable per square

Var is true if there is a queen there

\[
\begin{align*}
a8 &= \text{false} & a7 &= \text{false} \\
b8 &= \text{false} & b7 &= \text{false} \\
c8 &= \text{false} & c7 &= \text{false} \\
d8 &= \text{false} & d7 &= \text{true} \\
e8 &= \text{false} & e7 &= \text{false} \\
f8 &= \text{true} & f7 &= \text{false} \\
g8 &= \text{false} & g7 &= \text{false} \\
h8 &= \text{false} & h7 &= \text{false} \\
\end{align*}
\]

etc.
One queen per row

At least one queen per row:
\[(a_1 \lor b_1 \lor c_1 \lor d_1 \lor e_1 \lor f_1 \lor g_1 \lor h_1) \land (a_2 \lor b_2 \lor c_2 \lor d_2 \lor e_2 \lor f_2 \lor g_2 \lor h_2) \land \ldots\]

No more than one queen per row:
\[\neg a_1 \lor \neg b_1 \land \neg a_1 \lor \neg c_1 \land \neg a_1 \lor \neg d_1 \land \ldots \land \neg b_1 \lor \neg c_1 \land \neg b_1 \lor \neg d_1 \land \ldots \land \ldots\]
\[\neg a_2 \lor \neg b_2 \land \neg a_2 \lor \neg c_2 \land \ldots \land \ldots\]

...
One queen per column

At least one queen per column:
(a1 ∨ a2 ∨ a3 ∨ a4 ∨ a5 ∨ a6 ∨ a7 ∨ a8) ∧
(b1 ∨ b2 ∨ b3 ∨ b4 ∨ b5 ∨ b6 ∨ b7 ∨ b8) ∧

No more than one queen per column:
(¬a1 ∨ ¬a2) ∧
(¬a1 ∨ ¬a3) ∧
(¬a1 ∨ ¬a4) ∧ ... ∧
(¬a2 ∨ ¬a3) ∧
(¬a2 ∨ ¬a4) ∧ ... ∧ ...

(¬b1 ∨ ¬b2) ∧
(¬b1 ∨ ¬b3) ∧ ... ∧

...
At most one queen per diagonal

SW-NE diagonals:
\((\neg a_7 \lor \neg b_8) \land (\neg a_6 \lor \neg b_7) \land (\neg b_7 \lor \neg c_8) \land (\neg a_7 \lor \neg c_8) \land \ldots\)

NW-SE diagonals:
\((\neg g_8 \lor \neg h_7) \land (\neg f_8 \lor \neg g_7) \land (\neg g_7 \lor \neg h_6) \land (\neg f_8 \lor h_6) \land \ldots\)
N-queens demo
SAT solving

Brute force checking of all possible assignments, with some smarts
SAT solving

Brute force checking of all possible assignments, with some smarts

Backtracking search over all $2^n$ possible assignments

At any point, the assignment is partial
If \( \{ a = \text{true}, b = \text{false} \} \) is inconsistent with the formula, no need to explore \( c, d, \text{etc.} \)

Unit clause rule: In a clause, if one var is unassigned and all others are false, then the unassigned var must be true
DPLL algorithm (Davis–Putnam–Logemann–Loveland)

DPLL(cnf, a):
// cnf is a formula, a is a partial assignment

a ← unit-clause(cnf, a) // boolean constraint propagation (BCP)

switch eval(cnf, a):

  case true: return a

  case false: return “unsat”

  case unknown:

    v ← choose(unassigned-vars(cnf, a))

    return DPLL(cnf, a[v ↦ true]) or DPLL(cnf, a[v ↦ false])
DPLL algorithm (Davis–Putnam–Logemann–Loveland)

DPLL(cnf, a):

\[ a \leftarrow \text{unit-clause}(\text{cnf, } a) \]

\[
\text{switch eval}(\text{cnf, } a):
\]

- case true: return \( a \)
- case false: return “unsat”
- case unknown:

\[
\begin{align*}
v & \leftarrow \text{choose}(\text{unassigned-vars}(\text{cnf, } a)) \\
\text{return } & \text{DPLL}(\text{cnf, } a[v \mapsto \text{true}]) \text{ or DPLL}(\text{cnf, } a[v \mapsto \text{false}])
\end{align*}
\]

// cnf is a formula, a is a partial assignment

Idea: remember the combination of variables that made \( a \) unassignable. Avoid that combination in the future.

Idea: backtrack more than one level.

Idea: choose variables likely to lead to quick resolution.