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

Problem 1 $\xrightarrow{\text{encoding or translation}}$ Problem 2 $\xrightarrow{\text{solver}}$ Solution 2 $\xrightarrow{\text{decoding or translation}}$ Solution 1

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)

Problem → encoding or translation → SAT problem → SAT solver → SAT solution → decoding or translation → Solution

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 and (b or $\neg c$)) or ($\neg b$ and 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)

Problem $\xrightarrow{\text{encoding or translation}}$ SAT problem $\xrightarrow{\text{SAT solver}}$ SAT solution $\xrightarrow{\text{decoding or translation}}$ Solution

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* and (*b* or ¬*c*)) or (¬*b* 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)

Problem $\xrightarrow{\text{encoding or translation}}$ SAT problem $\xrightarrow{\text{SAT solver}}$ SAT solution $\xrightarrow{\text{decoding or translation}}$ Solution

Idea: Kautz & Selman, 1996

First implementation:

This paper appears in *Proceedings of the 15th International Joint Conference on Artificial Intelligence* (IJCAI-97), Nagoya, Aichi, Japan, August 23–29, 1997, pp. 1169-1176.

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,[1] 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)

Problem → (encoding or translation) → SAT problem → (SAT solver) → SAT solution → (decoding or translation) → Solution

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

1972:	A problem reduces to SAT, therefore it is hard.
Today:	A problem reduces to SAT, therefore it is easy.

Arguments in favor:

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

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):

$$(x1 \lor x2) \land (\neg x1 \lor a) \land (\neg x1 \lor b \lor \neg c) \land (\neg x2 \lor \neg b) \land (\neg x2 \lor c)$$

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

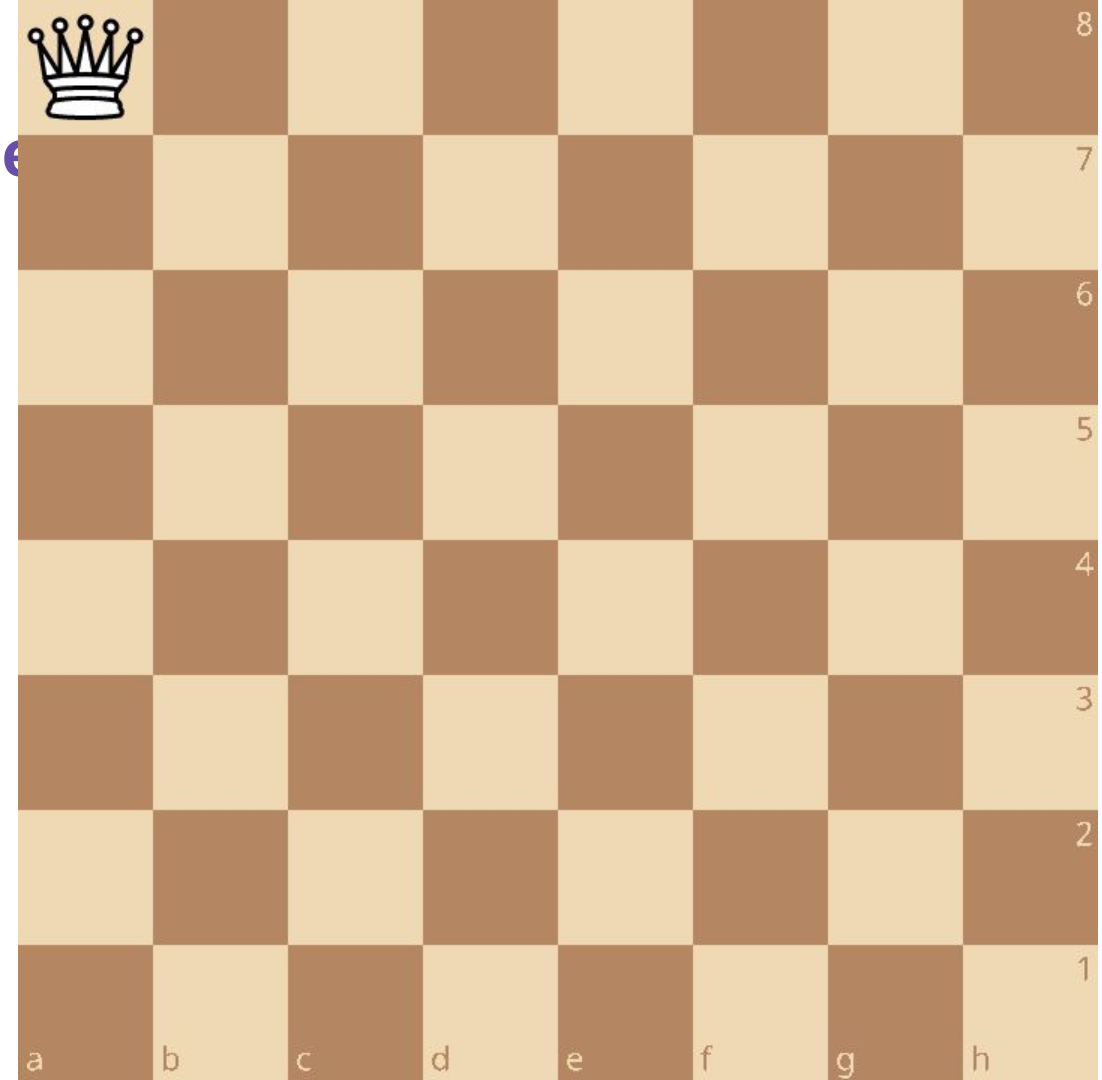
Brute force checking of all possible assignments, with some smarts

The n-queens problem [Bezzel, 1848]

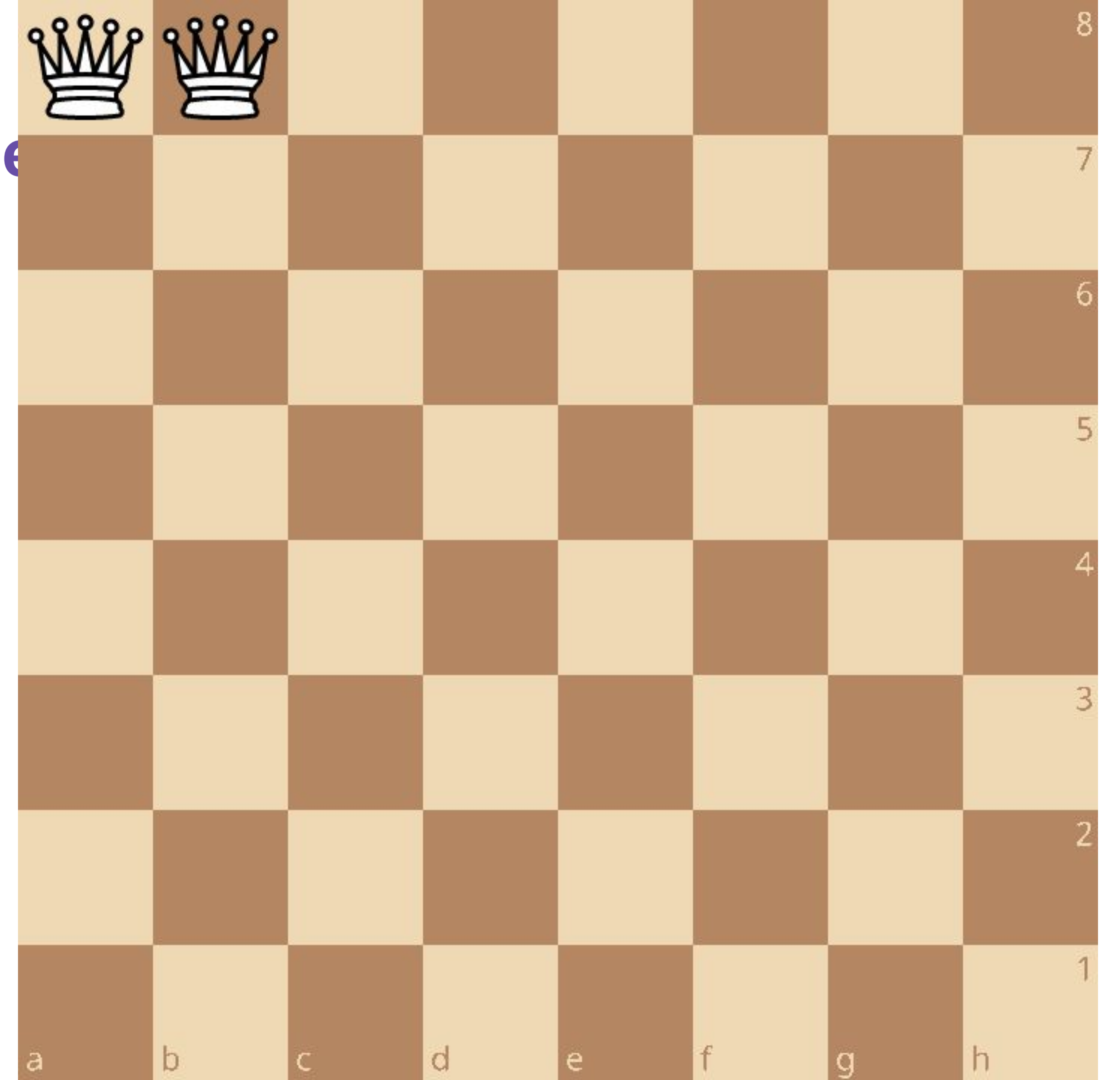
Place n queens on a $n \times n$ chessboard so that none is attacking any other.

Simple approach: brute-force search

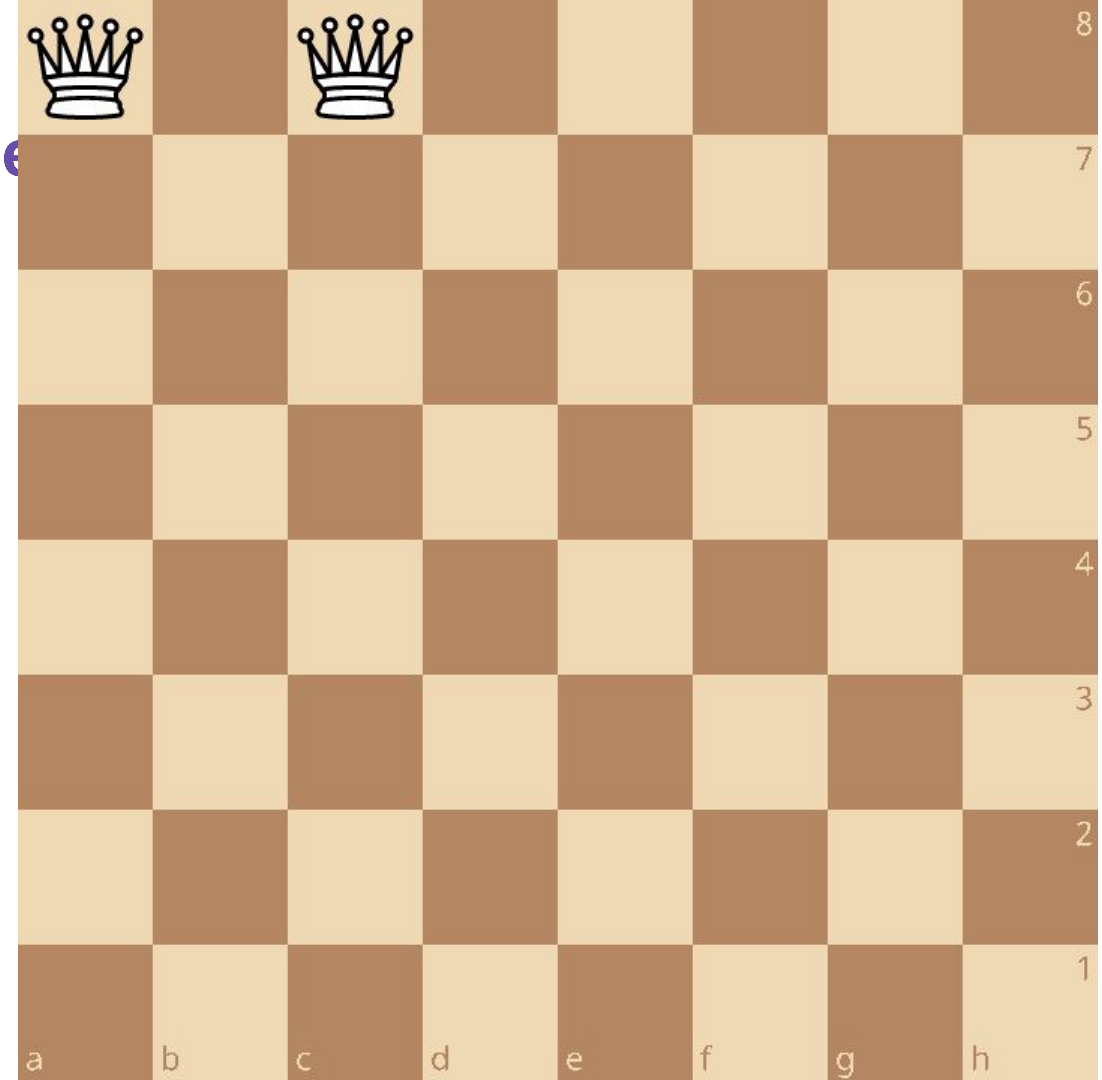
The n-quee



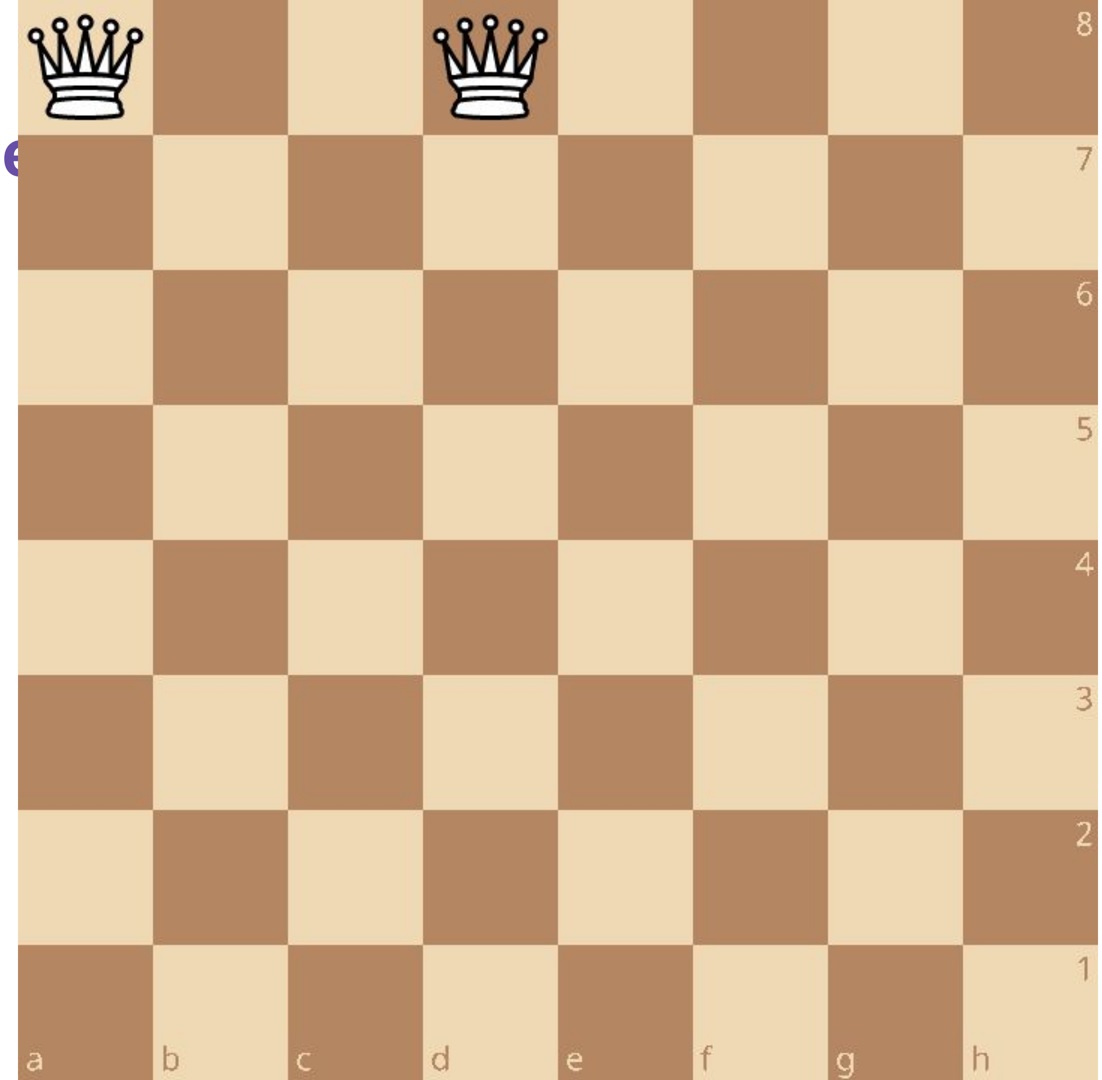
The n-quee



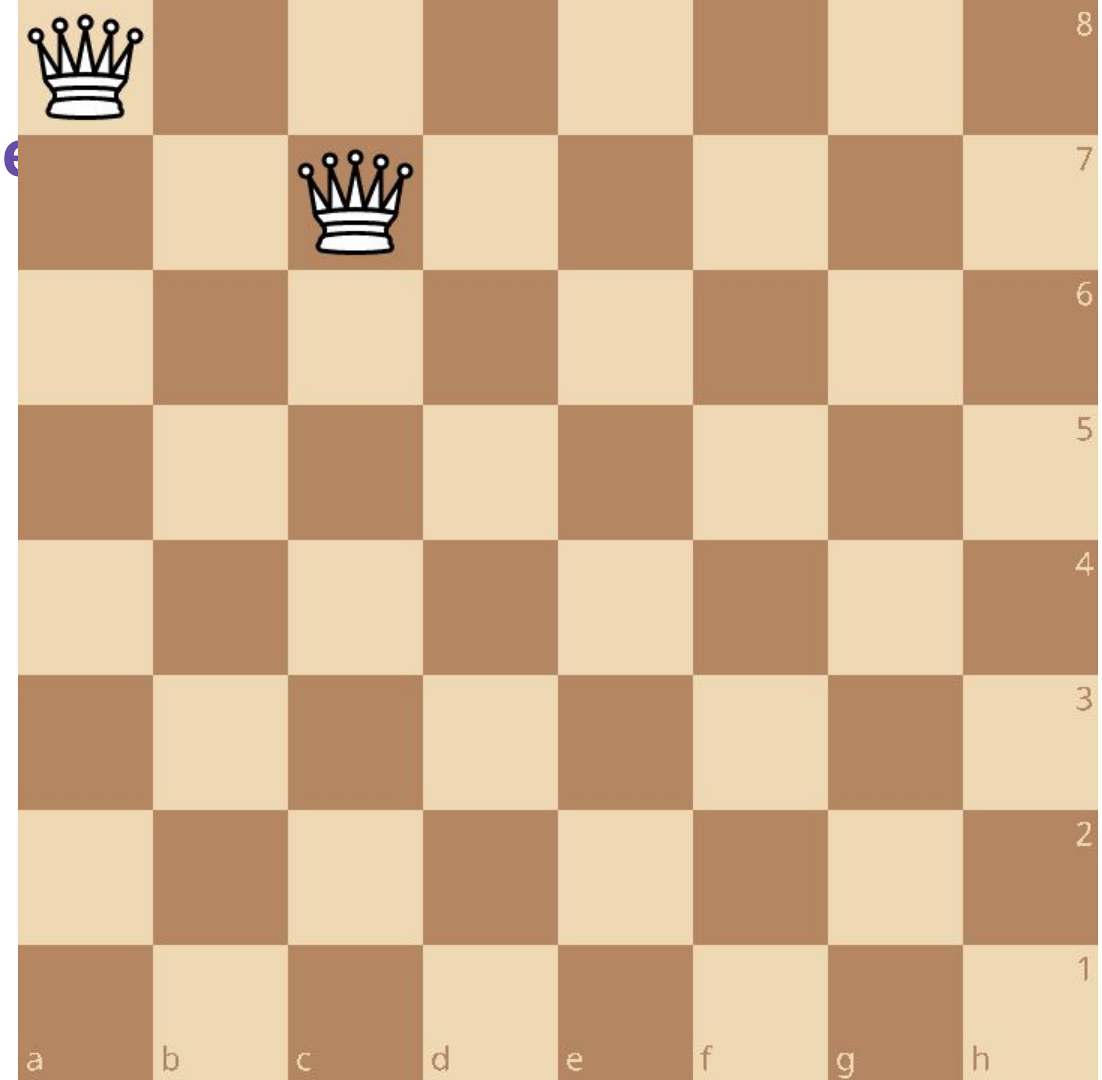
The n-quee



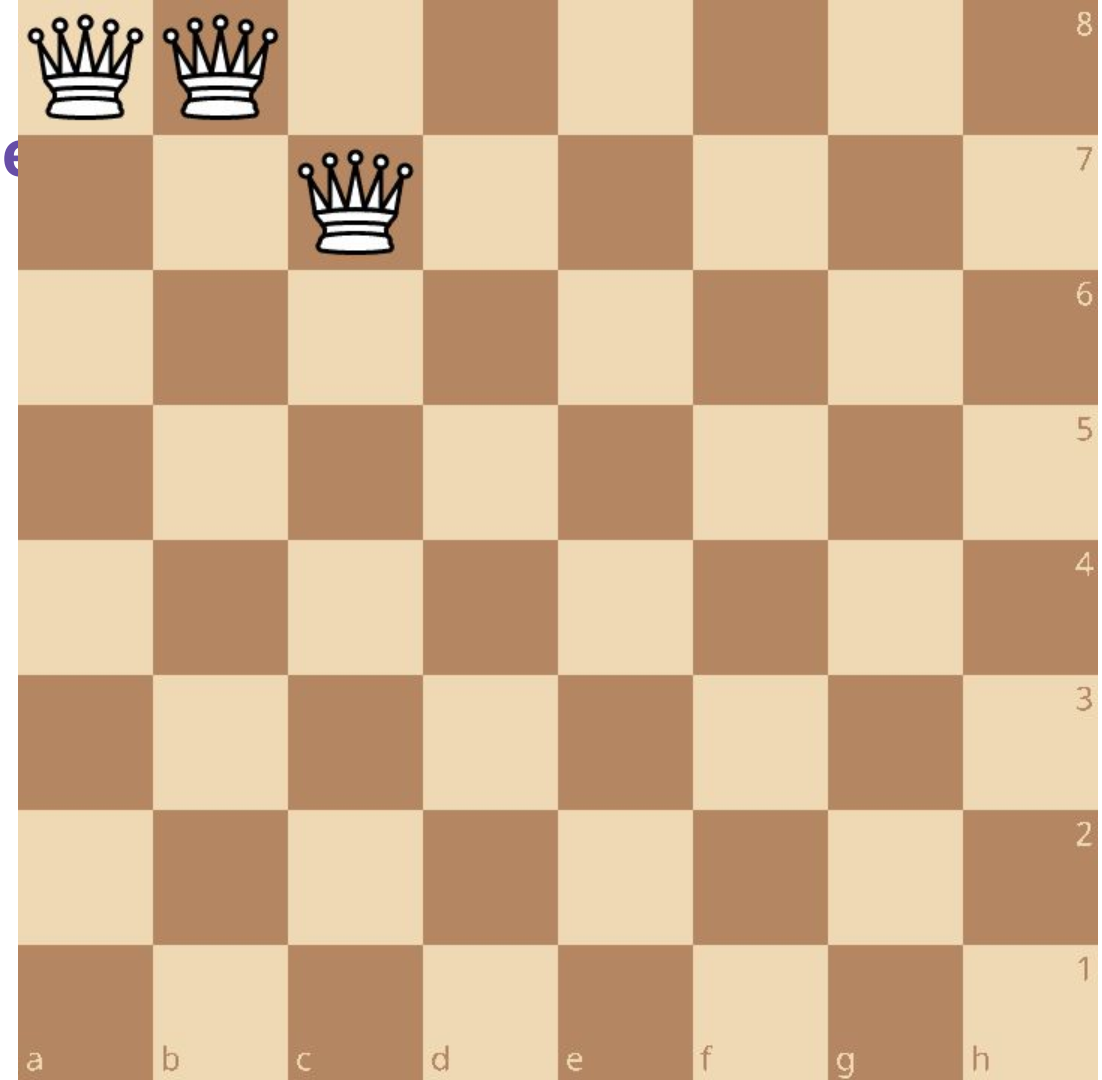
The n-quee



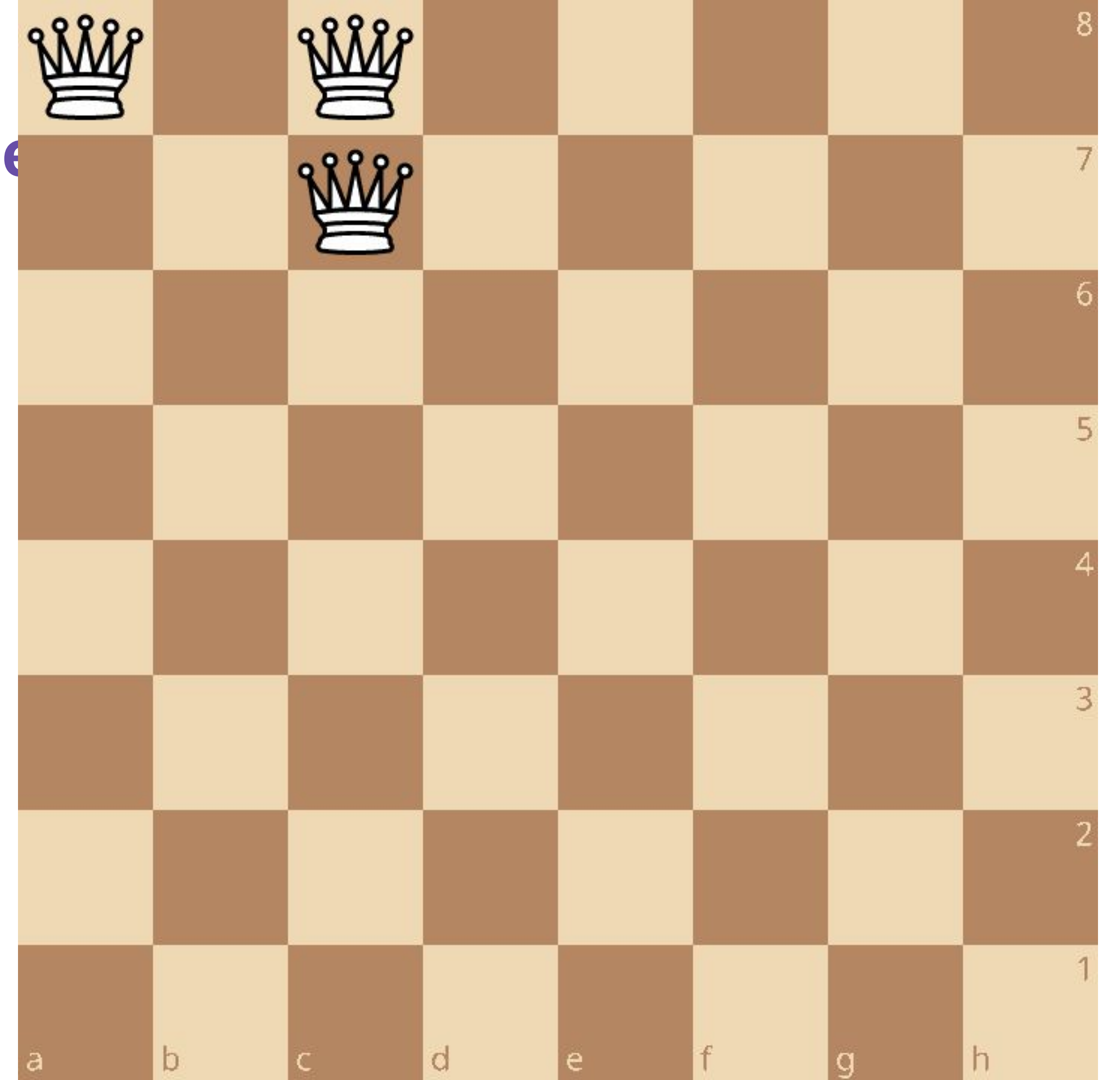
The n-quee



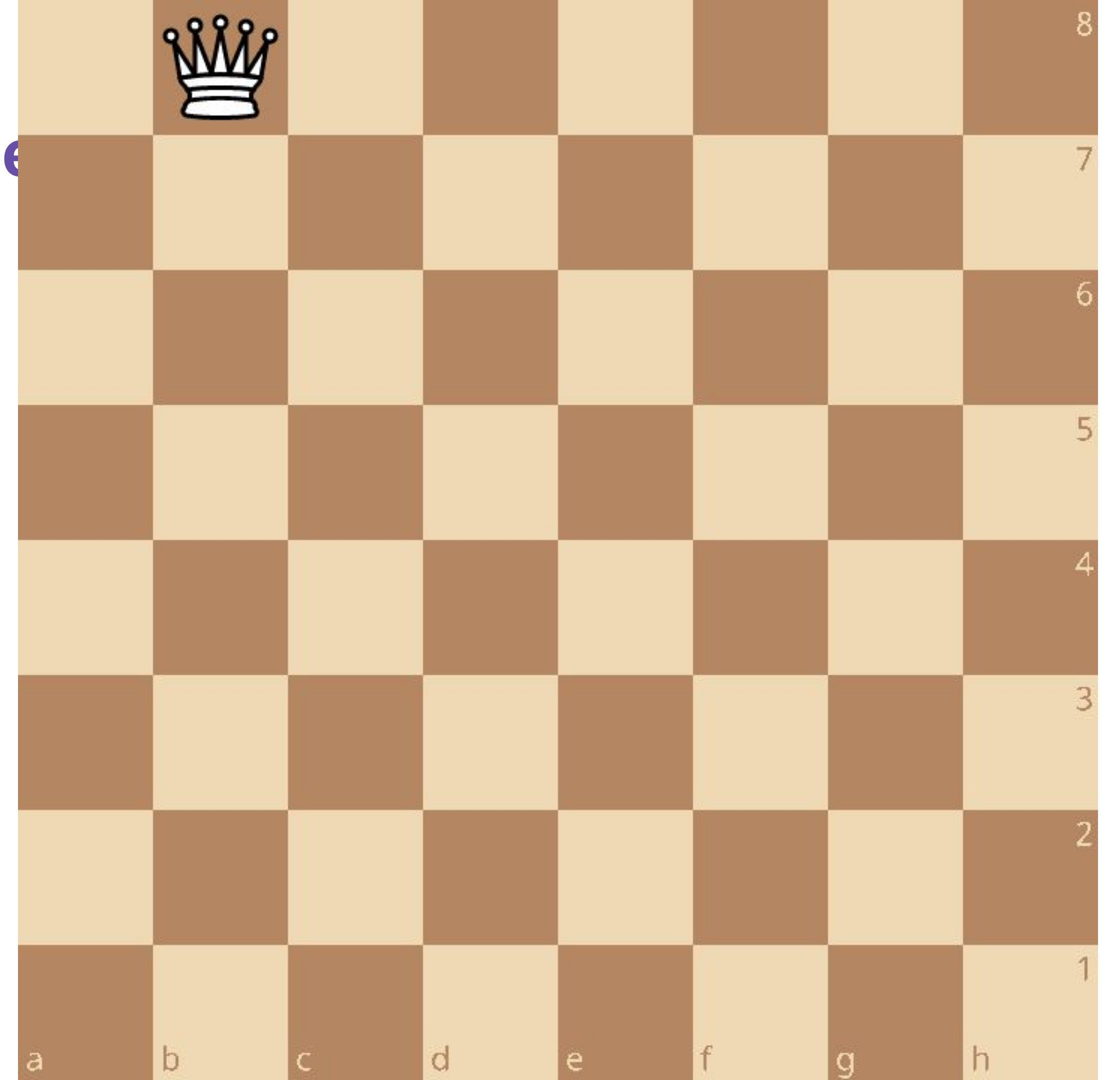
The n-quee



The n-quee



The n-quee



Exercise: Write a SAT formula for the 8-queens problem

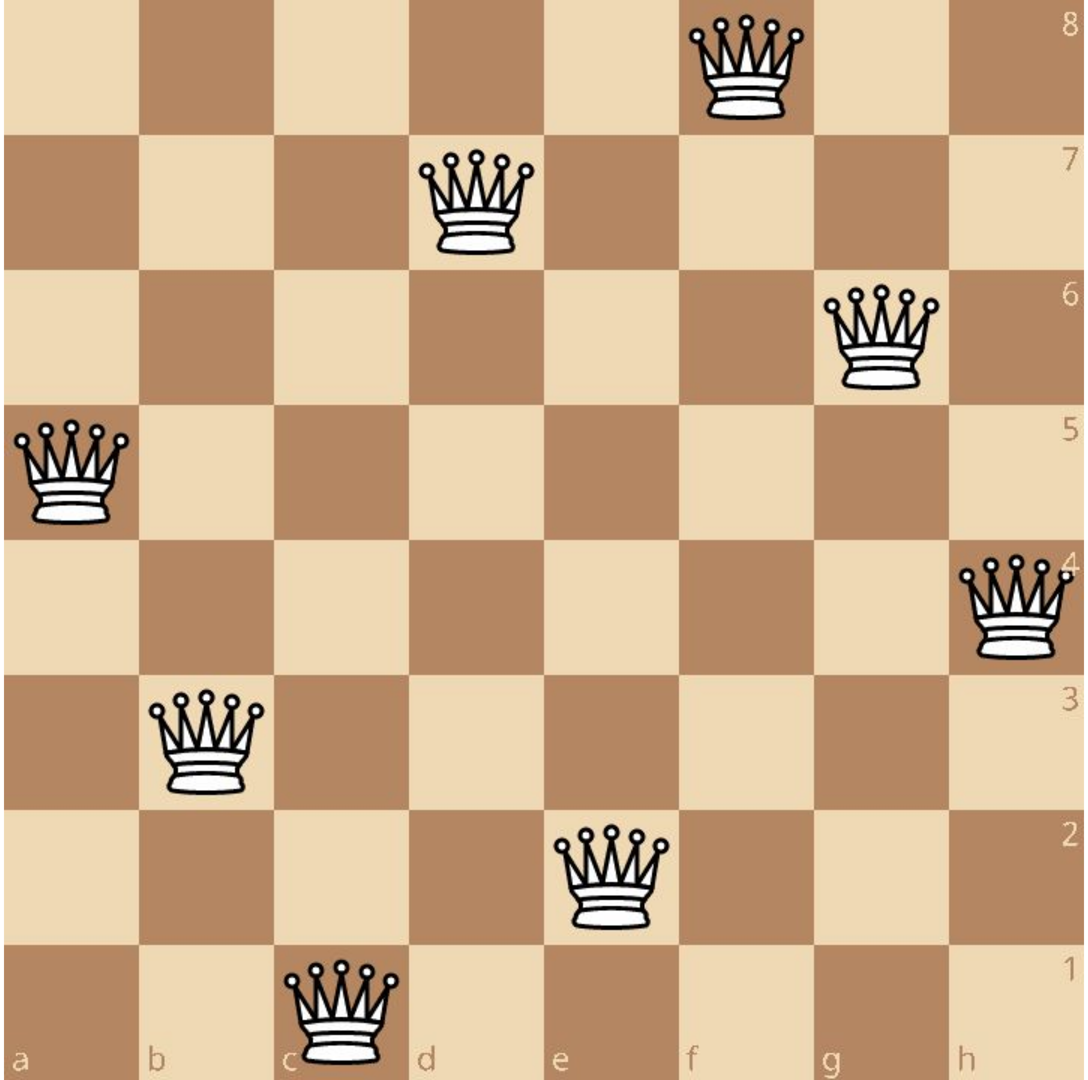
1. Write out the constraints in math or in English
2. Decide encoding as boolean variables
3. Translate the constraints to boolean formulas
   Does not have to be in CNF
4. Translate the variable assignment into a chess board

N-queens constraints

Exactly one queen per row

Exactly one queen per column

At most one queen per diagonal



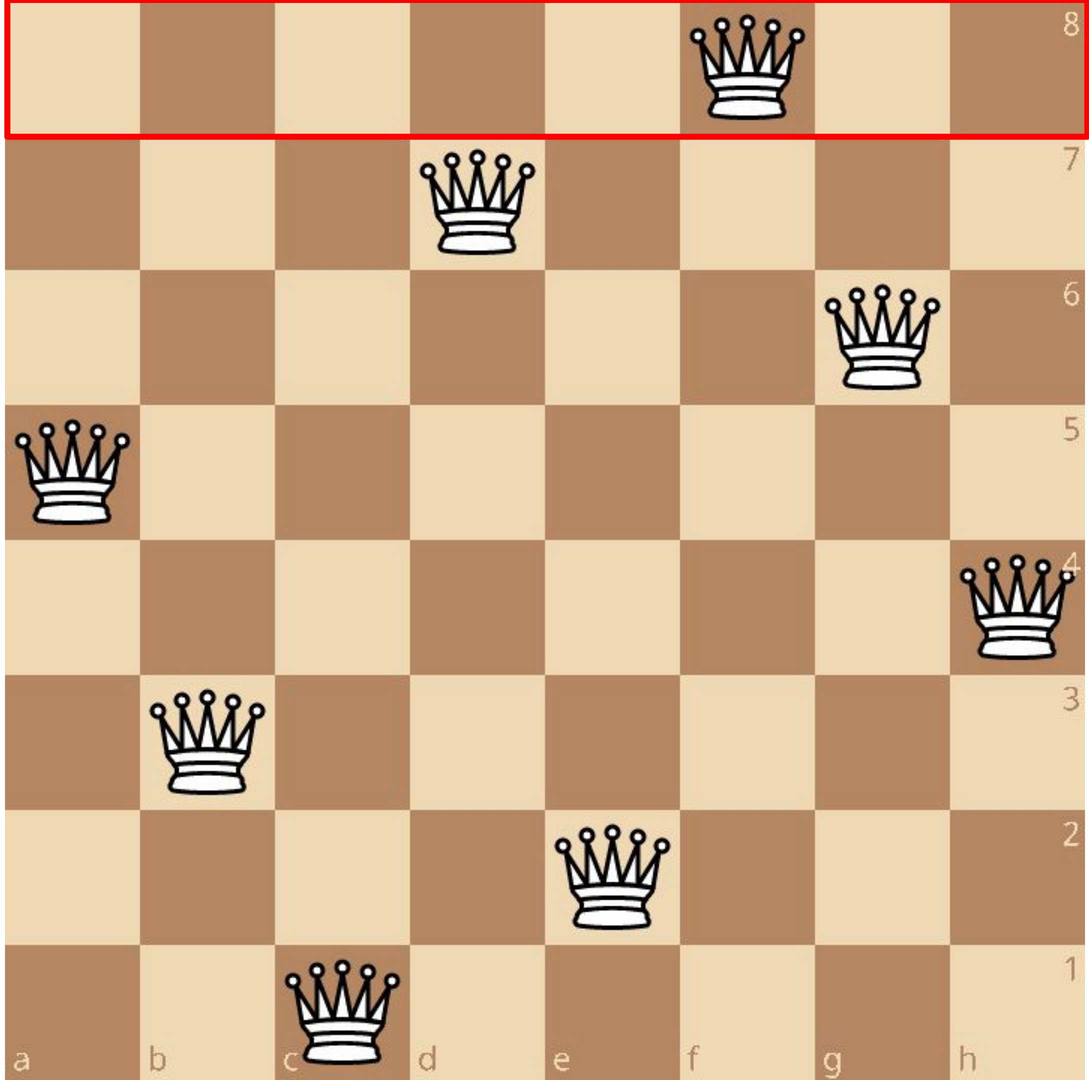
N-queens encoding

One boolean variable per square

Var is true if there is a queen there

a8 = false	a7 = false
b8 = false	b7 = false
c8 = false	c7 = false
d8 = false	d7 = true
e8 = false	e7 = false
f8 = true	f7 = false
g8 = false	g7 = false
h8 = false	h7 = false

etc.



One queen per row

At least one queen per row:

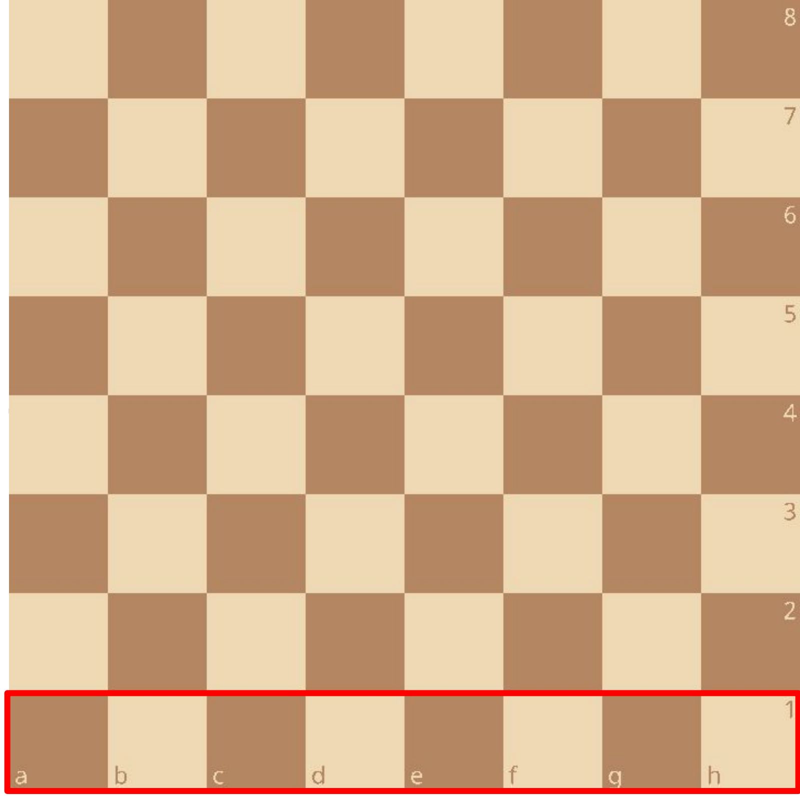
$(a1 \lor b1 \lor c1 \lor d1 \lor e1 \lor f1 \lor g1 \lor h1) \land$
$(a2 \lor b2 \lor c2 \lor d2 \lor e2 \lor f2 \lor g2 \lor h2) \land$

No more than one queen per row:

$(\neg a1 \lor \neg b1) \land$
$(\neg a1 \lor \neg c1) \land$
$(\neg a1 \lor \neg d1) \land \ldots \land$
$(\neg b1 \lor \neg c1) \land$
$(\neg b1 \lor \neg d1) \land \ldots \land \ldots$

$(\neg a2 \lor \neg b2) \land$
$(\neg a2 \lor \neg c2) \land \ldots \land$

…



One queen per column

At least one queen per column:

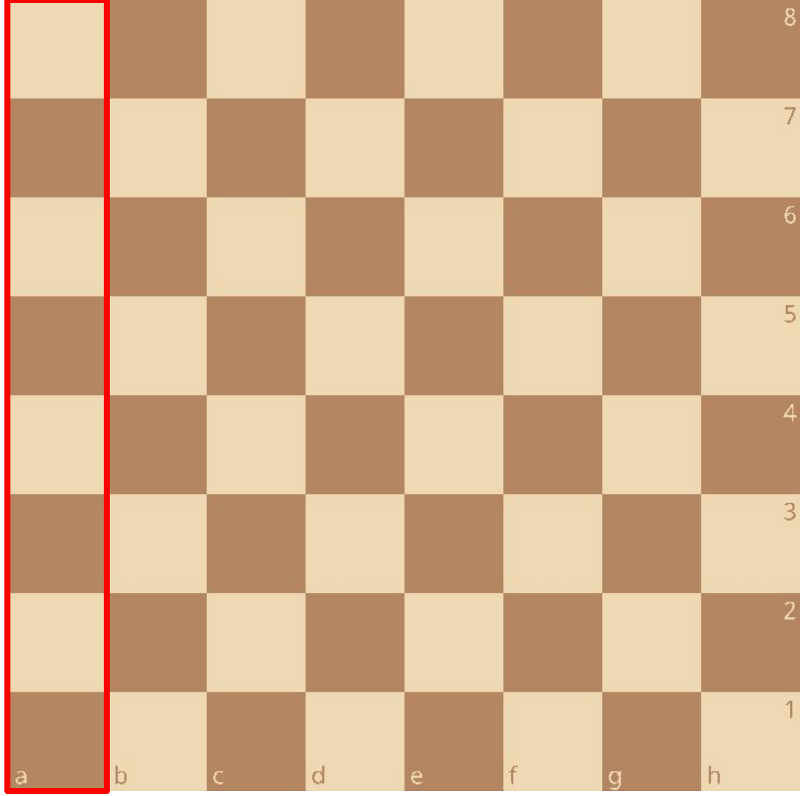
$(a1 \lor a2 \lor a3 \lor a4 \lor a5 \lor a6 \lor a7 \lor a8) \land$
$(b1 \lor b2 \lor b3 \lor b4 \lor b5 \lor b6 \lor b7 \lor b8) \land$

No more than one queen per column:

$(\neg a1 \lor \neg a2) \land$
$(\neg a1 \lor \neg a3) \land$
$(\neg a1 \lor \neg a4) \land \ldots \land$
$(\neg a2 \lor \neg a3) \land$
$(\neg a2 \lor \neg a4) \land \ldots \land \ldots$

$(\neg b1 \lor \neg b2) \land$
$(\neg b1 \lor \neg b3) \land \ldots \land$

…



At most one queen per diagonal

SW-NE diagonals:

$(\neg a7 \lor \neg b8) \land$

$(\neg a6 \lor \neg b7) \land (\neg b7 \lor \neg c8) \land (\neg a7 \lor \neg c8) \land$

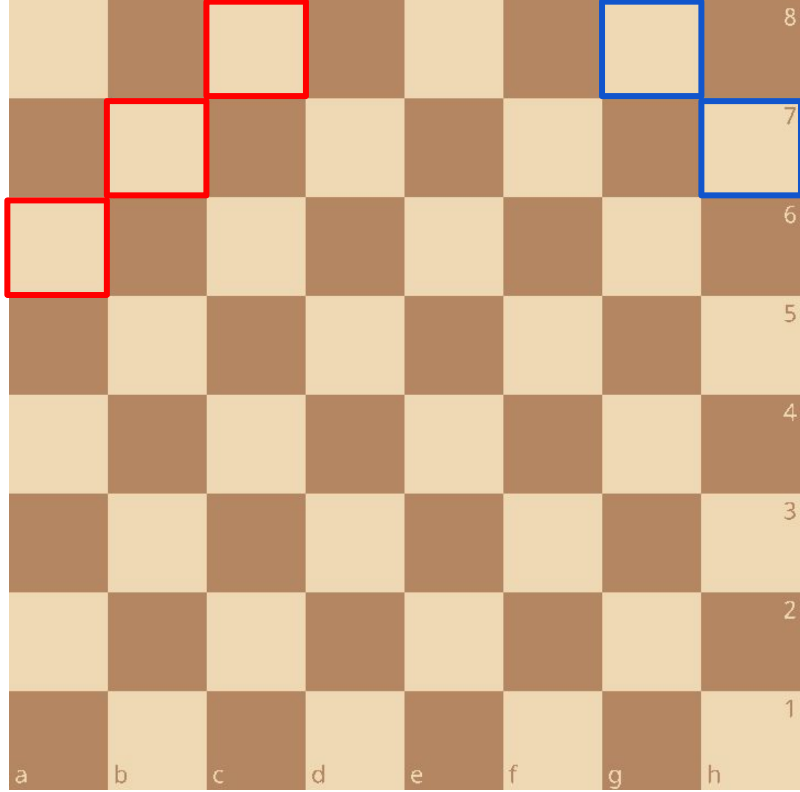
…

NW-SE diagonals:

$(\neg g8 \lor \neg h7) \land$

$(\neg f8 \lor \neg g7) \land (\neg g7 \lor \neg h6) \land (\neg f8 \lor \neg h6) \land$

…



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=true, b=false } is inconsistent with the formula, no need to explore c, d, 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])
```

Backtracking

DPLL algorithm (Davis–Putnam–Logemann–Loveland)

```
DPLL(cnf, a):                                // cnf is a formula, a is a partial assignment
    a ← unit-clause(cnf, a)                  ation (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])
```

Idea: remember the combination of variables that made *a* unassignable. Avoid that combination in the future.

Idea: choose variables likely to lead to quick resolution.

Idea: backtrack more than one level.