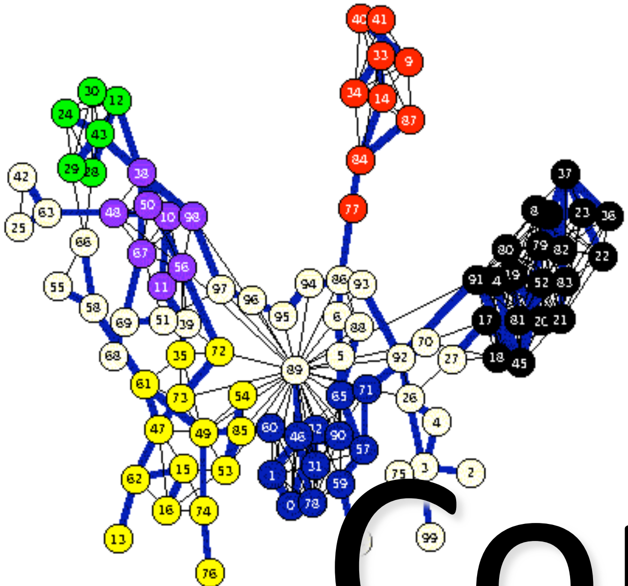


Constraint Satisfaction

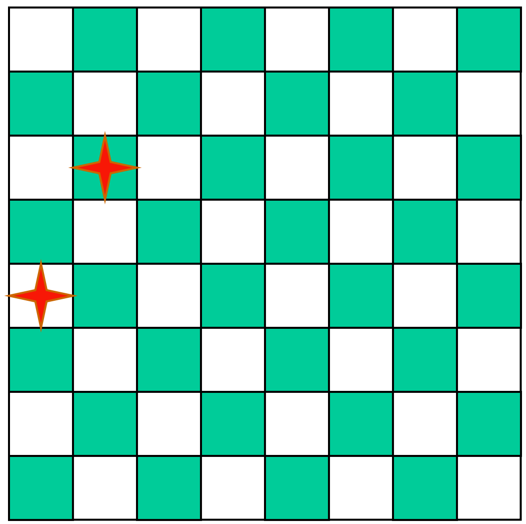
Russell & Norvig Ch. 6

Overview

- **Constraint satisfaction** is a powerful problem-solving paradigm
  - Problem: set of variables to which we must assign values satisfying problem-specific constraints
  - Constraint programming, constraint satisfaction problems (CSPs), constraint logic programming…
- Algorithms for CSPs
  - Backtracking (systematic search)
  - Constraint propagation (k-consistency)
  - Variable and value ordering heuristics
  - Backjumping and dependency-directed backtracking

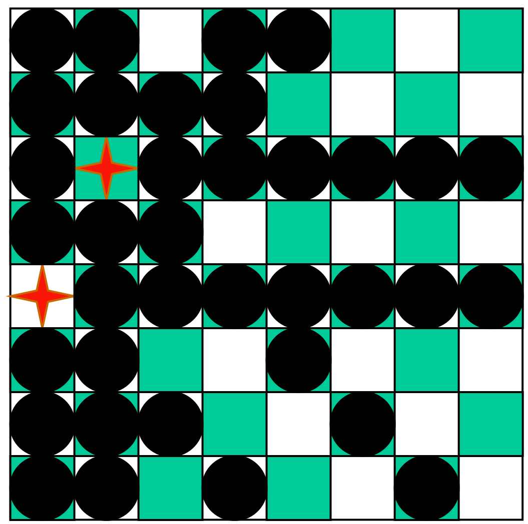
Motivating example: 8 Queens

Place 8 queens on a chess board such that none is attacking another



- **Generate-and-test** with no redundancies must try 8^8 (16,777,216) combinations!
- Unclear what heuristics might guide an algorithm A like search

Motivating example: 8-Queens



- After placing these two queens, it’s trivial to mark the squares we can no longer use
- greatly reducing the solution space!

What more do we need for 8 queens?

- Not just a successor function and goal test
- But also
  - way to **propagate constraints** imposed by one queen on placement of others
  - an early failure test

→ Explicit representation of constraints and constraint manipulation algorithms

Informal definition of CSP

- **CSP** (Constraint Satisfaction Problem), given

  (1) finite set of **variables**

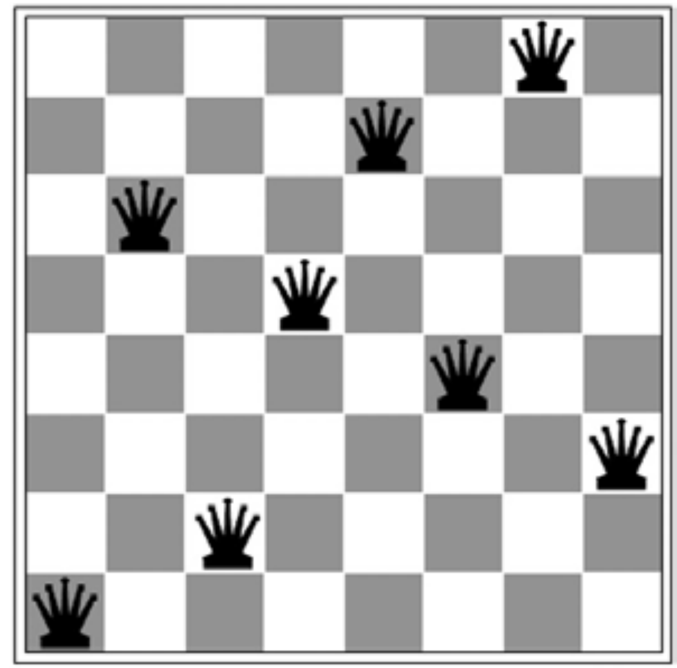
  (2) each with domain of **possible values** (often finite)

  (3) set of **constraints** on values variables can take

- **Solution:** assignment of a value to each variable such that all constraints are satisfied
- **Possible tasks:** (1) does solution exist, (2) find a solution, (3) find all solutions, (4) find *best solution w.r.t.* some metric (objective function)

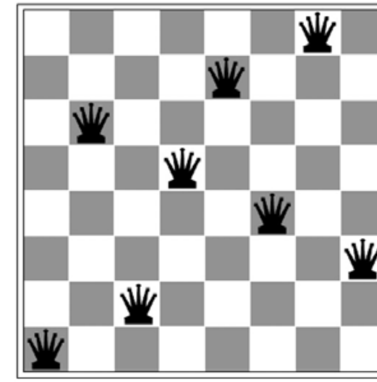
Example: 8-Queens Problem

- What are the **variables**?
- What are the variables' **domains**, i.e., sets of possible values
- What are the **constraints** between (pairs of) variables?



- Assume approach is to put one queen in each column, subject to constraints

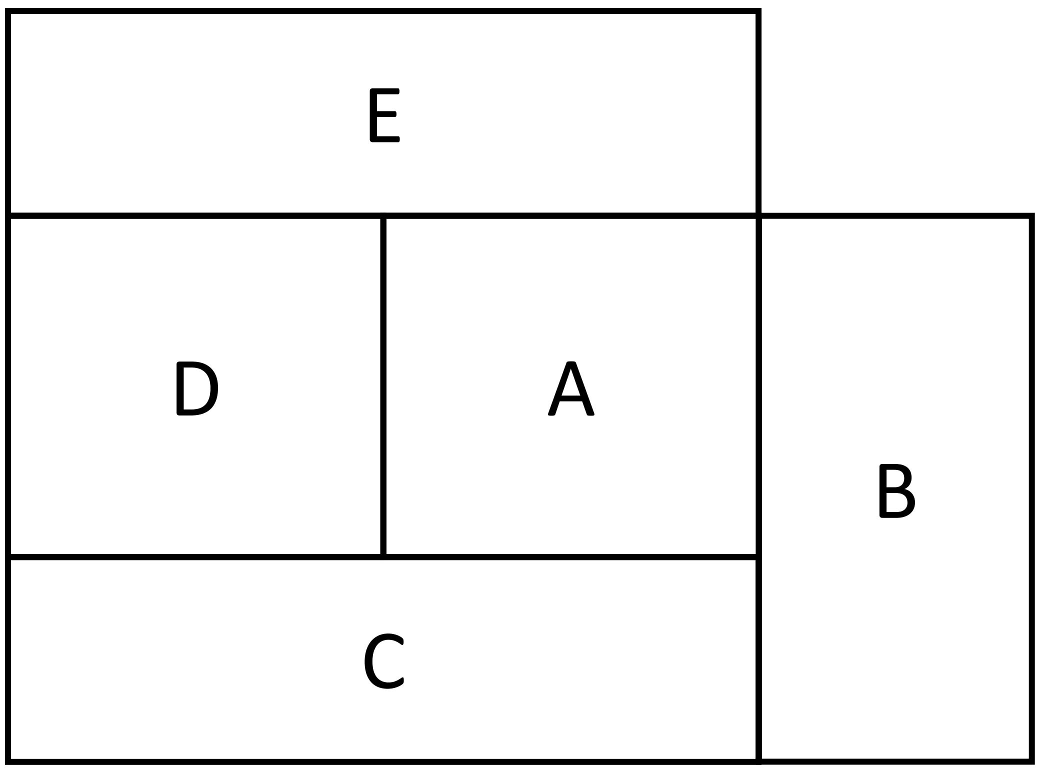
Example: 8-Queens Problem



- **Variables:** Qi, i = 1..8 where Qi is the row number of queen in column I, e.g., Q1=1, Q2=6, Q3=2, Q4=5, Q6=4, Q7=8, Q8=3
- **Variables domain:** same for each: {1,2,…,8}
- **Constraints:**
  - No queens on same row
    Qi = k ➔ Qj ≠ k for j = 1..8, j≠i
  - No queens on same diagonal
    Qi=rowi, Qj=rowj ➔|i-j|≠|rowi-rowj| for j = 1..8, j≠I
- We will **still need to search**, but constraints eliminate many possible states

Example 2: Map coloring

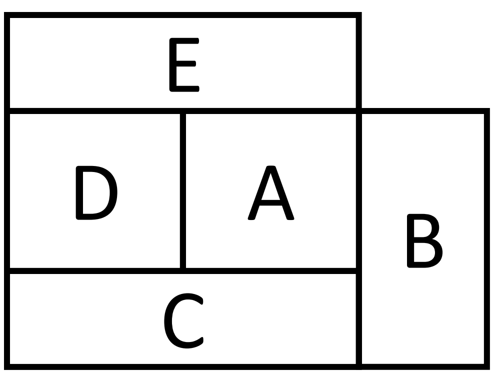
Color this map using three colors (red, green, blue) such that no two adjacent regions have the same color



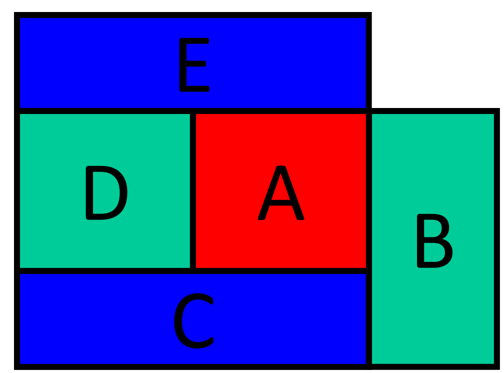
Map coloring

- **Variables:** A, B, C, D, E all of domain RGB
- **Domains:** RGB = {red, green, blue}
- **Constraints:** A≠B, A≠C, A≠E, A≠D, B≠C, C≠D, D≠E

- A solution: A=red, B=green, C=blue, D=green, E=blue



=>



Brute Force methods

- Finding a solution by a brute force search is easy
  - Just generate potential combinations and test each
  - Generate and test is a *weak method*
- Potentially very inefficient
  - With n variables where each can have one of 3 values, there are 3^n possible solutions to check
- There are ~190 countries in the world, which we can color using four colors
- 4^{190} is a big number!

Complete Prolog program to solves this

```
solve(A,B,C,D,E) :-
  color(A),
  color(B),
  color(C),
  color(D),
  color(E),
  not(A=B),
  not(A=B),
  not(B=C),
  not(A=C),
  not(C=D),
  not(A=E),
  not(C=D).

% possible colors
color(red).
color(green).
color(blue).
```

generate (color(A) … color(E)); test (not(A=B) … not(C=D))

4**190 is 2462625387274654950767440006258975862817483704404090416746768337765357610718575663213391640930307227550414249394176L

Example: Boolean SATisfiability

- Given a set of propositions, find assignment of variables to {true, false} making them all true (i.e., satisfying them)
- E.g., the 2 clauses: **$(A \lor B \lor \neg C)$, $(\neg A \lor D)$** are made true (i.e., satisfied) by assigning

  A = false, B = true, C = false, D = false
- Satisfiability known to be NP-complete

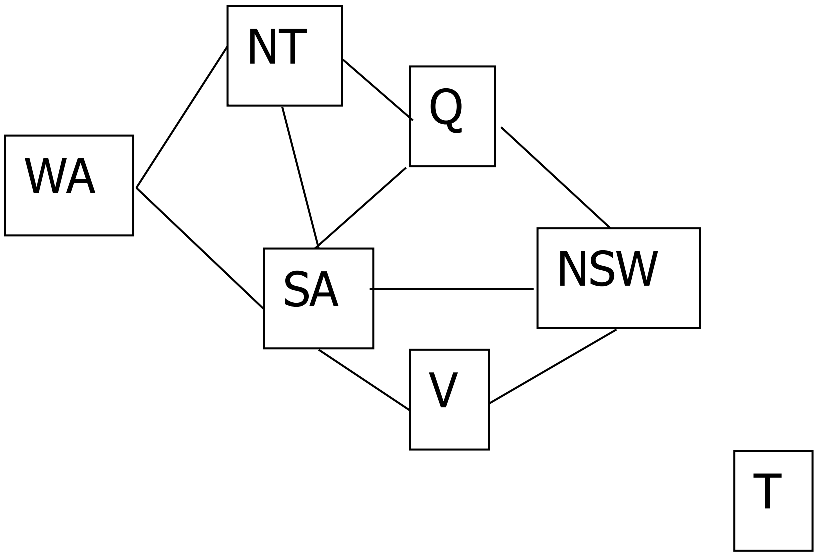
  ⇒ worst case, solving CSP problems requires exponential time
- Many real-world problems reduce to **SAT**

Real-world problems

CSPs are a good match for many practical problems that arise in the real world

- Scheduling
- Temporal reasoning
- Building design
- Planning
- Optimization/satisfaction
- Vision
- Graph layout
- Network management
- Natural language processing
- Molecular biology / genomics
- VLSI design

Running example: coloring Australia map



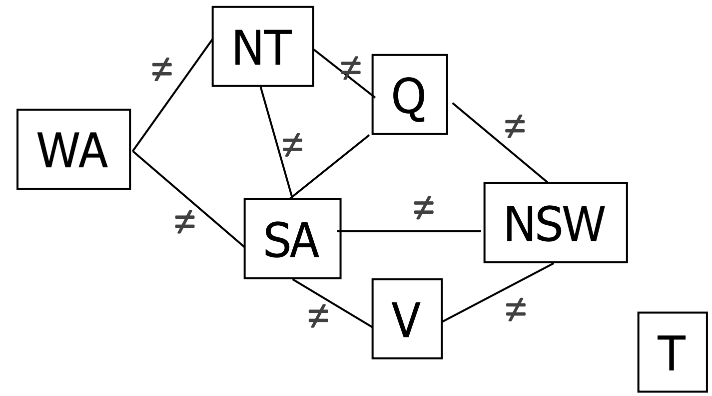
- Seven variables: {WA, NT, SA, Q, NSW, V, T}
- Each variable has same domain: **{red, green, blue}**
- No two adjacent variables can have same value:

WA≠NT, WA≠SA, NT≠SA, NT≠Q, SA≠Q, SA≠NSW, SA≠V,Q≠NSW, NSW≠V

Unary & binary constraints most common

For coloring Australia's map

- Each mainland region has a **binary** constraints with each of its neighbors that the two are different colors
- Tasmania has a **unary** constraint that its color must be R, G, or B



- Two variables are adjacent or neighbors if connected by an edge or an arc
- Possible to rewrite problems with higher-order constraints (i.e., involving >2 variables) as ones with just binary constraints

Constraint Network

- Instantiations
  - An instantiation of a subset of variables S is an assignment of a value (in its domain) to each variable in S
  - An instantiation is legal iff it violates no constraints
- A solution is a legal instantiation of all variables in the network

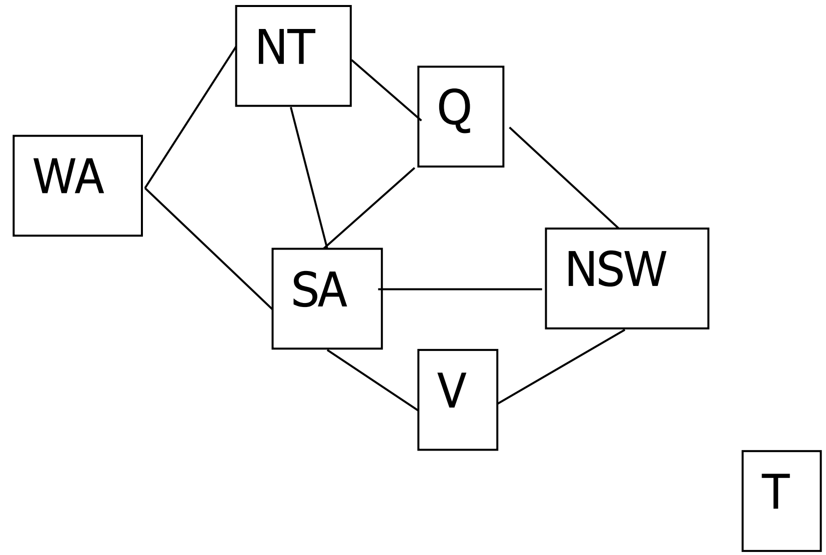
Typical tasks for CSP
Constraint Satisfaction Problem

- Possible solution related tasks:
  - Does a solution exist?
  - Find one solution
  - Find all solutions
  - Given a metric on solutions, find best one
  - Given a partial instantiation, do any of above
- Transform the constraint network into an equivalent one that's easier to solve

Binary CSP

- A **binary CSP** is one where all constraints involve two variables (or just one variable)
- Any non-binary CSP can be converted into a binary CSP by introducing additional variables
- Binary CSPs represented as a constraint graph, with a node for each variable and an arc between two nodes iff there's a constraint involving them
  - Unary constraints appear as self-referential arcs

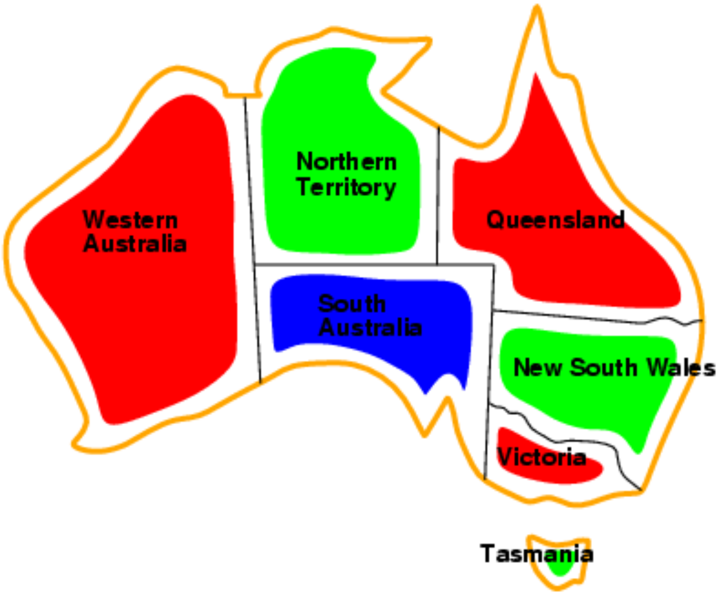
Running example: coloring Australia

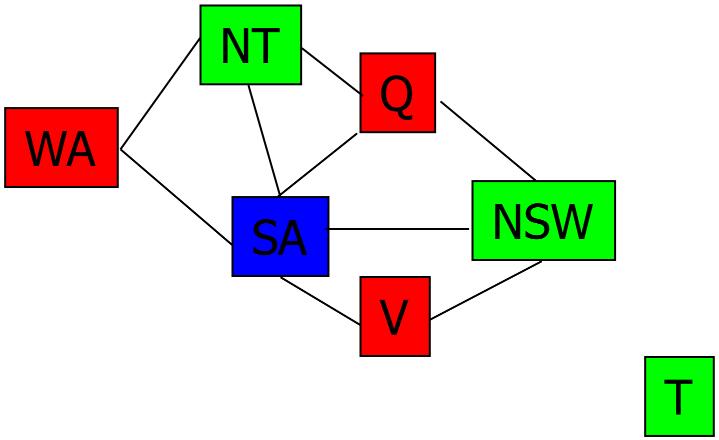


- Seven variables: {WA, NT, SA, Q, NSW, V, T}
- Each variable has same domain: {red, green, blue}
- No two adjacent variables can have same value:

WA≠NT, WA≠SA, NT≠SA, NT≠Q, SA≠Q, SA≠NSW, SA≠V,Q≠NSW, NSW≠V

A running example: coloring Australia





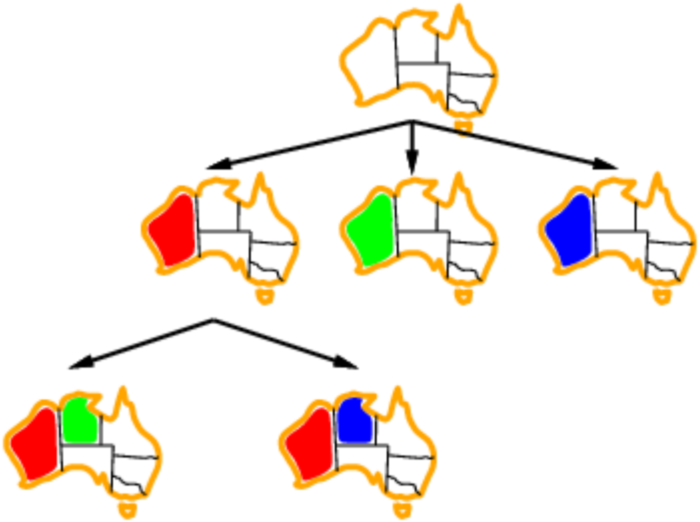
- Solutions: complete & consistent assignments
- Here is one of several solutions
- Constraints can often be expressed as relations, e.g., describe WA ≠ NT as a set of possible legal values, one for each variable

{(red,green), (red,blue), (green,red), (green,blue), (blue,red),(blue,green)}

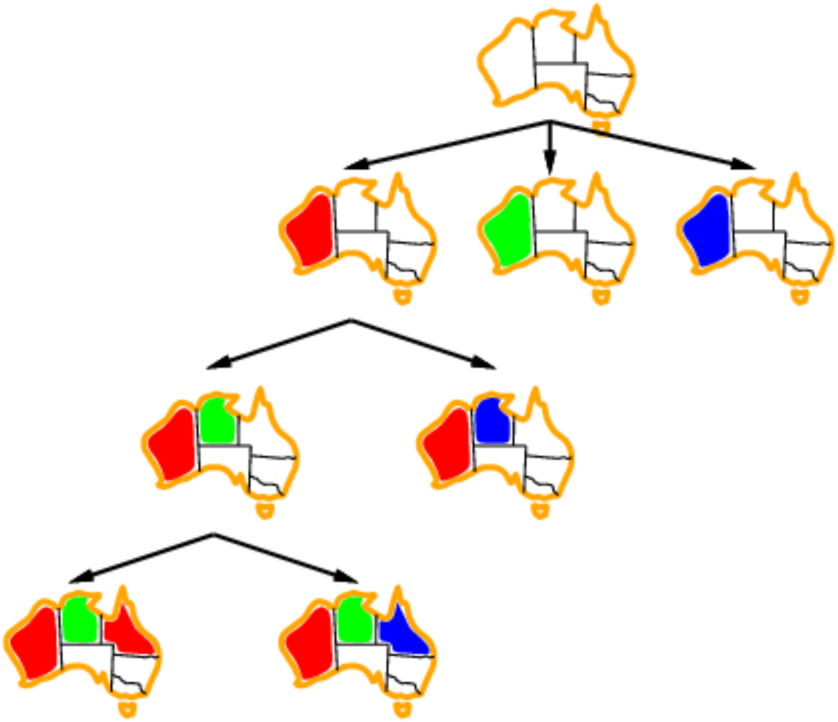
Simple Backtracking example

Backtracking example

Backtracking example



Backtracking example



Basic backtracking algorithm

CSP-backtracking(PartialAssignment A)

- If A is complete then return a
- X ← select an unassigned variable
- D ← select an ordering for the domain of X
- For each value v in D do

  If v consistent with a then

  - Add (X=v) to A
  - result ← CSP-BACKTRACKING(A)
  - If result ≠ failure then return result
  - Remove (X= v) from A
- Return failure

Start with CSP-BACKTRACKING({})

Note: depth first search can solve n-queens problems for n ~ 25

Problems with Backtracking

- Thrashing: keep repeating the same failed variable assignments
- Things that can help avoid this:
  - Consistency checking
  - Intelligent backtracking schemes
- Inefficiency: can explore areas of the search space that aren’t likely to succeed
  - Variable ordering can help

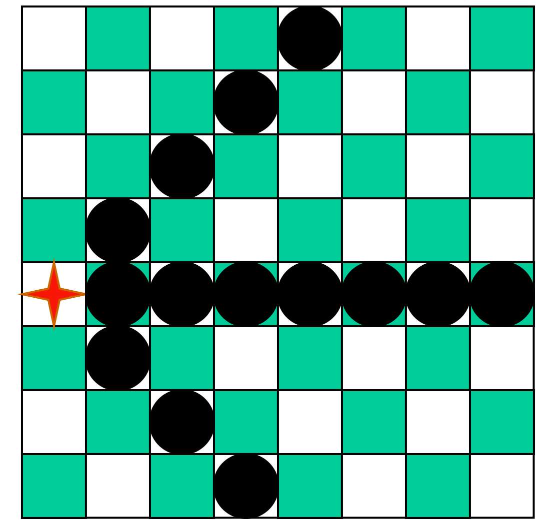
Improving backtracking efficiency

Here are some standard techniques to improve the efficiency of backtracking

- Can we detect inevitable failure early?
- Which variable should be assigned next?
- In what order should its values be tried?

Forward Checking

After variable X is assigned to value v, examine each unassigned variable Y connected to X by a constraint and delete values from Y's domain inconsistent with v

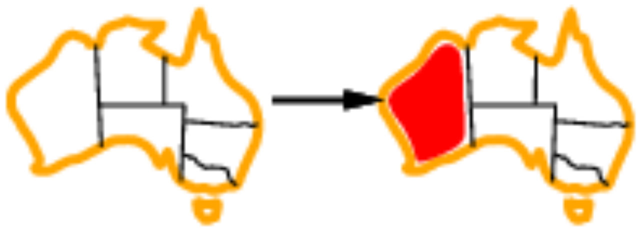


Using forward checking and backward checking roughly doubles the size of N-queens problems that can be practically solved

Forward checking

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values

Forward checking

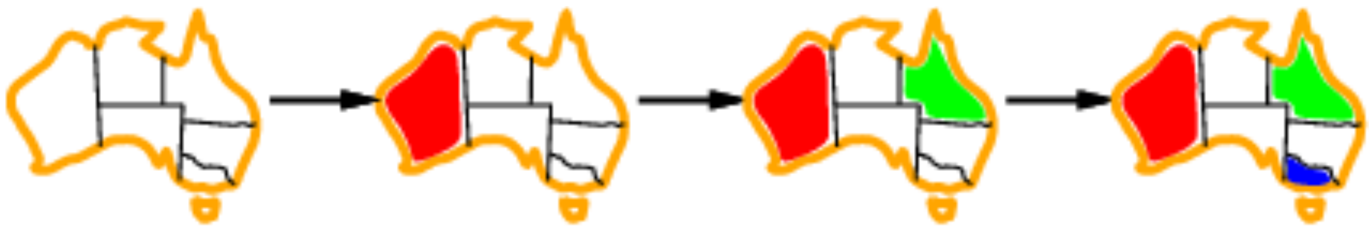


- Assign WA to red
- Propagate constraints by eliminating red from WA's neighbors' domains (NT & SA)

Forward checking

- Assign Q to **green** and eliminate **green** from Q's neighbors' domains (NT, SA,, NSW)

Forward checking

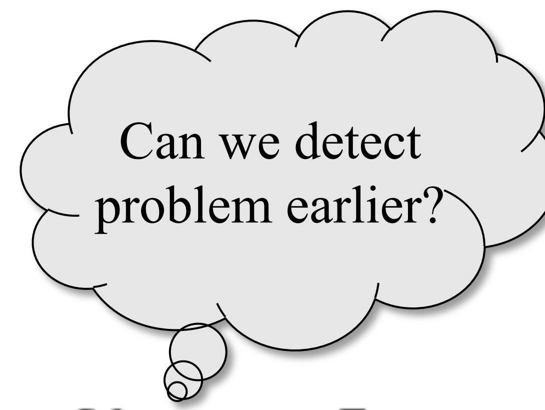


- Assign SA to **blue** and eliminate **blue** from SA's neighbors' domains
- The empty domain means failure

SA (South Australia) domain is empty!

Constraint propagation

- Forward checking propagates info. from assigned to unassigned variables, but doesn't provide **early detection for all failures**
- NT and SA cannot both be blue!

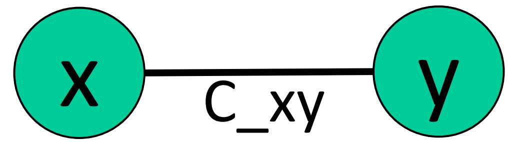


Definition: Arc consistency

- A constraint C_xy is arc consistent w.r.t. x if for each value v of x there is an allowed value of y
- Similarly define C_xy as arc consistent w.r.t. y
- Binary CSP is arc consistent iff **every** constraint C_xy is arc consistent w.r.t. x as well as y
- When a CSP is not arc consistent, we can make it arc consistent by using the AC3 algorithm
  - Also called "enforcing arc consistency"

Arc Consistency Example 1

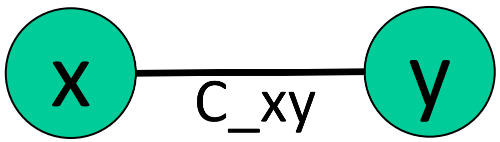
- **Initial domains**
  - D_x = {1, 2, 3}
  - D_y = {3, 4, 5, 6}



- **Constraint**
  - Note: for finite domains, we can represent a constraint as a set of legal value pairs
  - C_xy = {(1,3), (1,5), (3,3), (3,6)}
- C_xy isn't arc consistent w.r.t. **initial domains** of x or y
- **Enforcing arc consistency**, we get **reduced domains** for x and y:
  - D'_x = {1, 3}   *% x can't be 2*
  - D'_y={3, 5, 6}   *% y can't be 4*

Arc Consistency Example 2

- **Initial domains**
  - D_x = {1, 2, 3}
  - D_y = {1, 2, 3}



- **Constraint**: X must be less than Y
  - C_xy = `lambda v1,v2: v1 < v2`
- C_xy not arc consistent w.r.t. x or y; enforcing arc consistency, we get **reduced domains**:
  - D'_x = {1, 2}
  - D’_y = {2, 3}

Aside: Python lambda expressions

Previous slide expressed constraint between two variables as an *anonymous* Python function of two arguments

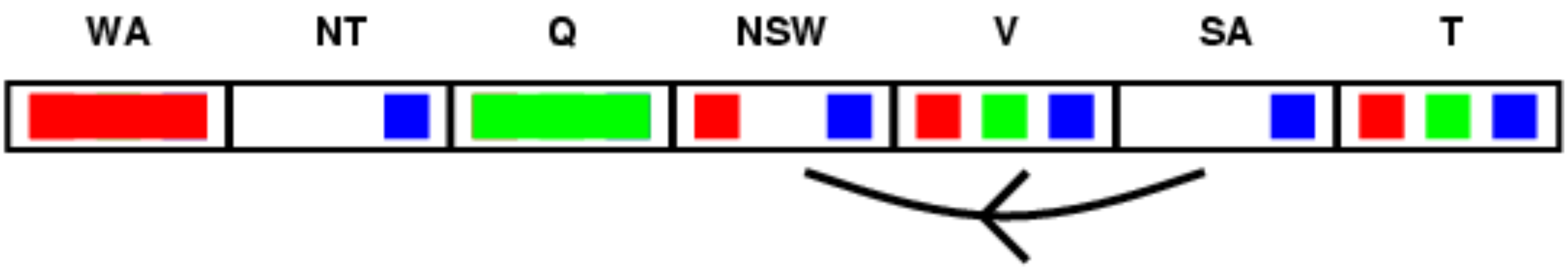
```
lambda v1,v2: v1 < v2
```

```
>>> f = lambda v1,v2: v1 < v2
>>> f
<function <lambda> at 0x10fcf21e0>
>>> f(100,200)
True
>>> f(200,100)
False
```

Python uses lambda after Alonzo Church's [lambda calculus] from the 1930s

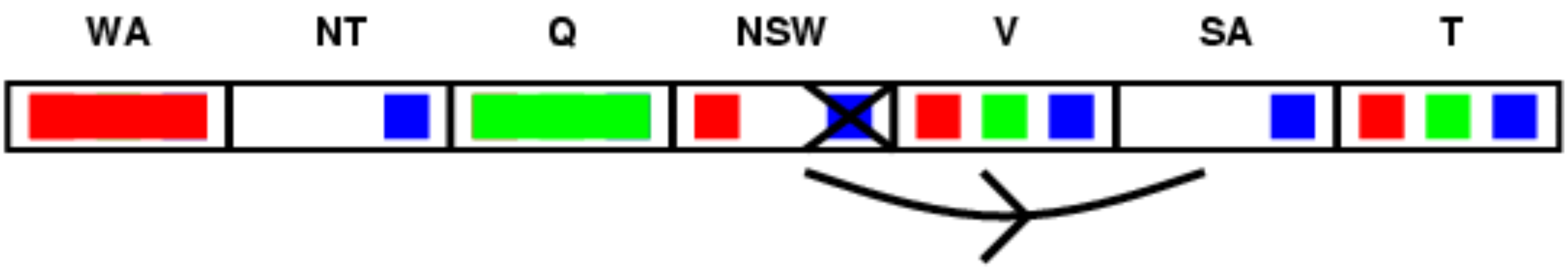
Arc consistency

- Simplest form of propagation makes each arc consistent
- X →Y is consistent iff for every value x_i of X there is some allowed value y_j in Y



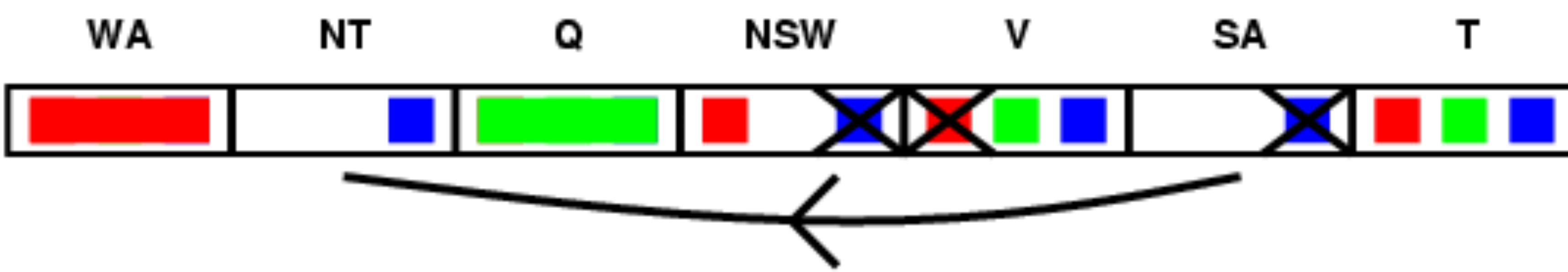
Arc consistency

- Simplest form of propagation makes each arc consistent
- X →Y is consistent iff for every value x_i of X there is some allowed value y_j in Y



Arc consistency

- Arc consistency detects failure earlier than simple forward checking
- WA=red and Q=green is quickly recognized as a **deadend**, i.e. an impossible partial instantiation
- The arc consistency algorithm can be run as a preprocessor or after each assignment



General CP for Binary Constraints

Algorithm AC3

contradiction ← false

Q ← stack of all variables

while Q is not empty and not contradiction do

  X ← UNSTACK(Q)

  For every variable Y adjacent to X do

    If REMOVE-ARC-INCONSISTENCIES(X,Y)

      If domain(Y) is non-empty then STACK(Y,Q)

      else return false

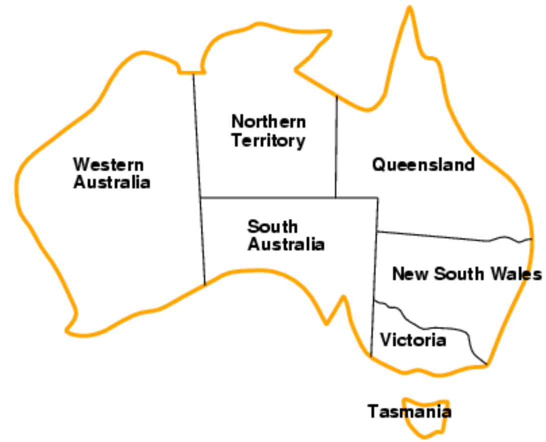
Complexity of AC3

- e = number of constraints (edges)
- d = number of values per variable
- Each variable inserted in queue up to d times
- REMOVE-ARC-INCONSISTENCY takes $O(d^2)$ time
- CP takes $O(ed^3)$ time

Improving backtracking efficiency

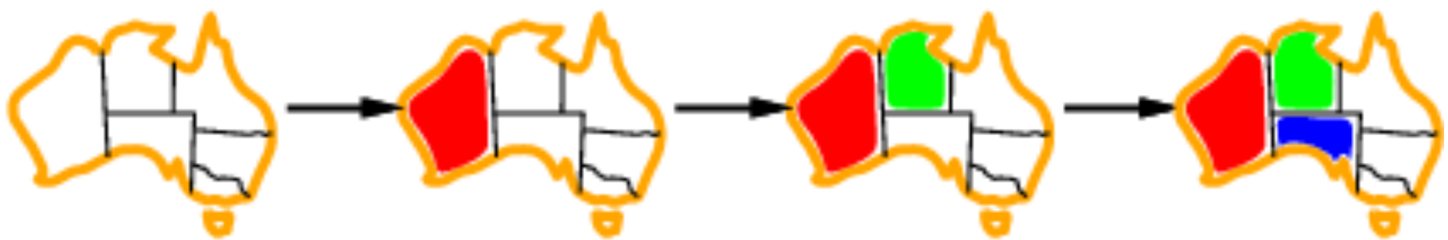
- Some standard techniques to improve the efficiency of backtracking
  - Can we detect inevitable failure early?
  - Which variable should be assigned next?
  - In what order should its values be tried?
- Combining constraint propagation with these heuristics makes **1000-queen puzzles feasible**

H1: pick var with fewest values



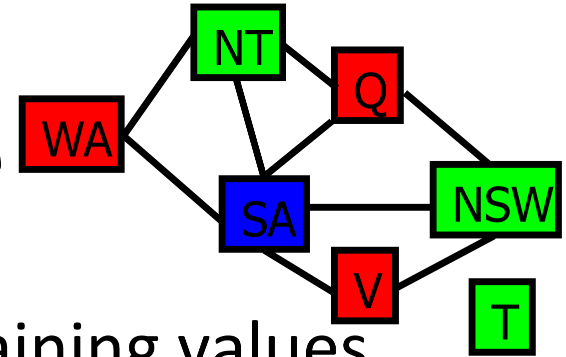
- AKA **most constrained variable**:

  choose the variable with the fewest legal values

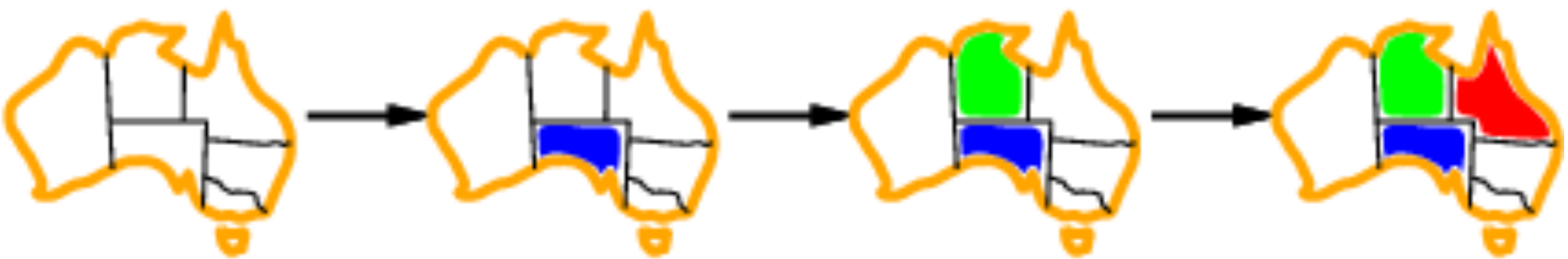


- a.k.a. minimum remaining values (MRV) heuristic
- After assigning value to WA, both NT and SA have only two values in their domains
  – choose one of them rather than Q, NSW, V or T

H2: most constraining variable



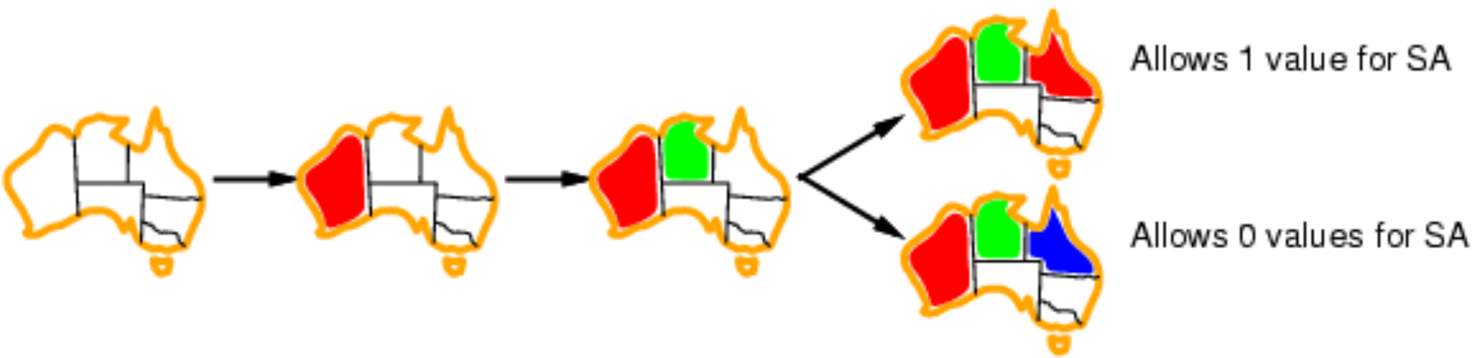
- Tie-breaker after H1, minimum remaining values
- Choose variable involved in largest # of constraints on remaining variables



- After assigning SA to be blue, WA, NT, Q, NSW and V all have just two values left.
- WA and V have only one constraint on remaining variables and T none, so choose one of NT, Q & NSW

H3: Least constraining value

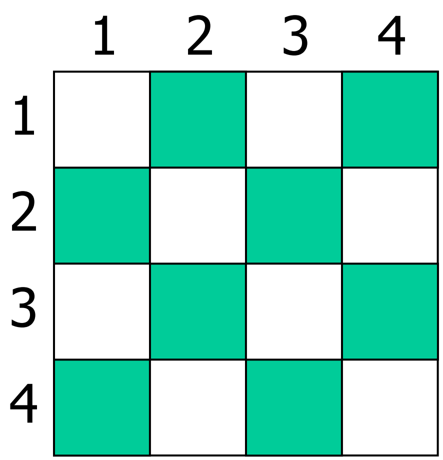
- Given variable, try value that's least constraining on its neighbors:
  - the one that rules out the fewest values in the remaining variables

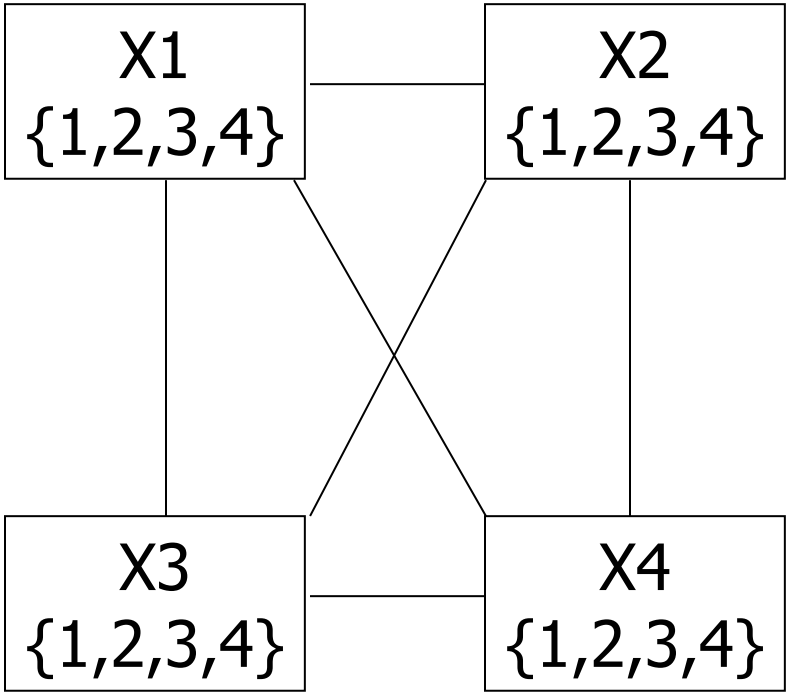


- Combining these heuristics makes 1000 queens feasible
- What's an intuitive explanation for this?

Is AC3 Alone Sufficient?

Consider the four queens problem

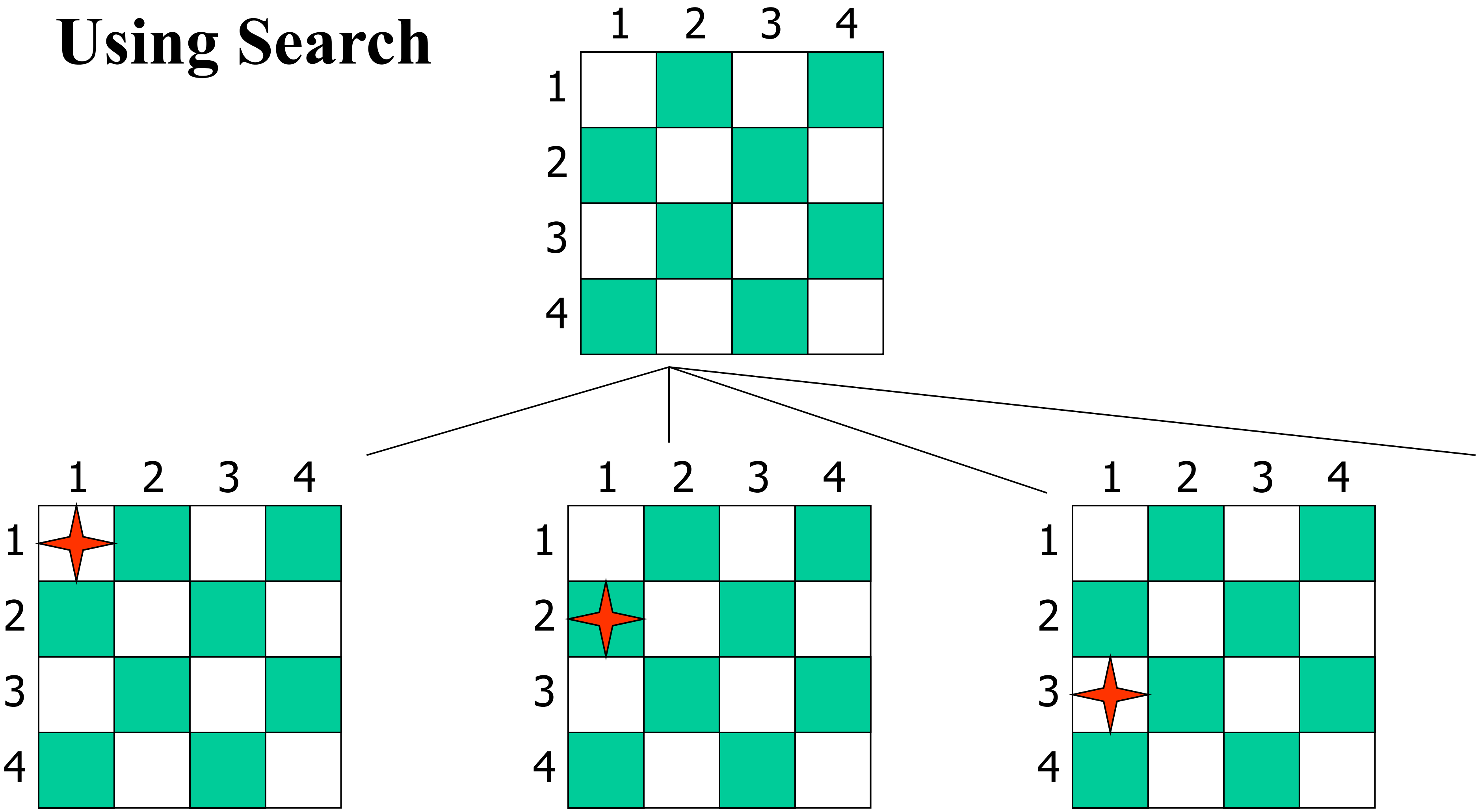




Solving a CSP still requires search

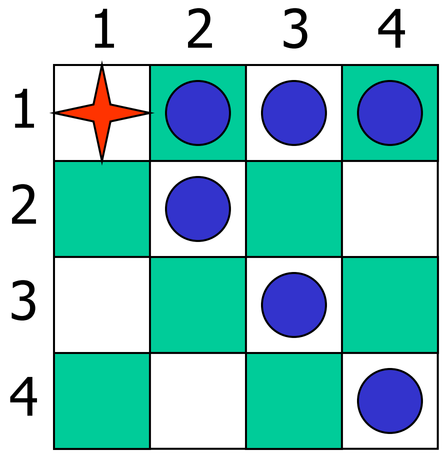
- Search:
  - can find good solutions, but must examine non-solutions along the way
- Constraint Propagation:
  - can rule out non-solutions, but this is not the same as finding solutions
- Interweave constraint propagation & search:
  - perform constraint propagation at each search step

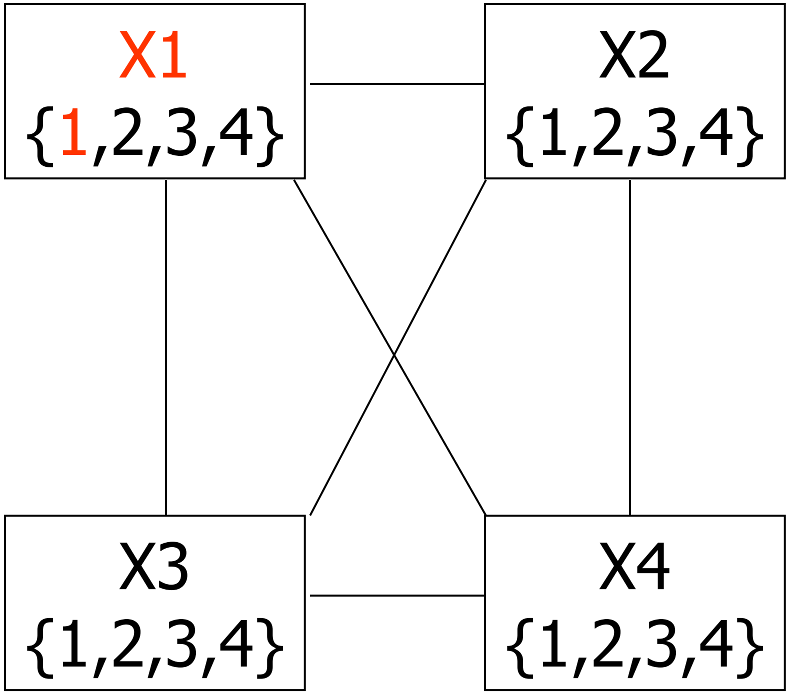
Using Search



Using CSP

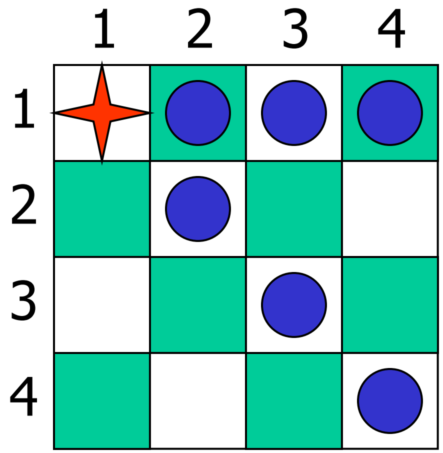
4-Queens Problem

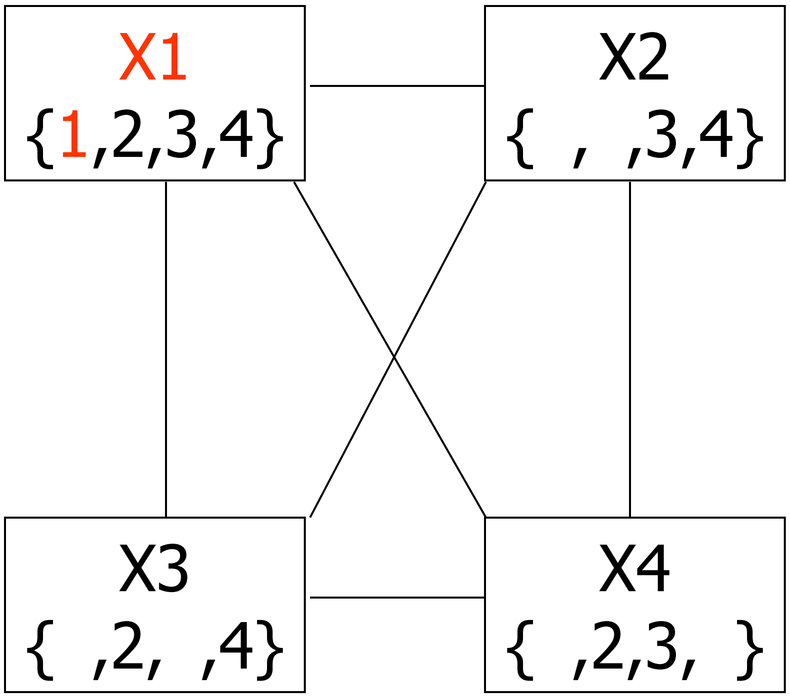




Try assigning X1=1

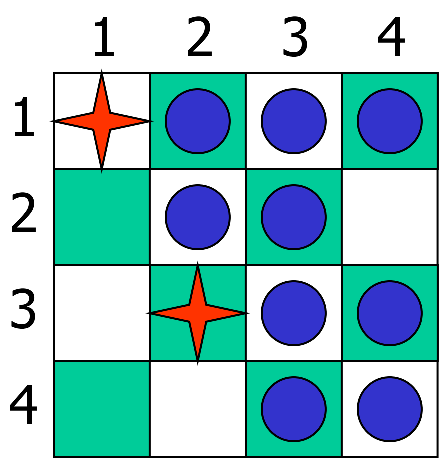
4-Queens Problem

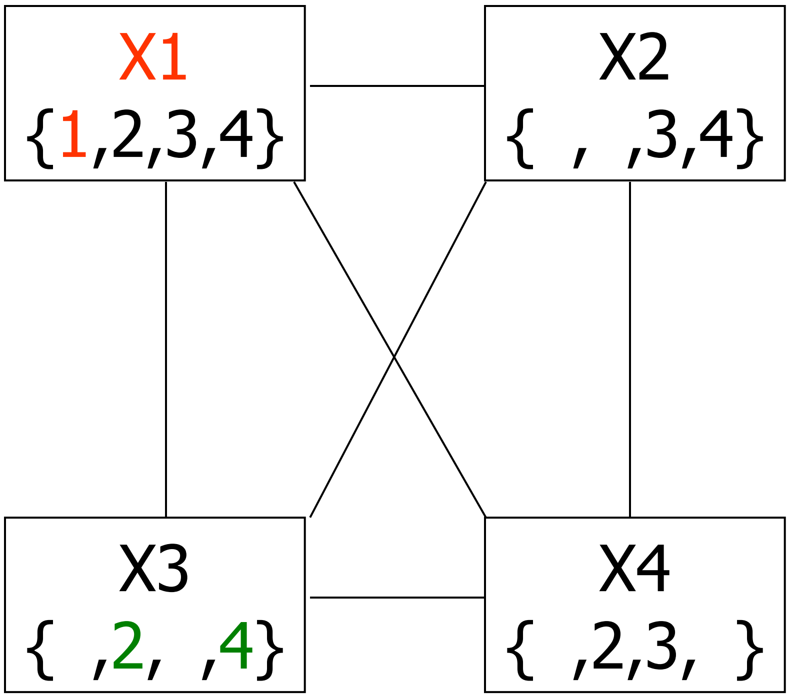




X1=1 eliminates { X2=1,2, X3=1,3, X4=1,4 }

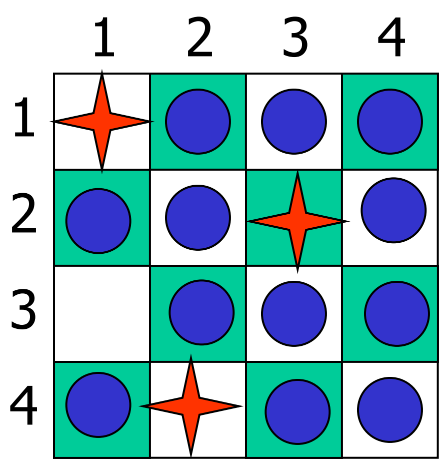
4-Queens Problem

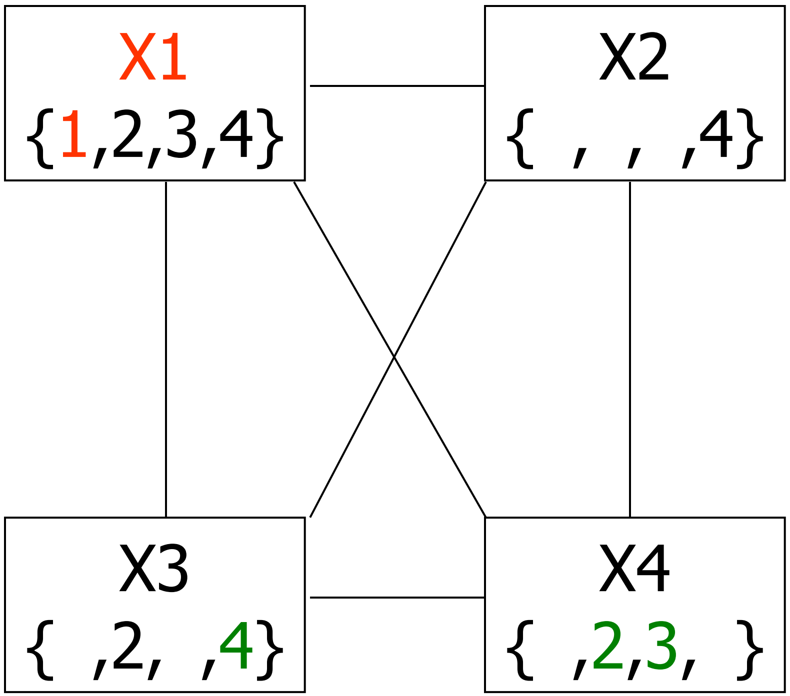




X2=3 eliminates { X3=2, X3=3, X3=4 }
⇒ inconsistent!

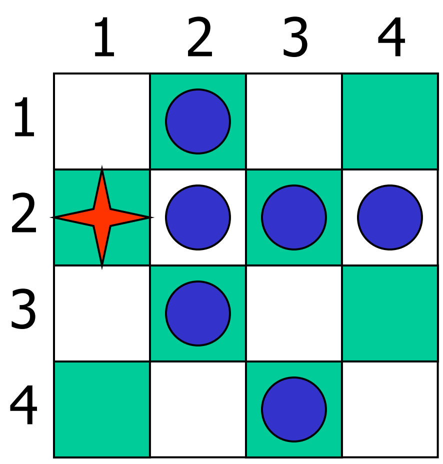
4-Queens Problem

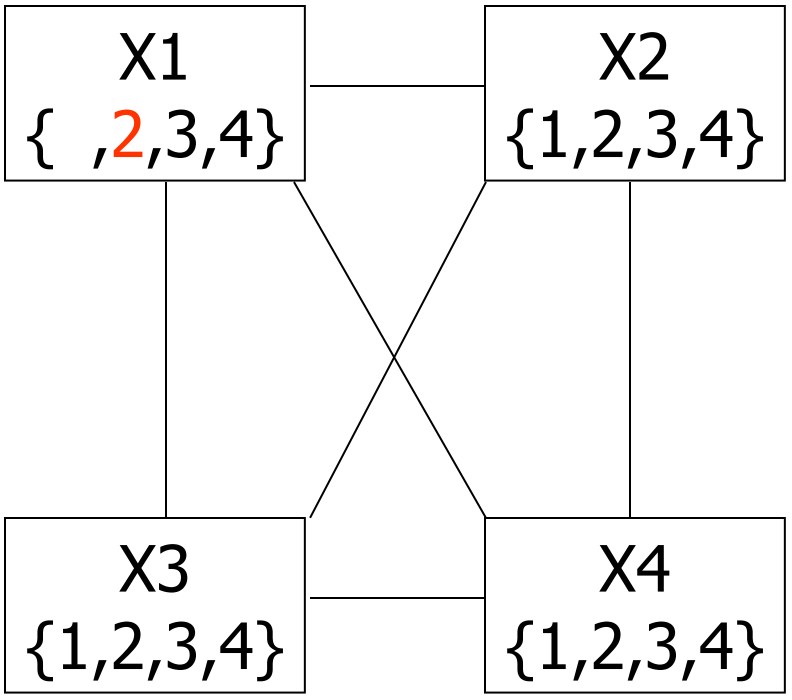




X2=4 ⇒ X3=2, which eliminates { X4=2, X4=3}
⇒ inconsistent!

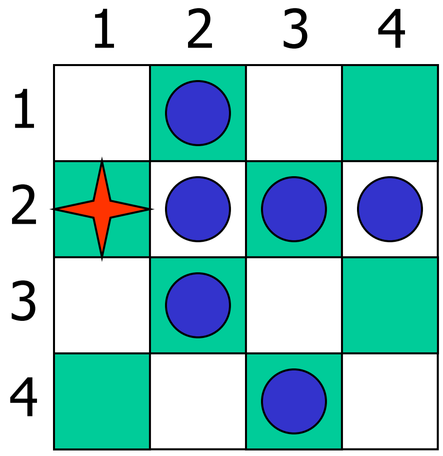
4-Queens Problem

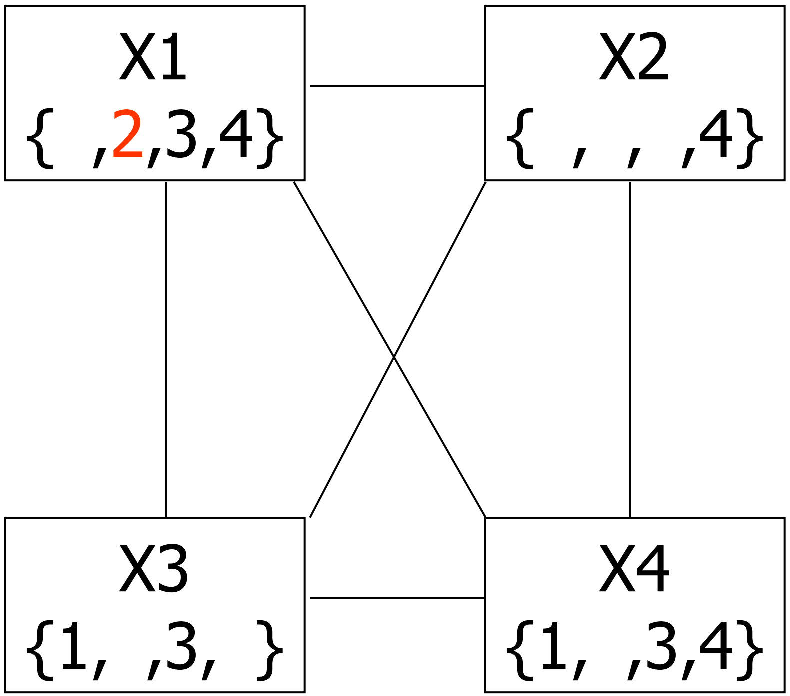




X1 can't be 1, let's try 2

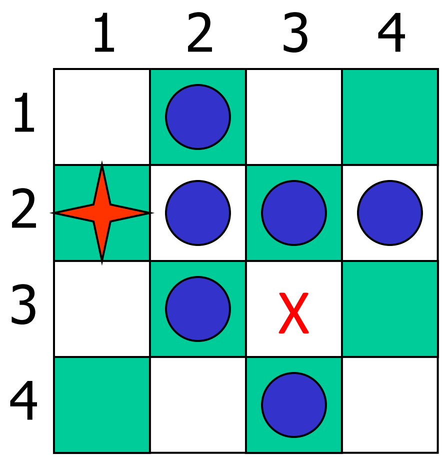
4-Queens Problem

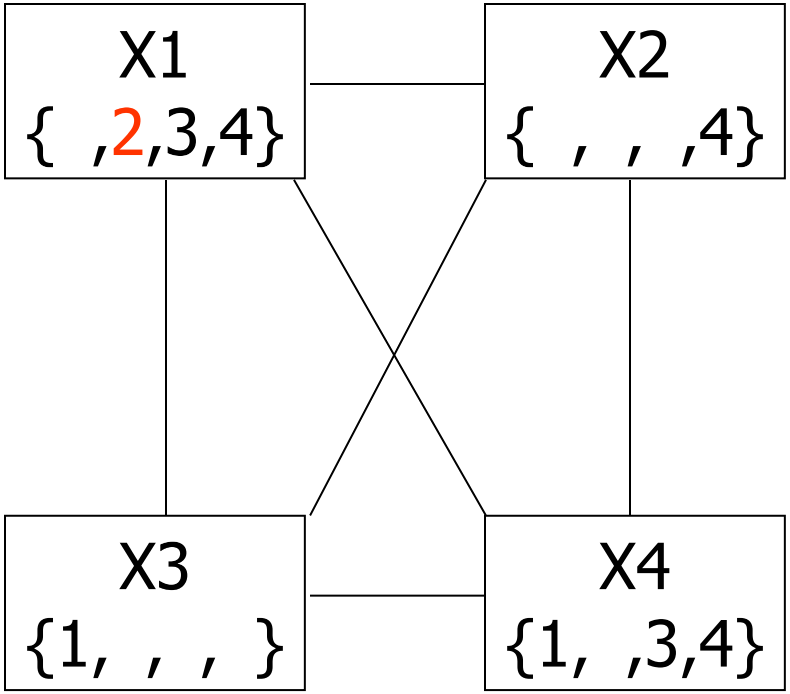




Can we eliminate any other values?

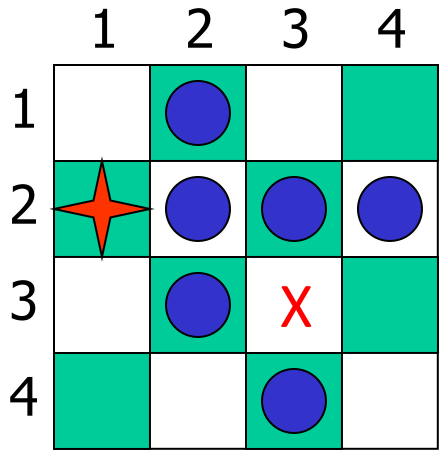
4-Queens Problem

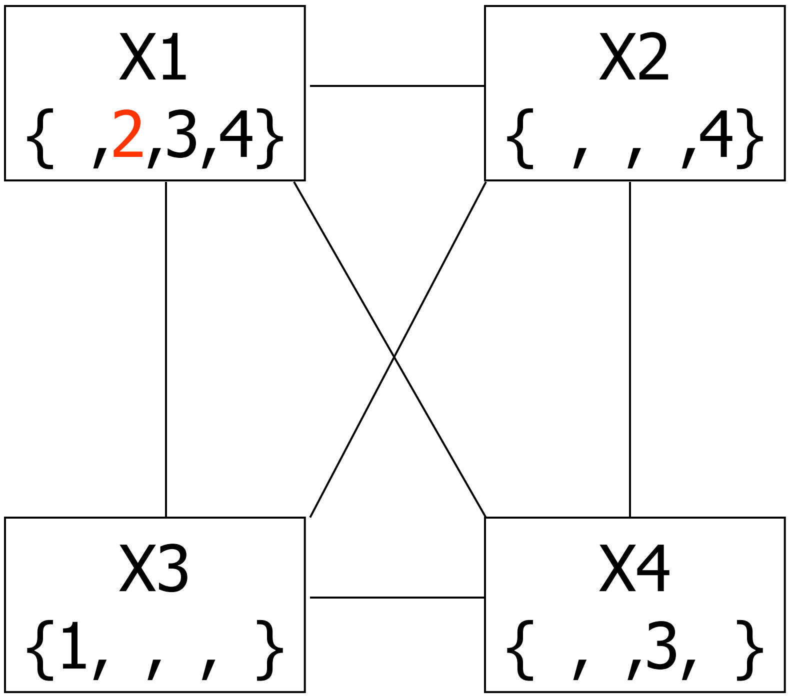




Yes! We know X2=4, so X3 can't be 3

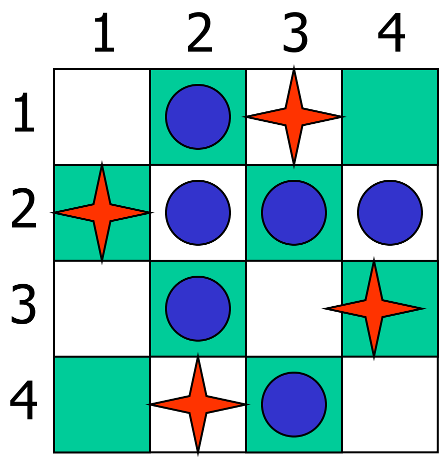
4-Queens Problem

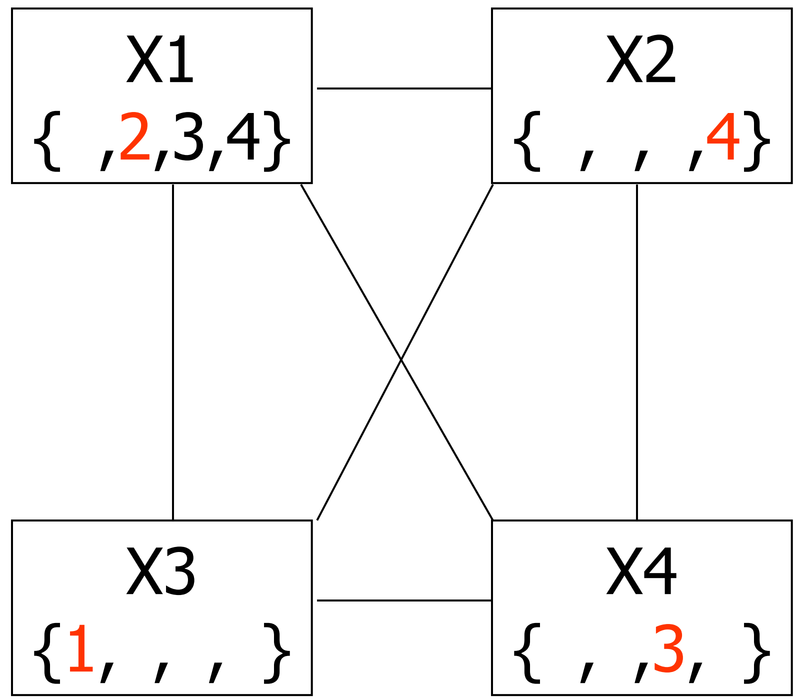




Arc constancy eliminates x3=3 because it's not consistent with X2's remaining values

4-Queens Problem





There is only one solution with X1=2

Sudoku

- Digit placement puzzle on 9x9 grid with unique answer
- Given an initial partially filled grid, fill remaining squares with a digit between 1 and 9
- Each column, row, and nine 3 × 3 sub-grids must contain all nine digits

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

- Some initial configurations are easy to solve and others very difficult

Sudoku Example

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

initial problem

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

a solution

How can we set this up as a CSP?

```python
def sudoku(initValue):
    p = Problem()
    for i in range(1, 10) : # Variable for each cell: 11,12,13...21,22,...98,99
        p.addVariables(range(i*10+1, i*10+10), range(1, 10))
    for i in range(1, 10) :  # Each row has different values
        p.addConstraint(AllDifferentConstraint(), range(i*10+1, i*10+10))
    for i in range(1, 10) : # Each column has different values
        p.addConstraint(AllDifferentConstraint(), range(10+i, 100+i, 10))
    # Each 3x3 box has different values
    p.addConstraint(AllDifferentConstraint(), [11,12,13,21,22,23,31,32,33])
    p.addConstraint(AllDifferentConstraint(), [41,42,43,51,52,53,61,62,63])
    p.addConstraint(AllDifferentConstraint(), [71,72,73,81,82,83,91,92,93])

    p.addConstraint(AllDifferentConstraint(), [14,15,16,24,25,26,34,35,36])
    p.addConstraint(AllDifferentConstraint(), [44,45,46,54,55,56,64,65,66])
    p.addConstraint(AllDifferentConstraint(), [74,75,76,84,85,86,94,95,96])

    p.addConstraint(AllDifferentConstraint(), [17,18,19,27,28,29,37,38,39])
    p.addConstraint(AllDifferentConstraint(), [47,48,49,57,58,59,67,68,69])
    p.addConstraint(AllDifferentConstraint(), [77,78,79,87,88,89,97,98,99])
    for i in range(1, 10) : # unary constraints for cells with initial non-zero values
        for j in range(1, 10):
            value = initValue[i-1][j-1]
            if value: p.addConstraint(lambda var, val=value: var == val, (i*10+j,))
    return p.getSolution() # find and return a solution
```

```python
# Sample problems
easy = [
  [0,9,0,7,0,0,8,6,0],
  [0,3,1,0,0,5,0,2,0],
  [8,0,6,0,0,0,0,0,0],
  [0,0,7,0,5,0,0,0,6],
  [0,0,0,3,0,7,0,0,0],
  [5,0,0,0,1,0,7,0,0],
  [0,0,0,0,0,0,1,0,9],
  [0,2,0,6,0,0,0,5,0],
  [0,5,4,0,0,8,0,7,0]]

hard = [
  [0,0,3,0,0,0,4,0,0],
  [0,0,0,0,7,0,0,0,0],
  [5,0,0,4,0,6,0,0,2],
  [0,0,4,0,0,0,8,0,0],
  [0,9,0,0,3,0,0,2,0],
  [0,0,7,0,0,0,5,0,0],
  [6,0,0,5,0,2,0,0,1],
  [0,0,0,0,9,0,0,0,0],
  [0,0,9,0,0,0,3,0,0]]

very_hard = [
  [0,0,0,0,0,0,0,0,0],
  [0,0,9,0,6,0,3,0,0],
  [0,7,0,3,0,4,0,9,0],
  [0,0,7,2,0,8,6,0,0],
  [0,4,0,0,0,0,0,7,0],
  [0,0,2,1,0,6,5,0,0],
  [0,1,0,9,0,5,0,4,0],
  [0,0,8,0,2,0,7,0,0],
  [0,0,0,0,0,0,0,0,0]]
```

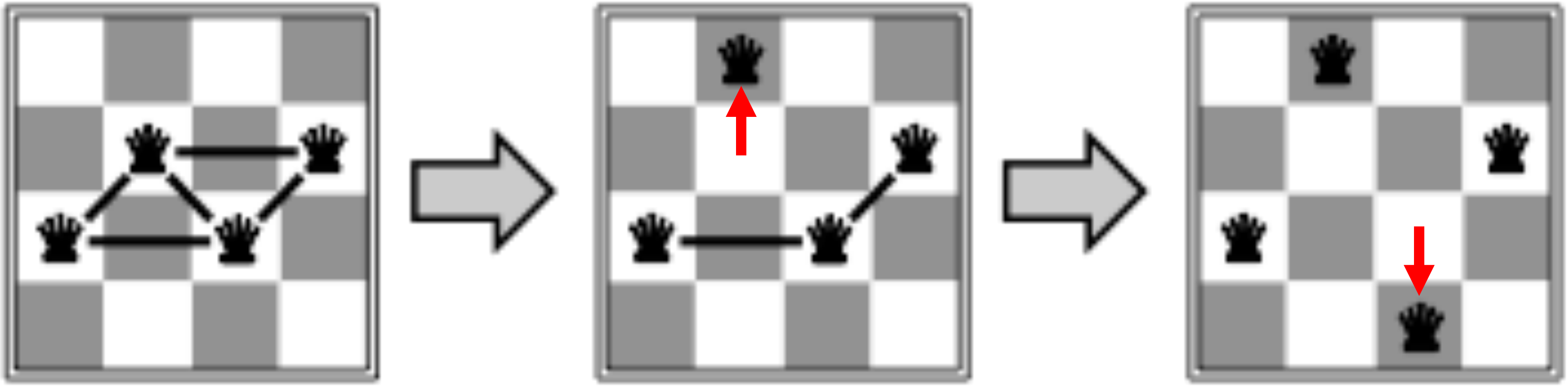
Local search for constraint problems

- Remember local search?
- There’s a version of local search for CSP problems
- Basic idea:
  - generate a random “solution”
  - Use metric “number of violated constraints”
  - Modifying solution by reassigning one variable at a time to decrease metric until solution found or no modification improves it
- Has all features and problems of local search

  like….?

Min Conflict Example

- **States:** 4 Queens, 1 per column
- **Operators:** Move a queen in its column
- **Goal test:** No attacks
- **Evaluation metric:** Total number of attacks



How many conflicts does each state have?

Basic Local Search Algorithm

Assign one domain value d_i to each variable v_i

while no solution & not stuck & not timed out:

    bestCost $\leftarrow \infty$;   bestList $\leftarrow$ [ ];

    for each variable v_i | Cost(Value(v_i)) > 0

        for each domain value d_i of v_i

            if Cost(d_i) < bestCost

                bestCost $\leftarrow$ Cost(d_i); bestList $\leftarrow$ [d_i];

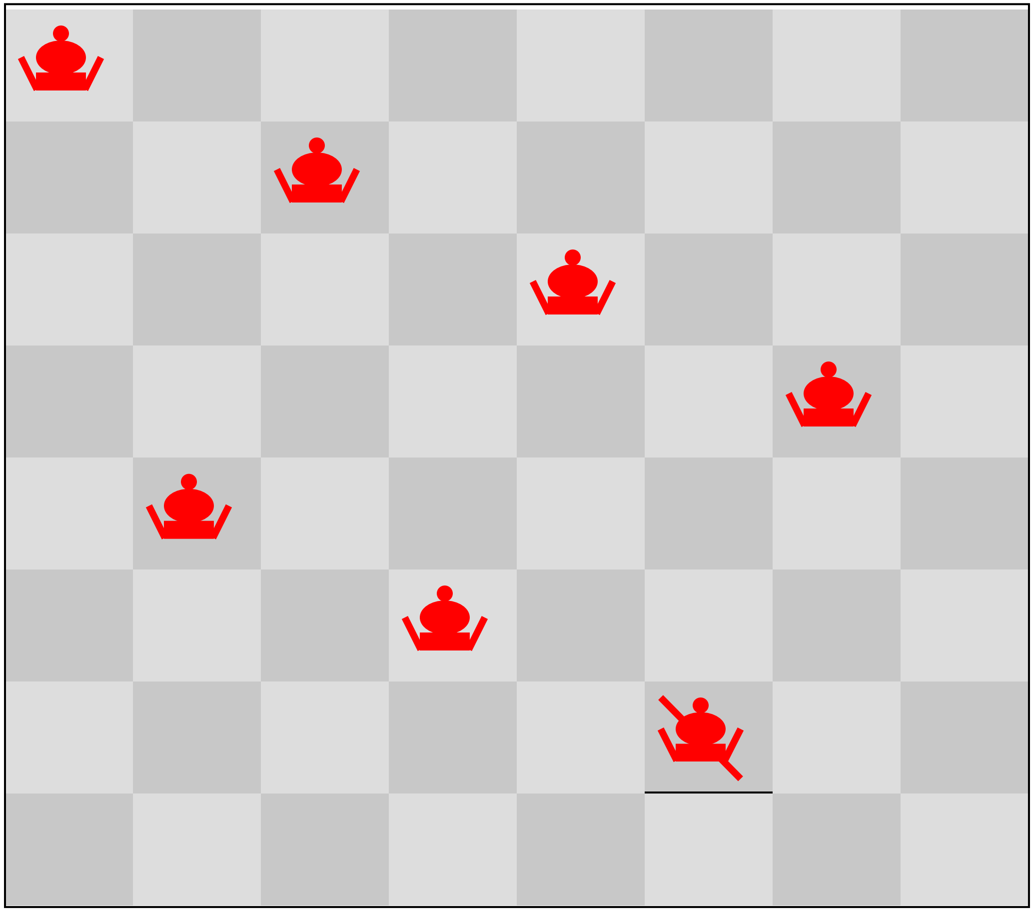
            else if Cost(d_i) = bestCost

                bestList $\leftarrow$ bestList $\cup$ d_i

    Take a randomly selected move from bestList

Eight Queens using Backtracking

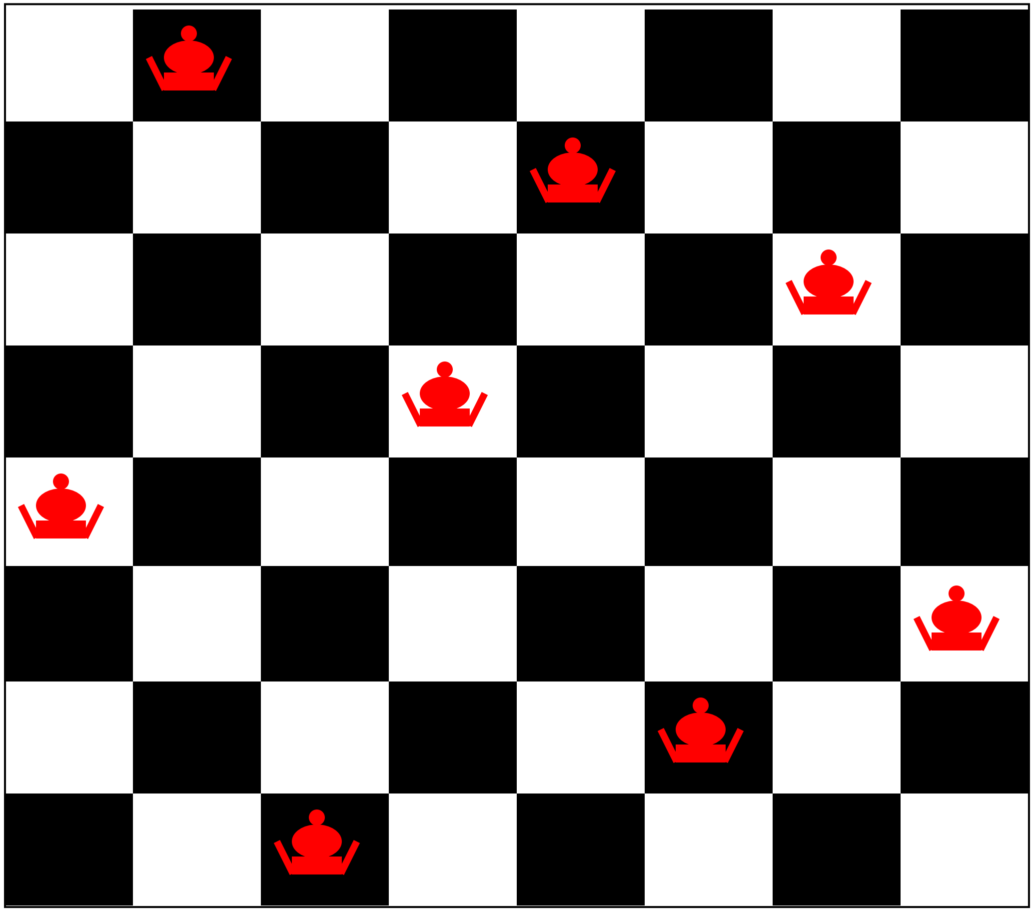
Undo move
for Queen 7
and so on...



Note: in this example we put one queen in each row, not column

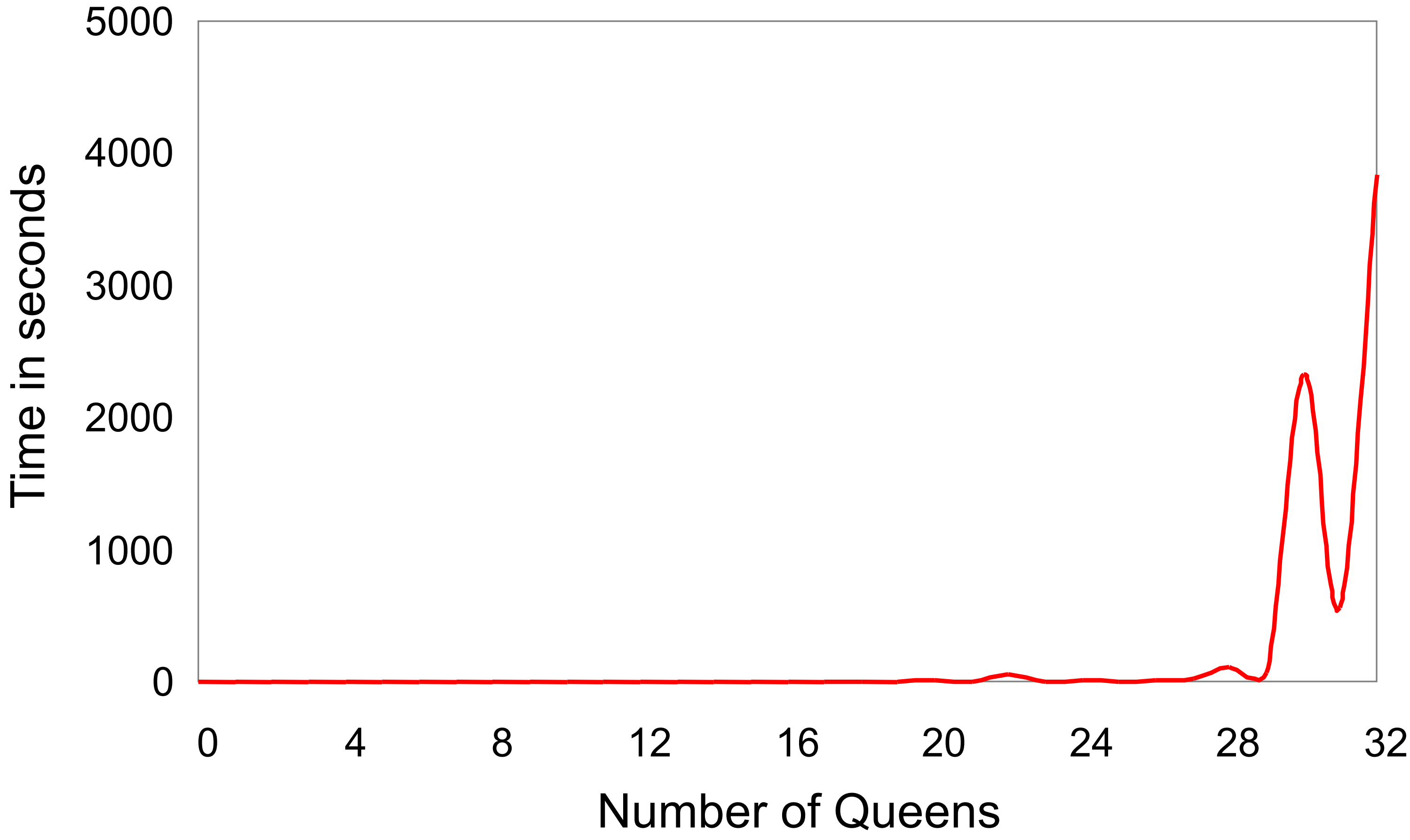
Eight Queens using Local Search

Answer Found

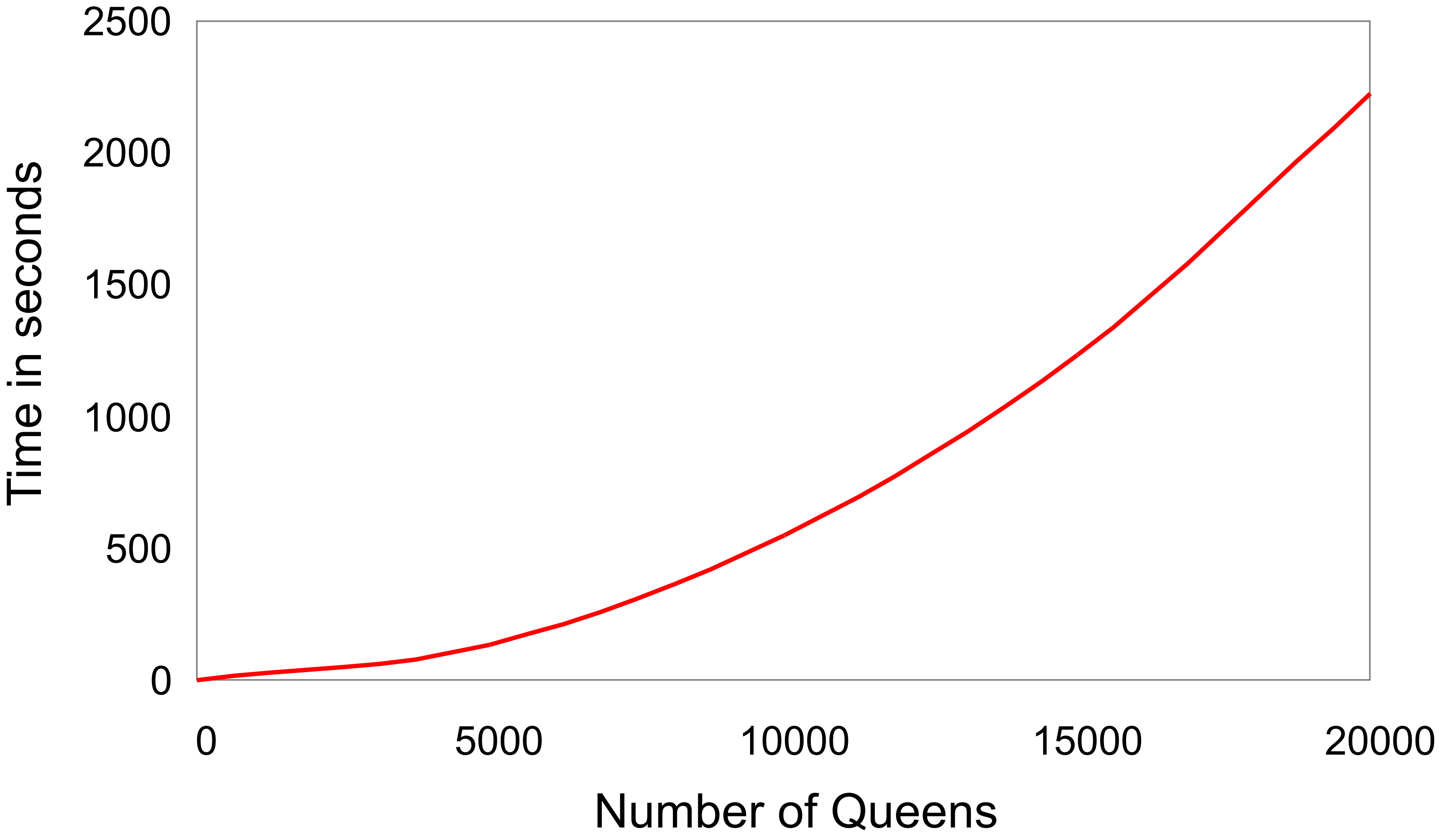


Note: in this example we put one queen in each row, not column

Backtracking Performance



Local Search Performance

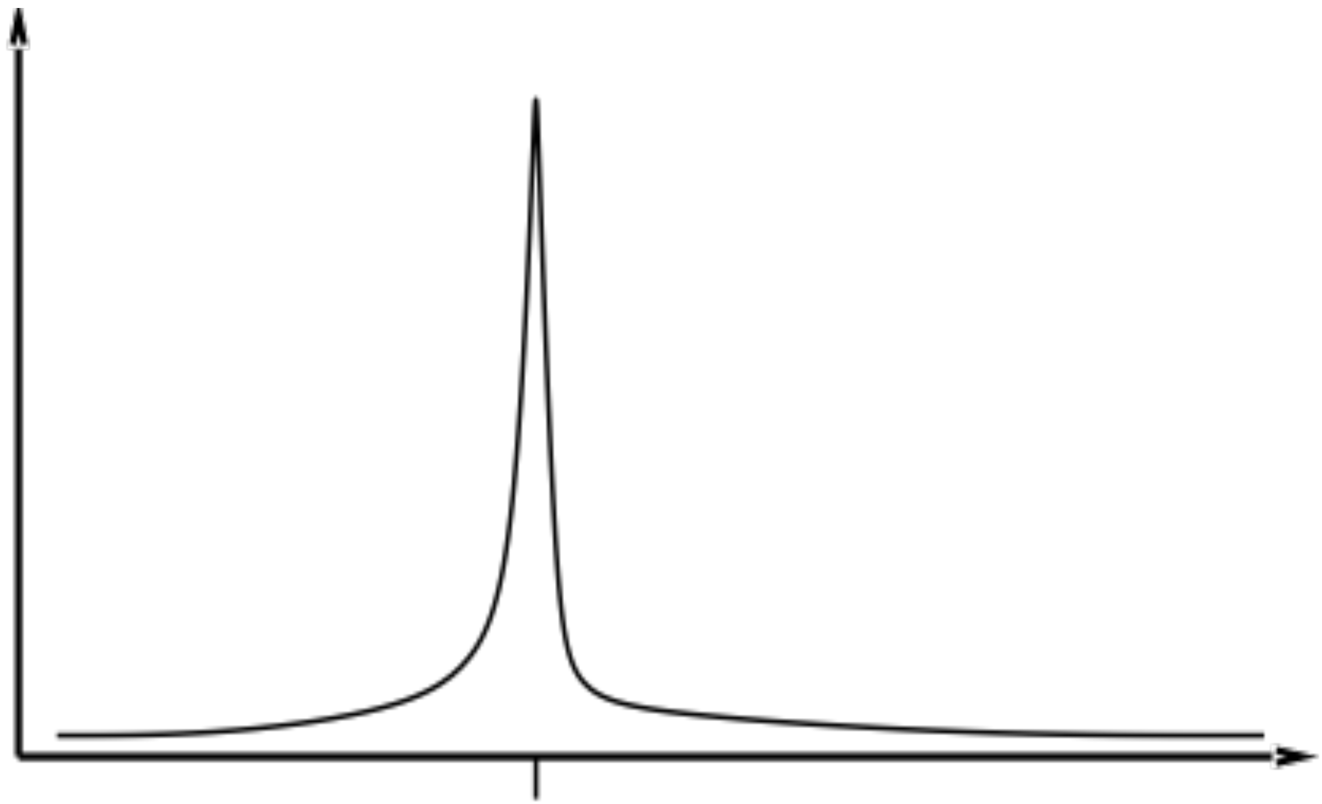


Min Conflict Performance

- Performance depends on quality and informativeness of initial assignment; inversely related to distance to solution
- Min Conflict often has astounding performance
- Can solve arbitrary size (i.e., millions) N-Queens problems in constant time
- Appears to hold for arbitrary CSPs with the caveat…

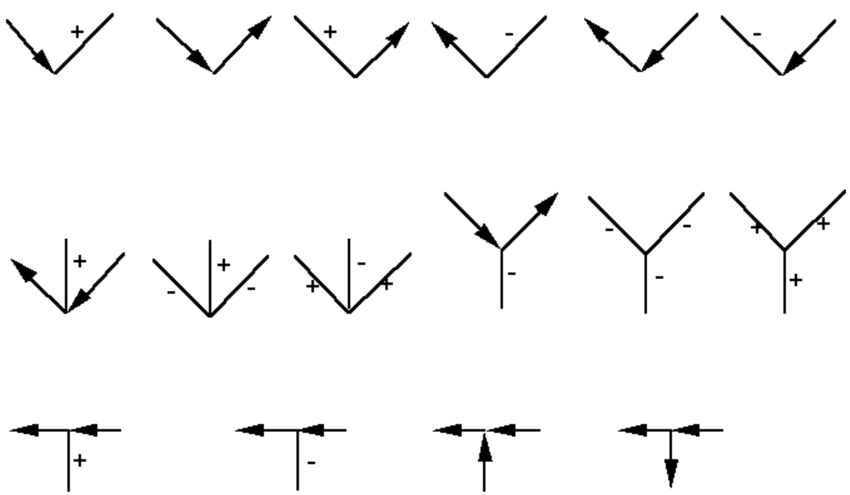
Min Conflict Performance

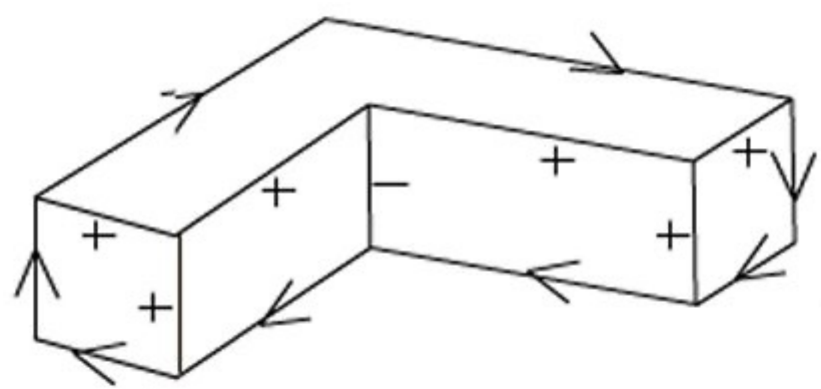
Except in a certain critical range of the ratio constraints to variables.



Famous example: labeling line drawings

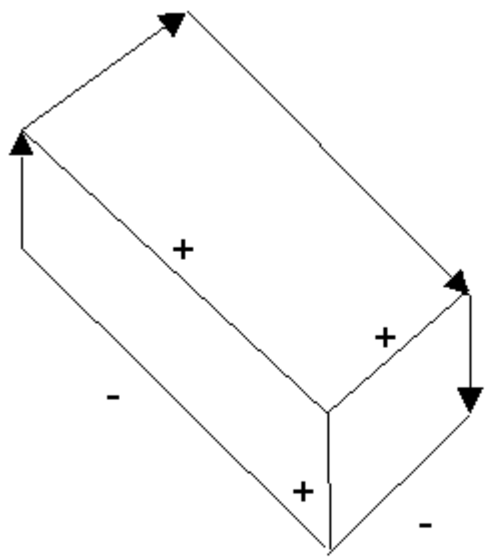
- Waltz labeling algorithm was one of the earliest AI CSP application (1972)
  - **Convex** interior lines labeled as **+**
  - **Concave** interior lines labeled as –
  - **Boundary lines** labeled as → with background to left
- 208 potential labelings for vertices, but only 18 are legal for simple blocks world scenes
- A line must have a single labeling



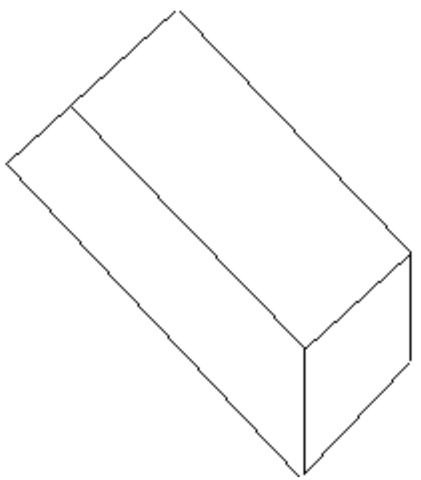


Labeling line drawings

Waltz labeling algorithm: propagate constraints repeatedly until a solution is found



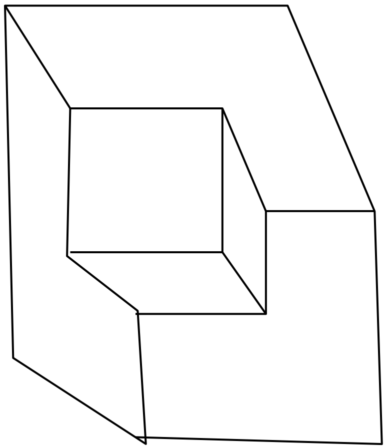
solution for one labeling problem



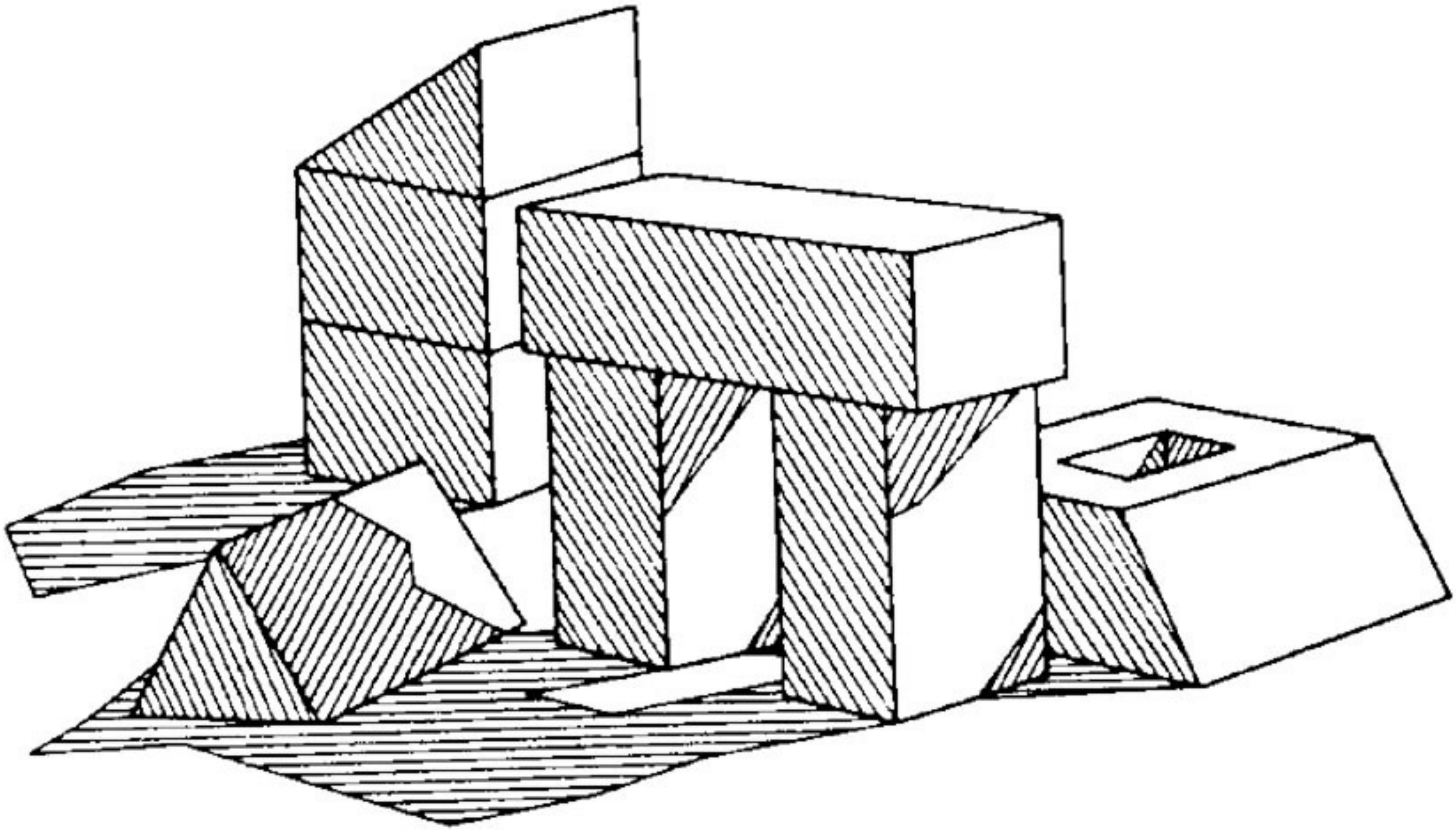
labeling problem with no solution

Labeling line drawings

This line drawing is ambiguous, with two interpretations



Shadows add complexity



CSP was able to label scenes where some of the lines were caused by shadows

Challenges for constraint reasoning

- What if not all constraints can be satisfied?
  - **Hard** vs. **soft** constraints vs. **preferences**
  - Degree of constraint satisfaction
  - Cost of violating constraints
- What if constraints are of different forms?
  - Symbolic constraints
  - Logical constraints
  - Numerical constraints [constraint solving]
  - Temporal constraints
  - Mixed constraints

Summary

- Many problems can be effectively modeled as constraints solving problems
- **Arc consistency** is simple yet powerful
- The approach is very good at **reducing search** needed
- Constraints are also useful for **local search**
- There's a lot of complexity in many real-world problems that require additional ideas and tools