CMSC 471
Artificial Intelligence

Constraint Satisfaction

Frank Ferraro – ferraro@umbc.edu

Many slides courtesy Tim Finin

A General Searching Algorithm

Core ideas:

1. Maintain a list of frontier (fringe) nodes
   1. Nodes coming *into* the frontier have been explored
   2. Nodes going out of the frontier have not been explored
2. Iteratively select nodes from the frontier and explore unexplored nodes from the frontier
3. Stop when you reach your **goal**

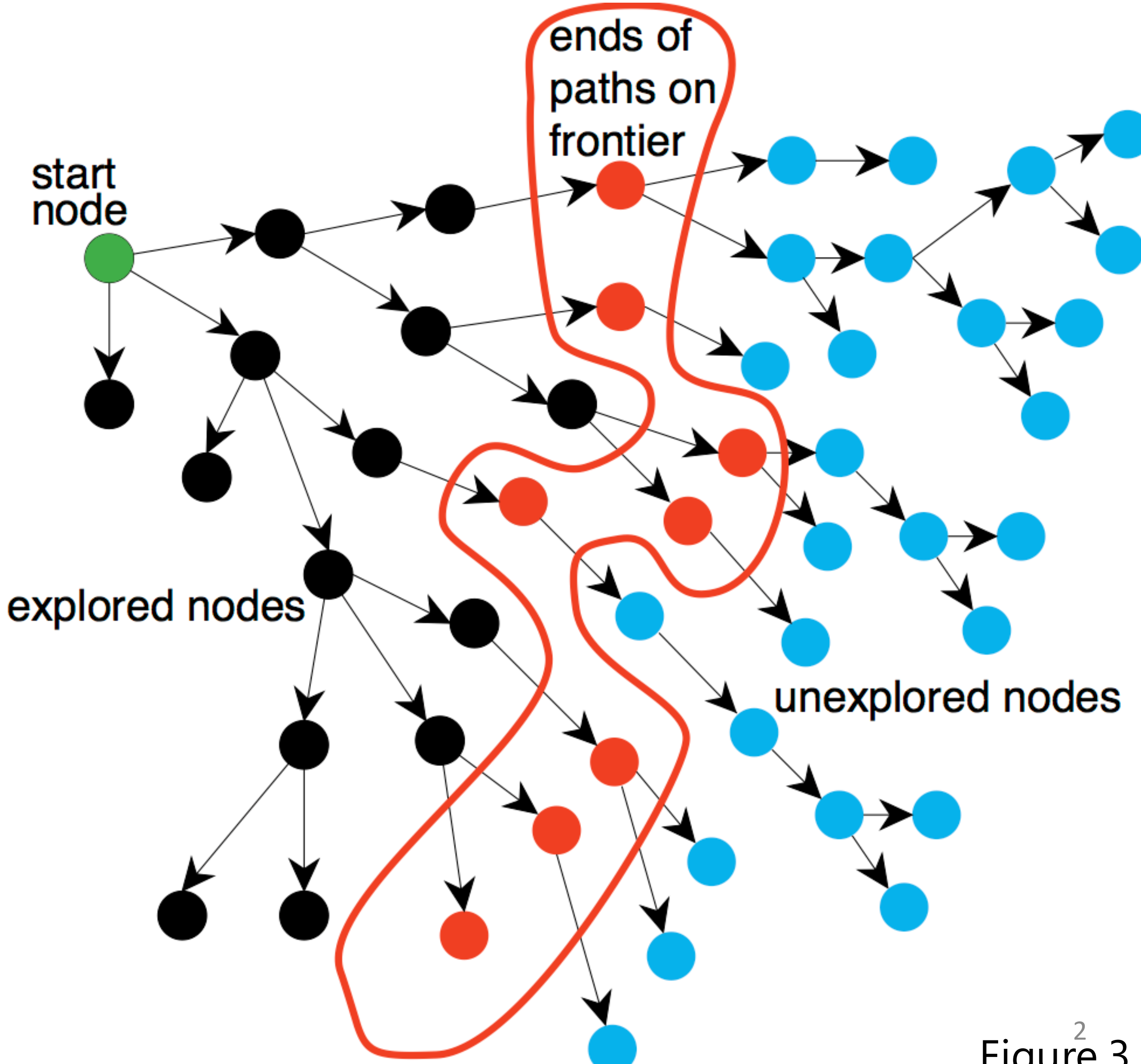


Figure 3.3

Recap

Informed vs. uninformed search

Uninformed search strategies (blind search)

- Use no information about likely *direction* of a goal
- Methods: breadth-first, depth-first, depth-limited, uniform-cost, depth-first iterative deepening, bidirectional

Informed search strategies (heuristic search)

- Use information about domain to (try to) (usually) head in the general direction of goal node(s)
- Methods: hill climbing, best-first, greedy search, beam search, algorithm A, algorithm A*

Recap

Evaluating search strategies

- **Completeness**
  - Guarantees finding a solution whenever one exists
- **Time complexity** (worst or average case)
  - Usually measured by *number of nodes expanded*
- **Space complexity**
  - Usually measured by maximum size of graph/tree during the search
- **Optimality/Admissibility**
  - If a solution is found, is it **guaranteed** to be an optimal one, i.e., one with minimum cost

Recap

Summary (Fig 3.11)

Strategy	Selection from Frontier	Path found	Space
Breadth-first	First node added	Fewest arcs	Exponential
Depth-first	Last node added	No	Linear
Iterative deepening	—	Fewest arcs	Linear
Greedy best-first	Minimal $h(p)$	No	Exponential
Lowest-cost-first	Minimal cost (p)	Least cost	Exponential
A^*	Minimal cost $(p) + h(p)$	Least cost	Exponential
IDA^*	—	Least cost	Linear

Overview

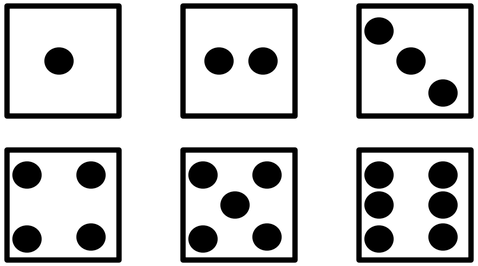
- Constraint satisfaction is a powerful problem-solving paradigm
  - Problem: set of variables to which we must assign values satisfying problem-specific constraints
  - Constraint programming, constraint satisfaction problems (CSPs), constraint logic programming…
- Algorithms for CSPs
  - Backtracking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

Some Core Terminology

- **(algebraic) variable** is a symbol used to denote features of possible worlds
  - If X is a variable, dom(X) is X's domain (the values X can take on)

Example: Variable

Let's consider rolling a standard, six-sided die

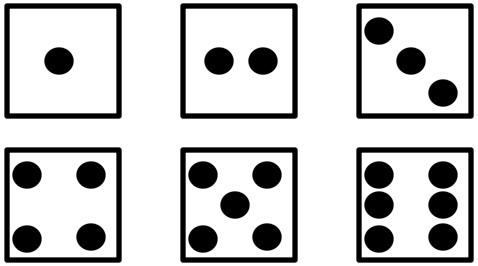


Let X_i be the variable corresponding to the outcome of the *i*th role

Q: What is $\text{dom}(X_i)$?

Example: Variable

Let's consider rolling a standard, six-sided die



Let X_i be the variable corresponding to the outcome of the *i*th role

Q: What is dom(X_i)?

A: dom(X_i) = {1, 2, 3, 4, 5, 6}

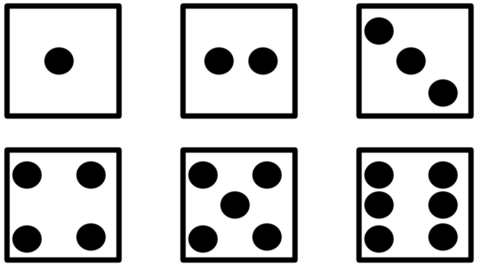
Types of Variables

- Discrete variables have finite or countable domains
  - Binary variables have two values in their domain
  - Boolean variables have two variables, TRUE and FALSE
  - Other examples?

- Continuous have uncountably infinite domains
  - Example types?

Example: Variable

Let’s consider rolling a standard, six-sided die



Let X_i be the variable corresponding to the outcome of the *i*th role

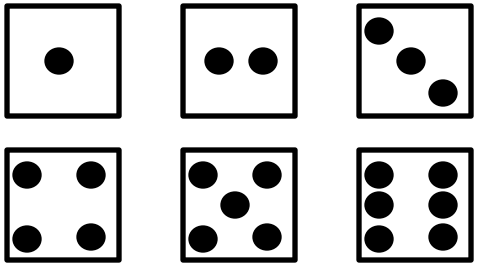
Q: What is dom(X_i)?

Q: Is X_i discrete or continuous?

A: dom(X_i) = {1, 2, 3, 4, 5, 6}

Example: Variable

Let's consider rolling a standard, six-sided die



Let X_i be the variable corresponding to the outcome of the *i*th role

Q: What is dom(X_i)?

A: dom(X_i) = {1, 2, 3, 4, 5, 6}

Q: Is X_i discrete or continuous?

A: Discrete

Variable Assignments

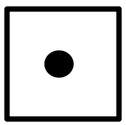
Given N variables $\mathbf{X} = \{X_1, X_2, \dots, X_N\}$

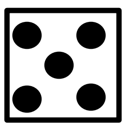
- An assignment is a setting of a subset X' of those variables
  - Total assignment: $X' = \boldsymbol{X}$
  - Partial assignment: $X' \neq \boldsymbol{X}$
- A **possible world** is a possible way the world (the real world or some imaginary world) could be

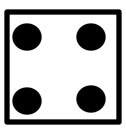
Full vs. Partial Assignment Example

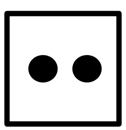
Let’s say there are N=9 rolls of the same die

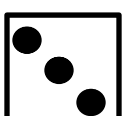
Full assignment

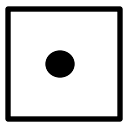
$X_1 =$ 

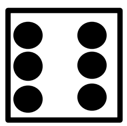
$X_2 =$ 

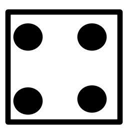
$X_3 =$ 

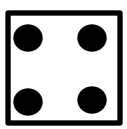
$X_4 =$ 

$X_5 =$ 

$X_6 =$ 

$X_7 =$ 

$X_8 =$ 

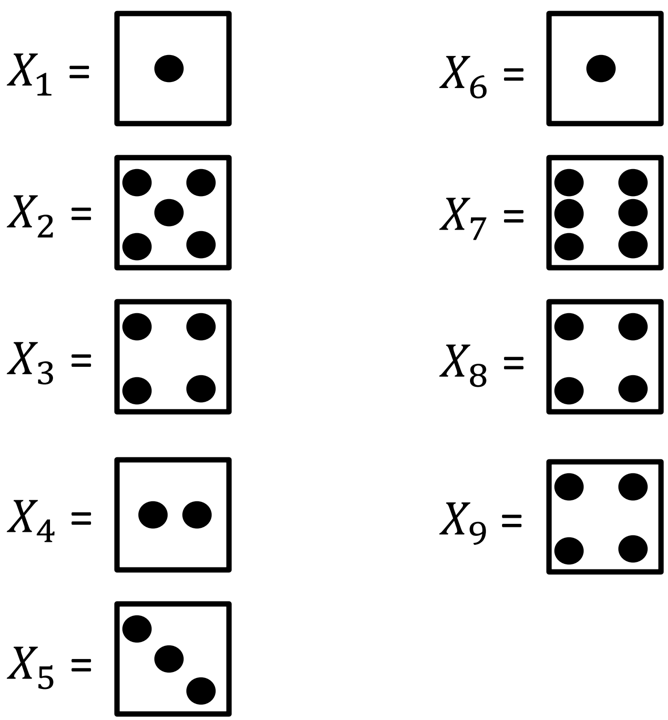
$X_9 =$ 

Partial assignment

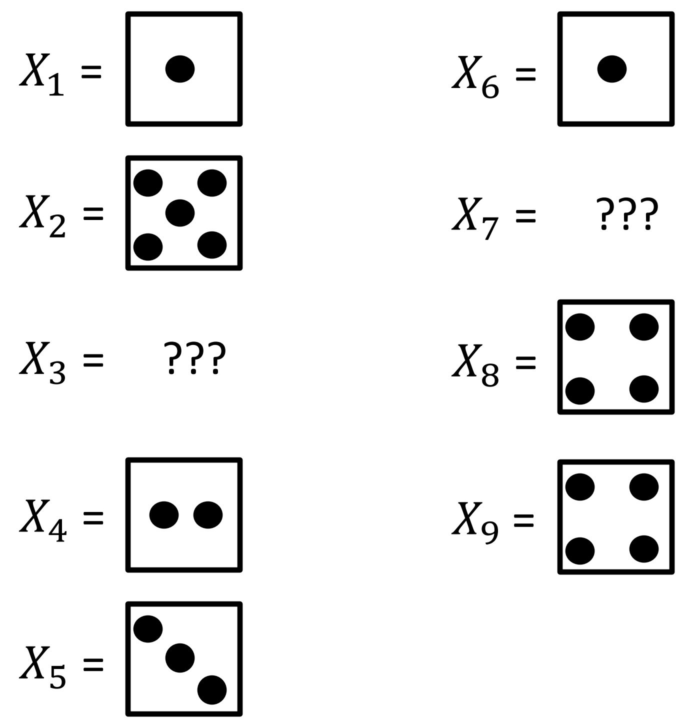
Full vs. Partial Assignment Example

Let’s say there are N=9 rolls of the same die

Full assignment



Partial assignment



Thinking About Possible Worlds

Let's say there are N variables. How many possible worlds are there if:

- Each variable's domain is of size 2?
- Each variable's domain is of size 10?
- Each variable's domain is uncountably infinite (the real numbers)?

Constraints

Many ***possible worlds***… but are all of those possible worlds "possible?"

Constraint: a specification of allowed / disallowed combinations of assignments to individual variables

- **Scope**: the set of variables involved in the constraint
- **Relation**: Boolean function on the scope that indicates if the constraint is satisfied

Constraints

Many **possible worlds**… but are all of those possible worlds "possible?"

Constraint: a specification of allowed / disallowed combinations of assignments to individual variables

- **Scope**: the set of variables involved in the constraint
- **Relation**: Boolean function on the scope that indicates if the constraint is satisfied

Scheduling example (4.7)

A, B, C are variables representing dates of events

Each has possible values {Jan, Feb, March, April}

"A can't happen later than B; and B must happen in January or February; and B must be before C; and either A and B can't happen at the same time, or C can't occur in April"

Constraints

Many **possible worlds**… but are all of those possible worlds "possible?"

Constraint: a specification of allowed / disallowed combinations of assignments to individual variables

- **Scope**: the set of variables involved in the constraint
- **Relation**: Boolean function on the scope that indicates if the constraint is satisfied

Scheduling example (4.7)

A, B, C are variables representing dates of events

Each has possible values {Jan, Feb, March, April}

"A can't happen later than B; and B must happen in January or February; and B must be before C; and either A and B can't happen at the same time, or C can't occur in April"

Constraints

Many **possible worlds**… but are all of those possible worlds "possible?"

Constraint: a specification of allowed / disallowed combinations of assignments to individual variables

- **Scope**: the set of variables involved in the constraint
- **Relation**: Boolean function on the scope that indicates if the constraint is satisfied

Scheduling example (4.7)

A, B, C are variables representing dates of events

Each has possible values {Jan, Feb, March, April}

"A can't happen later than B; and B must happen in January or February; and B must be before C; and either A and B can't happen at the same time, or C can't occur in April"

$$A \leq B \ \wedge$$
$$B < \text{March} \wedge$$
$$B < C \ \wedge$$
$$A \neq B \ \vee C < \text{April}$$

Constraints

Many ***possible worlds***… but are all of those possible worlds "possible?"

Constraint: a specification of allowed / disallowed combinations of assignments to individual variables

- **Scope**: the set of variables involved in the constraint
- **Relation**: Boolean function on the scope that indicates if the constraint is satisfied

Scheduling example (4.7)

A, B, C are variables representing dates of events

Each has possible values {Jan, Feb, March, April}

"A can't happen later than B; and B must happen in January or February; and B must be before C; and either A and B can't happen at the same time, or C can't occur in April"

$$A \leq B \ \wedge$$
$$B < \text{March} \wedge$$
$$B < C \ \wedge$$
$$A \neq B \ \vee C < \text{April}$$

Scope ({A, B})

Constraints

Many **possible worlds**… but are all of those possible worlds "possible?"

Constraint: a specification of allowed / disallowed combinations of assignments to individual variables

- **Scope**: the set of variables involved in the constraint
- **Relation**: Boolean function on the scope that indicates if the constraint is satisfied

Scheduling example (4.7)

A, B, C are variables representing dates of events

Each has possible values {Jan, Feb, March, April}

"A can't happen later than B; and B must happen in January or February; and B must be before C; and either A and B can't happen at the same time, or C can't occur in April"

$$A \le B \;\wedge$$
$$B < \text{March} \;\wedge$$
$$B < C \;\wedge$$
$$A \neq B \;\vee C < \text{April}$$

Scope ({A, B})

Relation (≤)

Constraints

Many **possible worlds**… but are all of those possible worlds "possible?"

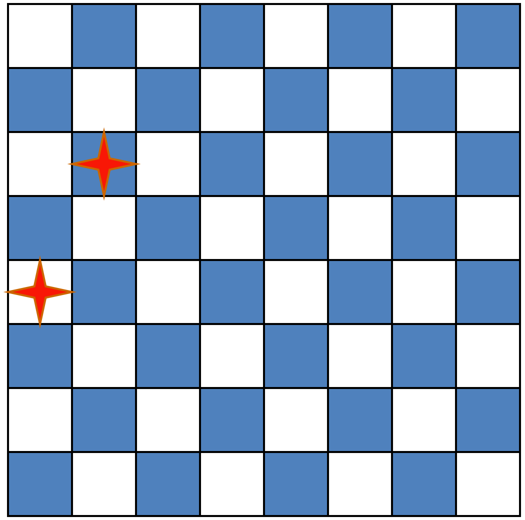
Constraint: a specification of allowed / disallowed combinations of assignments to individual variables

- **Scope**: the set of variables involved in the constraint
- **Relation**: Boolean function on the scope that indicates if the constraint is satisfied

Constraints are **satisfied** (an assignment that makes all constraints TRUE) or **violated**

Motivating example: 8 Queens

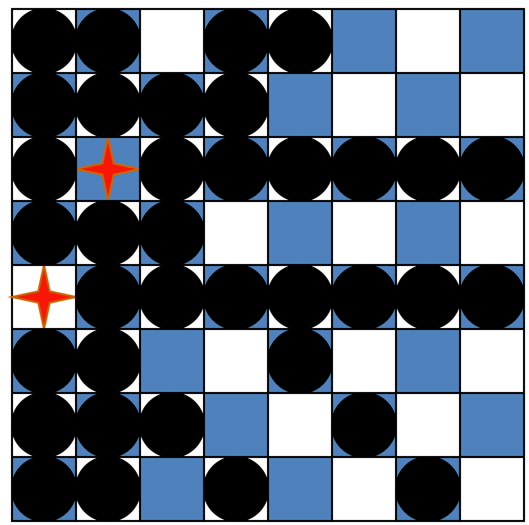
Place 8 queens on a chess board such
That none is attacking another.



Generate-and-test, with no
redundancies → "only" 8^8 combinations

8**8 is 16,777,216

Motivating example: 8-Queens



After placing these two queens, it's trivial to mark the squares we can no longer use

What more do we need for 8 queens?

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

→ Explicit representation of constraints and constraint manipulation algorithms

Informal definition of CSP

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

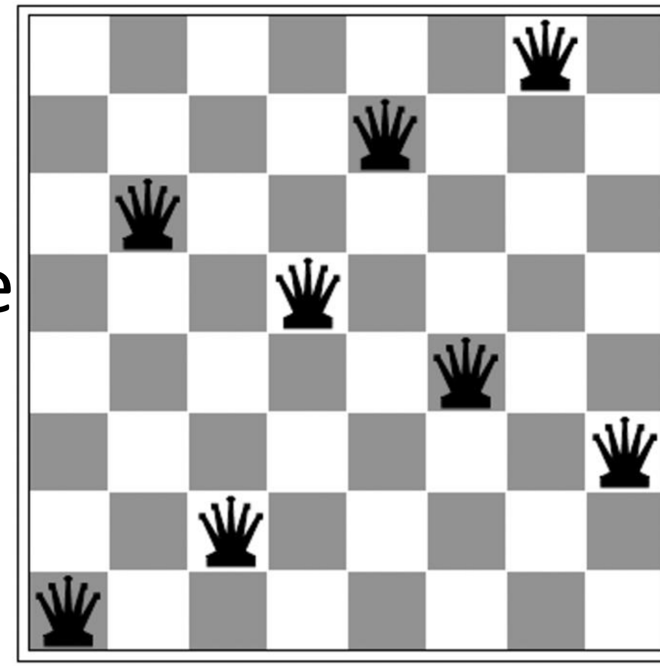
  (1) finite set of variables

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

  (3) set of constraints limiting values variables can take
- **Solution:** assignment of a value to each variable such that all constraints are satisfied
- **Possible tasks:** decide if solution exists, find a solution, find all solutions, find *best solution* according to 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?



Example: 8-Queens Problem

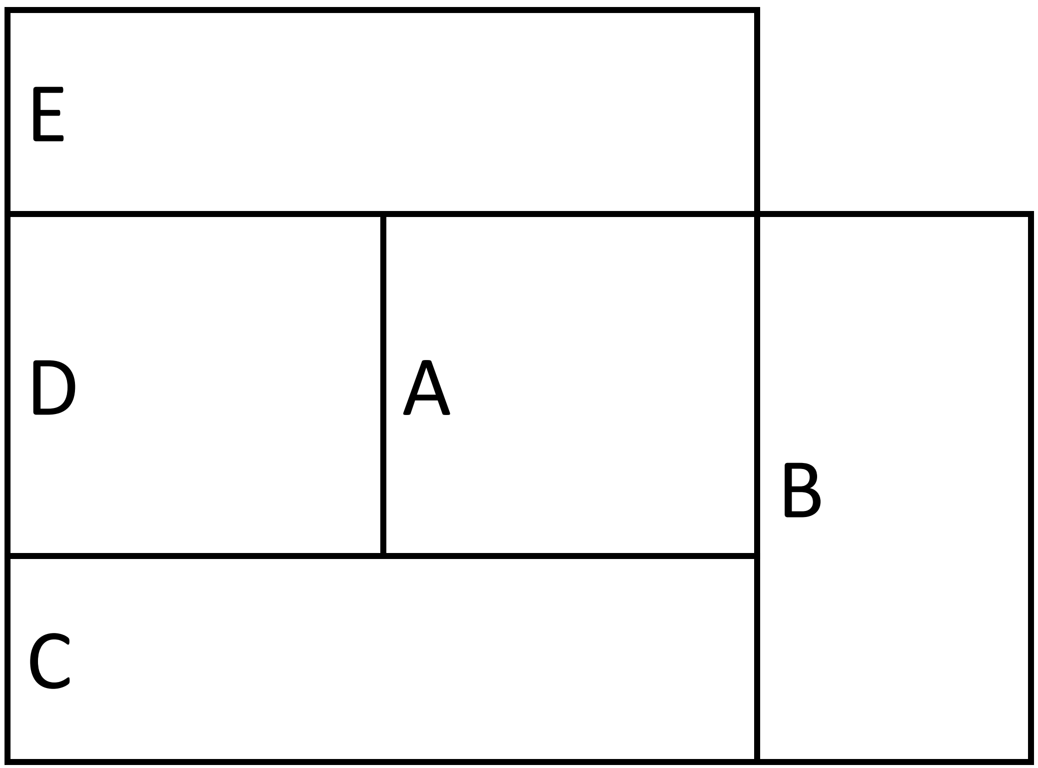
- Eight variables Qi, i = 1..8 where Qi is the row number of queen in column i
- Domain for each variable {1,2,…,8}
- Constraints are of the forms:
  - No queens on same row

    Qi = k ➔ Qj $\neq$ k for j = 1..8, j$\neq$i
  - No queens on same diagonal

    Qi=rowi, Qj=rowj ➔ |i-j| $\neq$ |rowi-rowj| for j = 1..8, j$\neq$i

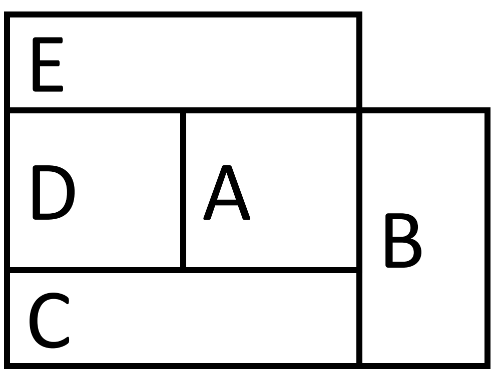
Example: Map coloring

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

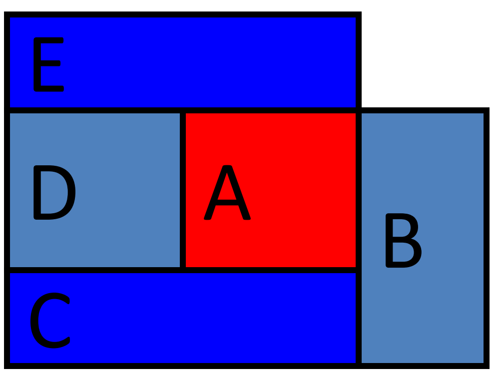


Map coloring

- 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



=>



Example: SATisfiability

- Given a set of logic propositions containing variables, find an assignment of the variables to {false, true} that satisfies them
- For example, the two clauses:
  - $(A \lor B \lor \neg C) \land (\neg A \lor D)$
  - equivalent to $(C \rightarrow A) \lor (B \land D \rightarrow A)$

  are satisfied by

  A = false, B = true, C = false, D = false
- Satisfiability known to be NP-complete
- ⇒ worst case, solving CSP problems requires exponential time

Real-world problems

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

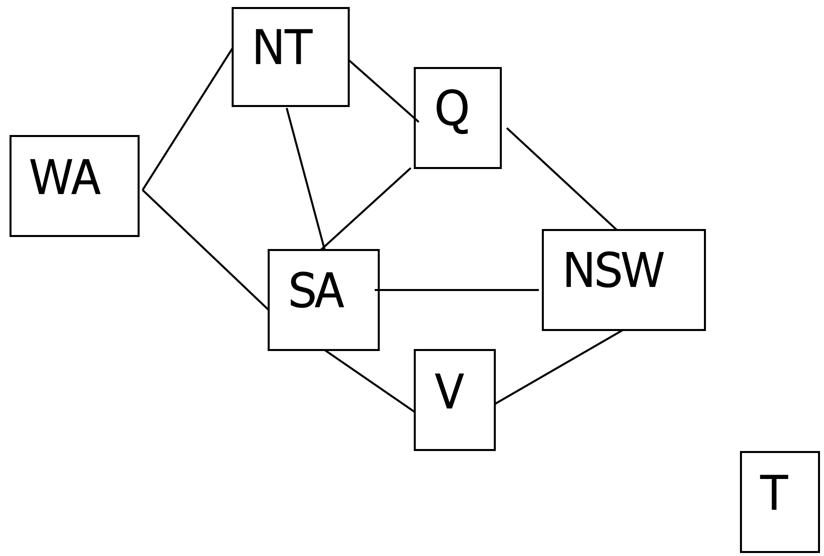
- Scheduling
- Temporal reasoning
- Building design
- Planning
- Optimization/satisfaction
- Vision
- Graph layout
- Network management
- Natural language processing
- Molecular biology / genomics
- VLSI design

Definition of a constraint network (CN)

A constraint network (CN) consists of

- Set of variables $X = \{x_1, x_2, \ldots x_n\}$
  - with associate domains $\{d_1, d_2, \ldots d_n\}$
  - domains are typically finite
- Set of constraints $\{c_1, c_2 \ldots c_m\}$ where
  - each defines a predicate that is a relation over a particular subset of variables (X)
  - e.g., C_i involves variables $\{X_{i1}, X_{i2}, \ldots X_{ik}\}$ and defines the relation $R_i \subseteq D_{i1} \times D_{i2} \times \ldots D_{ik}$

Running example: coloring Australia

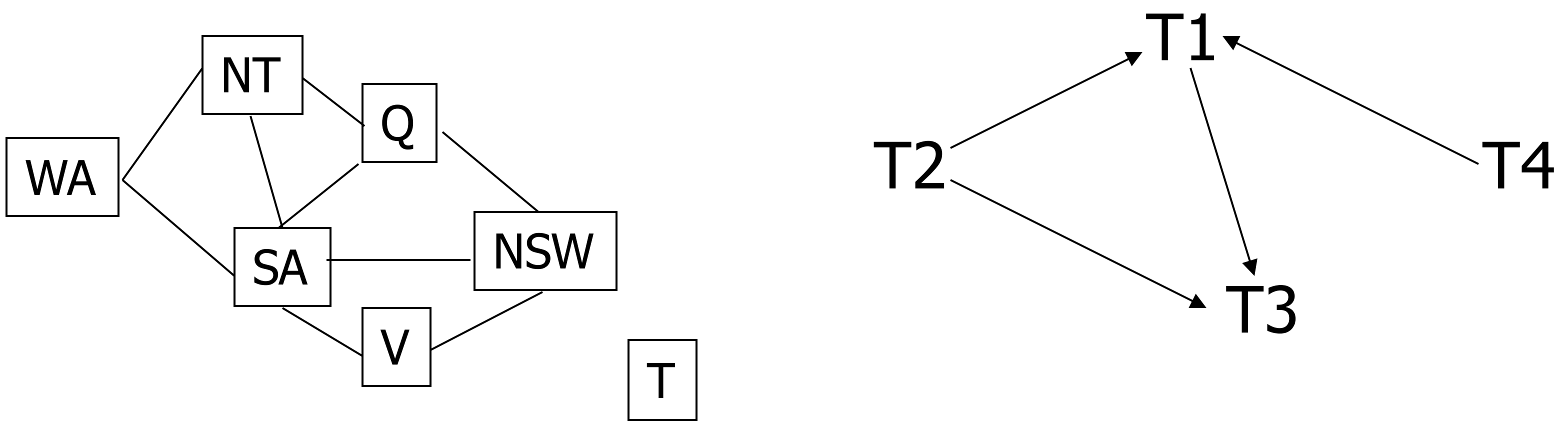


- Seven variables: {WA, NT, SA, Q, NSW, V, T}
- Each variable has same domain: {red, green, blue}
- No two adjacent variables can have same value:

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

Unary & binary constraints most common

Binary constraints



- Two variables are adjacent or neighbors if connected by an edge or an arc
- Possible to rewrite problems with higher-order constraints as ones with just binary constraints

Typical tasks for CSP

- 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 a CSP where all constraints are binary or unary
- Any non-binary CSP can be converted into a binary CSP by introducing additional variables
- A binary CSP can be 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

General Methods of Solving CSPs

- Generate-and-Test, aka Brute Force
- Search (backtracking)
- Consistency checking
  - Forward checking
  - Arc consistency
  - Domain splitting
  - Variable Elimination
- Localized search

Brute Force methods

- Finding a solution by a brute force search is easy
  - Generate and test is a *weak method*
  - Just generate potential combinations and test each
- 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!

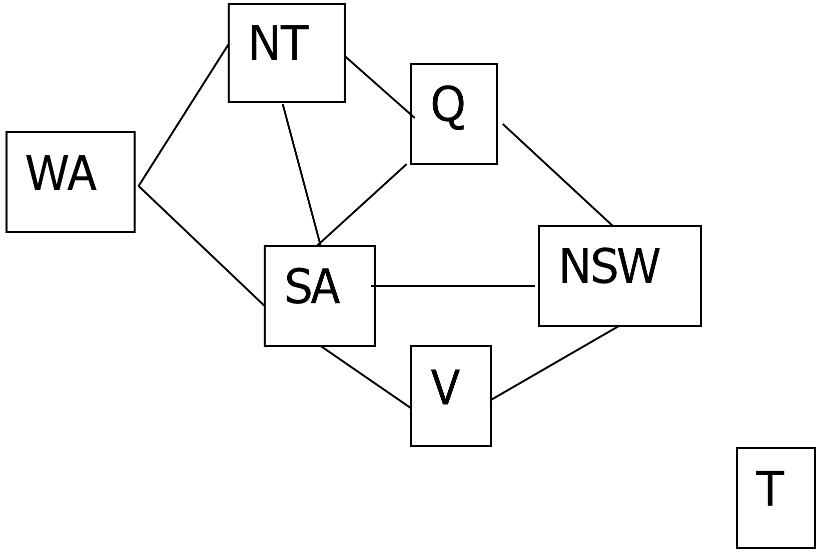
```
solve(A,B,C,D,E) :-
  color(A),
  color(B),
  color(C),
  color(D),
  color(E),
  not(A=B),
  not(A=B),
  not(B=C),
  not(A=C),
  not(C=D),
  not(A=E),
  not(C=D).

color(red).
color(green).
color(blue).
```

generate

test

4**190 is 2462625387274654950767440006258975862817483704404090416746768337765357610718575663213391640930307227550414249394176L

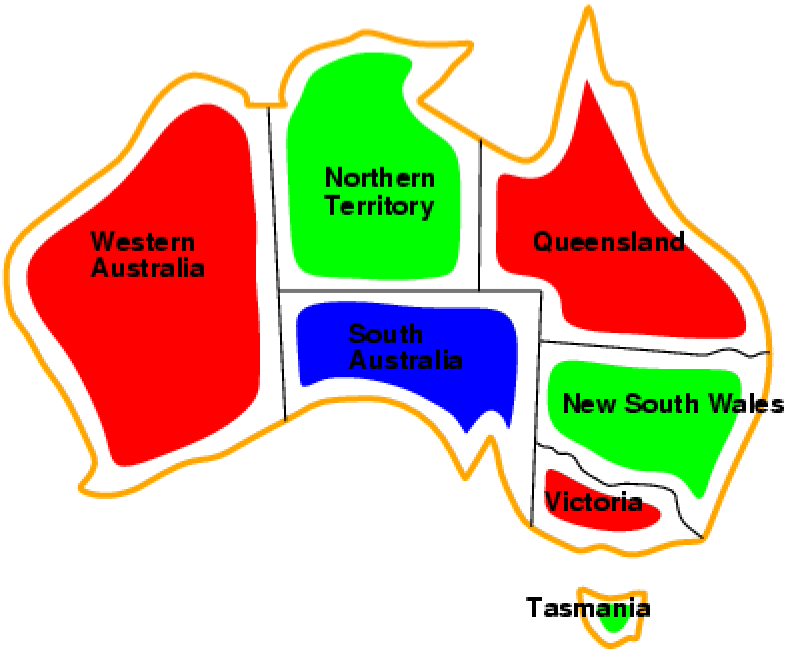
Running example: coloring Australia

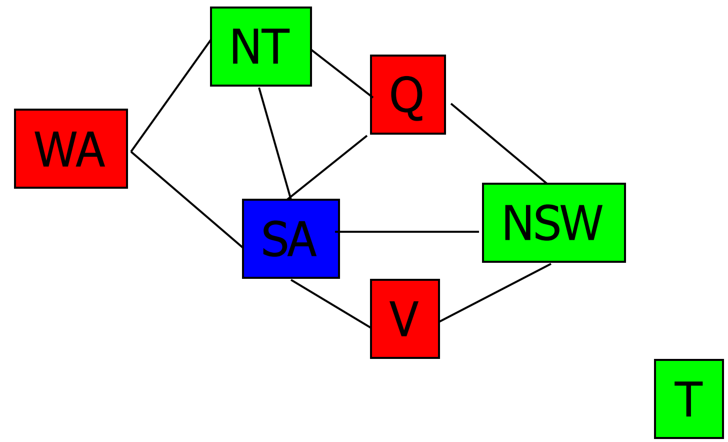


- Seven variables: {WA, NT, SA, Q, NSW, V, T}
- Each variable has same domain: {red, green, blue}
- No two adjacent variables can have same value:

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

A running example: coloring Australia

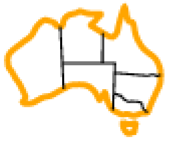




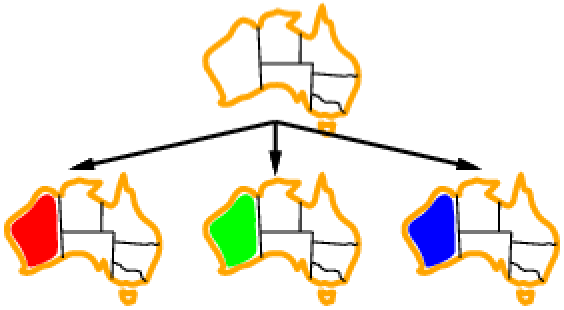
- Solutions: complete & consistent assignments
- Here is one of several solutions
- For generality, constraints can be expressed as relations, e.g., describe WA ≠ NT as {(red,green), (red,blue), (green,red), (green,blue), (blue,red),(blue,green)}

Slide

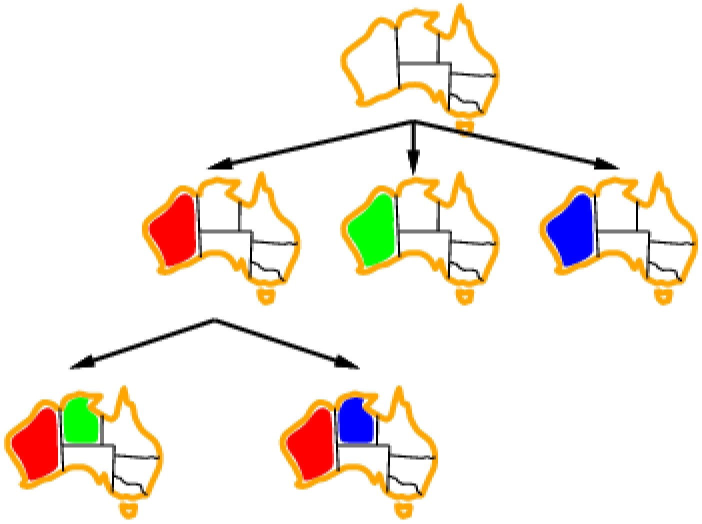
Backtracking example



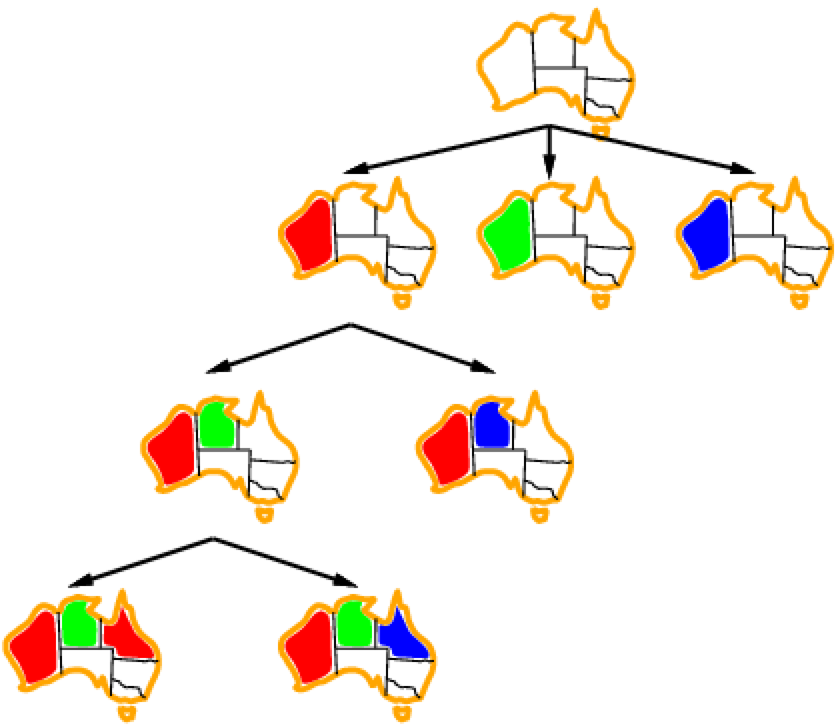
Backtracking example



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

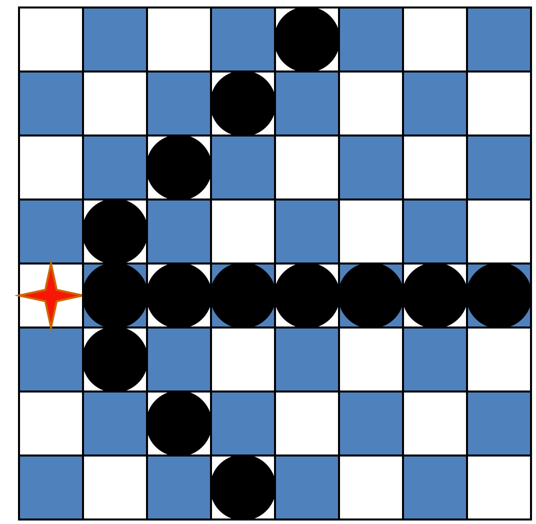
- Can we detect inevitable failure early?
- Which variable should be assigned next?
- In what order should its values be tried?

General Methods of Solving CSPs

- Generate-and-Test, aka Brute Force
- Search (backtracking)
- Consistency checking
  - Forward checking
  - Arc consistency
  - Domain splitting
  - Variable Elimination
- Localized search

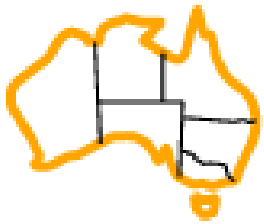
Forward Checking

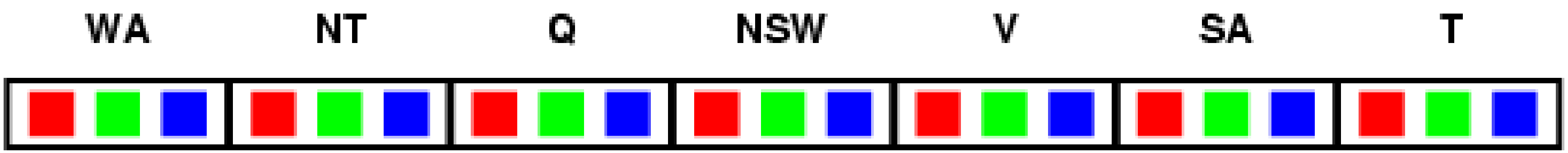
After variable X is assigned to value v, examine each unassigned variable Y connected to X by a constraint and delete values from Y's domain inconsistent with v



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

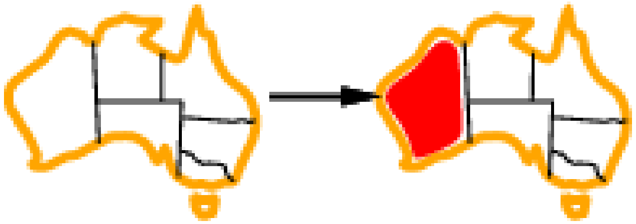
Forward checking

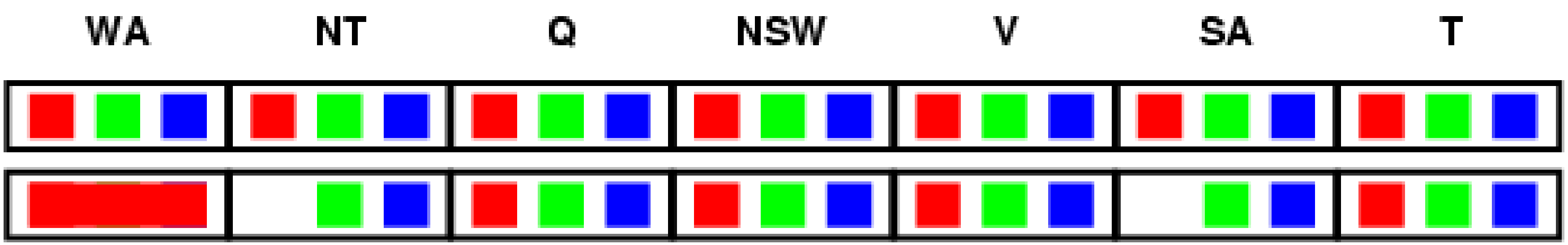




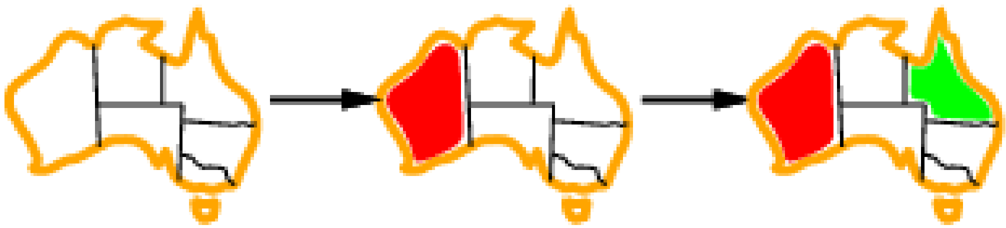
- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values

Forward checking

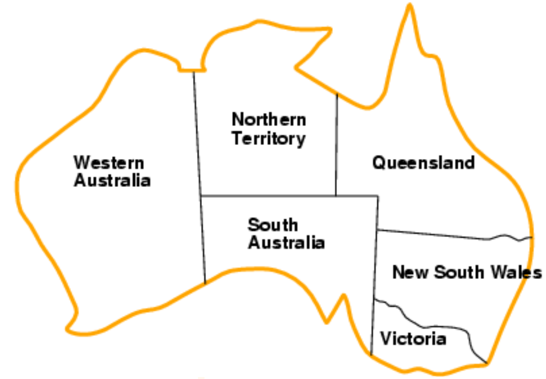


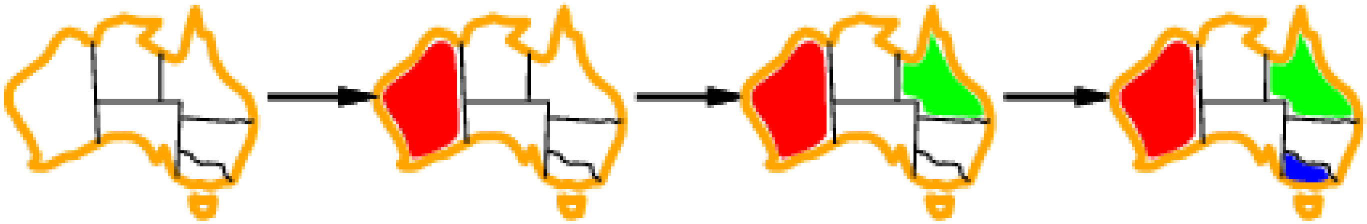


Forward checking



Forward checking





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!

Can we detect problem earlier?

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

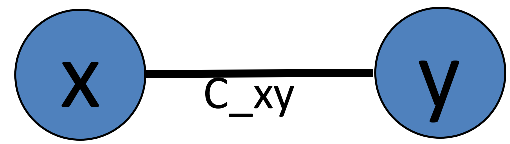
Binary CSP is arc consistent iff every constraint C_xy is arc consistent w.r.t. x as well as y

AC3 Algorithm: Enforcing Arc Consistency

When a CSP is not arc consistent, we can make it arc consistent by using the AC3 algorithm

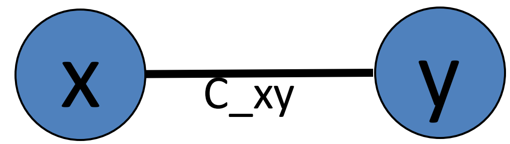
Arc Consistency Example 1

- Domains
  - D_x = {1, 2, 3}
  - D_y = {3, 4, 5, 6}



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

Arc Consistency Example 2



- Domains
  - D_x = {1, 2, 3}
  - D_y = {1, 2, 3}
- Constraint
  - 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

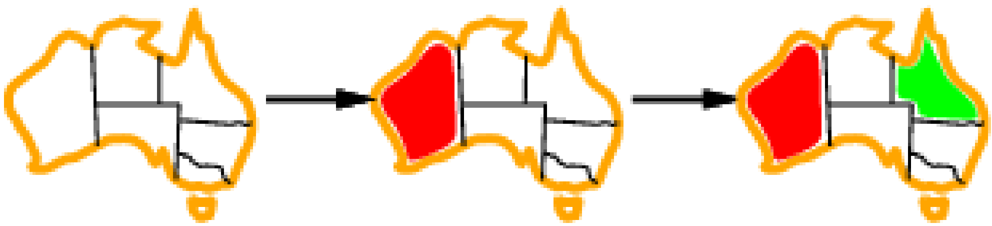
```
lambda v1,v2: v1 < v2
```

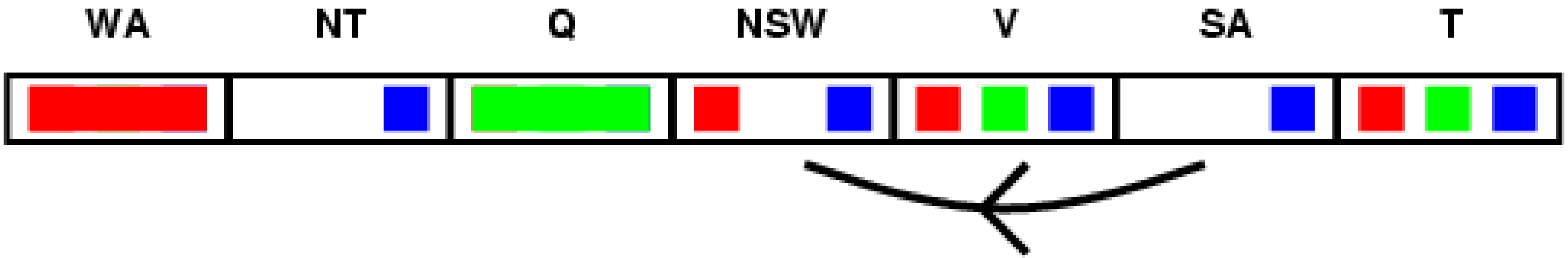
```
>>> f = lambda v1,v2: v1 < v2
>>> f
<function <lambda> at 0x10fcf21e0>
>>> f(100,200)
True
>>> f(200,100)
False
```

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

Arc consistency

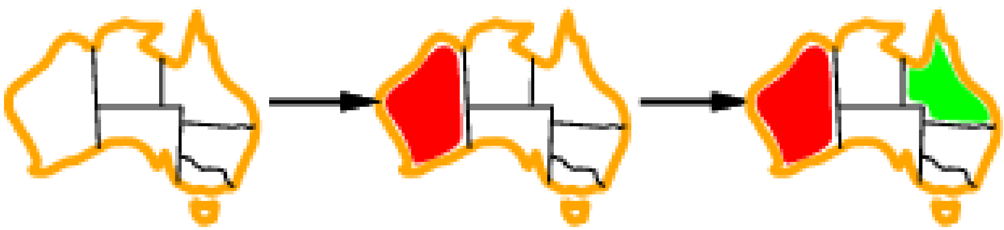
- Simplest form of propagation makes each arc consistent
- X →Y is consistent iff for every value x_i of X there is some allowed value y_j in Y

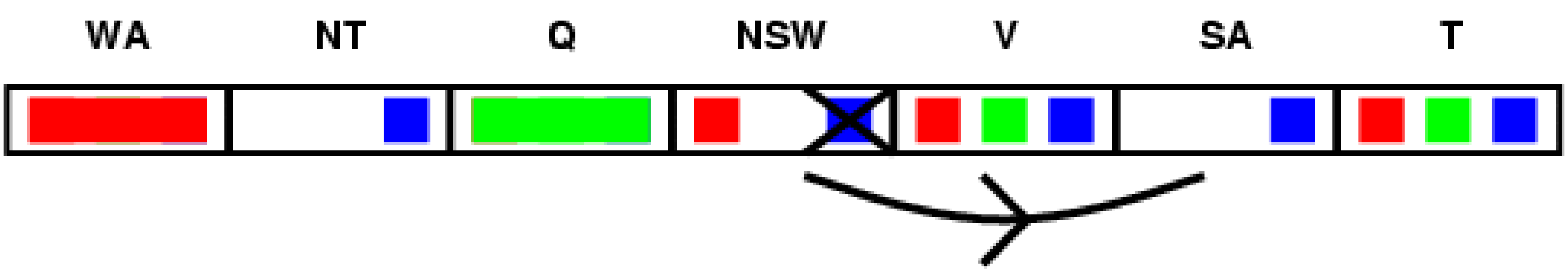




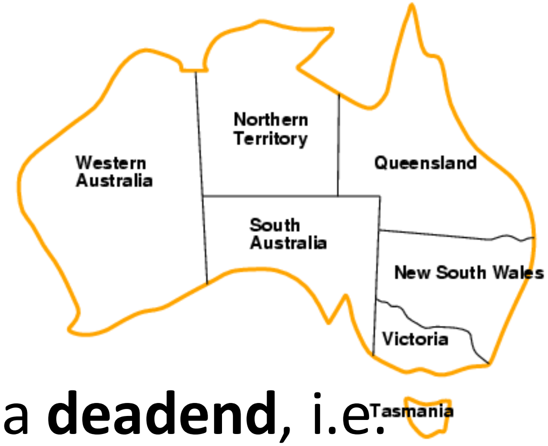
Arc consistency

- 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

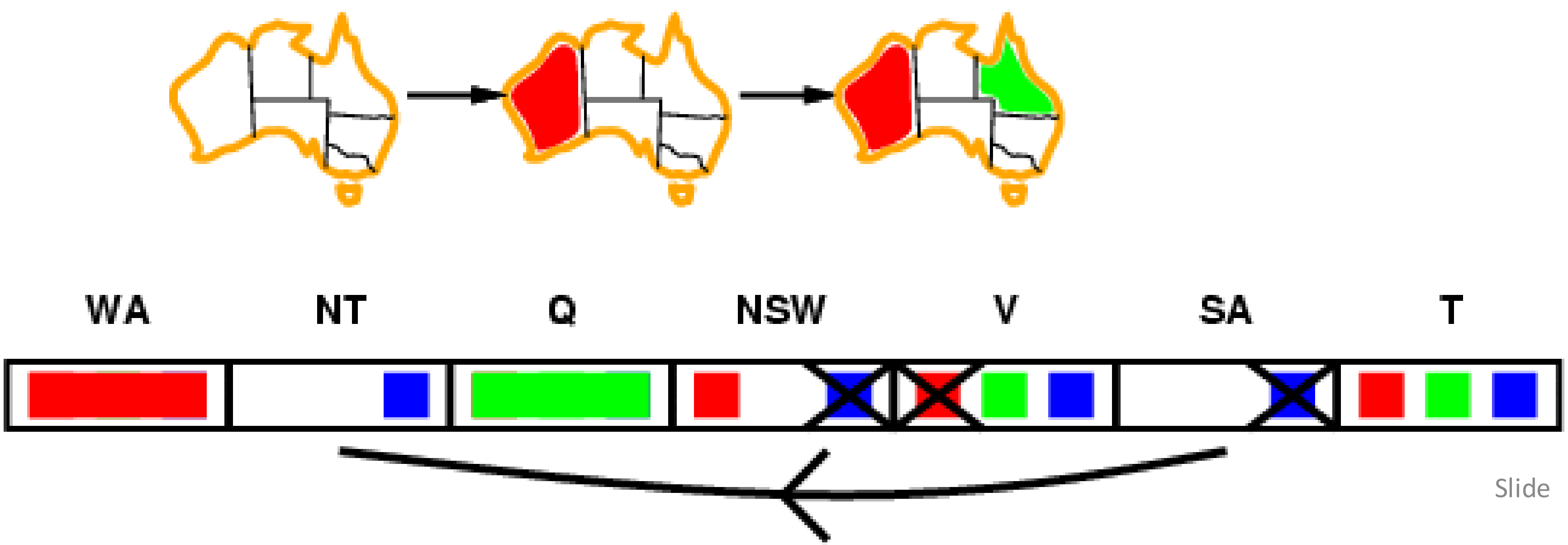




Arc consistency



- 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

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)

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

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)

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

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

Q: What is the time complexity of AC3?
e = # of constraints
d = # of values per variable

Complexity of AC3

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

A Poole & Mackworth Example (Fig 4.4)

Setup: 5 variables (A, B, C, D, E) each with domain {1,2,3,4}

Constraints:

$A \neq B$

$A = D$

$A \geq E$

$D \neq B$

$C \neq B$

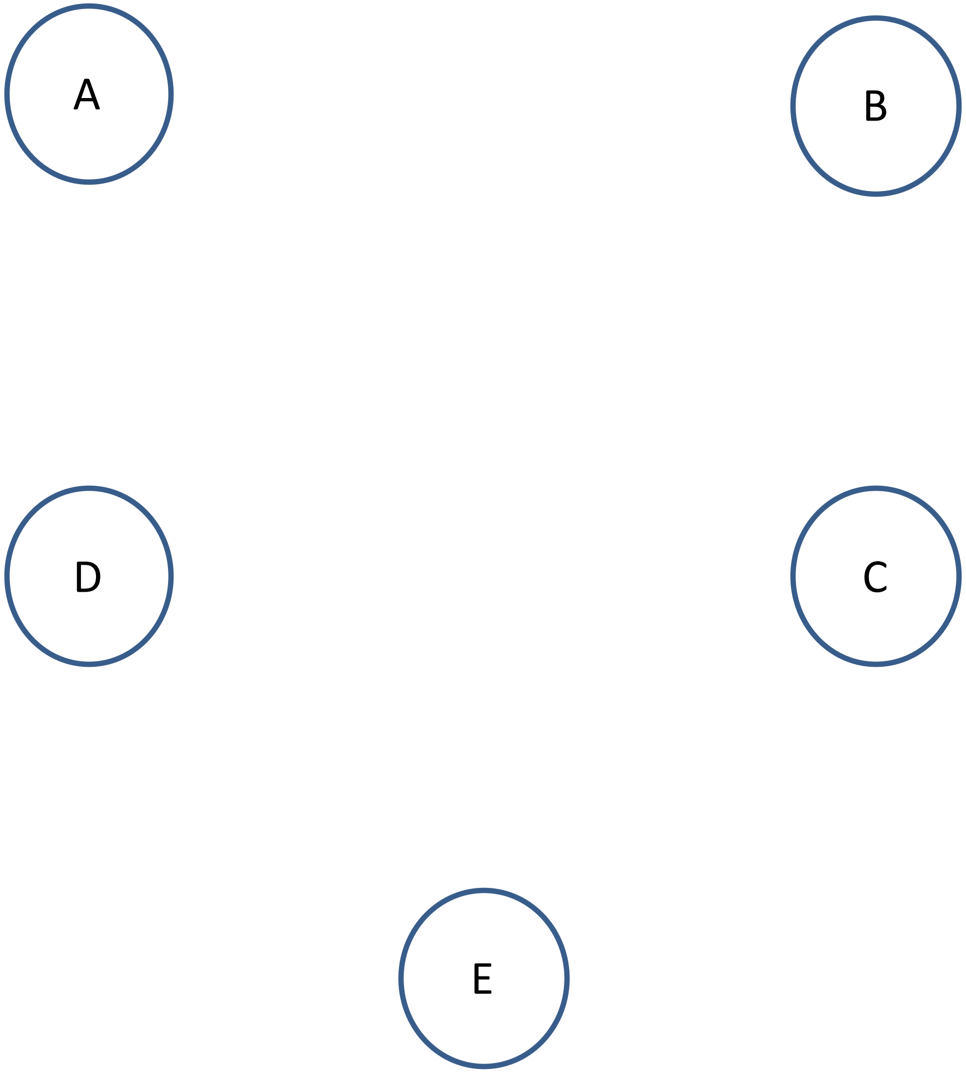
$E < B$

$C < D$

$E < C$

$E < D$

A Poole & Mackworth Example (Fig 4.4)



Setup: 5 variables (A, B, C, D, E) each with domain {1,2,3,4}

Constraints:

$A \neq B$

$A = D$

$A \geq E$

$D \neq B$

$C \neq B$

$E < B$

$C < D$

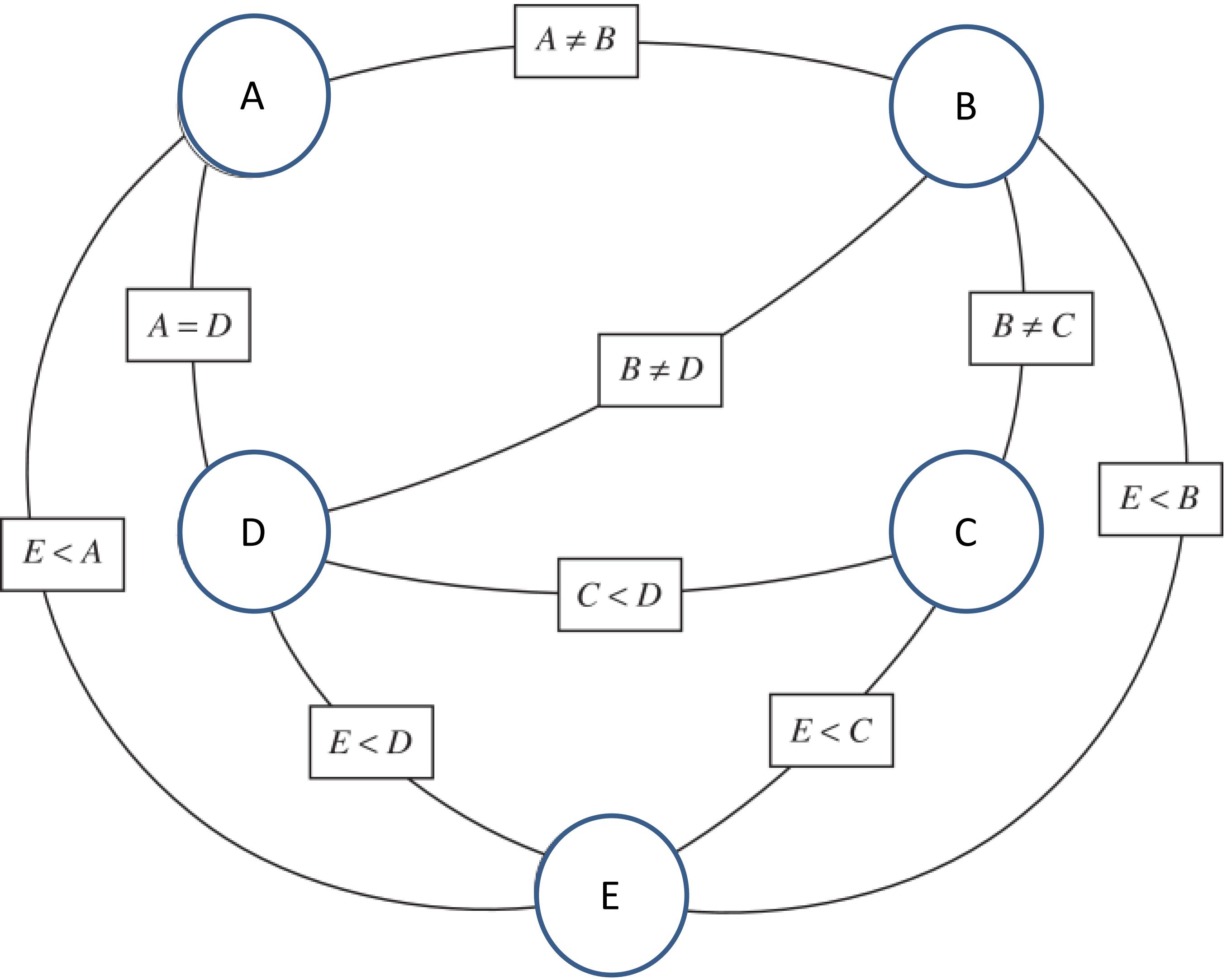
$E < C$

$E < D$

$B \neq 3$

$C \neq 2$

A Poole & Mackworth Example (Fig 4.4)

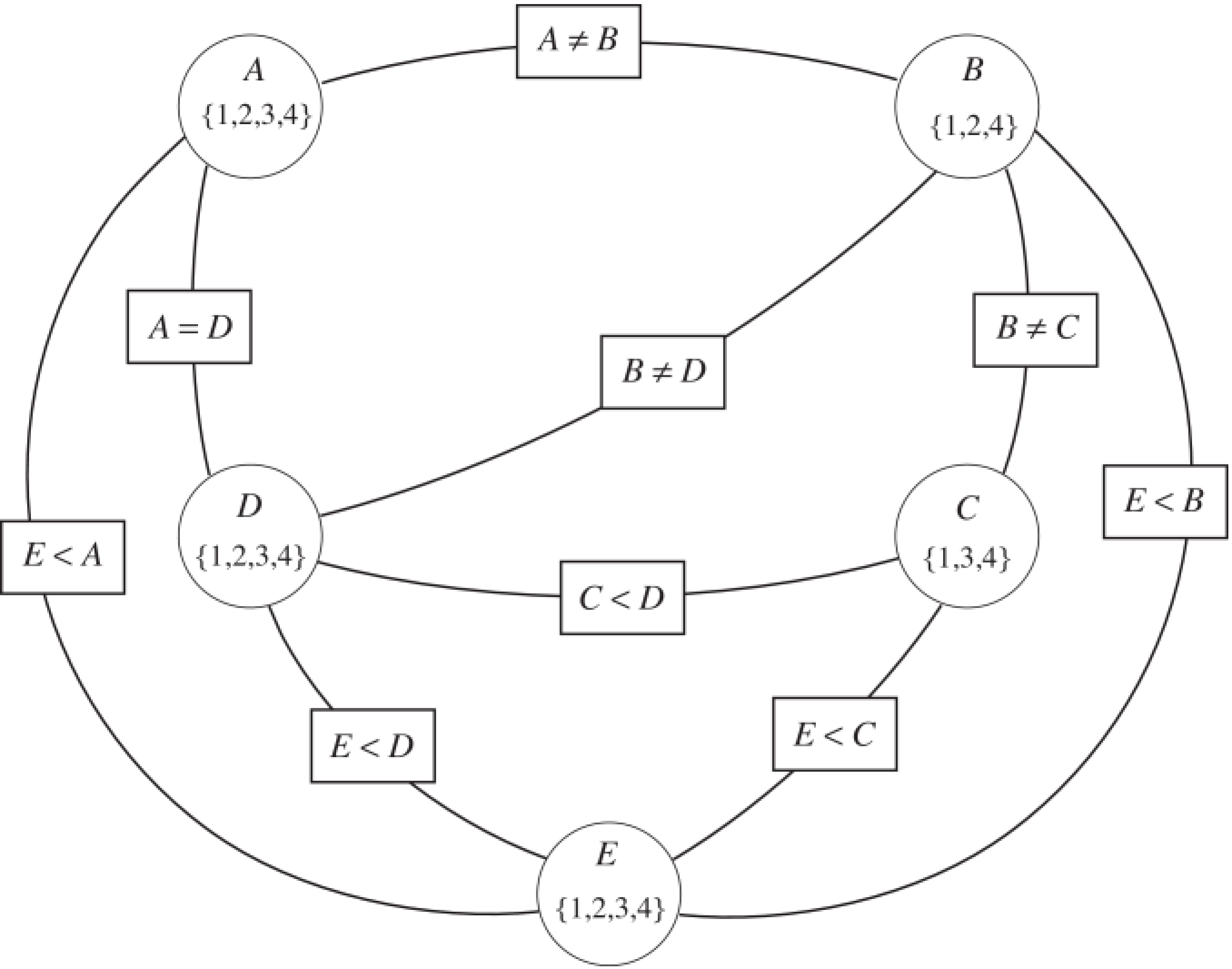


Setup: 5 variables (A, B, C, D, E) each with domain {1,2,3,4}

Constraints:

$A \neq B$
$A = D$
$A \geq E$
$D \neq B$
$C \neq B$
$E < B$
$C < D$
$E < C$
$E < D$
$B \neq 3$
$C \neq 2$

A Poole & Mackworth Example (Fig 4.4)



Setup: 5 variables (A, B, C, D, E) each with domain {1,2,3,4}

Constraints:

$A \neq B$

$A = D$

$A \geq E$

$D \neq B$

$C \neq B$

$E < B$

$C < D$

$E < C$

$E < D$

$B \neq 3$

$C \neq 2$

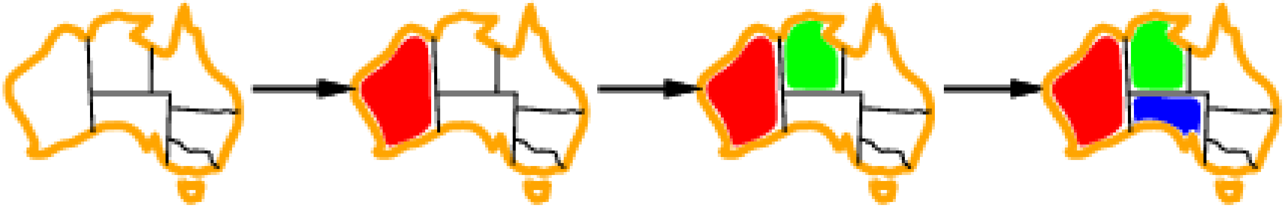
Q: Is this arc consistent?

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

Most constrained variable

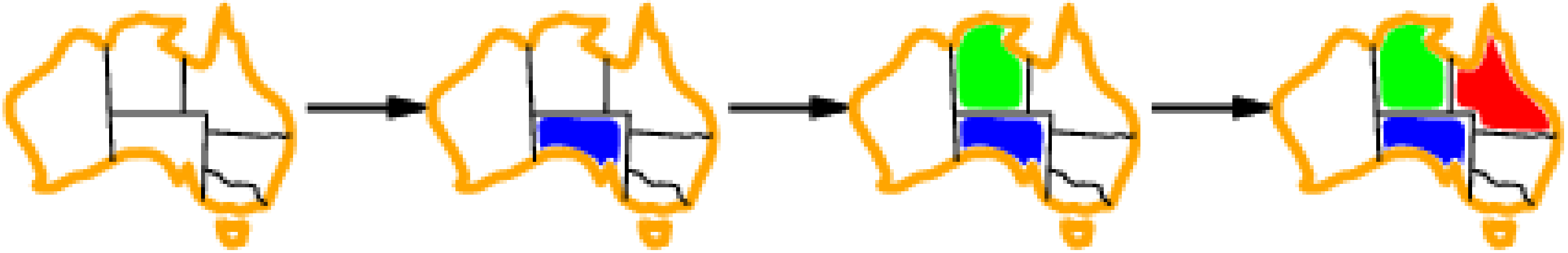
- Most constrained variable:
  choose the variable with the fewest legal values



- 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

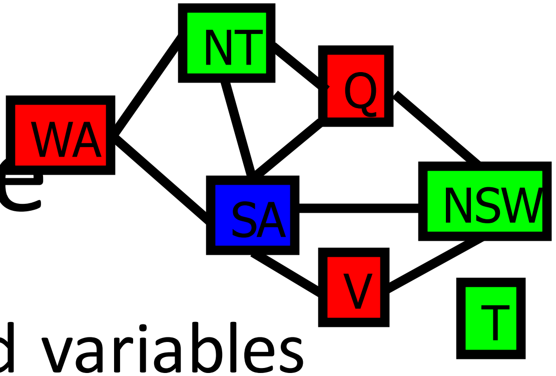
Most constraining variable

- Tie-breaker among most constrained variables
- Choose variable involved in largest # of constraints on remaining variables

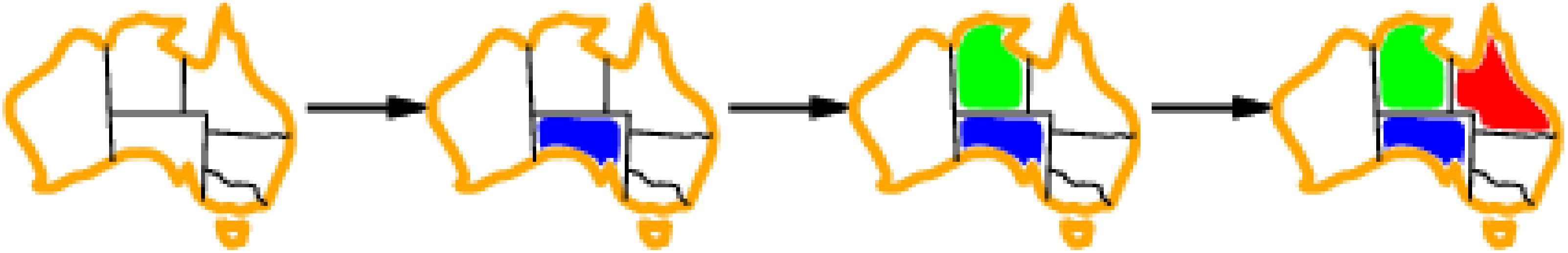


- After assigning SA to be blue, WA, NT, Q, NSW and V all have just two values left.
- WA and V have only one constraint on remaining variables and T none, so choose one of NT, Q & NSW

Most constraining variable



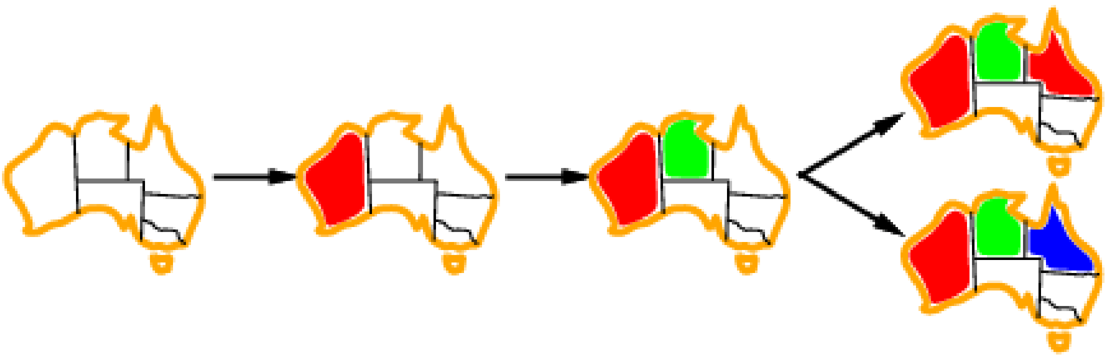
- Tie-breaker among most constrained variables
- Choose variable involved in largest # of constraints on remaining variables



- 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

Least constraining value

- Given a variable, choose least constraining value:
  - 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?

Domain Splitting

Also called “case analysis”

Split a variable’s domain into disjoint subsets, and consider them each separately

Domain Splitting

Also called “case analysis”

Split a variable’s domain into disjoint subsets, and consider them each separately

- If $\mathrm{dom}(X_i) = \{a_1, \ldots, a_M\}$, then for each possible setting of $X_i = a_m$, find an assignment to all other variables that satisfy the constraints
- This is solved a **reduced** problem

Domain Splitting

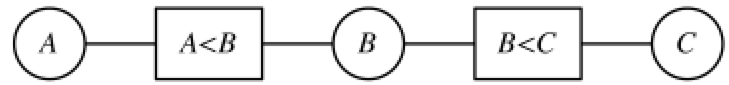
Also called “case analysis”

Split a variable’s domain into disjoint subsets, and consider them each separately

- If $\mathrm{dom}(X_i) = \{a_1, \dots, a_M\}$, then for each possible setting of $X_i = a_m$, find an assignment to all other variables that satisfy the constraints
- This is solved a **reduced** problem

Q: how does this relate to search?

Domain Splitting Example

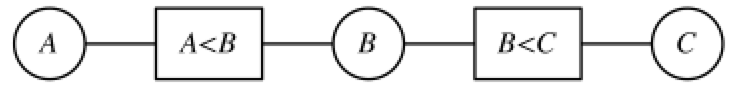


Setup: 3 variables (A, B, C) each with domain {1,2,3,4}

Constraints:
$A < B$
$B < C$

	A	B	C
Original Domains:	{1,2,3,4}	{1,2,3,4}	{1,2,3,4}
After arc consistency:	{1,2}	{2,3}	{3,4}
Domain Splitting:			
B=2	{1,~~2~~}	{2}	{3,4}
B=3	{1,2}	{3}	{~~3,~~4}

Domain Splitting Example



Setup: 3 variables (A, B, C) each with domain {1,2,3,4}

Constraints:
$A < B$
$B < C$

	A	B	C	
Original Domains:	{1,2,3,4}	{1,2,3,4}	{1,2,3,4}	
After arc consistency:	{1,2}	{2,3}	{3,4}	
Domain Splitting:				
B=2, C=3	{1,~~2~~}	{2}	{3,~~4~~}	✓
B=2, C=4	{1,~~2~~}	{2}	{~~3~~,4}	✓
B=3, A=1	{1,~~2~~}	{3}	{~~3~~,4}	✓
B=3, A=2	{~~1~~,2}	{3}	{~~3~~,4}	✓

Variable Elimination

- Simplify the network by incrementally removing variables
  - Remove a variable, and create a new constraint on the remaining variables to account for its removal

Variable Elimination Algorithm

```
procedure VE_CSP(Vs, Cs)
    Inputs
        Vs: a set of variables
        Cs: a set of constraints on Vs
    Output
        a relation containing all of the consistent variable assignments
    if Vs contains just one element then
        return the join of all the relations in Cs
    else
```

Variable Elimination Algorithm

```
procedure VE_CSP(Vs, Cs)
    Inputs
        Vs: a set of variables
        Cs: a set of constraints on Vs
    Output
        a relation containing all of the consistent variable assignments
    if Vs contains just one element then
        return the join of all the relations in Cs
    else
        select variable Xs to eliminate
```

Variable Elimination Algorithm

```
procedure VE_CSP(Vs, Cs)
    Inputs
        Vs: a set of variables
        Cs: a set of constraints on Vs
    Output
        a relation containing all of the consistent variable assignments
    if Vs contains just one element then
        return the join of all the relations in Cs
    else
        select variable Xs to eliminate
        Vs' := Vs \ {X}
```

Remove X from the set of variables

Variable Elimination Algorithm

procedure *VE_CSP*(Vs, Cs)
    Inputs
        Vs: a set of variables
        Cs: a set of constraints on Vs
    Output
        a relation containing all of the consistent variable assignments
    if Vs contains just one element **then**
        return the join of all the relations in Cs
    else
        select variable Xs to eliminate
        $\mathrm{Vs}' := \mathrm{Vs} \setminus \{X\}$
        $\mathrm{Cs}_X := \{T \in \mathrm{Cs} : T \text{ involves } X\}$

Identify the constraints involving X that need to be reformulated/accounted for

Variable Elimination Algorithm

procedure *VE_CSP*(Vs, Cs)
  Inputs
    Vs: a set of variables
    Cs: a set of constraints on Vs
  Output
    a relation containing all of the consistent variable assignments
  if Vs contains just one element **then**
    return the join of all the relations in Cs
  else
    select variable Xs to eliminate
    $\mathrm{Vs}' := \mathrm{Vs} \setminus \{X\}$
    $\mathrm{Cs}_X := \{T \in \mathrm{Cs} : T \text{ involves } X\}$
    let R be the join of all of the constraints in Cs_X
    let R' be R projected onto the variables other than X

Based on individual assignments to X, identify the set of allowed assignments to other variables in those constraints' scopes

Variable Elimination Algorithm

procedure *VE_CSP*(Vs, Cs)
    Inputs
        Vs: a set of variables
        Cs: a set of constraints on Vs
    Output
        a relation containing all of the consistent variable assignments
    if Vs contains just one element **then**
        return the join of all the relations in Cs
    else
        select variable Xs to eliminate
        $\mathrm{Vs}' := \mathrm{Vs} \setminus \{X\}$
        $\mathrm{Cs}_X := \{T \in \mathrm{Cs} : T \text{ involves } X\}$
        let R be the join of all of the constraints in Cs_X
        let R' be R projected onto the variables other than X
        $S := \mathrm{VE\_CSP}\,(\mathrm{Vs}', (\mathrm{Cs} \setminus \mathrm{Cs}_X) \cup \{R'\})$
        return $R \bowtie S$

Variable Elimination Example

Setup: 3 variables (A, B, C) each with domain {1,2,3,4}

Constraints:
$A < B$
$B < C$

A	B
1	2
1	3
1	4
2	3
2	4
3	4

B	C
1	2
1	3
1	4
2	3
2	4
3	4

Eliminate B

Variable Elimination Example

A	B
1	2
1	3
1	4
2	3
2	4
3	4

B	C
1	2
1	3
1	4
2	3
2	4
3	4

Setup: 3 variables (A, B, C) each with domain {1,2,3,4}

Initial Constraints:

$A < B$

$B < C$

Identify possible, legal combinations. Red rows are not feasible.

Variable Elimination Example

Setup: 3 variables (A, B, C) each with domain {1,2,3,4}

Initial Constraints:
$A < B$
$B < C$

A	B
1	2
1	3
1	4
2	3
2	4
3	4

B	C
1	2
1	3
1	4
2	3
2	4
3	4

A	B	C
1	2	3
1	2	4
1	3	4
2	3	4

Reformulate constraints/constraint table…

Variable Elimination Example

Setup: 3 variables (A, B, C) each with domain {1,2,3,4}

Initial Constraints:
$A < B$
$B < C$

A	C
1	3
1	4
2	4

Reformulate constraints/constraint table… into one that doesn't involve B, and solve the simpler problem

Characteristics of Variable Elimination

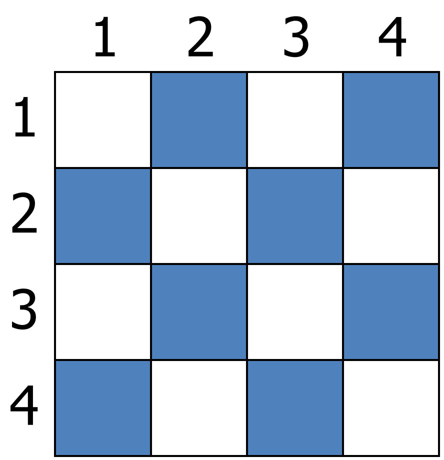
- Depends entirely on the tree-width
- Finding a good elimination order is NP-hard (!!!)
  – Heuristic 1: min-factor: select the variable that results in the smallest relation
  – Heuristic 2: minimum fill: select the variable that adds the fewest arcs to the resulting graph (don’t make the graph more complicated)

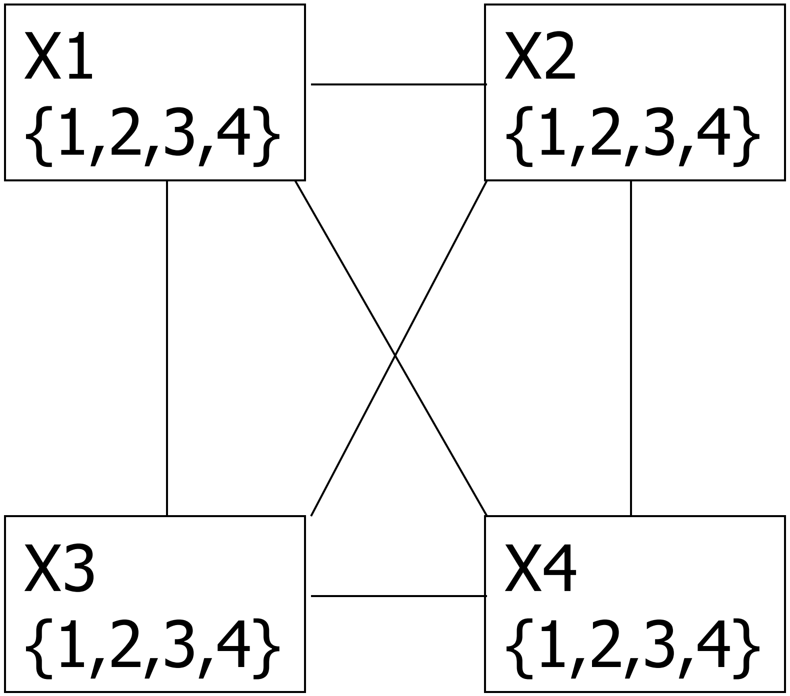
General Methods of Solving CSPs

- Generate-and-Test, aka Brute Force
- Search (backtracking)
- Consistency checking
  - Forward checking
  - Arc consistency
  - Domain splitting
  - Variable Elimination
- Localized search

Is AC3 Alone Sufficient?

Consider the four queens problem



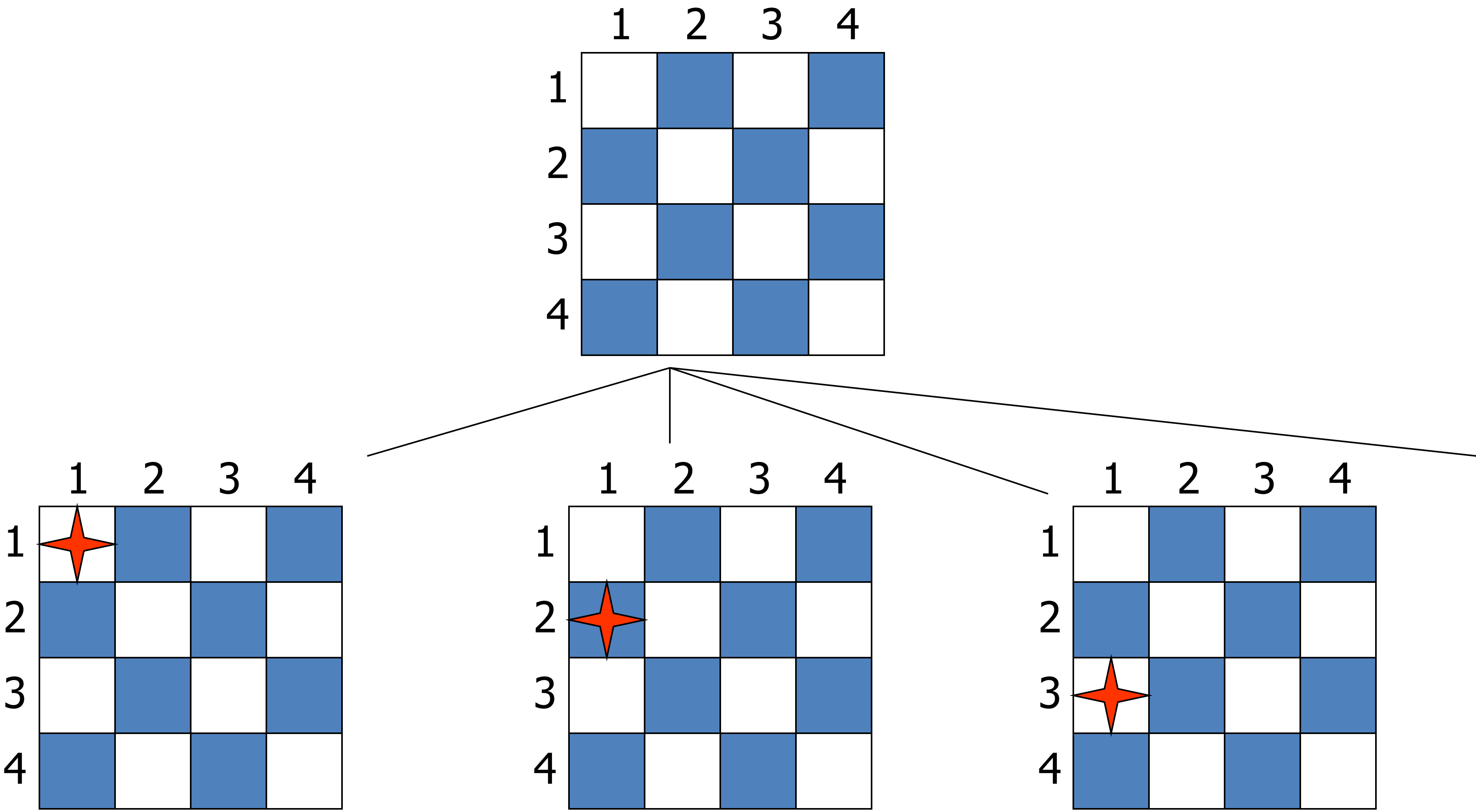


Solving a CSP still requires search

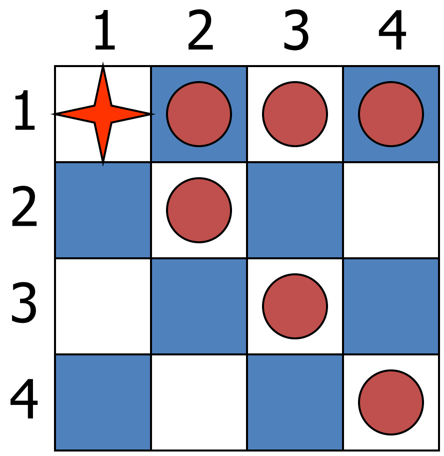
- Search:
  - can find good solutions, but must examine non-solutions along the way
- Constraint Propagation:
  - can rule out non-solutions, but this is not the same as finding solutions

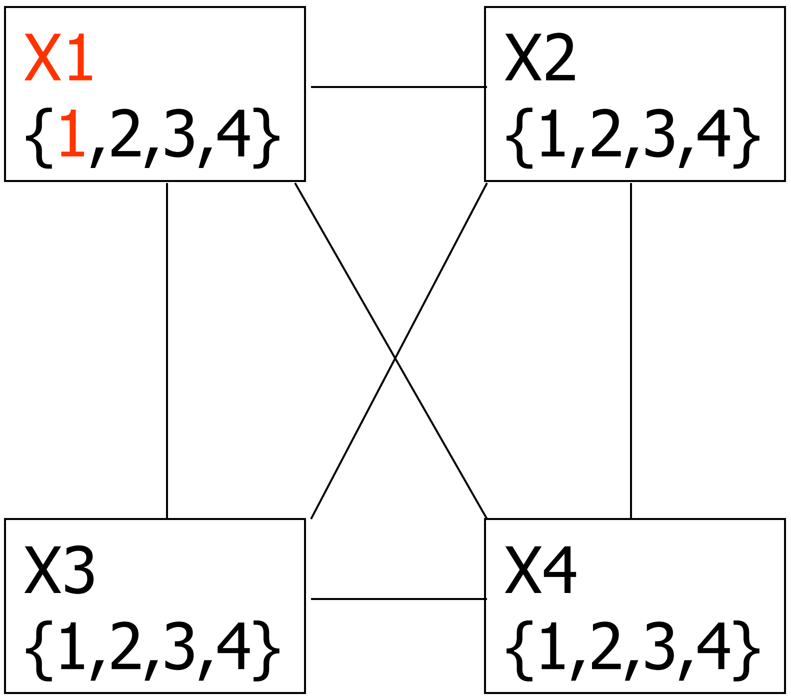
Solving a CSP still requires search

- 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

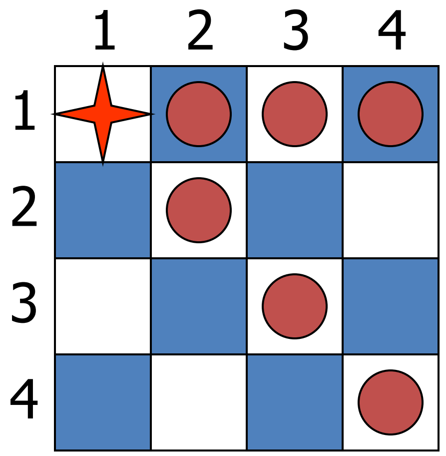

1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4

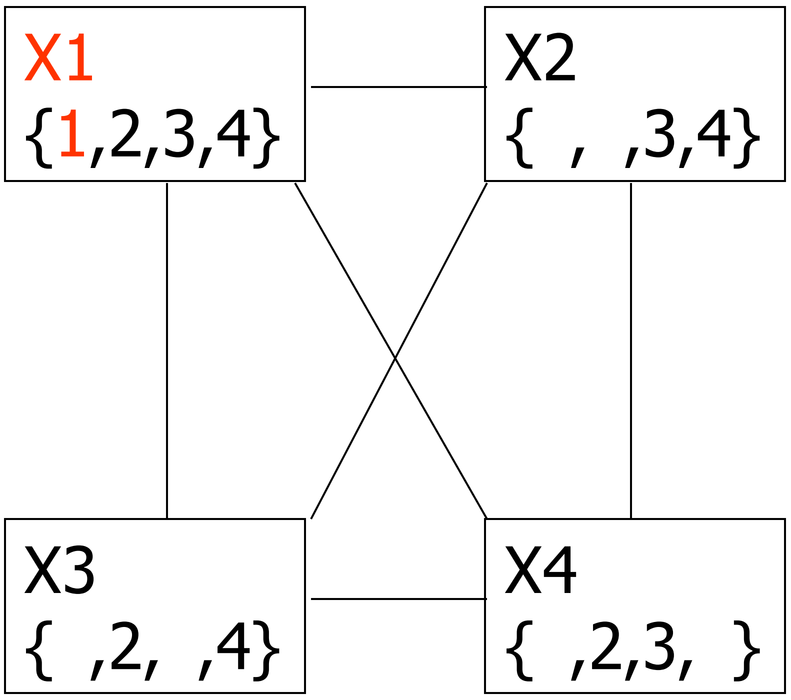
4-Queens Problem



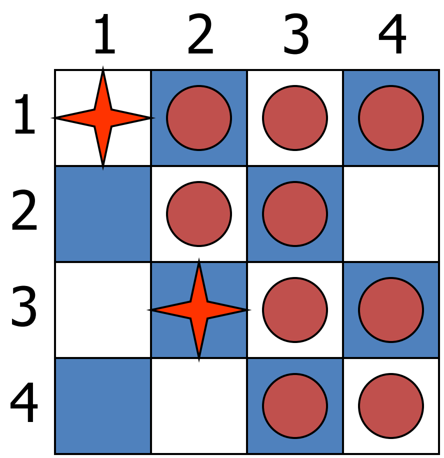


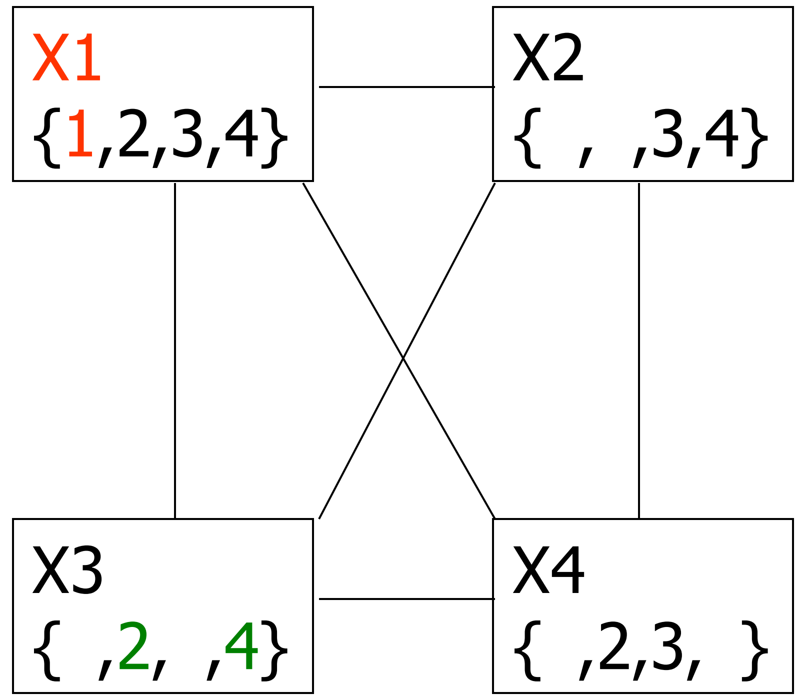
4-Queens Problem





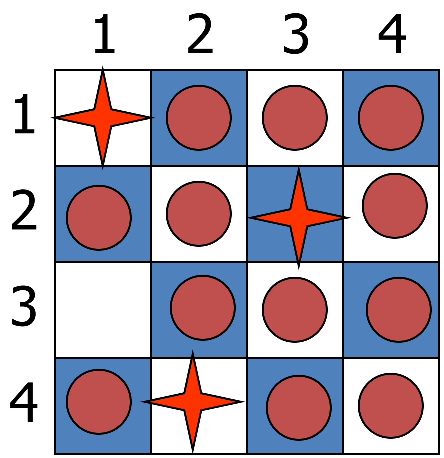
4-Queens Problem

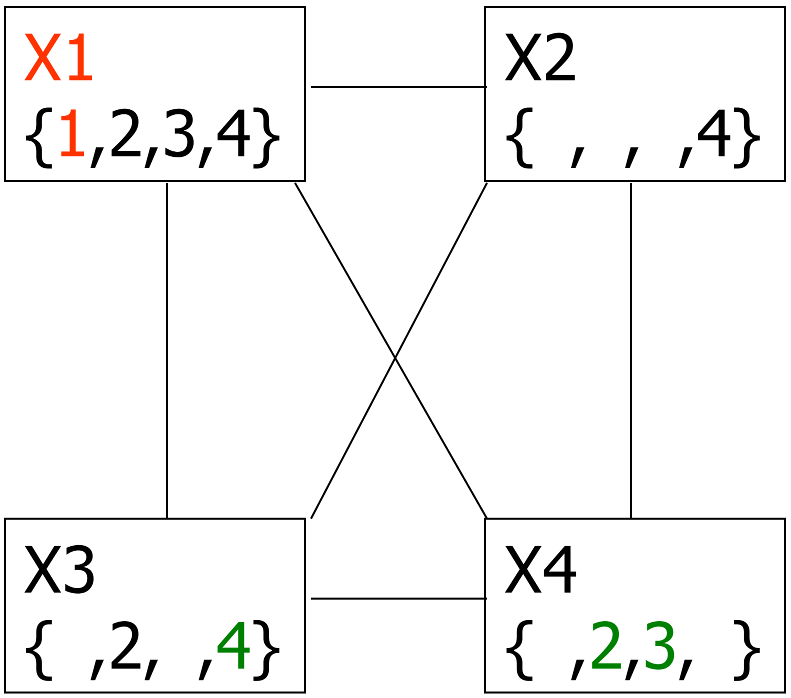




X2=3 eliminates { X3=2, X3=3, X3=4 }
⇒ inconsistent!

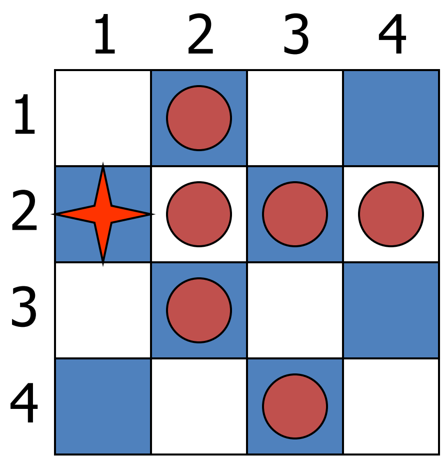
4-Queens Problem

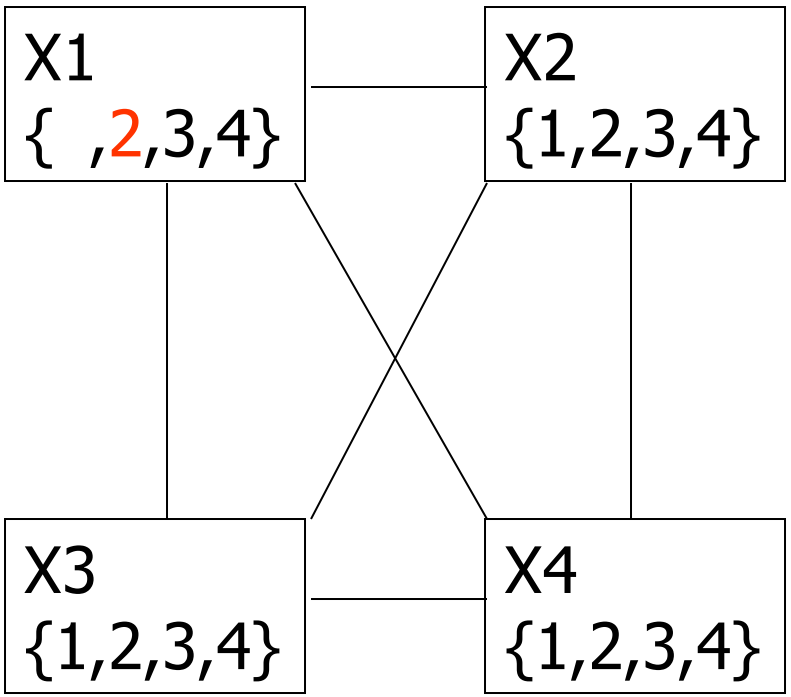




X2=4 ⇒ X3=2, which eliminates { X4=2, X4=3}
⇒ inconsistent!

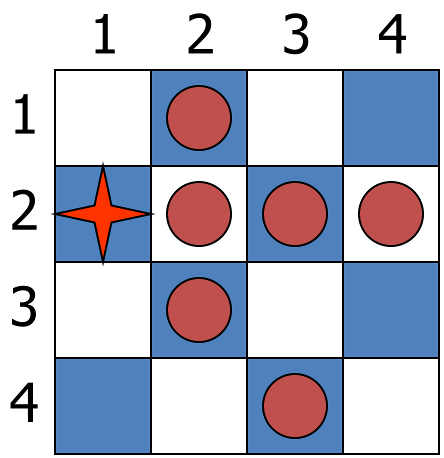
4-Queens Problem

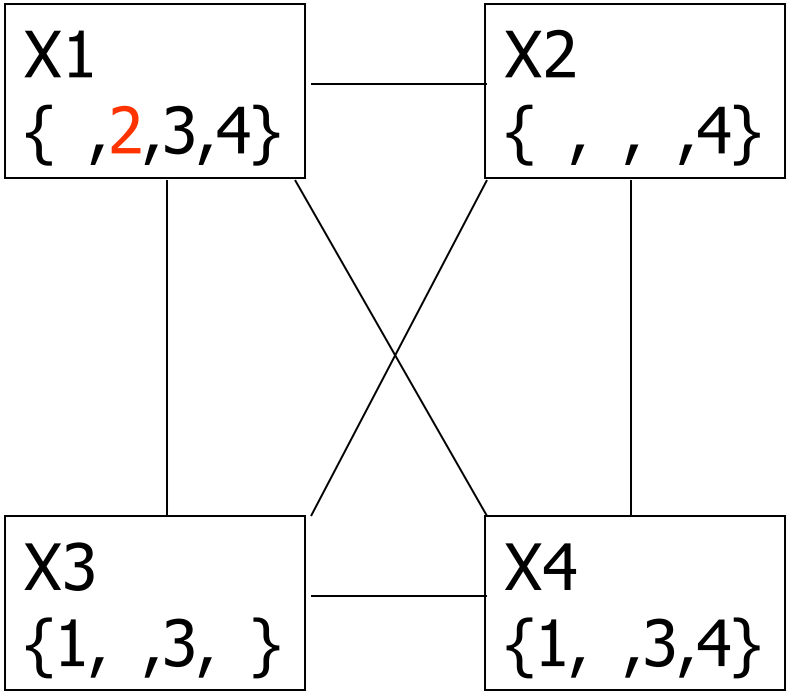




X1 can't be 1, let's try 2

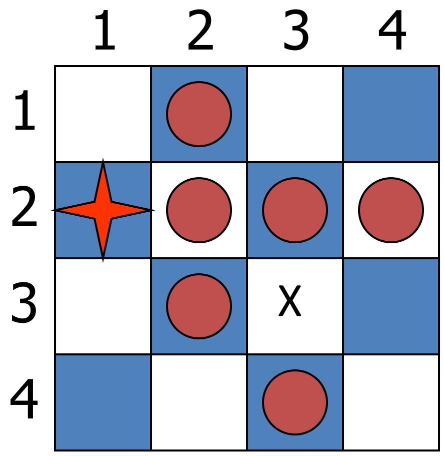
4-Queens Problem

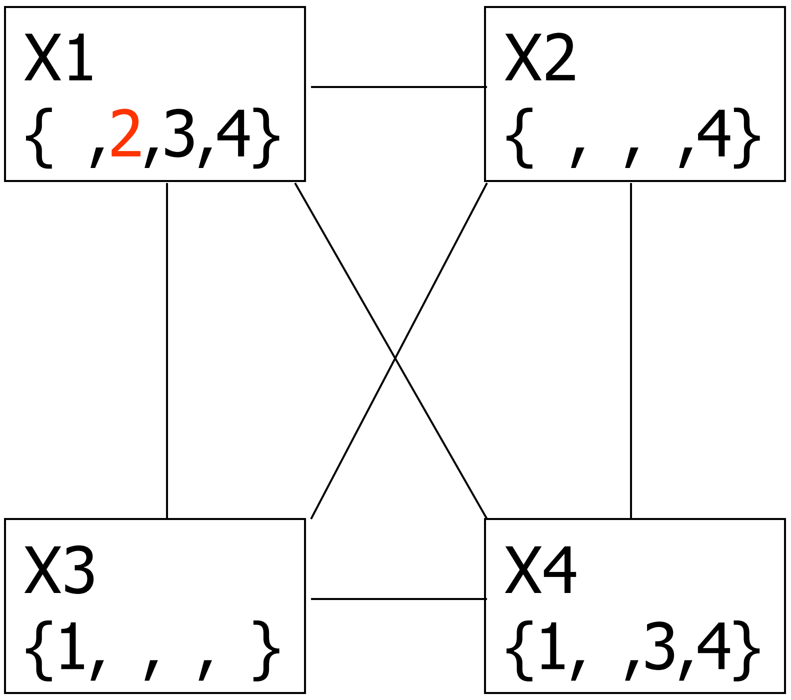




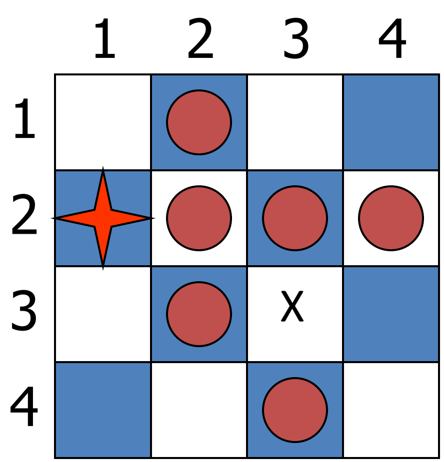
Can we eliminate any other values?

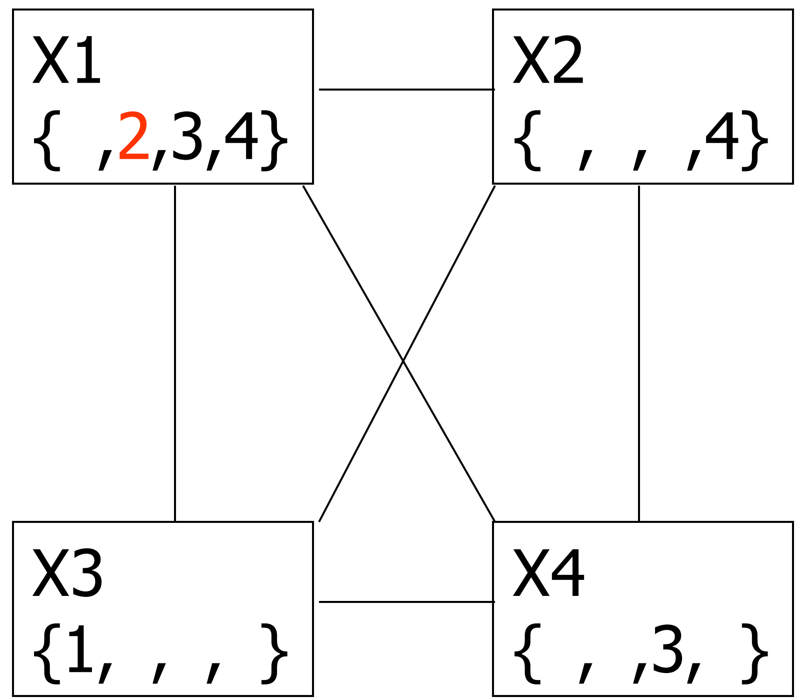
4-Queens Problem





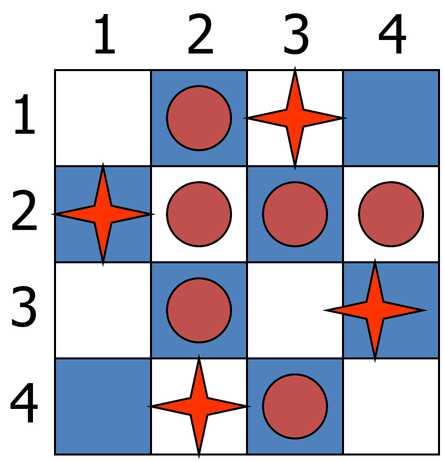
4-Queens Problem

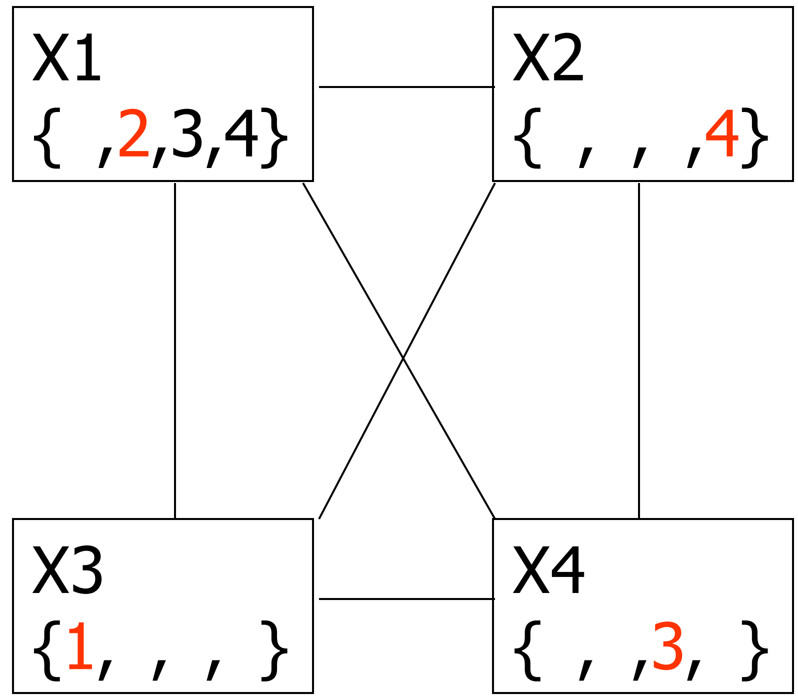




Arc constancy eliminates x3=3 because it's not consistent with X2's remaining values

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()
    # Define a variable for each cell: 11,12,13...21,22,23...98,99
    for i in range(1, 10) :
        p.addVariables(range(i*10+1, i*10+10), range(1, 10))
    # Each row has different values
    for i in range(1, 10) :
        p.addConstraint(AllDifferentConstraint(), range(i*10+1, i*10+10))
    # Each column has different values
    for i in range(1, 10) :
        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])

    # add unary constraints for cells with initial non-zero values
    for i in range(1, 10) :
        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()
```

```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]]
```

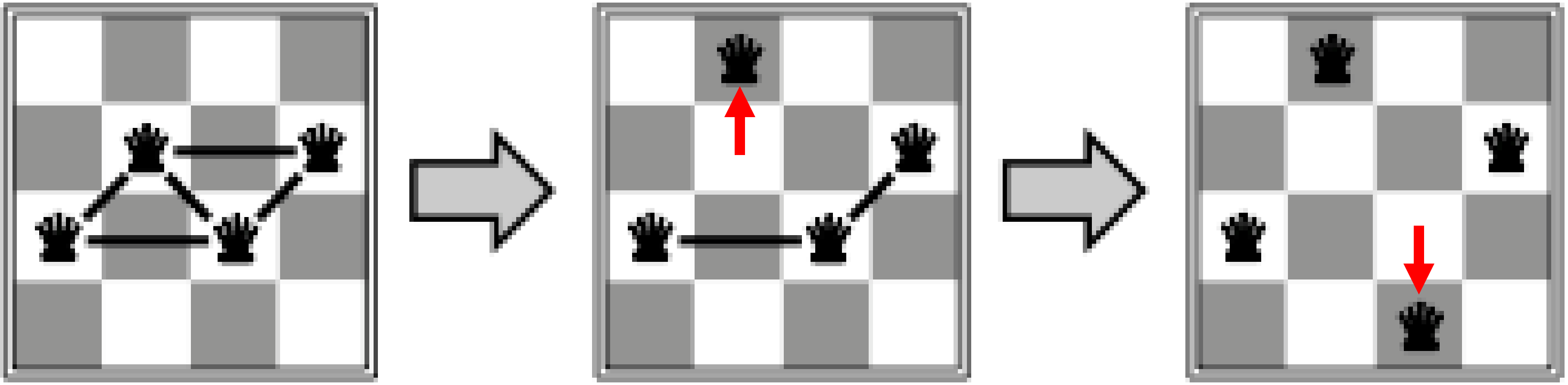
Local search for constraint problems

- Basic idea:
  - generate a random “solution”
  - Use metric of “number of conflicts”
  - 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 where 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]

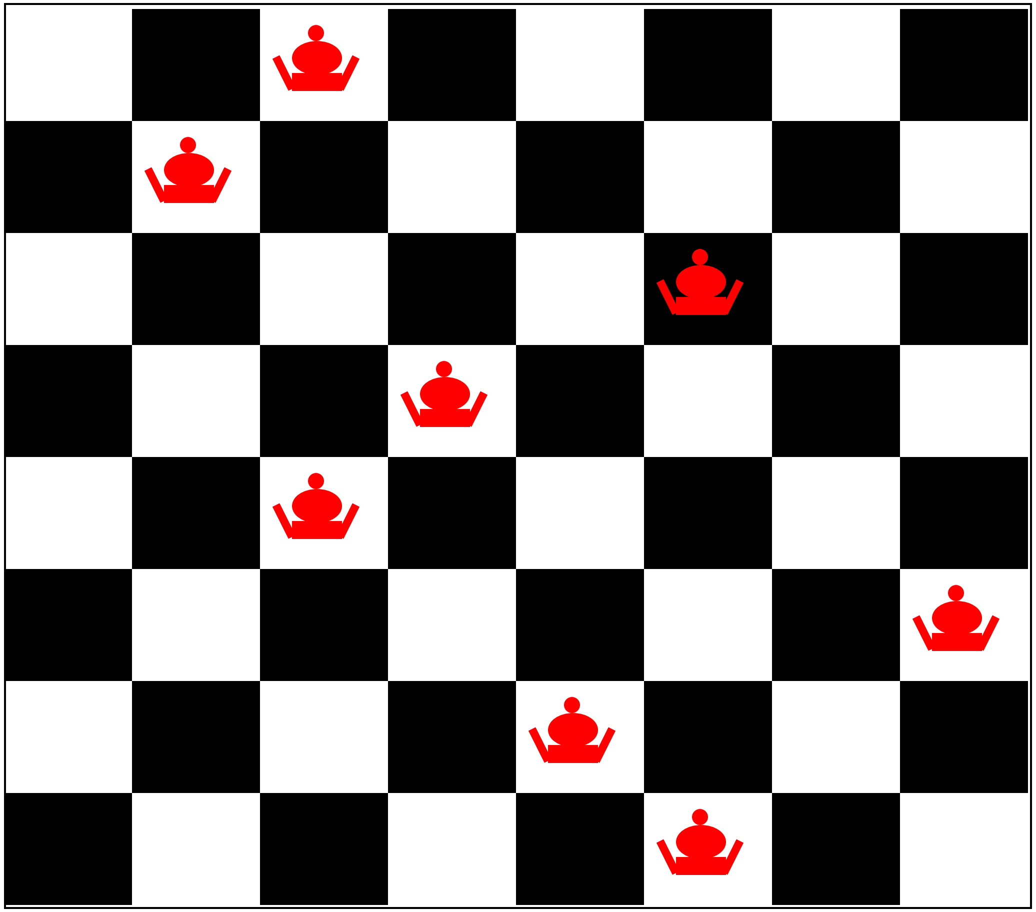
else if Cost(d_i) = bestCost

bestList $\leftarrow$ bestList $\cup$ d_i

Take a randomly selected move from bestList

Eight Queens using Local Search

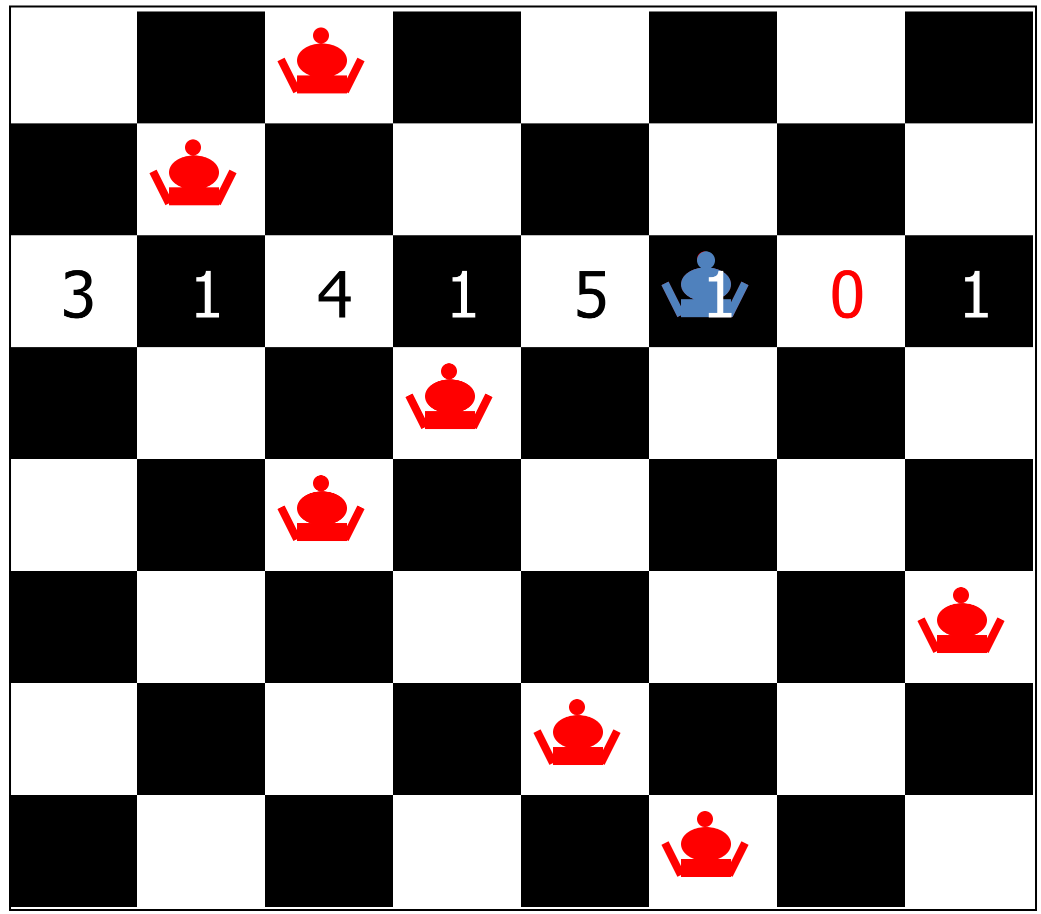
Place 8 Queens randomly on the board



Eight Queens using Local Search

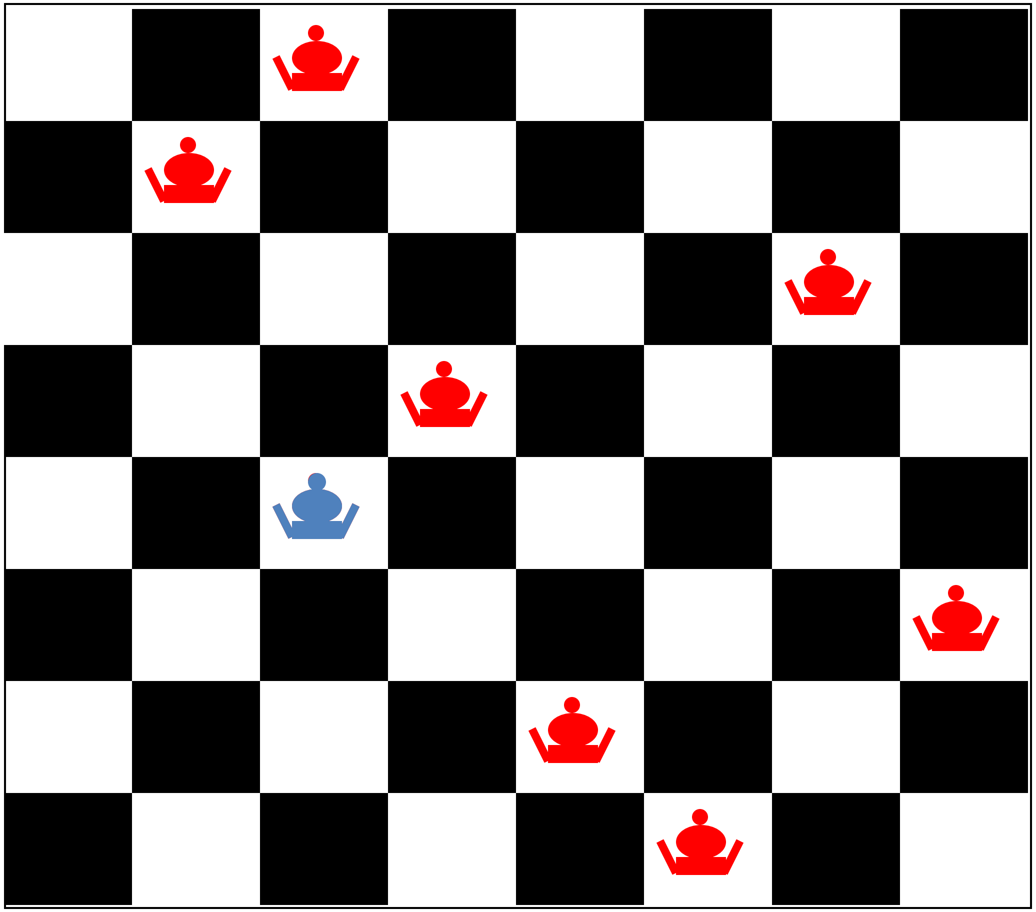
Pick a Queen:

Calculate cost of each move



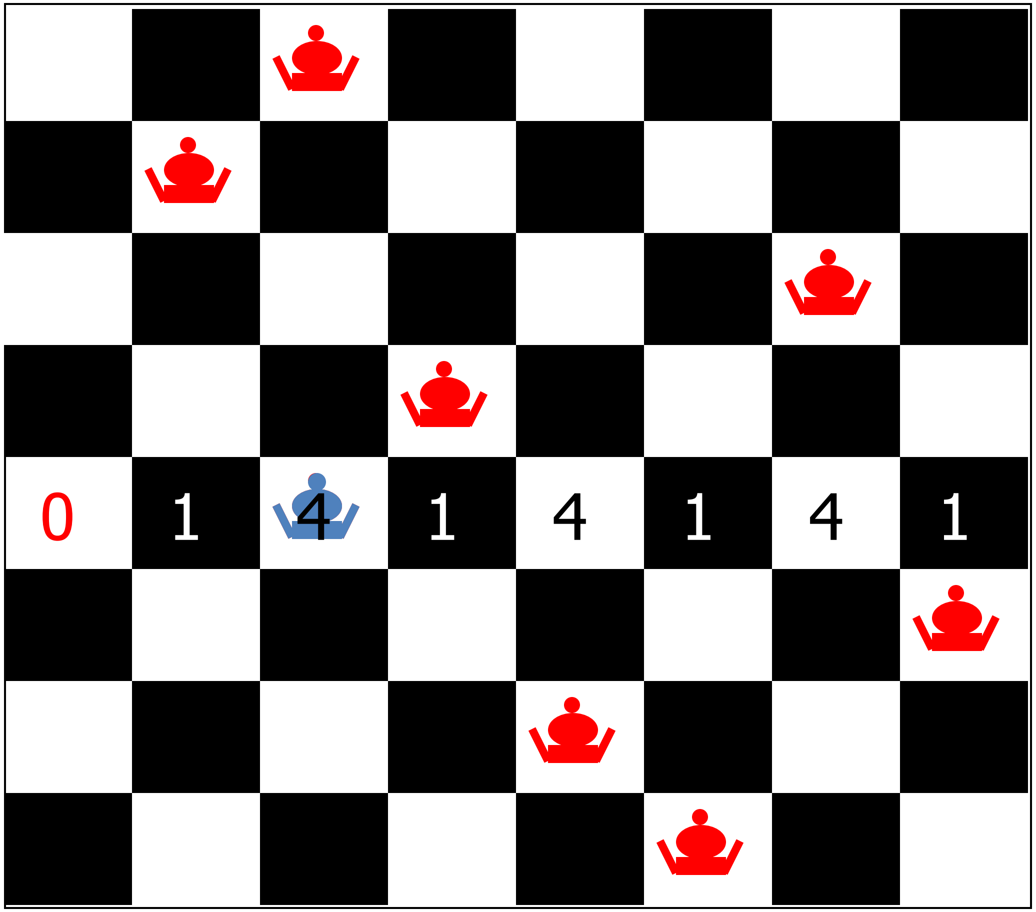
Eight Queens using Local Search

Take least cost move then try another Queen



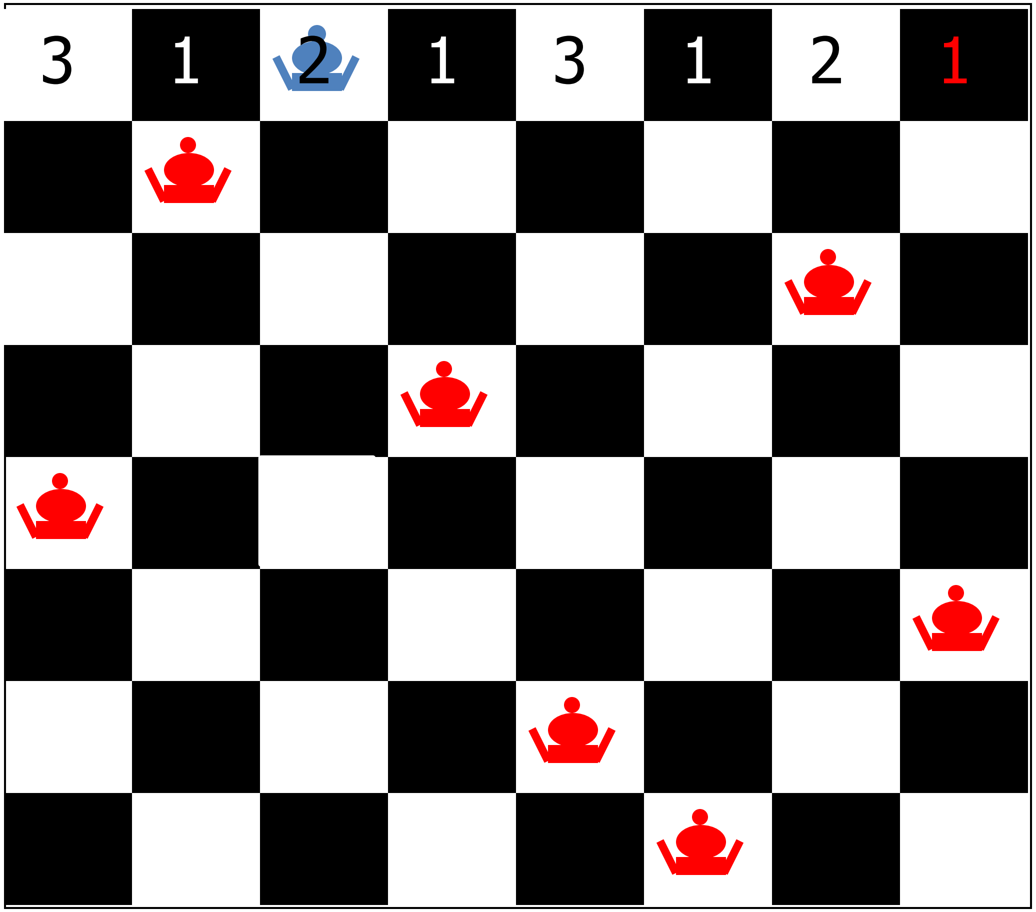
Eight Queens using Local Search

Take least cost move then try another Queen



Eight Queens using Local Search

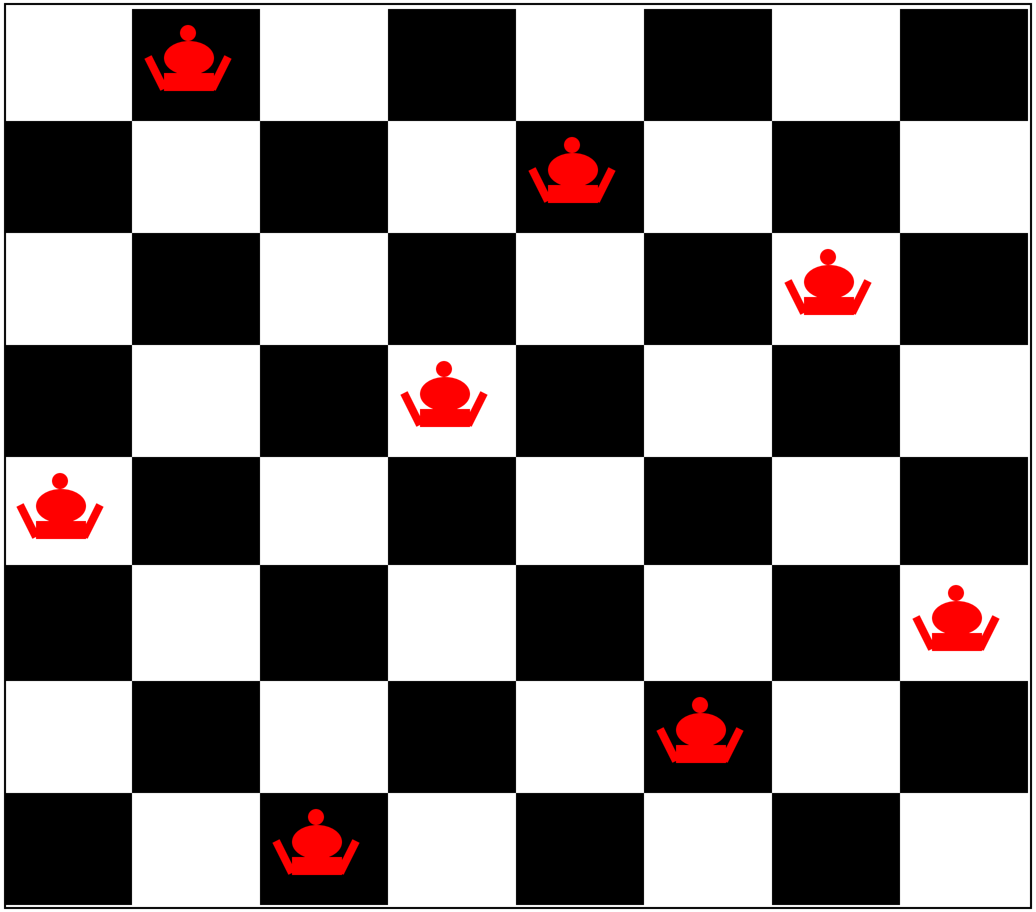
Take least cost move then try another Queen



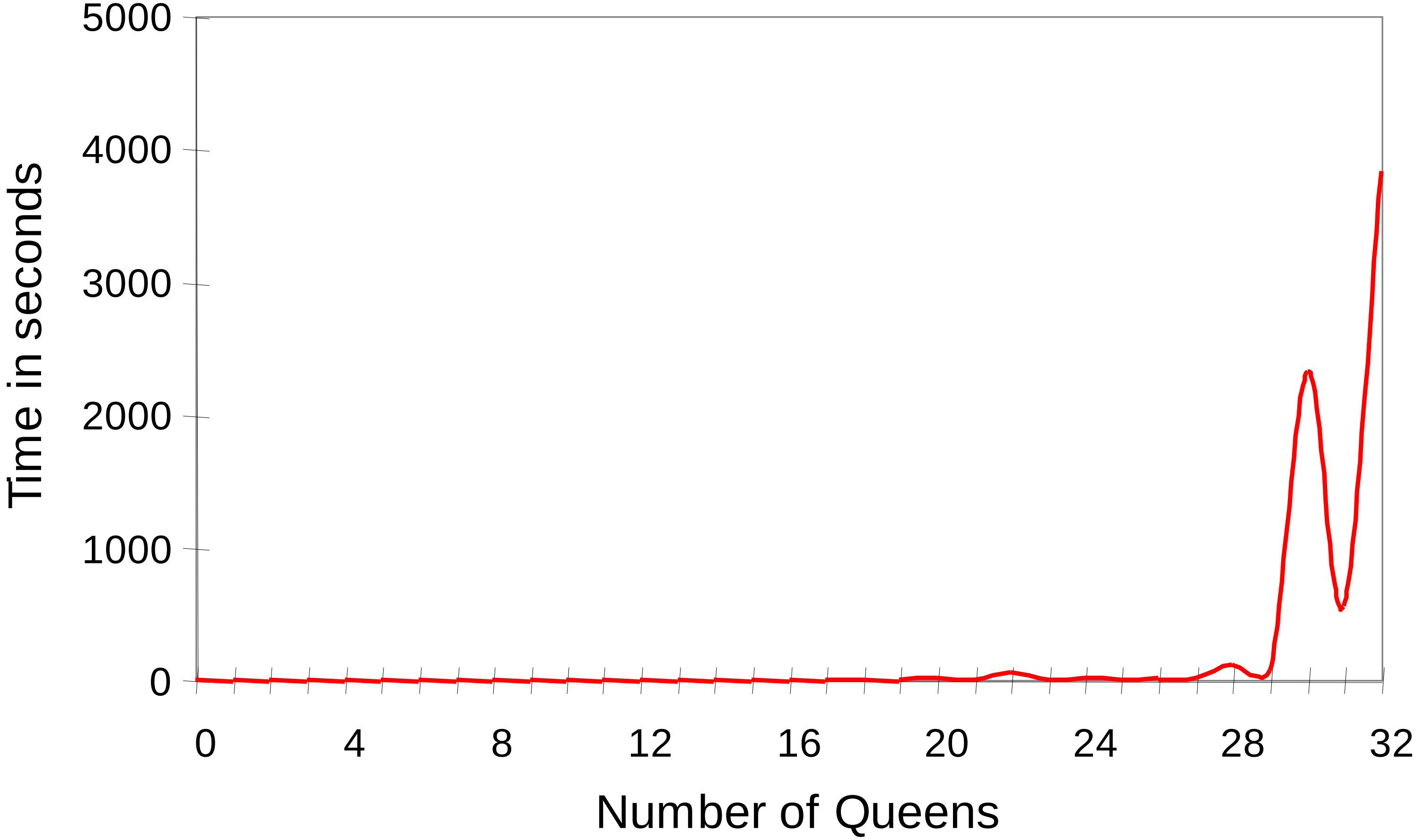
…and so on, until….

Eight Queens using Local Search

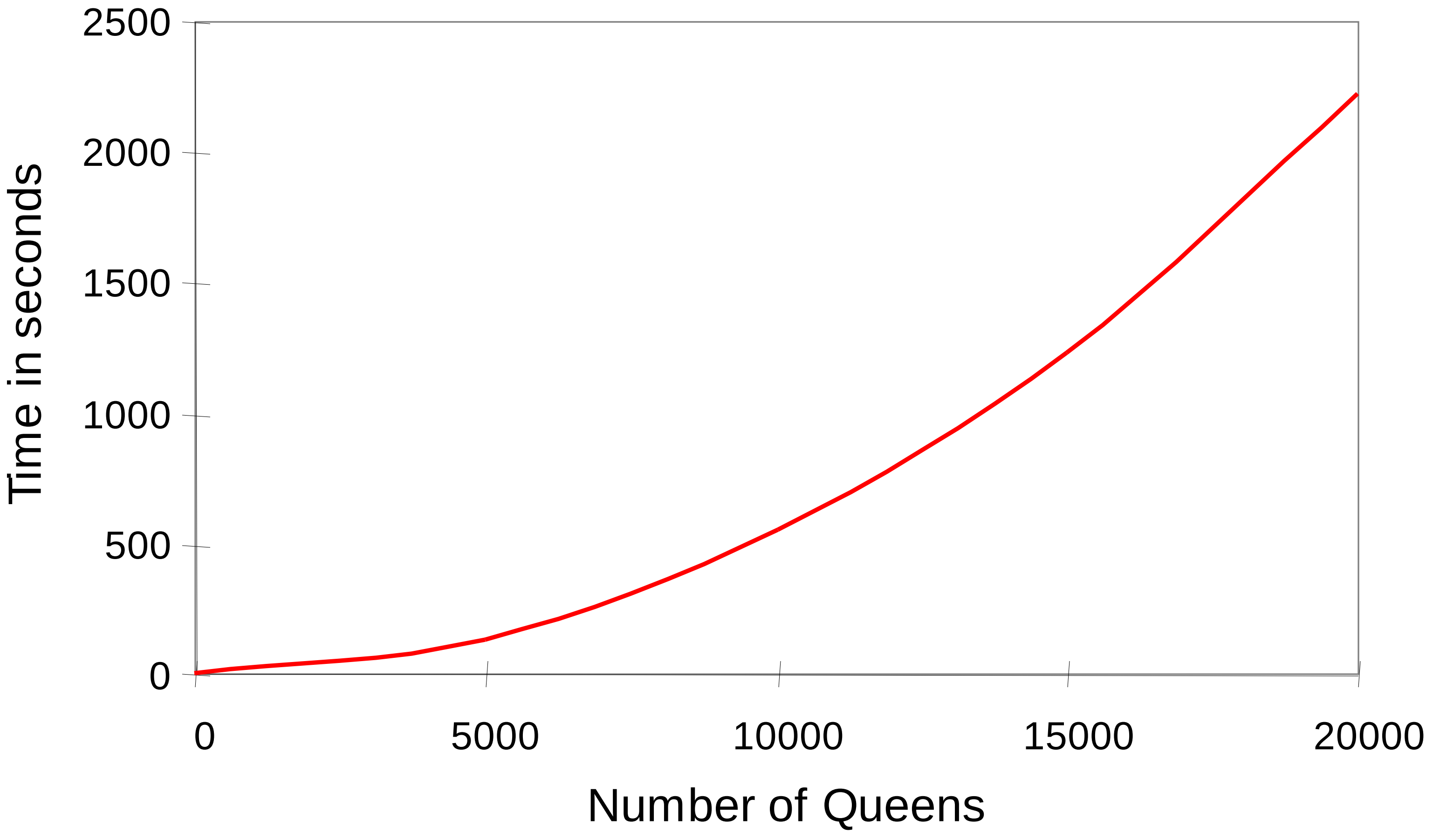
Answer Found



Backtracking Performance



Local Search Performance

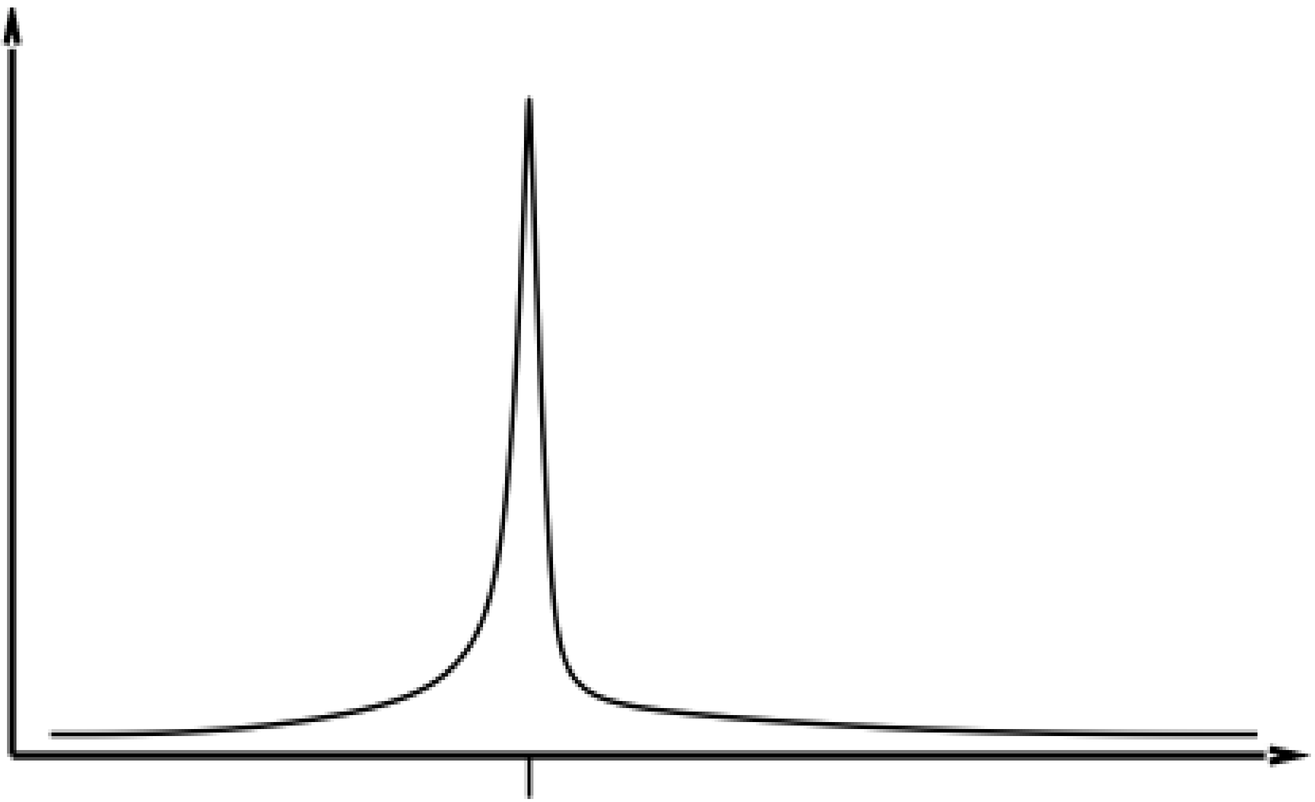


Min Conflict Performance

- 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…

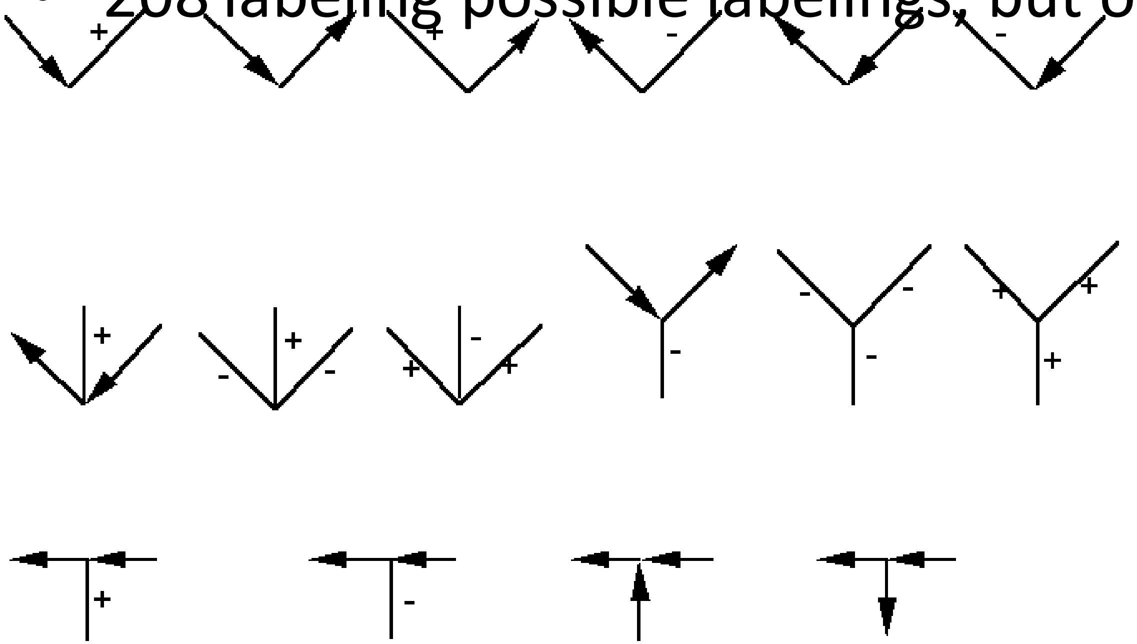
Min Conflict Performance

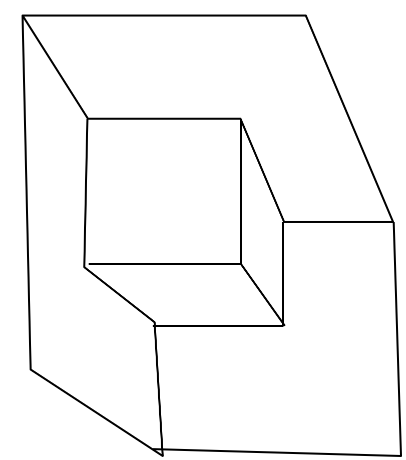
Except in a certain critical range of the ratio constraints to variables.



Famous example: labeling line drawings

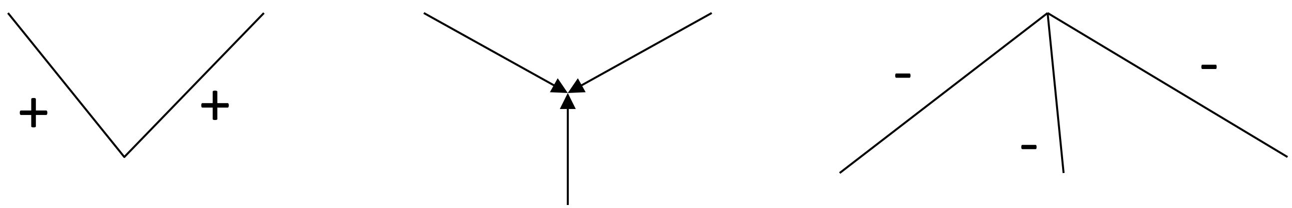
- Waltz labeling algorithm, earliest AI CSP application (1972)
  - Convex interior lines labeled as +
  - Concave interior lines labeled as –
  - Boundary lines labeled as with background to left
- 208 labeling possible labelings, but only 18 are legal





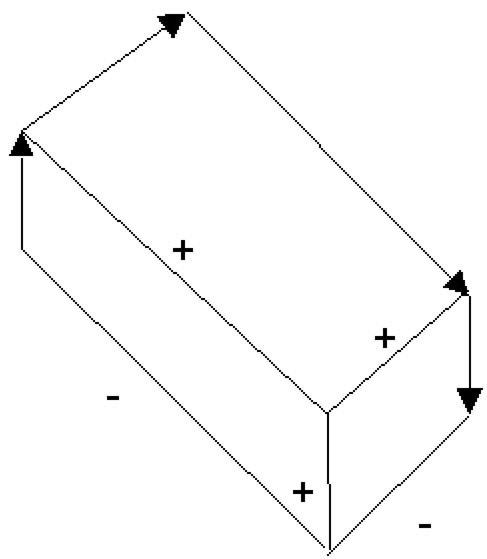
Labeling line drawings II

Here are some illegal labelings

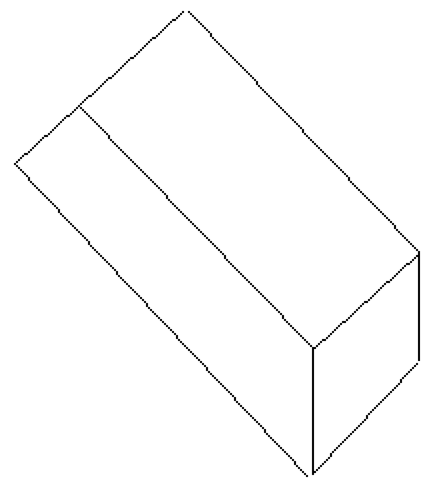


Labeling line drawings

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



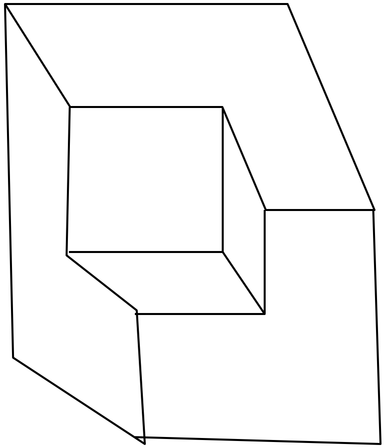
solution for one labeling problem



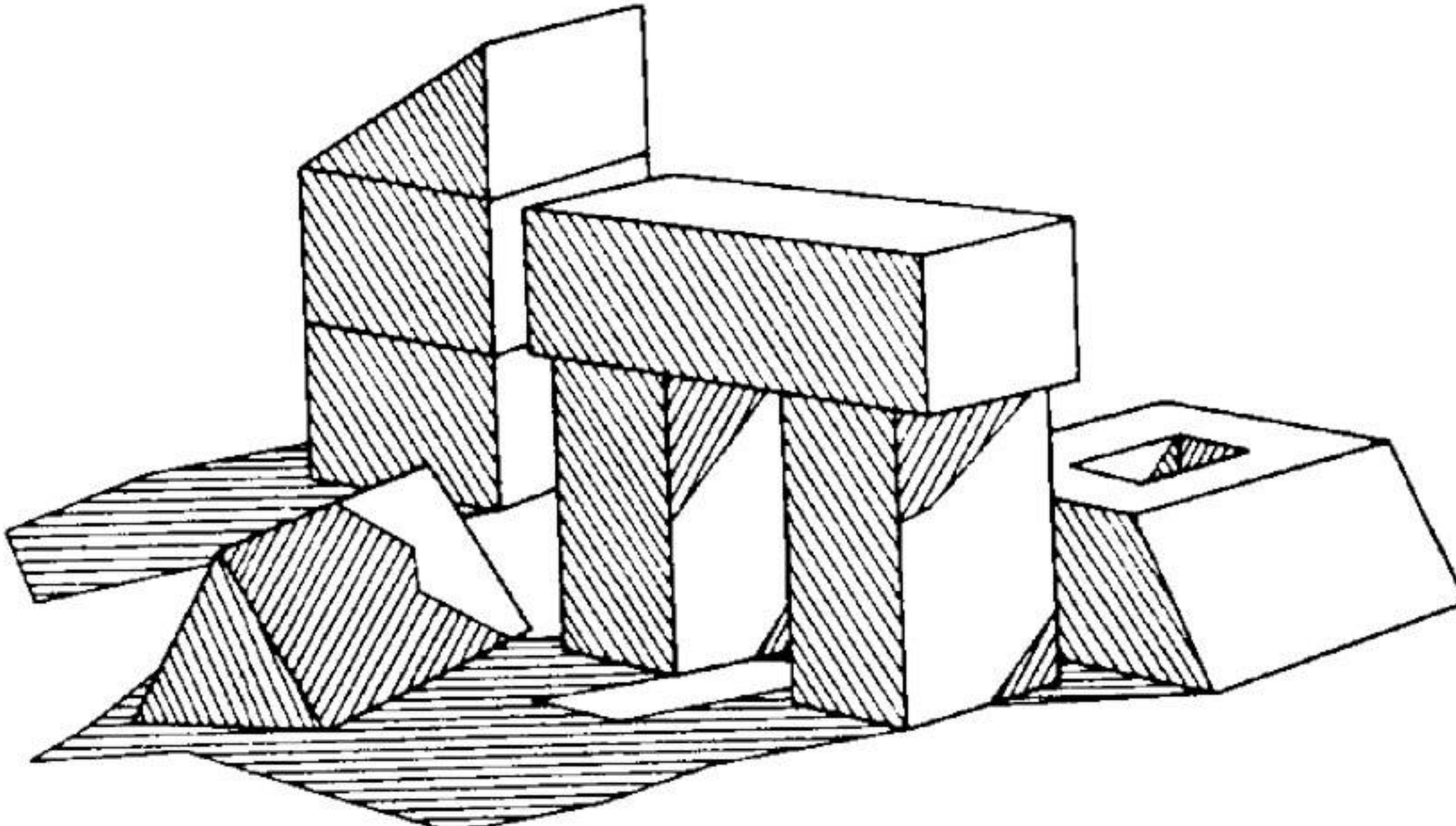
labeling problem with no solution

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

Challenges for constraint reasoning

- What if constraints are represented intentionally?
  - Cost of evaluating constraints (time, memory, resources)
- What if constraints, variables, and/or values change over time?
  - Dynamic constraint networks
  - Temporal constraint networks
  - Constraint repair
- What if multiple agents or systems are involved in constraint satisfaction?
  - Distributed CSPs
  - Localization techniques