

# COMP3411/9814: Artificial Intelligence

## Constraint Satisfaction Problems

# Lecture Overview

---

- ❑ Constraint Satisfaction Problems (CSPs)
- ❑ CSP examples
- ❑ Backtracking search and heuristics
- ❑ Forward checking and arc consistency
- ❑ Variable elimination
- ❑ Domain splitting and arc consistency
- ❑ Local search
  - Hill climbing
  - Simulated annealing

# Constraint Satisfaction Problems (CSPs)

---

- ❑ **Constraint Satisfaction Problems** are defined by a set of **variables**  $X_i$ , each with a **domain**  $D_i$  of possible values, and a set of **constraints**  $C$  that specify allowable combinations of values.
  
- ❑ The aim is to find an **assignment** of the variables  $X_i$  from the domains  $D_i$  in such a way that none of the constraints  $C$  are violated.
  - i.e. all of the constraints  $C$  are satisfied

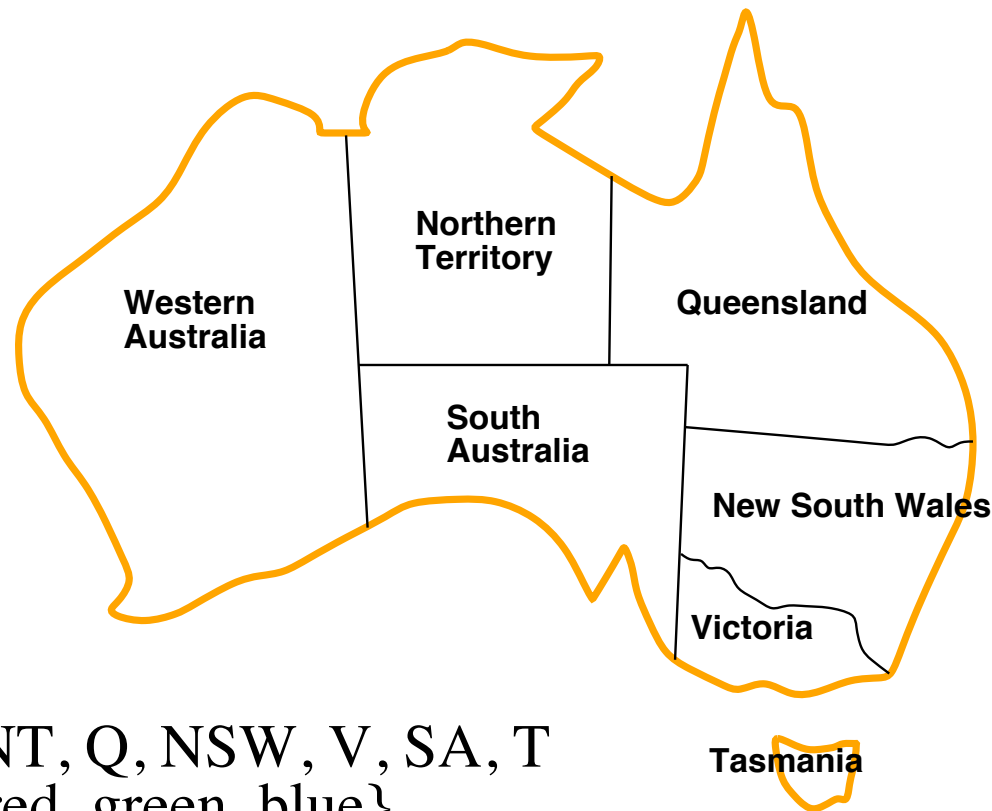
# Constraint Satisfaction Problems (CSPs)

---

- ❑ **Constraint Satisfaction Problems** are defined by a set of **variables**  $X_i$ , each with a **domain**  $D_i$  of possible values, and a set of **constraints**  $C$ .
- ❑ The aim is to find an **assignment** of the variables  $X_i$  from the domains  $D_i$  in such a way that none of the constraints  $C$  are violated.
  - i.e. all of the constraints  $C$  are satisfied

# Example: Map-Coloring

---



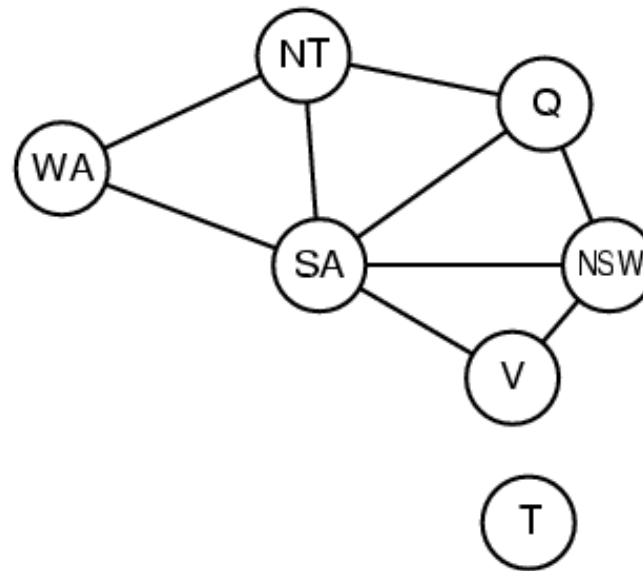
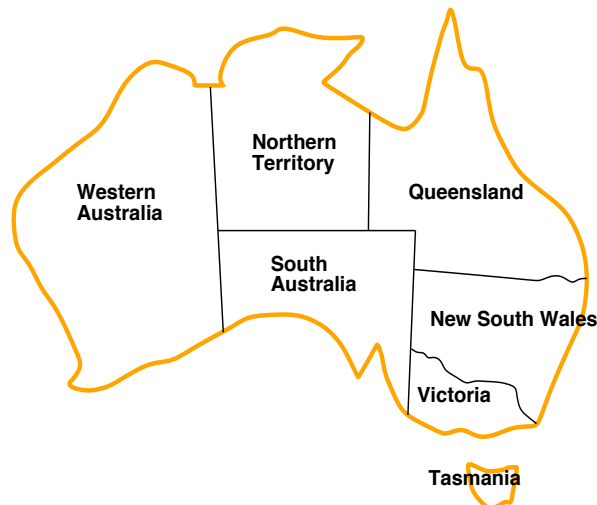
**Variables:** WA, NT, Q, NSW, V, SA, T

**Domains:**  $D_i = \{\text{red, green, blue}\}$

**Constraints:** adjacent regions must have different colors  
e.g.  $WA \neq NT$ , etc.

# Constraint graph

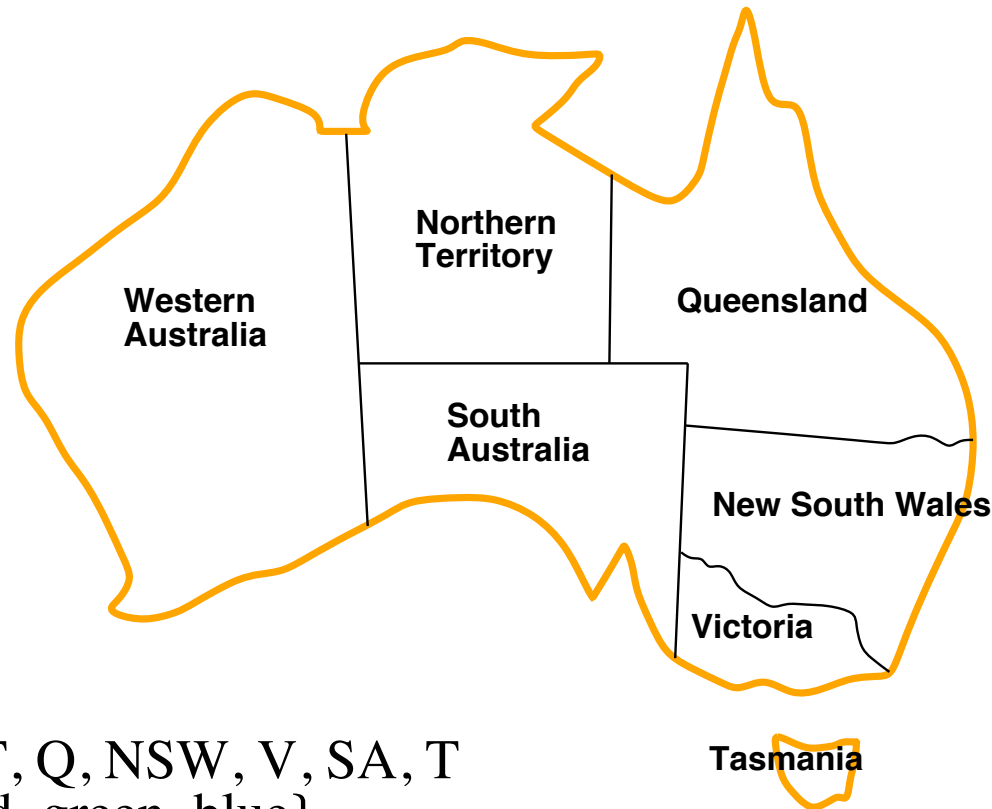
**Constraint graph:** nodes are variables, arcs are constraints



**Binary CSP:** each constraint relates two variables

# Example: Map-Coloring

---



**Variables:** WA, NT, Q, NSW, V, SA, T

**Domains:**  $D_i = \{\text{red, green, blue}\}$

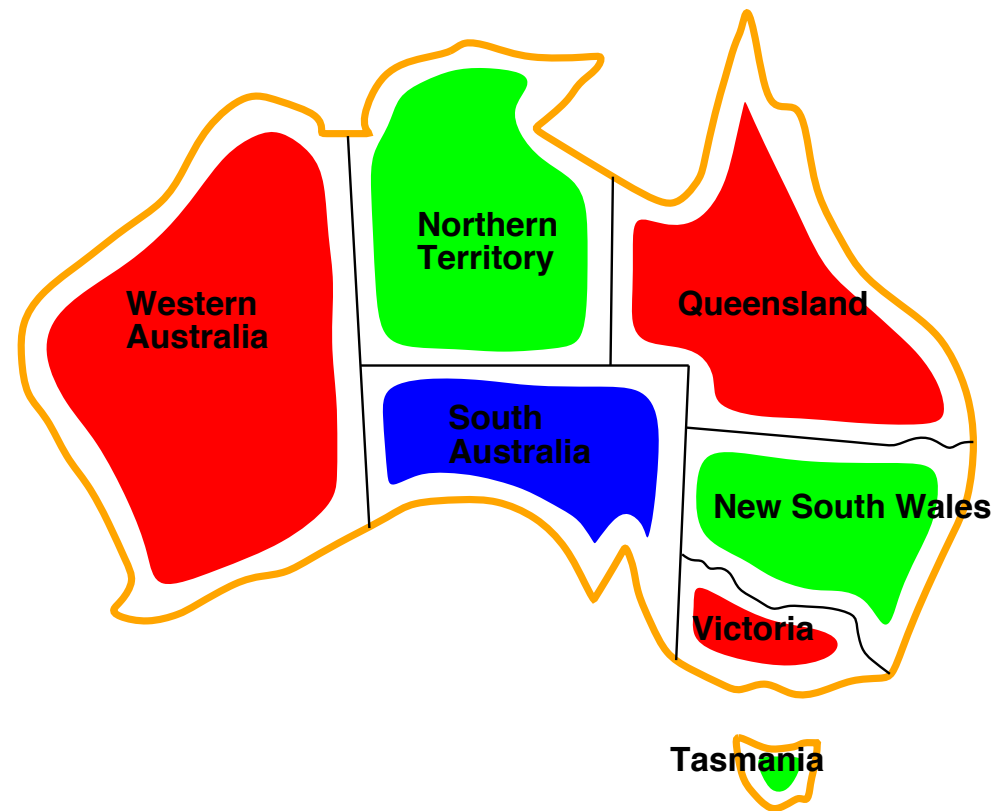
**Constraints:** adjacent regions must have different colors

e.g.  $WA \neq NT$ , etc.

or  $(WA, NT) \in \{(\text{red}, \text{green}), (\text{red}, \text{blue}), (\text{green}, \text{red}), (\text{green}, \text{blue}), (\text{blue}, \text{red}), (\text{blue}, \text{green})\}$

# Example: Map-Coloring

---



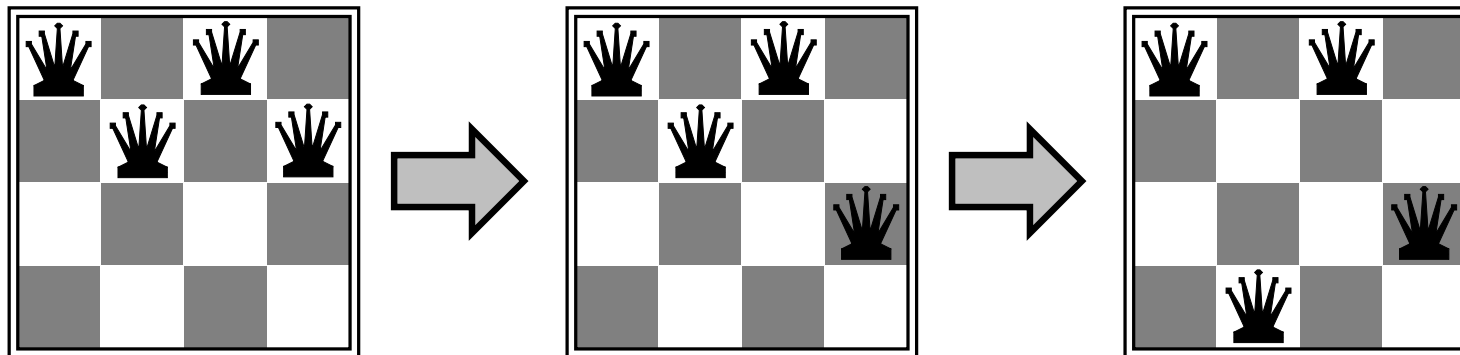
{WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green}



# n-Queens Puzzle as a CSP

Assume one queen in each column. Domains are possible positions of queen in a column. Assignment is when each domain has one element.

Which row does each one go in?



**Variables:**  $Q1, Q2, Q3, Q4$

**Domains:**  $D_i = \{1, 2, 3, 4\}$

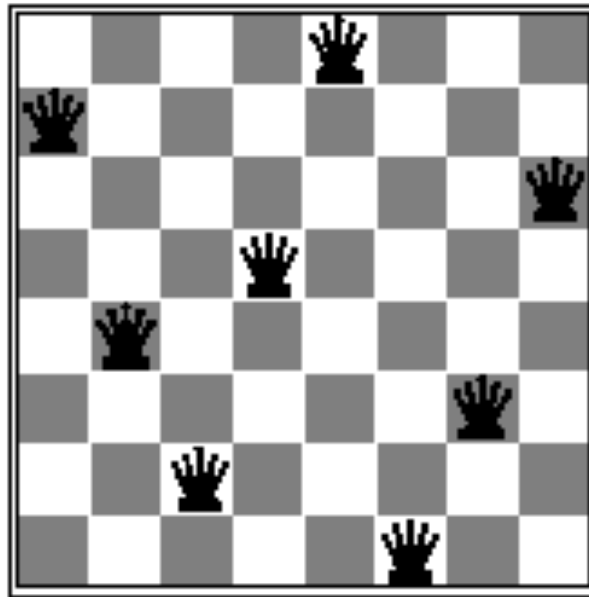
**Constraints:**

$Q_i \neq Q_j$  (cannot be in same row)

$|Q_i - Q_j| \neq |i - j|$  (or same diagonal)

# Example: n-Queens Puzzle

---



Put  $n$  queens on an  $n$ -by- $n$  chess board so that no two queens are attacking each other.

# Example: Cryptarithmic

---

$$\begin{array}{r} \phantom{+} S \phantom{+} E \phantom{+} N \phantom{+} D \\ + M \phantom{+} O \phantom{+} R \phantom{+} E \\ \hline M \phantom{+} O \phantom{+} N \phantom{+} E \phantom{+} Y \end{array}$$

Variables:

D E M N O R S Y

Domains:

$\{0,1,2,3,4,5,6,7,8,9\}$

Constraints:

$M \neq 0, S \neq 0$  (unary constraints)

$Y = D+E$  or  $Y = D+E - 10$ , etc.

$D \neq E, D \neq M, D \neq N$ , etc.

# Example: Cryptarithmic

---

$$\begin{array}{r} \text{T W O} \\ + \text{T W O} \\ \hline \text{F O U R} \end{array}$$

**Variables:** F T U W R O  $X_1$   $X_2$   $X_3$

**Domains:** {0,1,2,3,4,5,6,7,8,9}

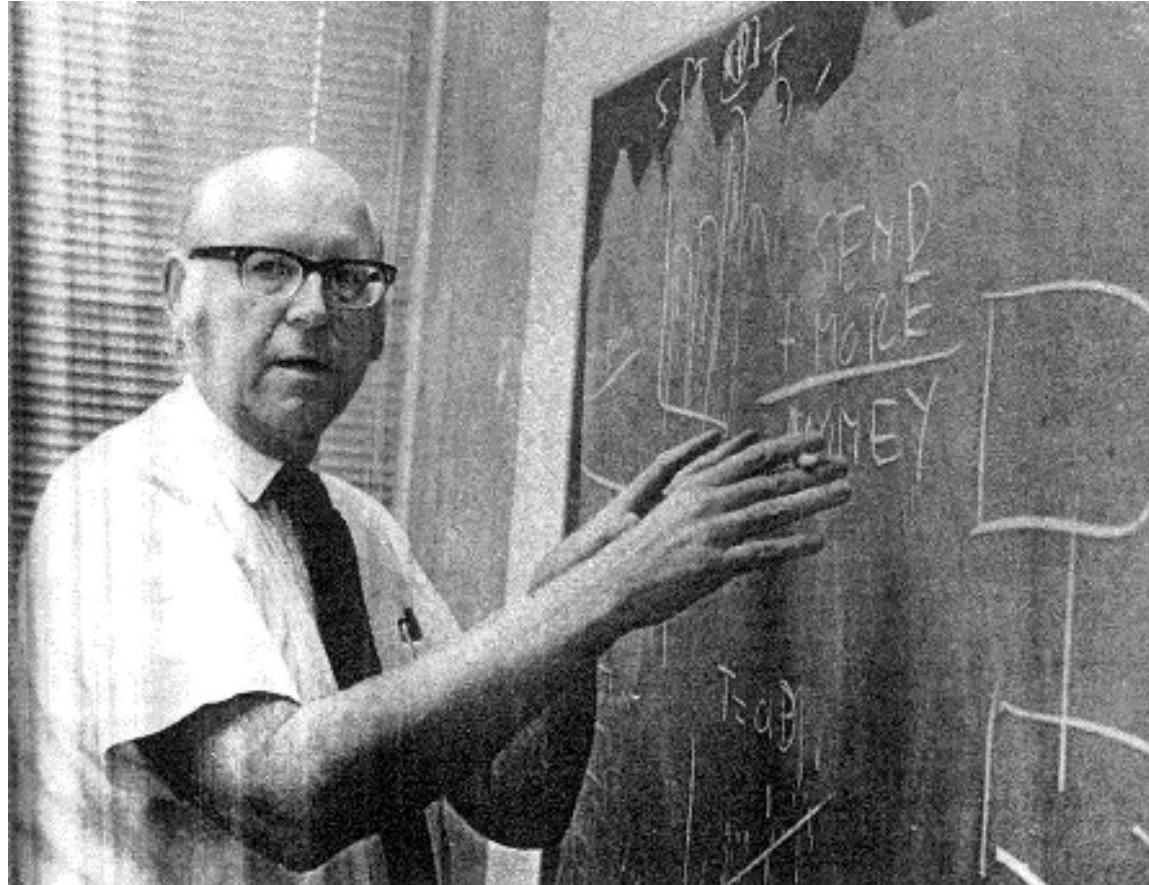
**Constraints:**

AllDifferent(F,T,U,W,R,O)

$O + O = R + 10 \cdot X_1$ , etc.

# Cryptarithmic with Allen Newell

---



Book: Intended rational Behavior

# Cryptarithmic with Allen Newell

---

## 7.1. Cryptarithmic

Let us start with **cryptarithmic**. This task was first analyzed by Bartlett (1958) and later by Herb Simon and myself (**Newell** & Simon, 1972). It plays an important role in the emergence of cognitive psychology—at least for me, and perhaps for others. It has been the strongest convincer that humans really do use problem spaces and do search in them, just as the AI theory of heuristic search says.

A **cryptarithmic** task is just a small arithmetical puzzle (see Figure 7-1). The words *DONALD*, *GERALD*, and *ROBERT* represent three six-digit numbers. Each letter is to be replaced by a distinct digit (that is, *D* and *T* must each be a digit, say  $D = 5$  and  $T = 0$ , but they cannot be the same digit). This replacement must lead to a correct sum, such that  $DONALD + GERALD = ROBERT$ . Mathematically viewed, the problem is one of satisfying multiple integer constraints involving equality, inequality, and un-equality.

Humans can be set to solving **cryptarithmic** tasks, and pro-

# Cryptarithmic with Allen Newell

---

Intendedly Rational Behavior ■ 365

Assign each letter a unique digit to make a correct sum

$$\begin{array}{r} \text{DONALD} \\ + \text{GERALD} \\ \hline \text{ROBERT} \end{array} \quad D = 5$$

Figure 7-1. The cryptarithmic task.

protocols can be obtained from transcripts of their verbalizations while they work (Newell & Simon, 1972). Analysis shows that people solve the task by searching in a problem space and that the search can be plotted explicitly.<sup>1</sup> Figure 7-2 shows the behavior of S3 on *DONALD + GERALD = ROBERT*.<sup>2</sup> The plot is called a *problem-behavior graph (PBG)*; it is just a way of spreading out the search so it can be examined (Figure 1-4 showed a PBG for chess). S3

# Cryptarithm etic with Allen Newell

## 366 ■ Unified Theories of Cognition

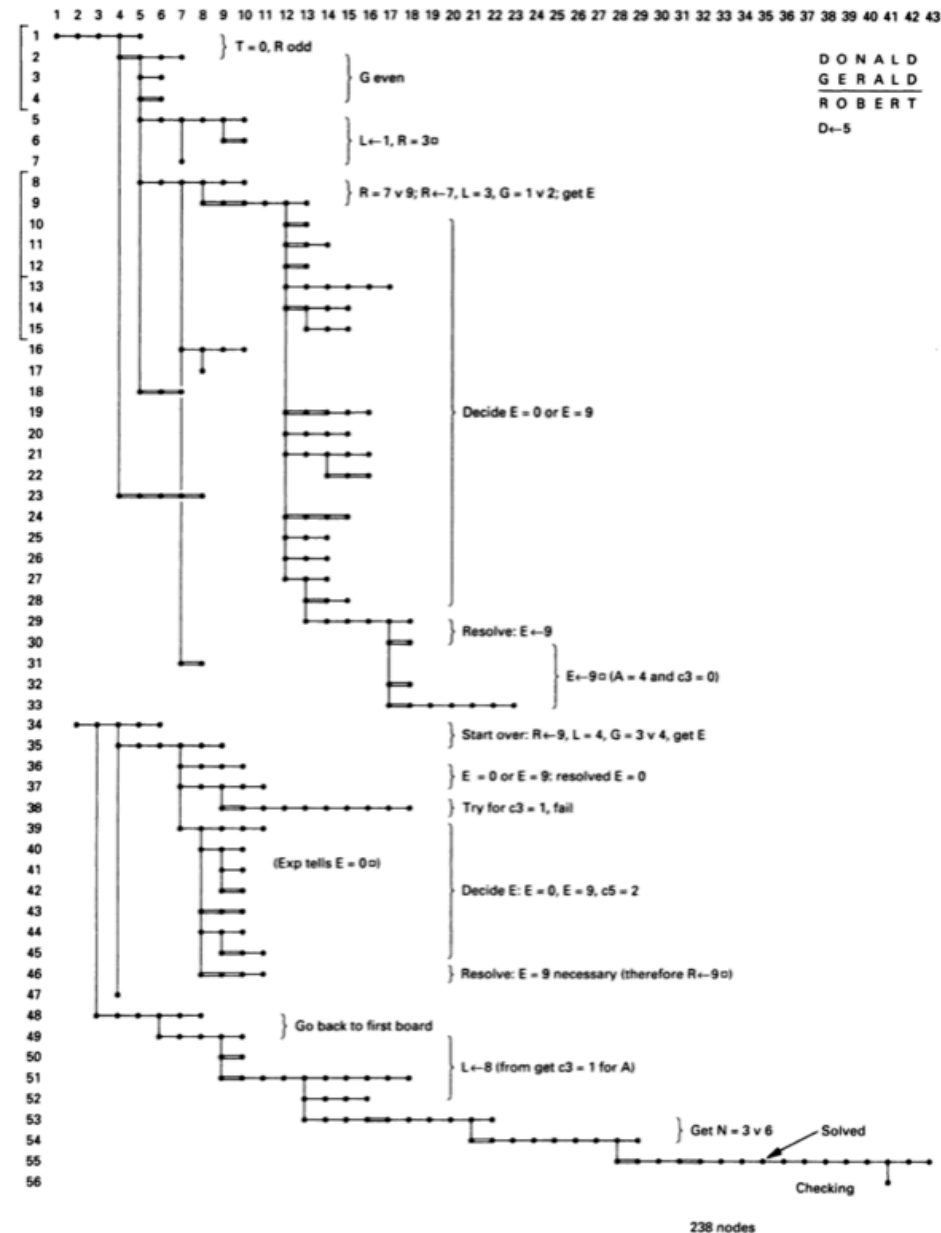


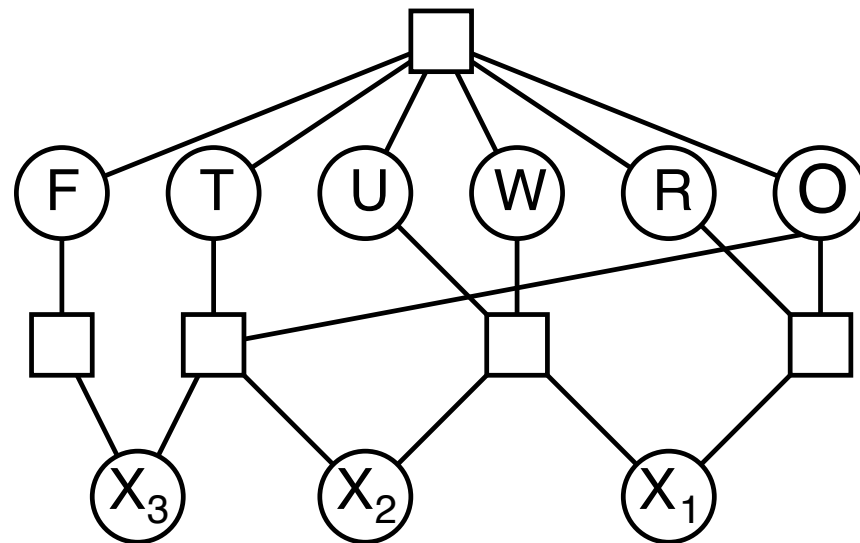
Figure 7-2. Problem-behavior graph of subject S3 on  
*DONALD + GERALD = ROBERT*.



# Cryptarithmic with Hidden Variables

- We can add “hidden” variables to simplify the constraints.

$$\begin{array}{r}
 \text{ T W O} \\
 + \text{ T W O} \\
 \hline
 \text{ F O U R}
 \end{array}$$



**Variables:** F T U W R O  $X_1$   $X_2$   $X_3$

**Domains:**  $\{0,1,2,3,4,5,6,7,8,9\}$

**Constraints:**

AllDifferent(F,T,U,W,R,O)

$O + O = R + 10 \cdot X_1$ , etc.

# Example: Sudoku

---

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

# Example: Sudoku

	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

**Constraints:** no digit appears twice in any row, column, or 3X3 box. (unit)

AllDiff constraints (A1, A2, A3, A4, A5, A6, A7, A8, A9)

AllDiff constraints (B1, B2, B3, B4, B5, B6, B7, B8, B9)

AllDiff constraints (A1, B1, C1, D1, E1, F1, G1, H1, I1) ....

# Real-world CSPs

---

- ❑ Assignment problems (e.g. who teaches what class)
- ❑ Timetabling problems (e.g. which class is offered when and where?)
- ❑ Hardware configuration (e.g. minimize space for circuit layout)
- ❑ Transport scheduling (e.g. courier delivery, vehicle routing)
- ❑ Factory scheduling (optimize assignment of jobs to machines)
- ❑ Gate assignment (assign gates to aircraft to minimize transit)

Many real world CSPs are also optimization problems

# Real-world CSPs - Factory scheduling

---

- ❑ A robot (agent) needs to schedule a set of activities for a manufacturing process, involving casting, milling, drilling, and bolting.
- ❑ Each activity has a set of possible times at which it may start.
- ❑ The robot has to satisfy various constraints arising from prerequisite requirements and resource use limitations.
- ❑ For each activity there is a variable that represents the time that it starts:
  - B – start of bolting
  - D – start of drilling
  - C – start of casting

# Real-world CSPs - Factory scheduling

---

- ❑ A robot (agent) needs to schedule a set of activities for a manufacturing process, involving casting, milling, drilling, and bolting.
- ❑ Each activity has a set of possible times at which it may start.
- ❑ The robot has to satisfy various constraints arising from prerequisite requirements and resource use limitations.
- ❑ For each activity there is a **variable** that represents the time that it starts:
  - B – start of bolting
  - D – start of drilling
  - C – start of casting
- ❑ **Constraints:**  $D < B$ ,  $C \neq D$ ,  $B = C + 3$

# Real-world CSPs - Factory scheduling

---

□ Consider a constraint on the possible dates for three activities.

*Variables*:  $A, B, C$  - variables that represent the date of each activity

*Domain* of each variable is:  $\{1, 2, 3, 4\}$

A *constraint* with scope:  $(A \leq B) \wedge (B < 3) \wedge (B < C) \wedge \neg(A = B \wedge C \leq 3)$

This formula says that  $A$  is on the same date or before  $B$

and it cannot be that  $A$  and  $B$  are on the same date and  $C$  is on or before day 3.

# Real-world CSPs - Factory scheduling

- Consider a constraint on the possible dates for three activities.

*Variables*:  $A, B, C$  - variables that represent the date of each activity

*Domain* of each variable is:  $\{1, 2, 3, 4\}$

A *constraint* with scope:

$$(A \leq B) \wedge (B < 3) \wedge (B < C) \wedge \neg(A = B \wedge C \leq 3),$$

This formula says that  $A$  is on the same date or before  $B$  and it cannot be that  $A$  and  $B$  are on the same date and  $C$  is on or before day 3.

This constraint could instead have its relation defined its **extension**, as a table specifying the legal assignments:

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



# Types of constraints

---

- ❑ **Unary** constraints involve a single variable
  - $M \neq 0$
- ❑ **Binary** constraints involve pairs of variables
  - $SA \neq WA$
- ❑ **Higher-order** constraints involve 3 or more variables
  - $Y = D + E$  or  $Y = D + E - 10$
- ❑ **Inequality** constraints on Continuous variables
  - $\text{EndJob1} + 5 \leq \text{StartJob3}$
- ❑ **Soft** constraints (Preferences)
  - 11am lecture is better than 8am lecture!

# Path Search vs Constraint Satisfaction

---

Important difference between path search problems and CSPs

## □ Path Search Problems (e.g. Rubik's Cube)

- Knowing the final state is easy
- Difficult part is how to get there

## □ Constraint Satisfaction Problems (e.g. $n$ -Queens)

- Difficult part is knowing the final state
- How to get there is easy

# Backtracking Search

---

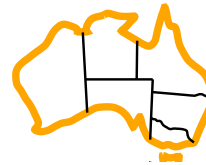
CSPs can be solved by assigning values to variables one by one, in different combinations. Whenever a constraint is violated, we go back to the most recently assigned variable and assign it a new value.

This can be achieved by a Depth First Search on a special kind of state space, where states are defined by the values assigned so far:

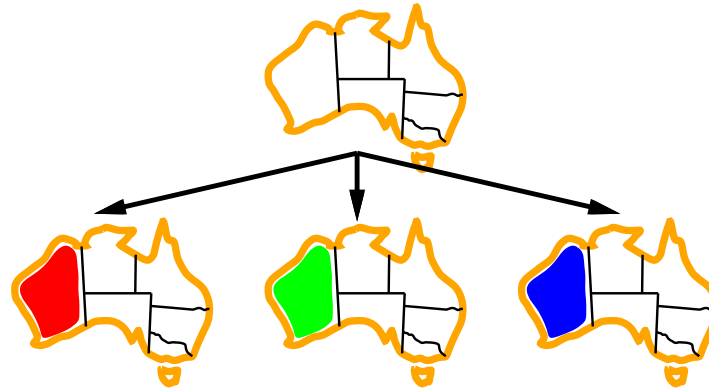
- **Initial state**: the empty assignment.
- **Successor function**: assign a value to an unassigned variable that does not conflict with previously assigned values of other variables. (If no legal values remain, the successor function fails.)
- **Goal test**: all variables have been assigned a value, and no constraints have been violated.

# Backtracking example

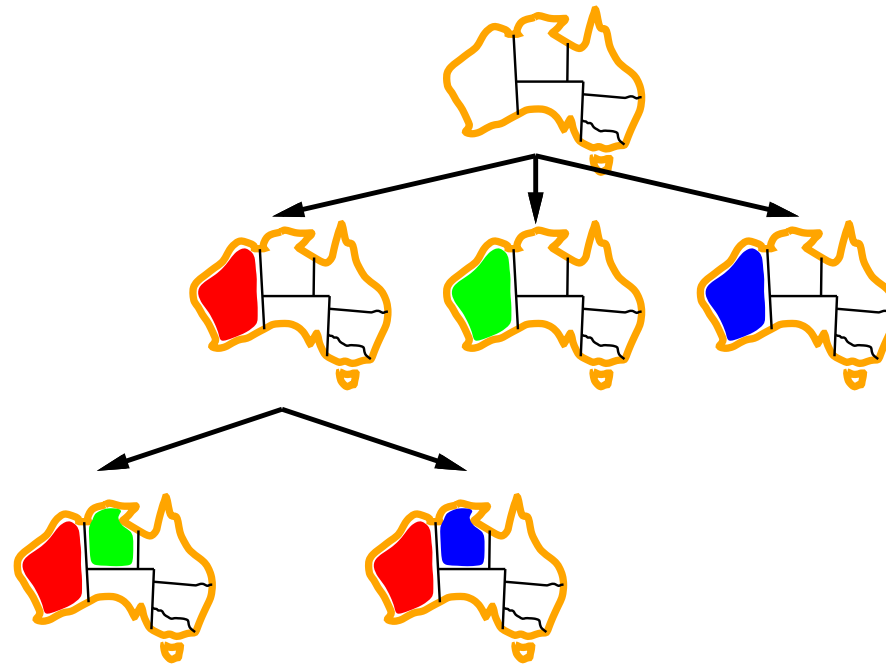
---



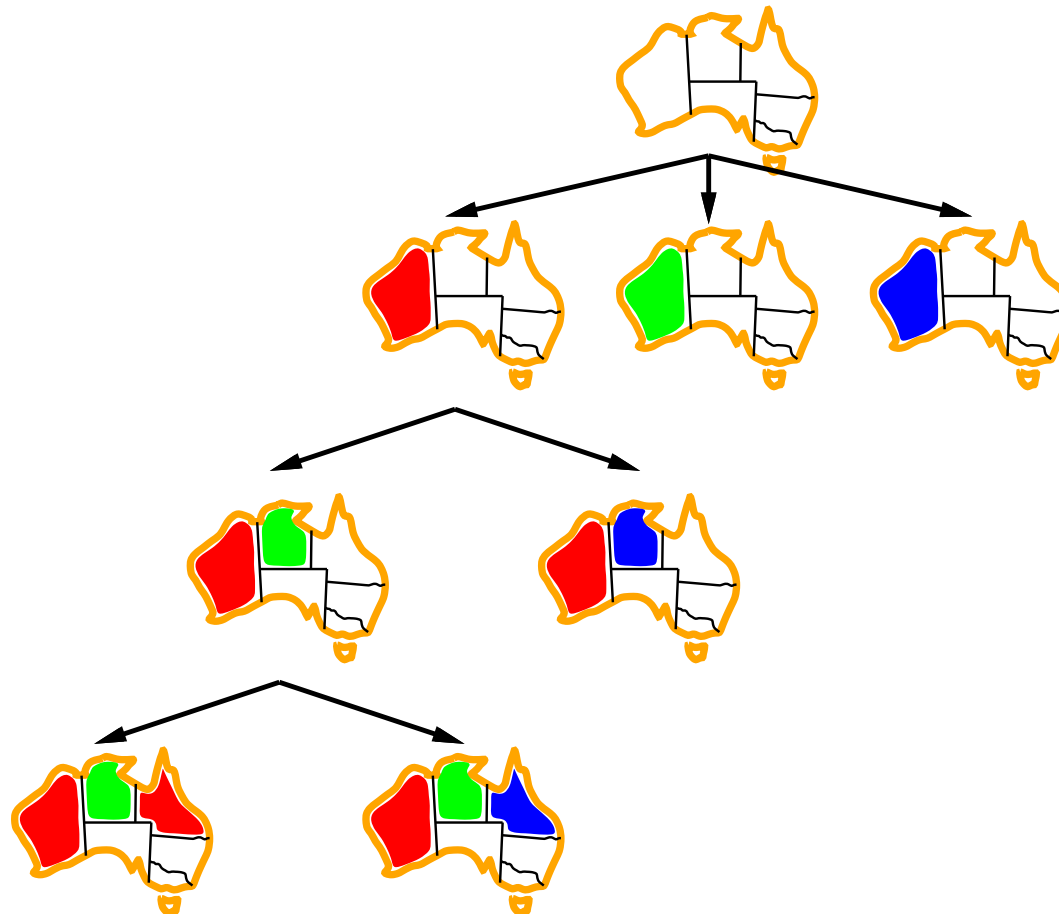
# Backtracking example



# Backtracking example



# Backtracking example



# Backtracking search Space Properties

---

The search space for this Depth First Search has certain very specific properties:

- ❑ If there are  $n$  variables, every solution will occur at exactly depth  $n$ .
- ❑ Variable assignments are **commutative**  
[WA = red then NT = green] same as [NT = green then WA = red]

Backtracking search can solve  $n$ -Queens for  $n \approx 25$



# Improvements to Backtracking search

---

General-purpose methods can give huge gains in speed:

1. Which variable should be assigned next?
2. In what order should its values be tried?
3. Can we detect inevitable failure early?

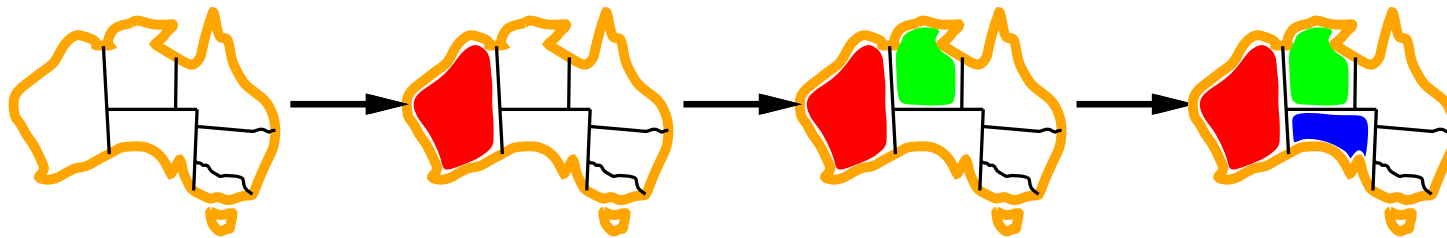
Improving backtracking efficiency

# Minimum Remaining Values



## Minimum Remaining Values (MRV)

- choose the variable with the fewest legal values



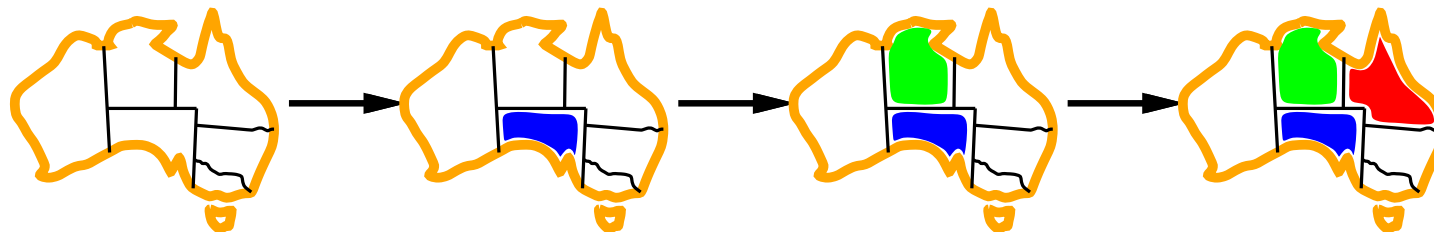
- Most constrained variable

## Minimum remaining values (MRV) heuristic

# Degree Heuristic



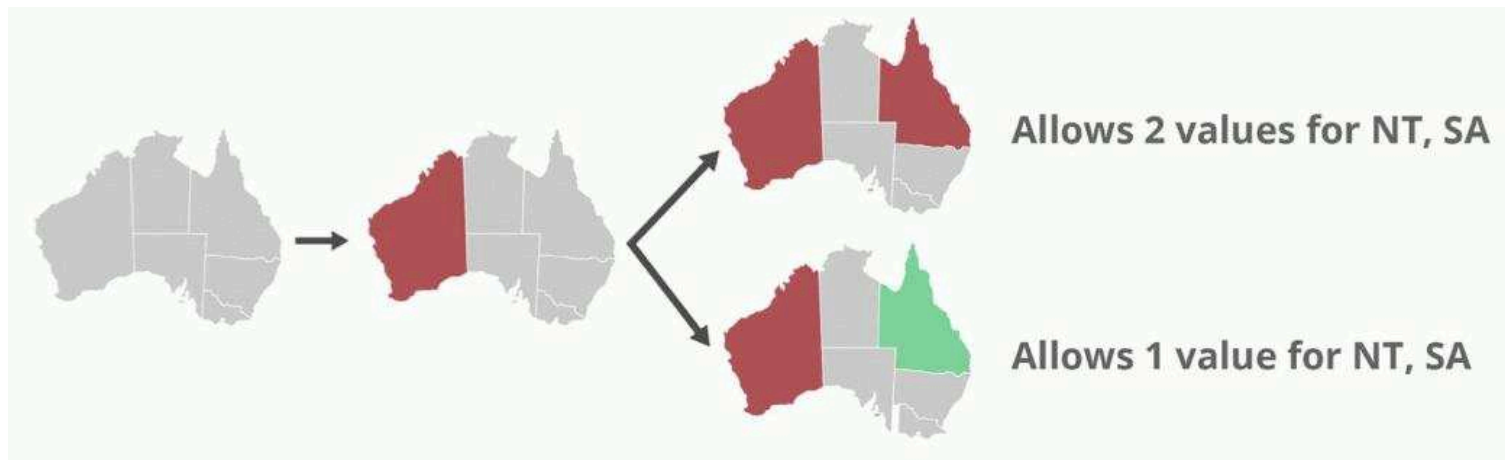
- ❑ Tie-breaker among MVR variables
- ❑ Degree heuristic:
  - choose the variable with the most constraints on remaining variables



# Least Constraining Value



- Given a variable, choose the least constraining value: the one that rules out the fewest values in the remaining variables



- (More generally, 3 allowed values would be better than 2, etc.)  
Combining these heuristics makes 1000 queens feasible.

# Forward checking



Idea:

- ☐ Keep track of remaining legal values for unassigned variables
- ☐ Terminate search when any variable has no legal values
  - prune off that part of the search tree, and backtrack



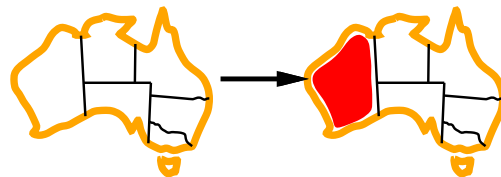
Initially, all values are available.

# Forward checking



Idea:

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



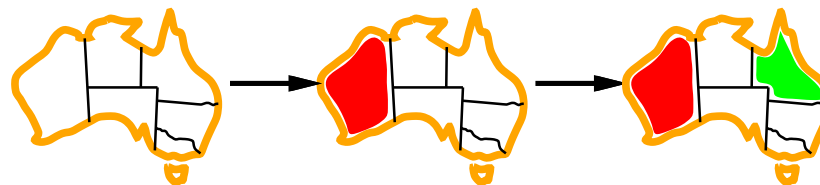
WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

# Forward checking



Idea:

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



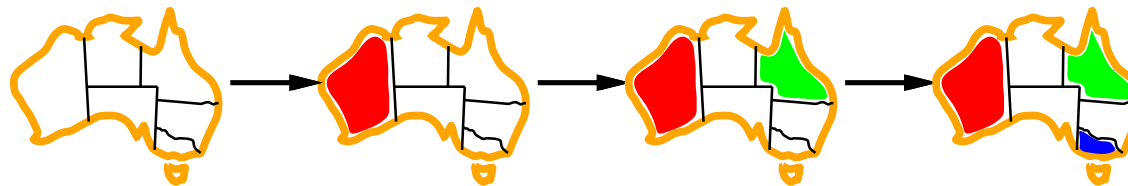
WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

# Forward checking



Idea:

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



WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

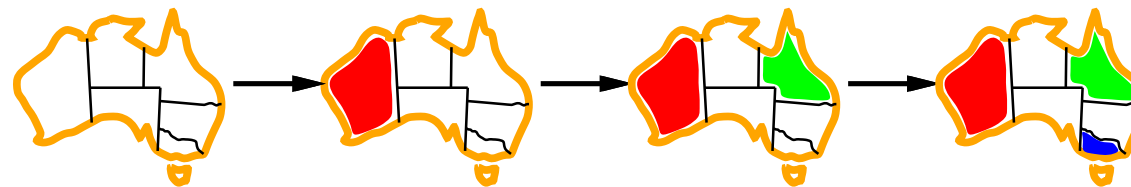


# Forward checking



Idea:

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



WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

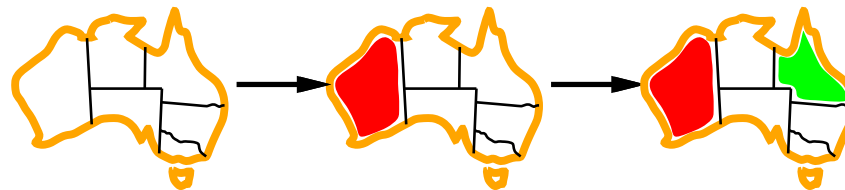
After colouring Victoria Blue, Blue is not available for NSW or SA.

SA has no values remaining, so we terminate this part of the search.

# Constraint propagation



Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:



WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

NT and SA cannot both be blue!

Constraint propagation repeatedly enforces constraints locally

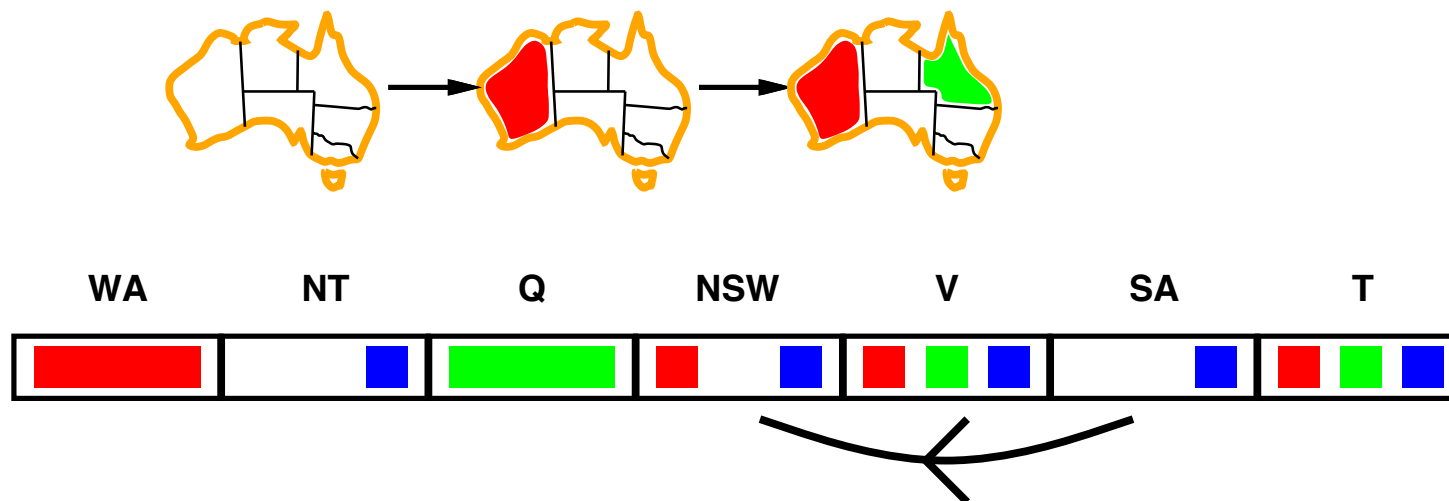
# Arc consistency



Simplest form of propagation makes each arc consistent

$X \rightarrow Y$  is consistent if

for **every** value  $x$  of  $X$  there is **some** allowed  $y$



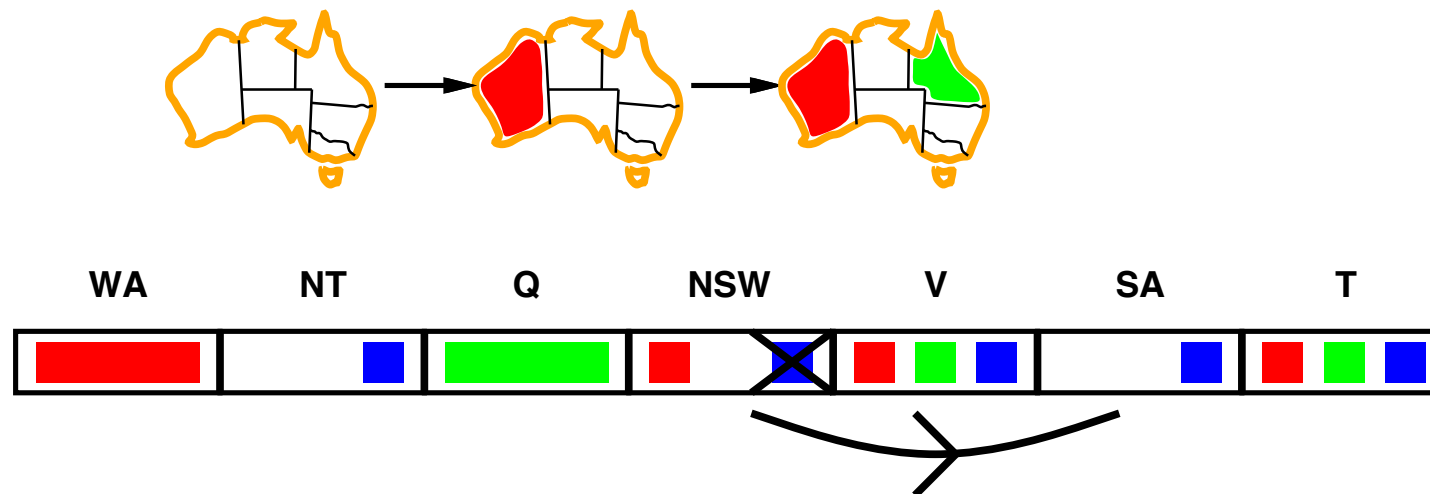
# Arc consistency



Simplest form of propagation makes each arc consistent

$X \rightarrow Y$  is consistent if

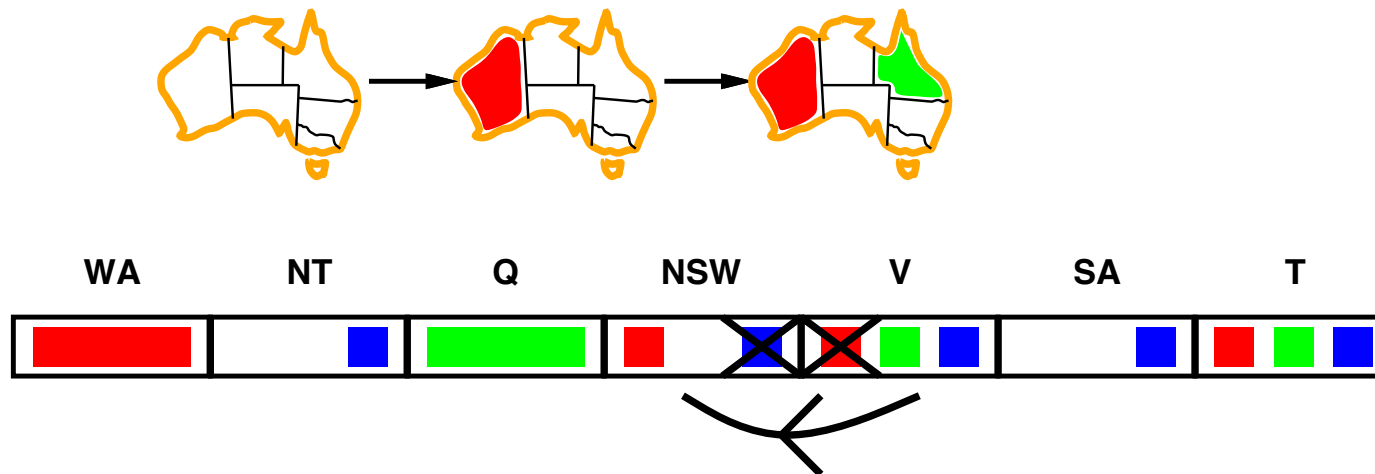
for **every** value  $x$  of  $X$  there is **some** allowed  $y$



# Arc consistency



$X \rightarrow Y$  is consistent if  
for **every** value  $x$  of  $X$  there is **some** allowed  $y$

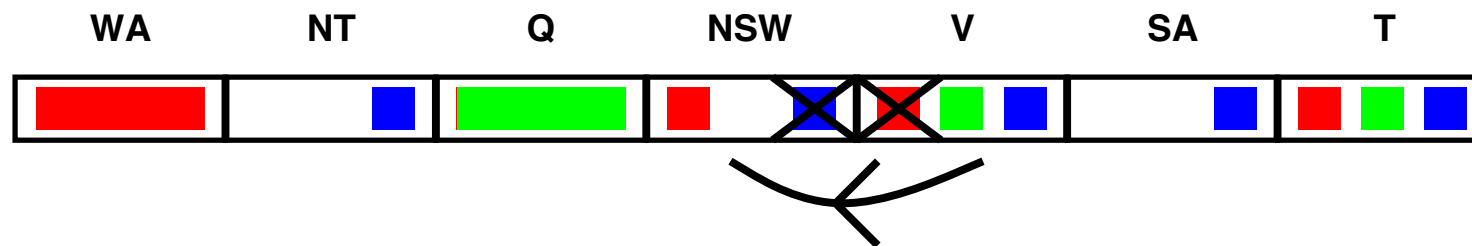
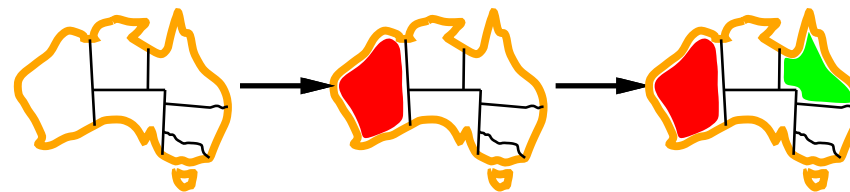


If  $X$  loses a value, neighbors of  $X$  need to be rechecked.

# Arc consistency



$X \rightarrow Y$  is consistent if  
for **every** value  $x$  of  $X$  there is **some** allowed  $y$

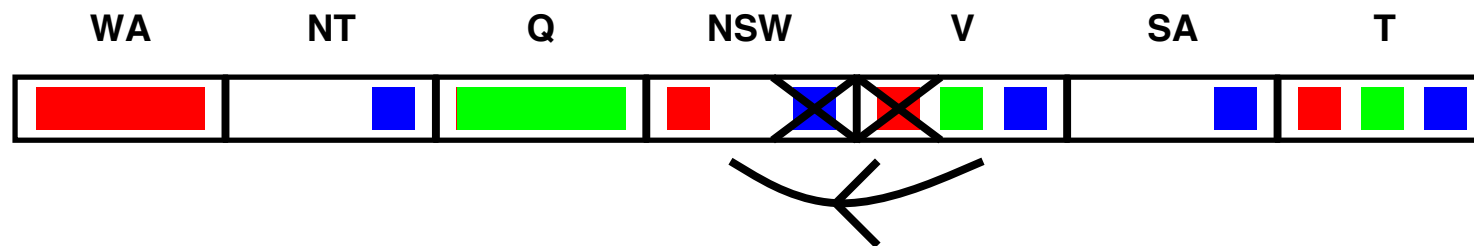
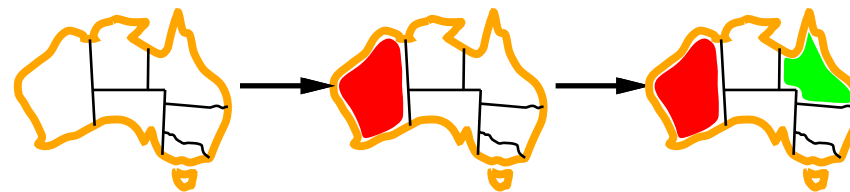


If  $X$  loses a value, neighbors of  $X$  need to be rechecked  
Arc consistency detects failure earlier than forward checking  
Can be run as a preprocessor after each assignment

# Arc consistency



$X \rightarrow Y$  is consistent if  
for **every** value  $x$  of  $X$  there is **some** allowed  $y$



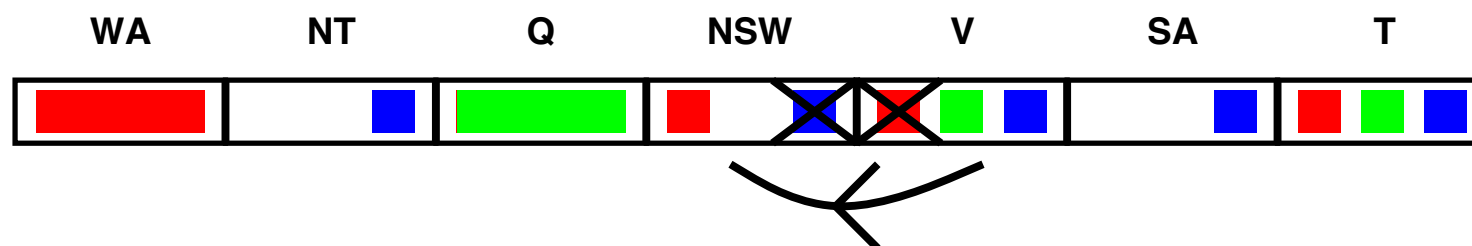
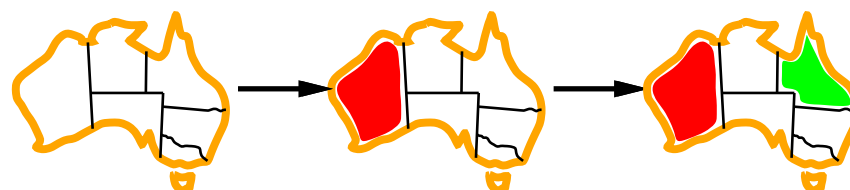
If  $X$  loses a value, neighbors of  $X$  need to be rechecked  
Arc consistency detects failure earlier than forward checking  
Can be run as a preprocessor after each assignment

# Arc consistency



$X \rightarrow Y$  is consistent if

for **every** value  $x$  of  $X$  there is **some** allowed  $y$



If  $X$  loses a value, neighbors of  $X$  need to be rechecked

Arc consistency detects failure earlier than forward checking

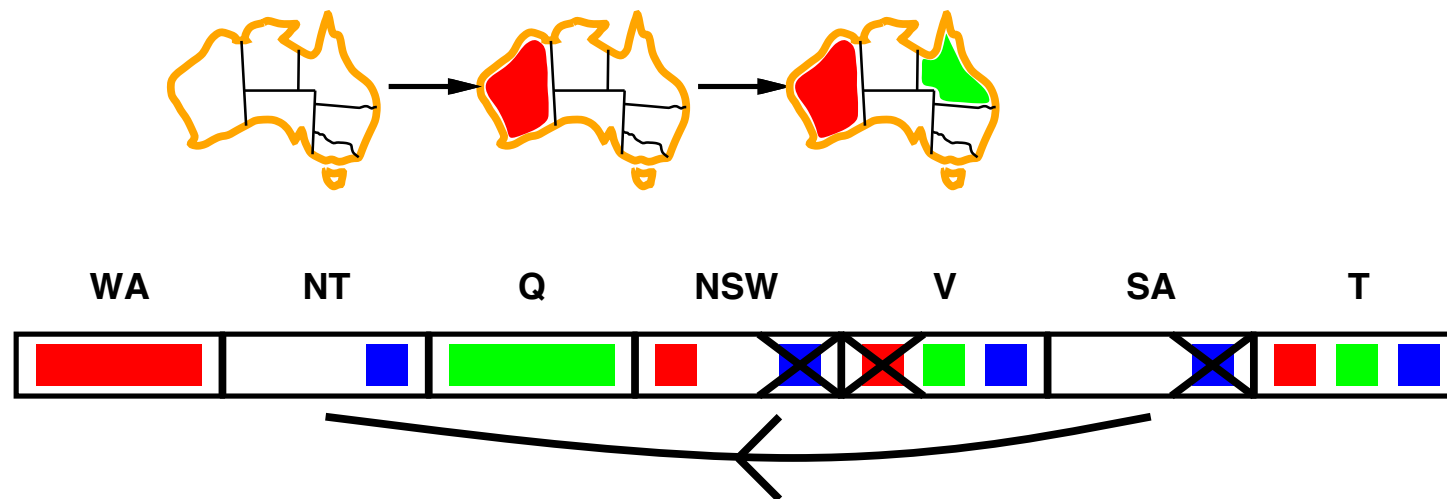
Can be run as a preprocessor after each assignment



# Arc consistency



$X \rightarrow Y$  is consistent if  
for **every** value  $x$  of  $X$  there is **some** allowed  $y$

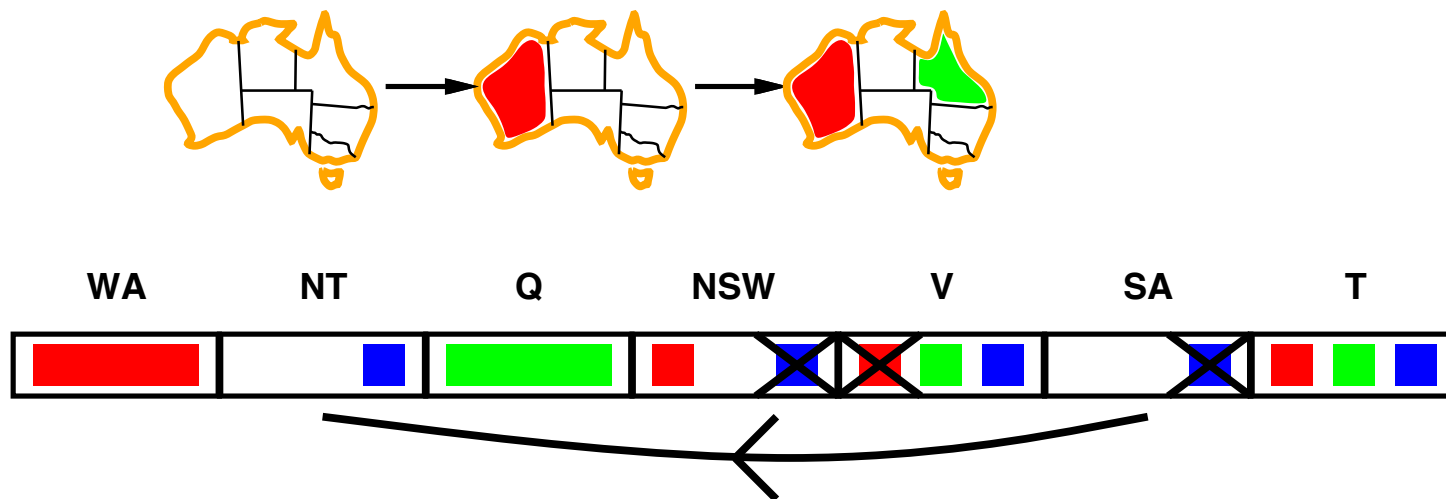


Arc consistency detects failure earlier than forward checking.  
For some problems, it can speed up the search enormously.  
For others, it may slow the search due to computational overheads.

# Arc consistency



$X \rightarrow Y$  is consistent if  
for **every** value  $x$  of  $X$  there is **some** allowed  $y$



Arc consistency detects failure earlier than forward checking.  
For some problems, it can speed up the search enormously.  
For others, it may slow the search due to computational overheads.

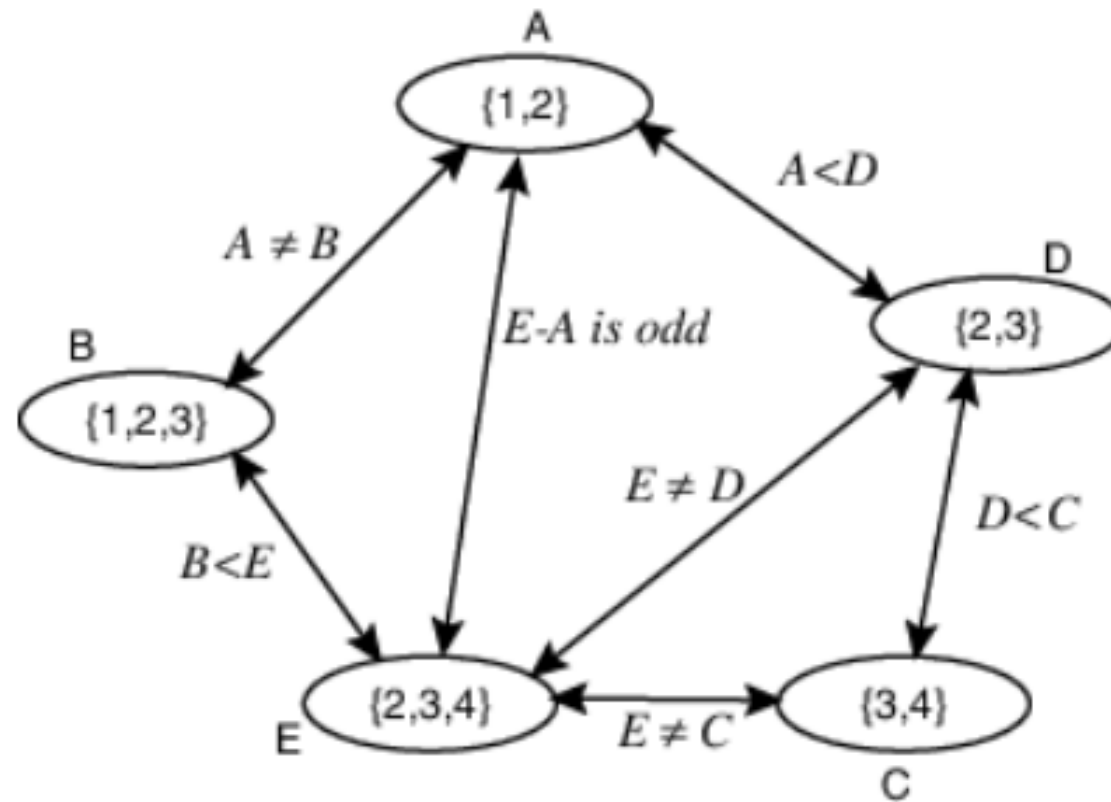
Blue for SA is not compatible with any remaining value for NT, **leaving SA with no legal values.**

# Variable Elimination

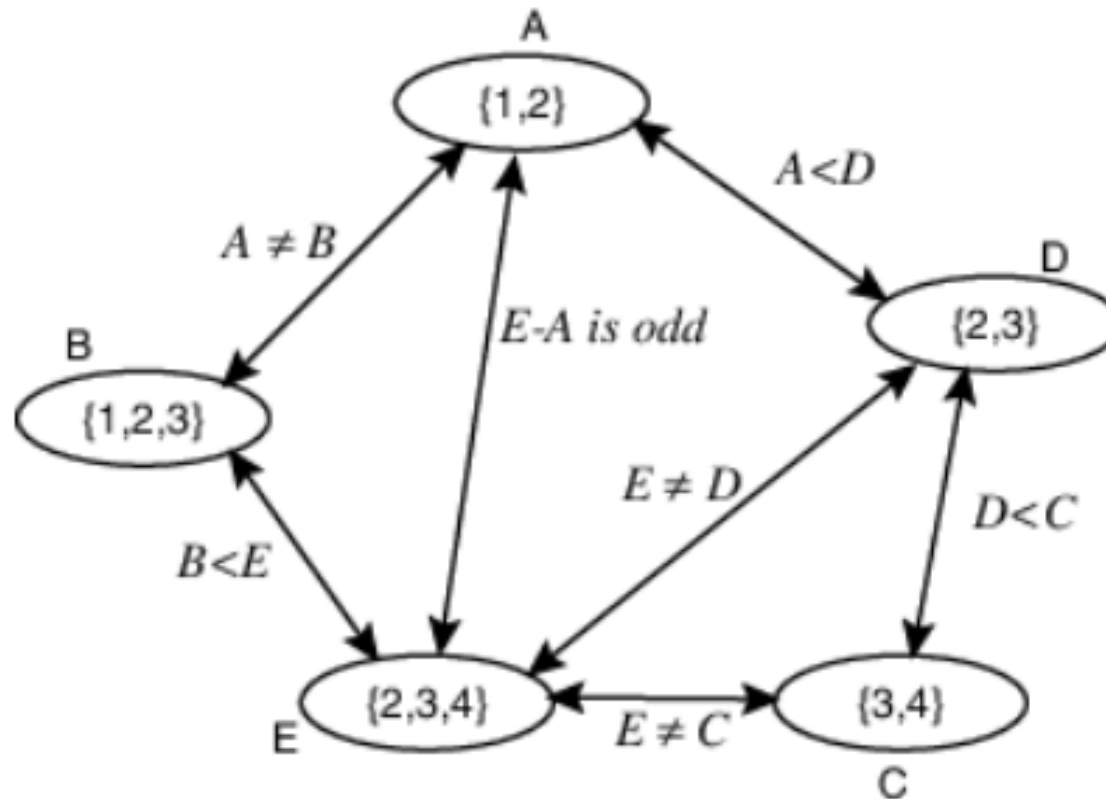
---

- ❑ If there is only one variable, return the intersection of its (unary) constraints
- ❑ Otherwise
  - Select a variable  $X$
  - Join the constraints in which  $X$  appears, forming constraint  $R1$
  - Project  $R1$  onto its variables other than  $X$ , forming  $R2$
  - Replace all of the constraints in which  $X$  appears by  $R2$
  - Recursively solve the simplified problem, forming  $R3$
  - Return  $R1$  joined with  $R3$

# Variable Elimination



# Variable Elimination



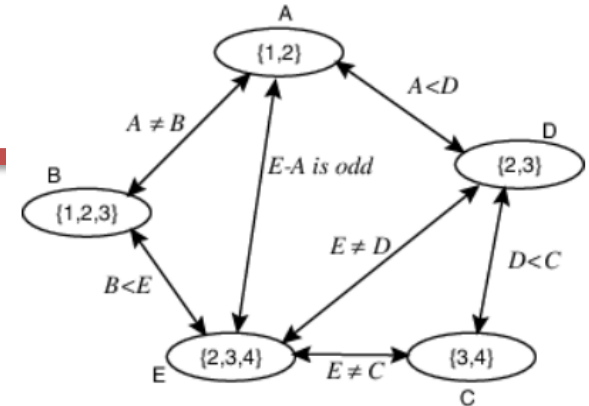
**Variables:** A, B, C, D, E

**Domains:** A = {1,2}, B = {1,2,3}, C = {3,4}, D = {2,3}, E = {2,3,4}

**Constraints:**  $A \neq B$ ,  $E \neq C$ ,  $E \neq D$ ,  $A < D$ ,  $B < E$ ,  $D < C$ ,  $E - A$  is odd

# Variable Elimination Example

$r_1 : C \neq E$	$C$	$E$
	3	2
	3	4
	4	2
	4	3



# Variable Elimination Example

---

$r_1 : C \neq E$	$C$	$E$
	3	2
	3	4
	4	2
	4	3

$r_2 : C > D$	$C$	$D$
	3	2
	4	2
	4	3

# Variable Elimination Example

---

$r_1 : C \neq E$	$C$	$E$	
	3	2	
	3	4	
	4	2	
	4	3	


$r_2 : C > D$	$C$	$D$	
	3	2	
	4	2	
	4	3	

$r_3 : r_1 \bowtie r_2$	$C$	$D$	$E$
	3	2	2
	3	2	4
	4	2	2
	4	2	3
	4	3	2
	4	3	3

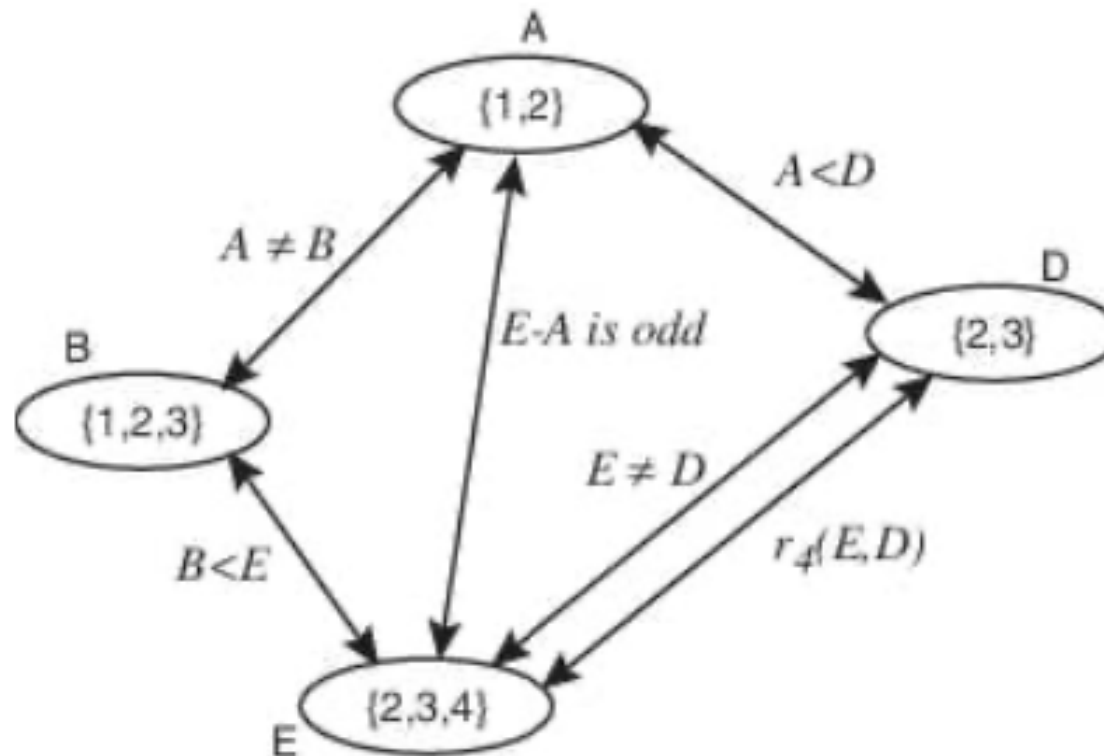


# Variable Elimination Example

$r_1 : C \neq E$	$C$	$E$		$r_2 : C > D$	$C$	$D$
	3	2			3	2
	3	4			4	2
	4	2			4	3
	4	3				
$r_3 : r_1 \bowtie r_2$	$C$	$D$	$E$	$r_4 : \pi_{\{D,E\}} r_3$	$D$	$E$
	3	2	2		2	2
	3	2	4		2	3
	4	2	2		2	4
	4	2	3		3	2
	4	3	2		3	3
	4	3	3			

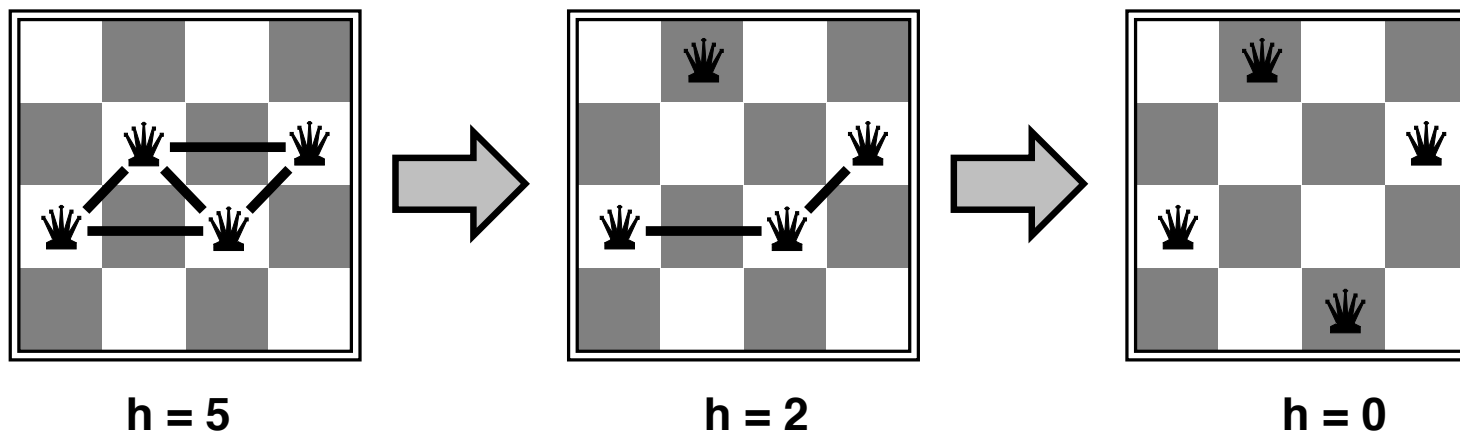
 new constraint

# Variable Elimination Example



# Local Search

There is another class of algorithms for solving CSP's, called “**Iterative Improvement**” or “Local Search”.

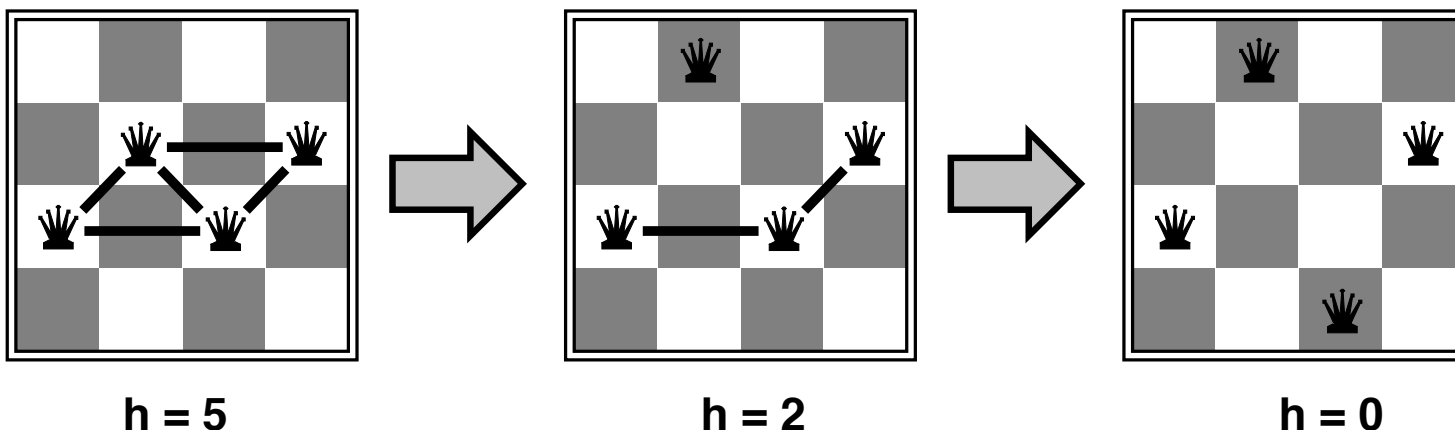


# Local Search

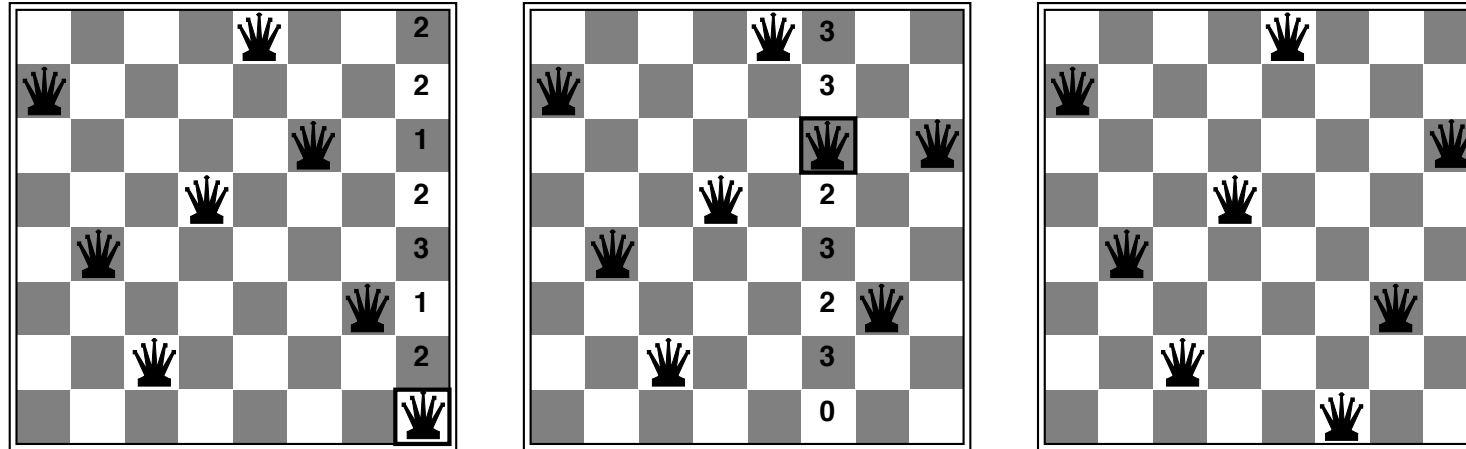
There is another class of algorithms for solving CSP's, called “**Iterative Improvement**” or “Local Search”.

## Iterative Improvement

- assign all variables randomly in the beginning (thus violating several constraints),
- change one variable at a time, trying to reduce the number of violations at each step.
- Greedy Search with  $h$  = number of constraints violated

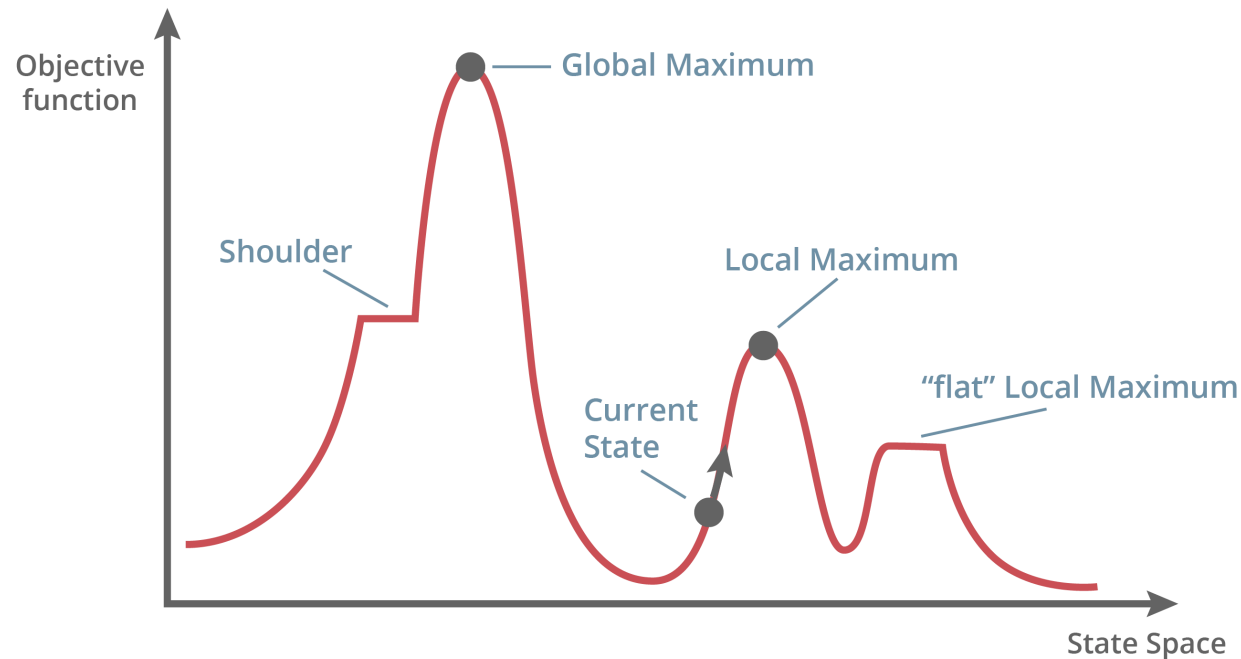


# Hill-climbing by min-conflicts



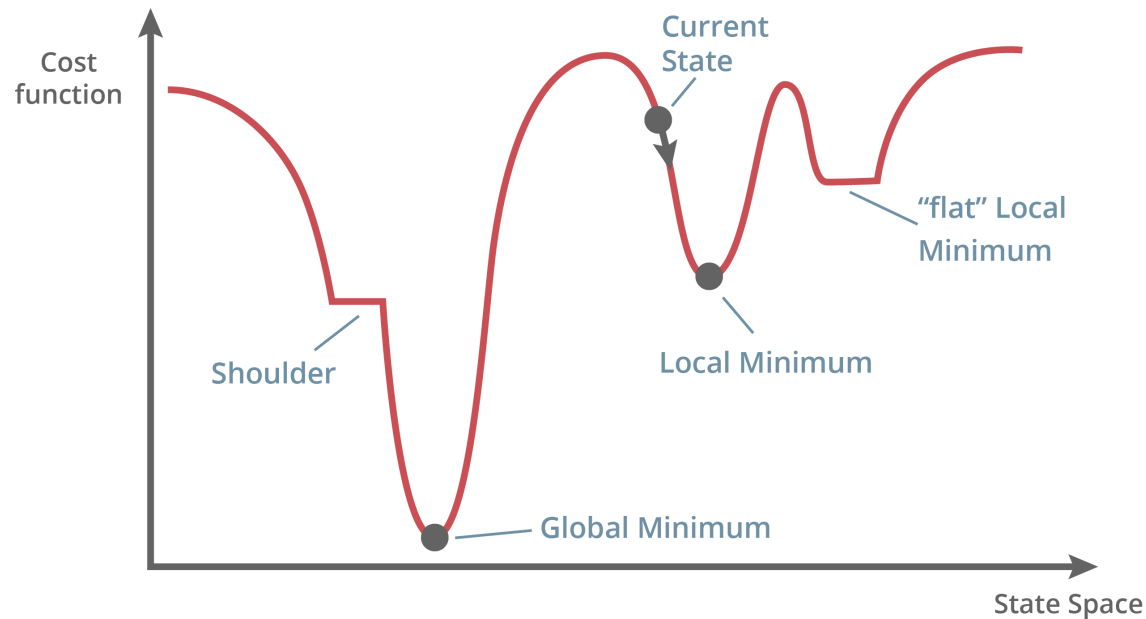
- ❑ Variable selection: randomly select any conflicted variable
- ❑ Value selection by **min-conflicts** heuristic
  - choose value that violates the fewest constraints

# Flat regions and local optima



- ❑ Sometimes, have to go sideways or even backwards in order to make progress towards the actual solution.

# Inverted View



- When we are minimizing violated constraints, it makes sense to think of starting at the top of a ridge and climbing down into the valleys.

# Simulated Annealing

---

- ❑ **Stochastic** hill climbing based on difference between evaluation of previous state ( $h_0$ ) and new state ( $h_1$ ).

- If  $h_1 < h_0$ , definitely make the change
- Otherwise, make the change with probability

$$e^{-(h_1 - h_0)/T}$$

where  $T$  is a “temperature” parameter.

- ❑ Reduces to ordinary hill climbing when  $T = 0$
- ❑ Becomes totally random search as  $T \rightarrow \infty$
- ❑ Sometimes, we gradually decrease the value of  $T$  during the search

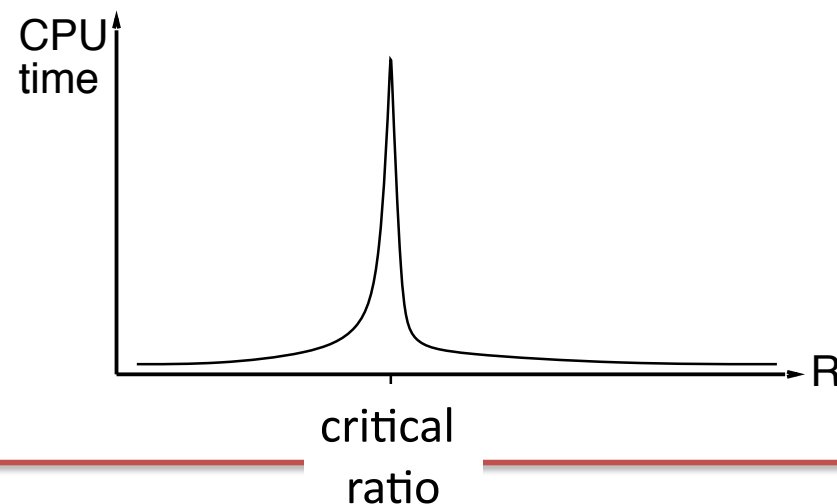


# Phase transition in CSP's

Given random initial state, hill climbing by min-conflicts with random restarts can solve  $n$ -queens in almost constant time for arbitrary  $n$  with high probability (e.g.,  $n = 10,000,000$ ).

In general, randomly-generated CSP's tend to be easy if there are very few or very many constraints. They become extra hard in a narrow range of the ratio

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$



# Summary

---

- ❑ CSPs are a special kind of problem:
  - states defined by values of a fixed set of variables
  - goal test defined by constraints on variable values
- ❑ Backtracking = depth-first search with one variable assigned per node
- ❑ Variable ordering and value selection heuristics help significantly
- ❑ Forward checking prevents assignments that guarantee later failure
- ❑ Constraint propagation (e.g., arc consistency) does additional work to constrain values and detect inconsistencies
- ❑ Iterative min-conflicts is usually effective in practice

# Summary

---

- ❑ Much interest in CSP's for real-world applications
- ❑ Backtracking = depth-first search
- ❑ Variable and Value ordering heuristics help significantly
- ❑ Forward Checking for detecting inevitable failure early
- ❑ Hill Climbing by min-conflicts often effective in practice
- ❑ Simulated Annealing can help to escape from local optima
- ❑ Which method(s) are best? It varies from one task to another!

# Domain Splitting and Arc Consistency

---

States are whole CSPs (not partial assignments)

❑ Make CSP domain consistent and arc consistent

➤ Domain consistent = all **unary** constraints are satisfied

❑ To solve CSP using **Depth First Search**

➤ Choose a variable  $v$  with more than one value in domain

➤ Split the domain of  $v$  into two subsets

➤ This gives two smaller CSPs

➤ Make each CSP **arc consistent**

❑ Solve each resulting CSP (or backtrack if unsolvable)

# Constraint Optimization Problems

---

States are whole CSPs (not partial assignments) **with costs**

- ❑ Make CSP **domain consistent** and **arc consistent**

- ❑ Add CSP to priority queue

- ❑ To solve CSP using **Greedy Search**

- Remove CSP with minimal  $h$  from priority queue
- Choose a variable  $v$  with more than one value in domain
- Split the domain of  $v$  into two subsets
- This gives two smaller CSPs
- Make each CSP **arc consistent** – add to priority queue

- ❑  $cost(CSP) = \text{sum of costs to violate soft constraints}$



# Local search for CSPs

---

- ❑ Hill-climbing, simulated annealing typically work with "complete" states, i.e., all variables assigned

To apply to CSPs:

- allow states with unsatisfied constraints
- operators reassign variable values

Variable selection: randomly select any conflicted variable

- ❑ Value selection by min-conflicts heuristic:
- ❑ choose value that violates the fewest constraints
- ❑ i.e., hill-climb with  $h(n)$  = total number of violated constraints

---

□ Russell & Norvig, Chapter 6.1,6.2,6.3,6.4,4.1





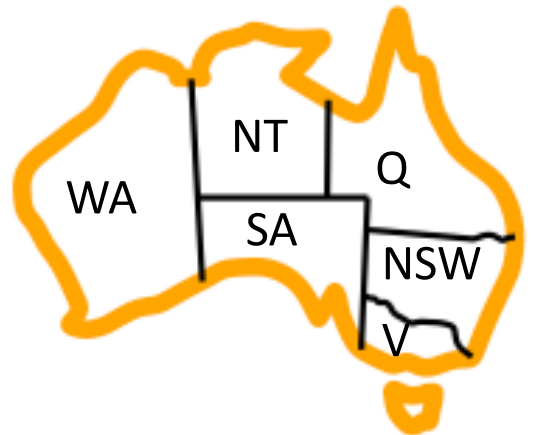


# MVR - Least constraining value

---

# MVR - Least constraining value

---



# Most constrained variable

---

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



- ❑ a.k.a. minimum remaining values (MRV) heuristic

# Most constrained variable

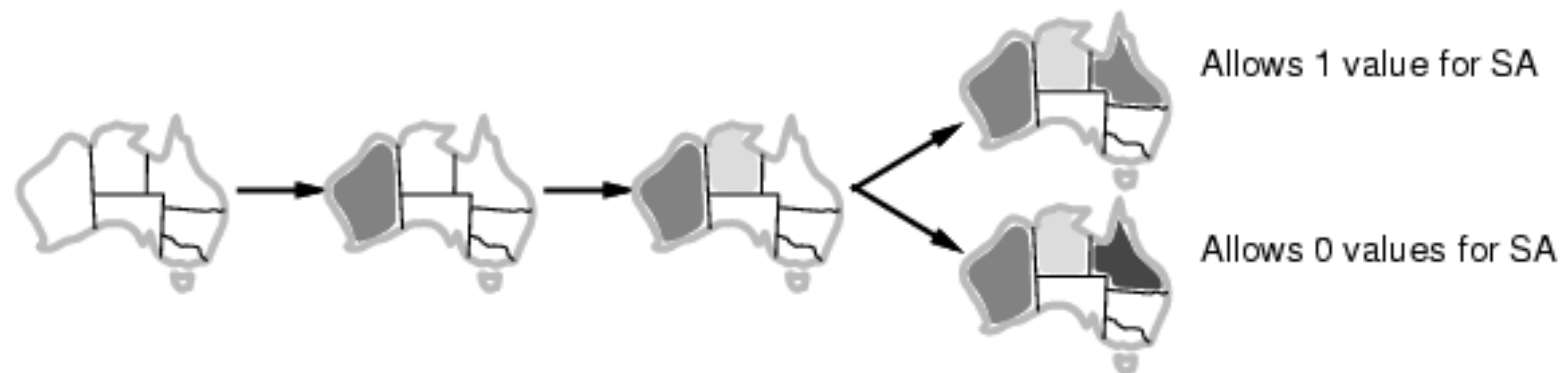
---

- ❑ Tie-breaker among most constrained variables
- ❑ Most constraining variable:
  - choose the variable with the most constraints on



# Least constraining value

- ❑ Given a variable, choose the least constraining value:
- ❑ the one that rules out the fewest values in the remaining variables



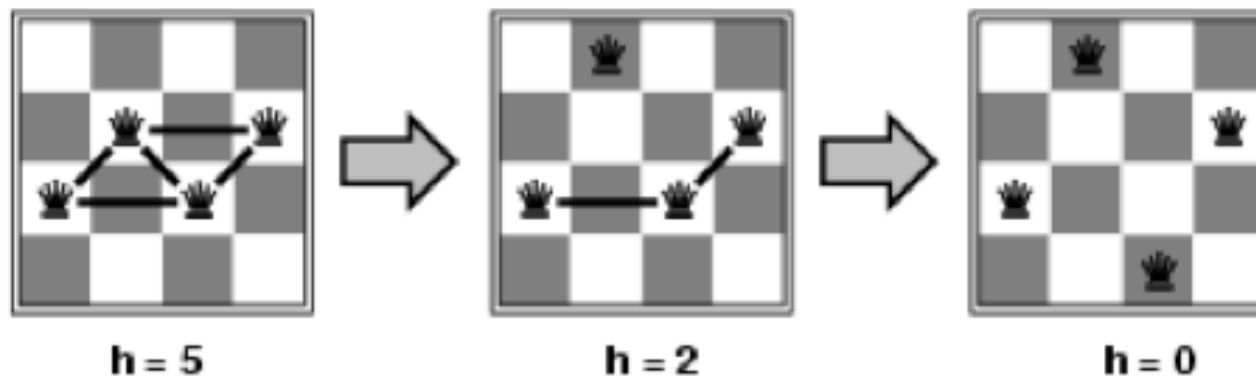
- ❑ Combining these heuristics makes 1000 queens feasible





## Example: 4-Queens \*

- ❑ States: 4 queens in 4 columns ( $4^4 = 256$  states)
- ❑ Actions: move queen in column
- ❑ Goal test: no attacks
- ❑ Evaluation:  $h(n)$  = number of attacks



- ❑ Given random initial state, can solve  $n$ -queens in almost constant time for arbitrary  $n$  with high probability (e.g.,  $n = 10,000,000$ )