

Artificial Intelligence

CSE 4617

Ahnaf Munir

Assistant Professor

Islamic University of Technology

What is Search for?

- Assumptions about the world
 - Single agent \rightarrow No adversaries
 - Deterministic actions
 - Fully observed state
 - Discrete state space

What is Search for?

- Assumptions about the world
 - Single agent \rightarrow No adversaries
 - Deterministic actions
 - Fully observed state
 - Discrete state space
- Planning: sequences of actions
 - The path to the goal is the important thing
 - Paths have various costs, depths
 - Heuristics give problem-specific guidance

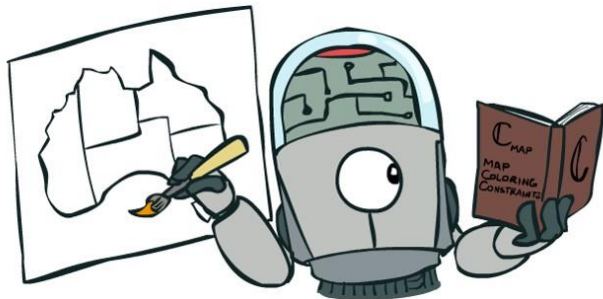


What is Search for?

- Assumptions about the world
 - Single agent \rightarrow No adversaries
 - Deterministic actions
 - Fully observed state
 - Discrete state space
- **Planning:** sequences of actions
 - The path to the goal is the important thing
 - Paths have various costs, depths
 - Heuristics give problem-specific guidance
- **Identification:** assignments to variables
 - The goal itself is important, not the path
 - All paths at the same depth (for some formulations)
 - CSPs are a specialized class of identification problems

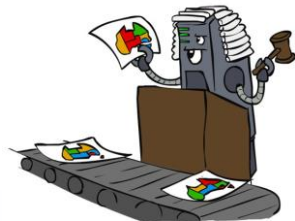


Constraint Satisfaction Problems



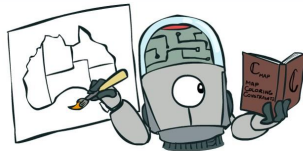
Constraint Satisfaction Problems

- Standard search problems
 - State is a “black box”: arbitrary data structure
 - Goal test can be any function over states
 - Successor function can also be anything



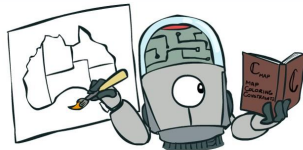
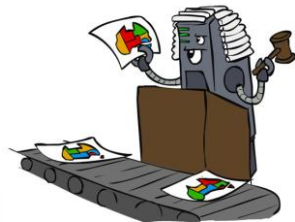
Constraint Satisfaction Problems

- Standard search problems
 - State is a “black box”: arbitrary data structure
 - Goal test can be any function over states
 - Successor function can also be anything
- Constraint satisfaction problems (CSPs):
 - A special subset of search problems
 - State is defined by **variables X_i** with values from a **domain D** (sometimes D depends on i)
 - Goal test is a **set of constraints** specifying allowable combinations of values for subsets of variables



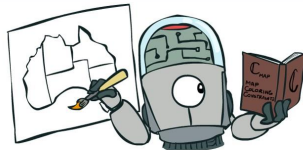
Constraint Satisfaction Problems

- Standard search problems
 - State is a “black box”: arbitrary data structure
 - Goal test can be any function over states
 - Successor function can also be anything
- Constraint satisfaction problems (CSPs):
 - A special subset of search problems
 - State is defined by **variables X_i** with values from a **domain D** (sometimes D depends on i)
 - Goal test is a **set of constraints** specifying allowable combinations of values for subsets of variables
- Simple example of a *formal representation language*

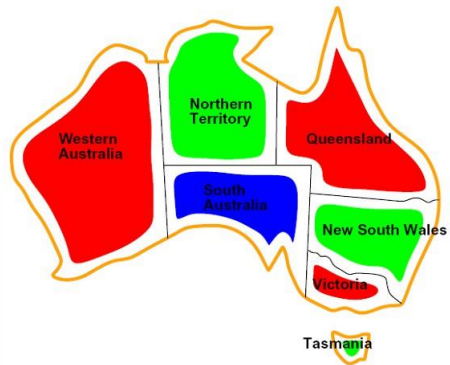


Constraint Satisfaction Problems

- Standard search problems
 - State is a “black box”: arbitrary data structure
 - Goal test can be any function over states
 - Successor function can also be anything
- Constraint satisfaction problems (CSPs):
 - A special subset of search problems
 - State is defined by variables X_i with values from a domain D (sometimes D depends on i)
 - Goal test is a set of constraints specifying allowable combinations of values for subsets of variables
- Simple example of a *formal representation language*
- Allows useful general-purpose algorithms with more power than standard search algorithms

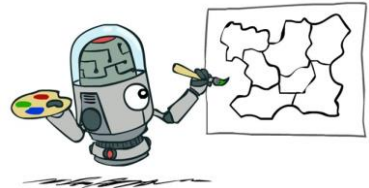
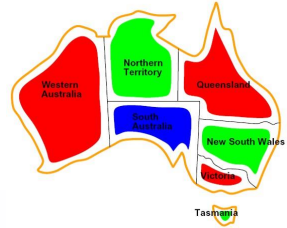


CSP Examples



Example: Map Coloring

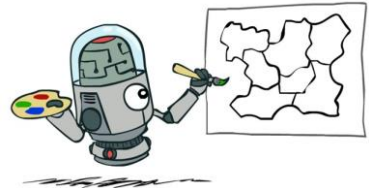
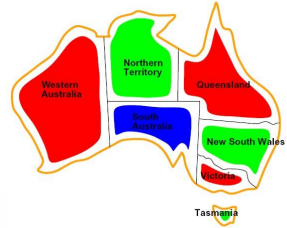
■ Variables:



Example: Map Coloring

■ Variables:

- WA, NT, Q, NSW, V, SA, T

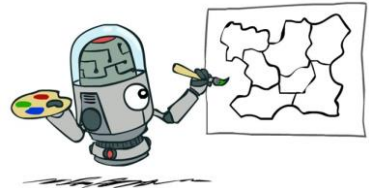
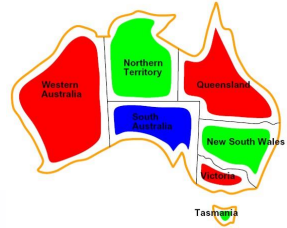


Example: Map Coloring

- Variables:

- WA, NT, Q, NSW, V, SA, T

- Domains:



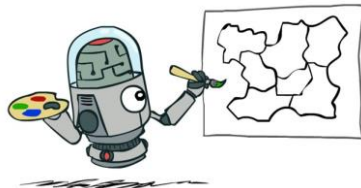
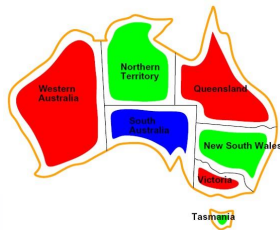
Example: Map Coloring

- Variables:

- WA, NT, Q, NSW, V, SA, T

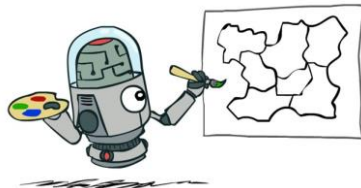
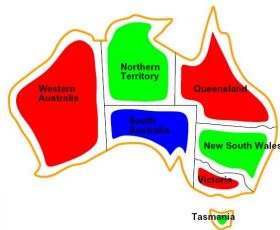
- Domains:

- $D = \{\text{red, green, blue}\}$



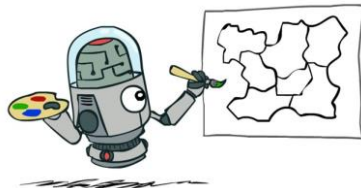
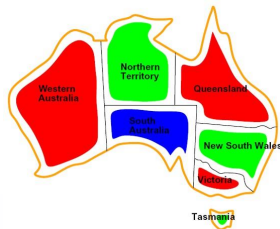
Example: Map Coloring

- Variables:
 - WA, NT, Q, NSW, V, SA, T
- Domains:
 - $D = \{\text{red, green, blue}\}$
- Constraints: adjacent regions must have different colors



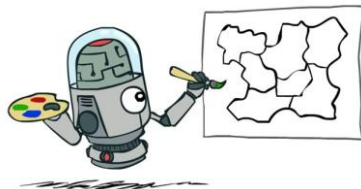
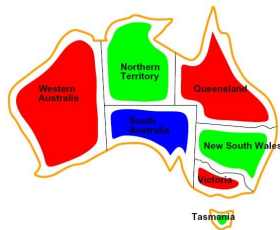
Example: Map Coloring

- Variables:
 - WA, NT, Q, NSW, V, SA, T
- Domains:
 - $D = \{\text{red, green, blue}\}$
- Constraints: adjacent regions must have different colors
 - Implicit: $WA \neq NT$



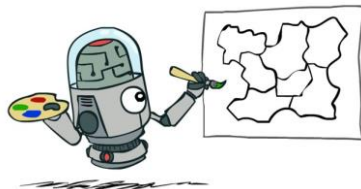
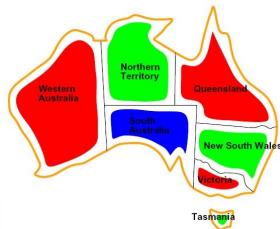
Example: Map Coloring

- Variables:
 - WA, NT, Q, NSW, V, SA, T
- Domains:
 - $D = \{\text{red, green, blue}\}$
- Constraints: adjacent regions must have different colors
 - Implicit: $WA \neq NT$
 - Explicit:
 $(WA, NT) \in \{(\text{red, green}), (\text{red, blue}), \dots\}$



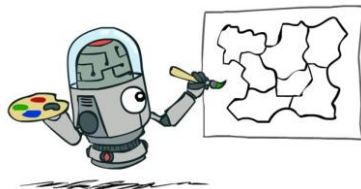
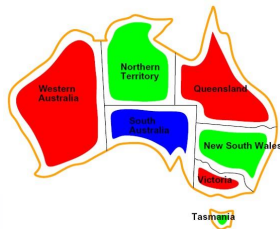
Example: Map Coloring

- Variables:
 - WA, NT, Q, NSW, V, SA, T
- Domains:
 - $D = \{\text{red, green, blue}\}$
- Constraints: adjacent regions must have different colors
 - Implicit: $WA \neq NT$
 - Explicit:
 $(WA, NT) \in \{(\text{red, green}), (\text{red, blue}), \dots\}$
- Solutions are assignments satisfying all constraints

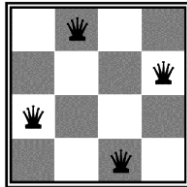


Example: Map Coloring

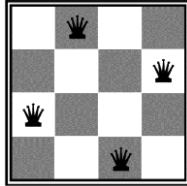
- Variables:
 - WA, NT, Q, NSW, V, SA, T
- Domains:
 - $D = \{\text{red, green, blue}\}$
- Constraints: adjacent regions must have different colors
 - **Implicit:** $WA \neq NT$
 - **Explicit:**
 $(WA, NT) \in \{(\text{red, green}), (\text{red, blue}), \dots\}$
- Solutions are assignments satisfying all constraints
 - $\{WA = \text{red}, NT = \text{green}, Q = \text{red}, NSW = \text{green}, V = \text{red}, SA = \text{blue}, T = \text{green}\}$



Example: N-Queens

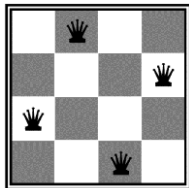


Example: N-Queens



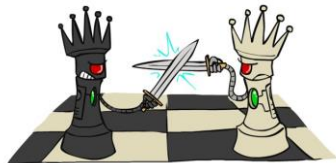
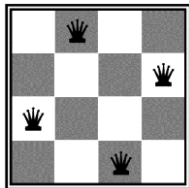
Example: N-Queens

■ Formulation 1:



Example: N-Queens

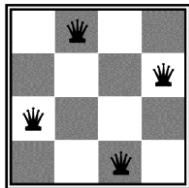
- Formulation 1:
 - Variables: X_{ij}



Example: N-Queens

■ Formulation 1:

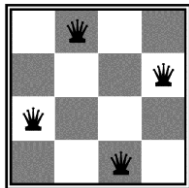
- Variables: X_{ij}
- Domains: $\{0, 1\}$



Example: N-Queens

■ Formulation 1:

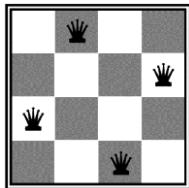
- Variables: X_{ij}
- Domains: $\{0, 1\}$
- Constraints:



Example: N-Queens

■ Formulation 1:

- Variables: X_{ij}
- Domains: $\{0, 1\}$
- Constraints:

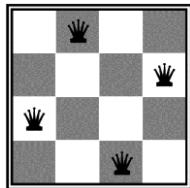


$$\forall i, j, k (X_{ij}, X_{ik}) \in \{(0, 0), (0, 1), (1, 0)\}$$

Example: N-Queens

■ Formulation 1:

- Variables: X_{ij}
- Domains: $\{0, 1\}$
- Constraints:



$$\forall i, j, k (X_{ij}, X_{ik}) \in \{(0, 0), (0, 1), (1, 0)\}$$

$$\forall i, j, k (X_{ij}, X_{kj}) \in \{(0, 0), (0, 1), (1, 0)\}$$

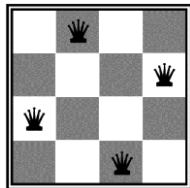
$$\forall i, j, k (X_{ij}, X_{i+k, j+k}) \in \{(0, 0), (0, 1), (1, 0)\}$$

$$\forall i, j, k (X_{ij}, X_{i+k, j-k}) \in \{(0, 0), (0, 1), (1, 0)\}$$

Example: N-Queens

■ Formulation 1:

- Variables: X_{ij}
- Domains: $\{0, 1\}$
- Constraints:



$$\forall i,j,k (X_{ij}, X_{ik}) \in \{(0, 0), (0, 1), (1, 0)\}$$

$$\forall i,j,k (X_{ij}, X_{kj}) \in \{(0, 0), (0, 1), (1, 0)\}$$

$$\forall i,j,k (X_{ij}, X_{i+k,j+k}) \in \{(0, 0), (0, 1), (1, 0)\}$$

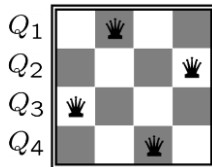
$$\forall i,j,k (X_{ij}, X_{i+k,j-k}) \in \{(0, 0), (0, 1), (1, 0)\}$$

$$\sum_{i,j} X_{ij} = N$$

Example: N-Queens

■ Formulation 2:

- Variables: Q_k
- Domains: $\{1, 2, 3, \dots, N\}$
- Constraints:

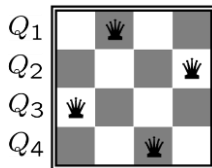


Example: N-Queens

■ Formulation 2:

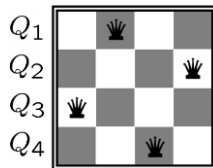
- Variables: Q_k
- Domains: $\{1, 2, 3, \dots, N\}$
- Constraints:

■ Implicit: $\forall i, j$ non-threatening (Q_i, Q_j)

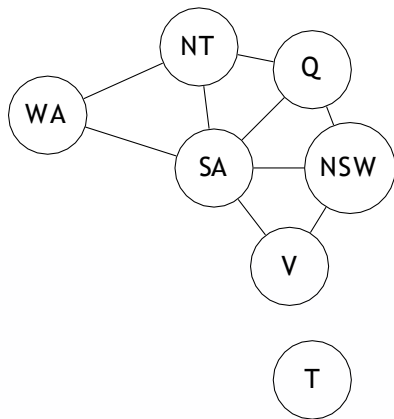


Example: N-Queens

- Formulation 2:
 - Variables: Q_k
 - Domains: $\{1, 2, 3, \dots, N\}$
 - Constraints:
- Implicit: $\forall i, j$ non-threatening (Q_i, Q_j)
- Explicit: $(Q_i, Q_j) \in \{(1, 3), (1, 4), \dots\}$
- ...

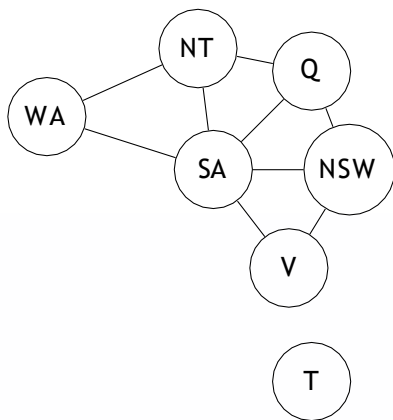


Constraint Graphs



Constraint Graphs

- Binary CSP: each constraint relates (at most) two variables
- Binary constraint graph: nodes are variables, arcs show constraints
- General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!



Example: Cryptarithmic

- Variables
- Domains:
- Constraints:

$$\begin{array}{r} \text{TWO} \\ + \text{TWO} \\ \hline \text{FOUR} \end{array}$$



Example: Cryptarithmic

- Variables
 - $F, T, U, W, R, O, X_1, X_2, X_3$
- Domains:
- Constraints:

$$\begin{array}{r} \text{TWO} \\ + \text{TWO} \\ \hline \text{FOUR} \end{array}$$



Example: Cryptarithmic

- 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:

$$\begin{array}{r} \text{TWO} \\ + \text{TWO} \\ \hline \text{FOUR} \end{array}$$



Example: Cryptarithmic

- 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:
 - $\text{alldiff}(F, T, U, W, R, O)$

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



Example: Cryptarithmic

- 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:
 - $\text{alldiff}(F, T, U, W, R, O)$
 - $O + O = R + 10 \times X_1$
 - ...

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



Example: Cryptarithmic

■ Variables

- $F, T, U, W, R, O, X_1, X_2, X_3$

■ Domains:

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

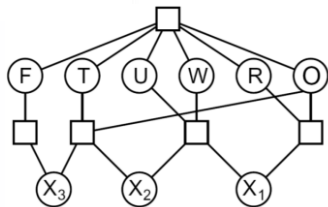
■ Constraints:

$\text{alldiff}(F, T, U, W, R, O)$

$$O + O = R + 10 \times X_1$$

...

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



Example: Sudoku

- Variables
 - Each (open) square
- Domains
 - $\{1, 2, \dots, 9\}$
- Constraints

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

Example: Sudoku

- Variables
 - Each (open) square
- Domains
 - $\{1, 2, \dots, 9\}$
- Constraints
 - Unary constraints for given values

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

Example: Sudoku

■ Variables

- Each (open) square


■ Domains

- $\{1, 2, \dots, 9\}$

■ Constraints

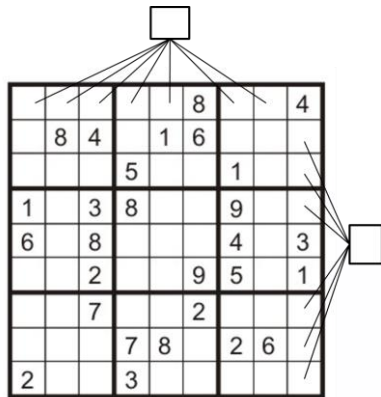
- Unary constraints for given values
- 9-way alldiff for each column

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



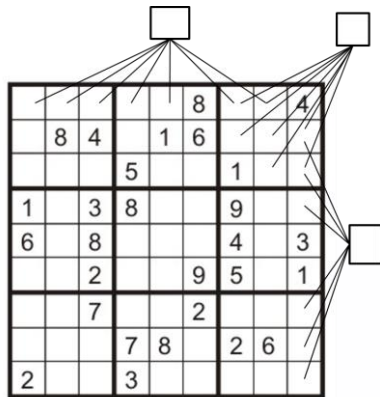
Example: Sudoku

- Variables
 - Each (open) square
- Domains
 - $\{1, 2, \dots, 9\}$
- Constraints
 - Unary constraints for given values
 - 9-way alldiff for each column
 - 9-way alldiff for each row



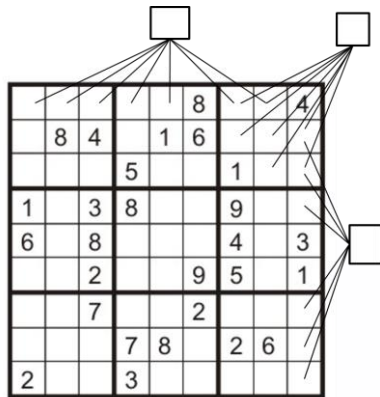
Example: Sudoku

- Variables
 - Each (open) square
- Domains
 - $\{1, 2, \dots, 9\}$
- Constraints
 - Unary constraints for given values
 - 9-way alldiff for each column
 - 9-way alldiff for each row
 - 9-way alldiff for each region

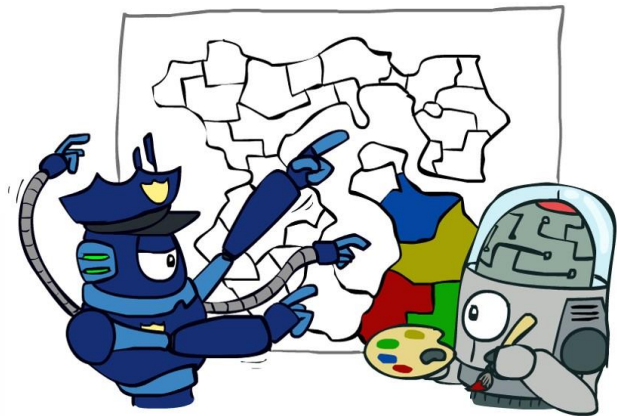


Example: Sudoku

- Variables
 - Each (open) square
- Domains
 - $\{1, 2, \dots, 9\}$
- Constraints
 - Unary constraints for given values
 - 9-way alldiff for each column
 - 9-way alldiff for each row
 - 9-way alldiff for each region
 - Can also have a bunch of pairwise inequalities



Varieties of CSPs and Constraints



Varieties of CSPs

- Discrete Variables
 - Finite domains
 - ▶ Size d means $O(d^n)$ complete assignments



Varieties of CSPs

■ Discrete Variables

- Finite domains
 - ▶ Size d means $O(d^n)$ complete assignments
- Infinite domains (integers, strings, etc.)
 - ▶ E.g., job scheduling, variables are start/end times for each job



Varieties of CSPs

■ Discrete Variables

- Finite domains
 - ▶ Size d means $O(d^n)$ complete assignments
- Infinite domains (integers, strings, etc.)
 - ▶ E.g., job scheduling, variables are start/end times for each job



Continuous Variables

- E.g., start/end times for Hubble Telescope observations



Varieties of Constraints

- Unary constraints involve a single variable (equivalent to reducing domains), e.g.: $SA \neq green$



Varieties of Constraints

- Unary constraints involve a single variable (equivalent to reducing domains), e.g.: $SA \neq green$
- Binary constraints involve pairs of variables, e.g.: $SA \neq WA$



Varieties of Constraints

- Unary constraints involve a single variable (equivalent to reducing domains), e.g.: $SA \neq \text{green}$
- Binary constraints involve pairs of variables, e.g.: $SA \neq WA$
- Higher-order constraints involve 3 or more variables, e.g., cryptarithmic column constraints



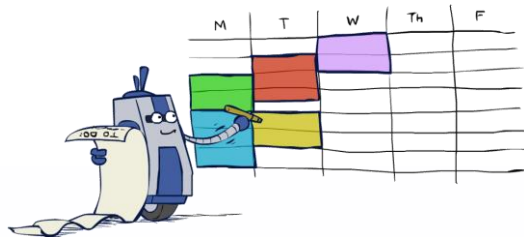
Varieties of Constraints

- **Unary constraints** involve a single variable (equivalent to reducing domains), e.g.: $SA \neq \text{green}$
- **Binary constraints** involve pairs of variables, e.g.: $SA \neq WA$
- **Higher-order constraints** involve 3 or more variables, e.g., cryptarithmic column constraints
- Preferences (soft constraints)
 - E.g., red is better than green
 - Often representable by a cost for each variable assignment
 - Gives constrained optimization problems
 - (We'll ignore these until we get to Bayes' nets)



Real-World CSPs

- Scheduling problem
- Timetabling problem
- Assignment problem
- Hardware configuration
- Transportation scheduling
- Factory scheduling
- Circuit layout
- Fault diagnosis
- ...lots more!
- Many real-world problems involve real-valued variables...

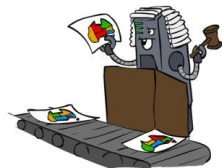


Solving CSPs



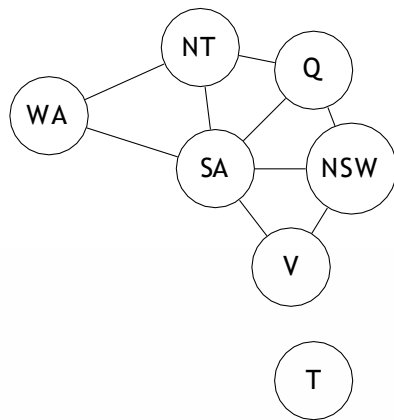
Standard Search Formulation

- States defined by the values assigned so far (partial assignments)
 - Initial state: the empty assignment, $\{\}$
 - Successor function: assign a value to an unassigned variable
 - Goal test: the current assignment is complete and satisfies all constraints



Search Methods

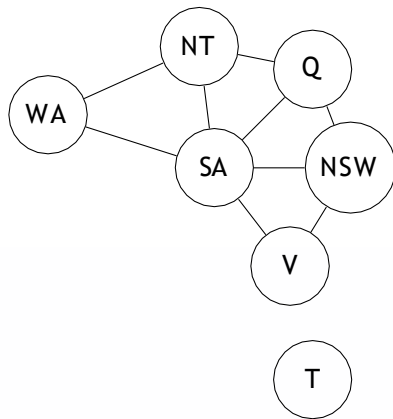
- What would BFS do?



Website: [simple-naive](#)

Search Methods

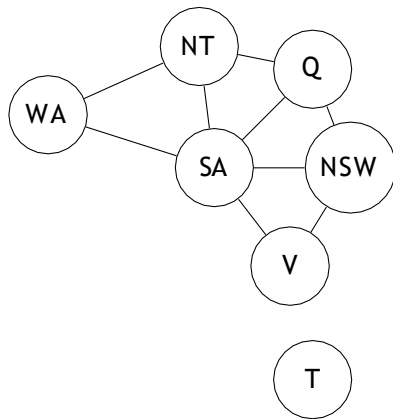
- What would BFS do?
- What would DFS do?



Website: [simple -naive](#)

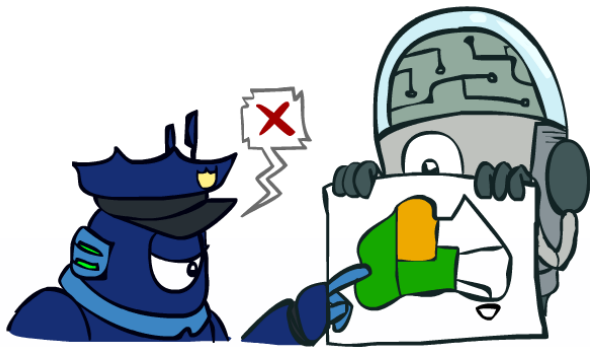
Search Methods

- What would BFS do?
- What would DFS do?
- What problems does naïve search have?



Website: [simple -naive](#)

Backtracking Search



Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs

Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
- Idea 1: One variable at a time
 - Variable assignments are commutative \rightarrow Any ordering is OK!
 - i.e., $[WA = \text{red then } NT = \text{green}]$ same as $[NT = \text{green then } WA = \text{red}]$
 - Only need to consider assignments to a single variable at each step

Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
- Idea 1: One variable at a time
 - Variable assignments are commutative \rightarrow Any ordering is OK!
 - i.e., $[WA = \text{red then } NT = \text{green}]$ same as $[NT = \text{green then } WA = \text{red}]$
 - Only need to consider assignments to a single variable at each step
- Idea 2: Check constraints as you go
 - i.e. consider only values which do not conflict with previous assignments
 - Might have to do some computation to check the constraints
 - “Incremental goal test”

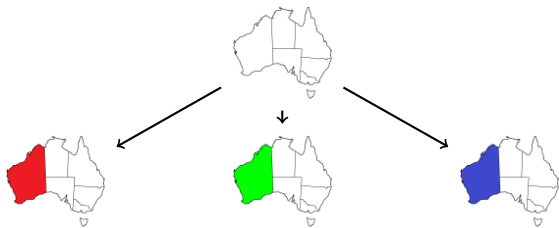
Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
- Idea 1: One variable at a time
 - Variable assignments are commutative \rightarrow Any ordering is OK!
 - i.e., $[WA = \text{red then } NT = \text{green}]$ same as $[NT = \text{green then } WA = \text{red}]$
 - Only need to consider assignments to a single variable at each step
- Idea 2: Check constraints as you go
 - i.e. consider only values which do not conflict with previous assignments
 - Might have to do some computation to check the constraints
 - “Incremental goal test”
- Depth-first search with these two improvements is called *backtracking search*

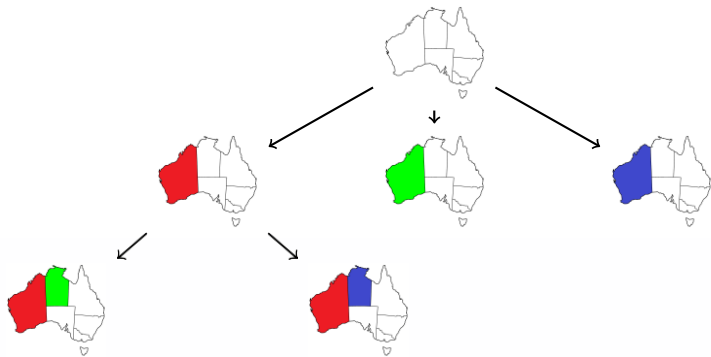
Backtracking Search



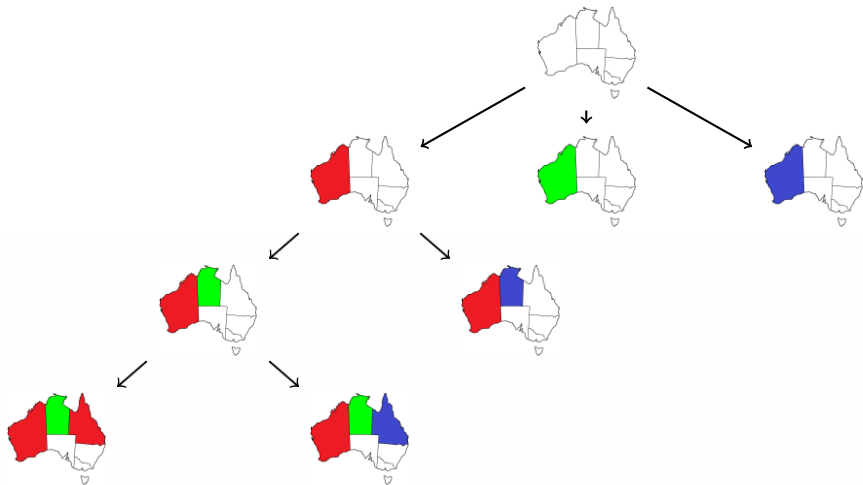
Backtracking Search



Backtracking Search



Backtracking Search



Backtracking Search

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure  
  return RECURSIVE-BACKTRACKING({}, csp)
```

```
function RECURSIVE-BACKTRACKING(assignment, csp) returns a solution, or failure  
  if assignment is complete then return assignment  
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)  
  for each value in ORDER-DOMAIN-VALUE(var, assignment, csp) do  
    if value is consistent with assignment given CONSTRAINTS[csp] then  
      add {var = value} to assignment  
      result ← RECURSIVE-BACKTRACKING(assignment, csp)  
      if result ≠ failure then return result  
      remove {var = value} from assignment  
  return failure
```

■ Backtracking = DFS + variable-ordering + fail-on-violation

Website: [simple -backtracking](#)

Improving Backtracking

- General-purpose ideas give huge gains in speed
- Filtering: Can we detect inevitable failure early?
- Ordering:
 - Which variable should be assigned next?
 - In what order should its values be tried?
- Structure: Can we exploit the problem structure?



Filtering



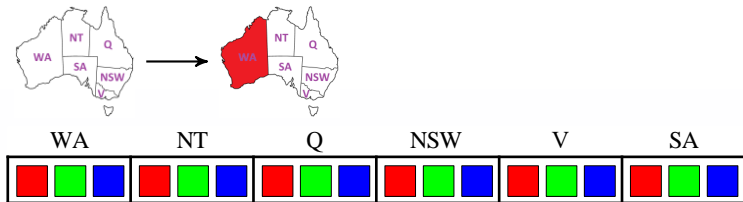
Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment



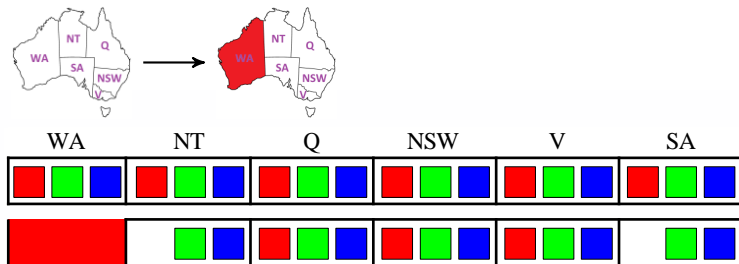
Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment



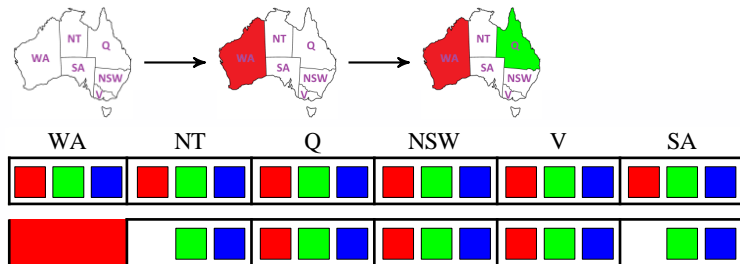
Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment



Filtering: Forward Checking

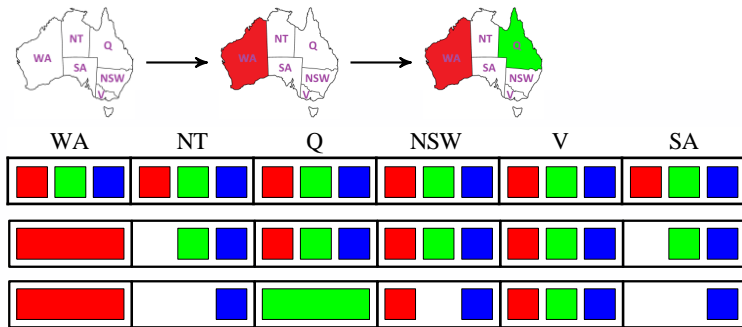
- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment



Website: [simple -
backtracking, forward](#)

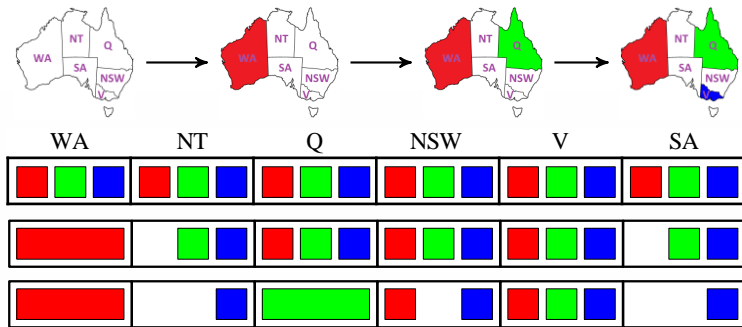
Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment



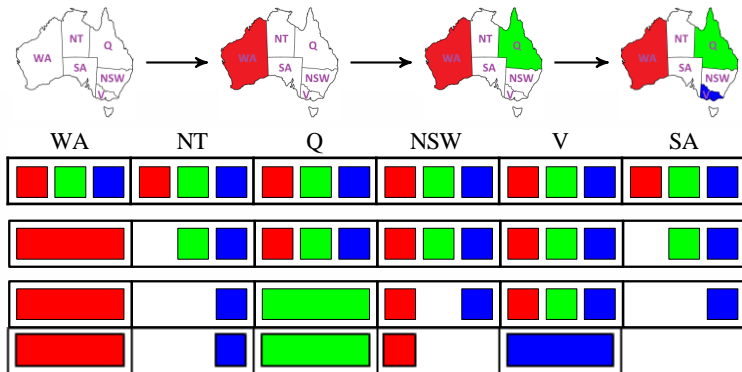
Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment



Filtering: Forward Checking



















- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment



Filtering: 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
					
					
					

Filtering: 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
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<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!

Filtering: 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
<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>
<div><div>Red</div><div>Red</div><div>Red</div></div>	<div><div></div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div></div><div>Green</div><div>Blue</div></div>
<div><div>Red</div><div>Red</div><div>Red</div></div>	<div><div></div><div></div><div>Blue</div></div>	<div><div>Green</div><div>Green</div><div>Green</div></div>	<div><div>Red</div><div></div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div></div><div></div><div>Blue</div></div>

- NT and SA cannot both be blue!
- Why didn't we detect this yet?

Filtering: Constraint Propagation

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

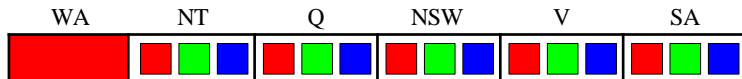


WA	NT	Q	NSW	V	SA
<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>
<div><div>Red</div></div>	<div><div></div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div></div><div>Green</div><div>Blue</div></div>
<div><div>Red</div></div>	<div><div></div><div>Blue</div></div>	<div><div>Green</div></div>	<div><div>Red</div><div></div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div></div><div></div><div>Blue</div></div>

- NT and SA cannot both be blue!
- Why didn't we detect this yet?
- Constraint propagation*: reason from constraint to constraint

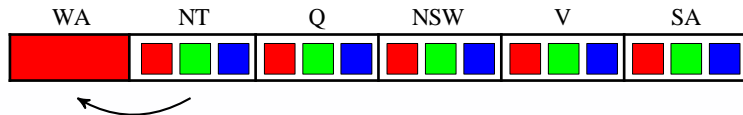
Consistency of A Single Arc

- An arc $X \rightarrow Y$ is **consistent** iff for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint



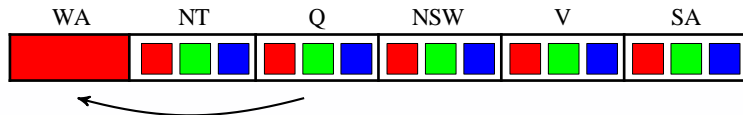
Consistency of A Single Arc

- An arc $X \rightarrow Y$ is **consistent** iff for every x in the tail there is *some* y in the head which could be assigned without violating a constraint



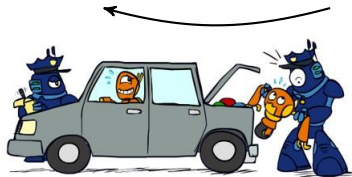
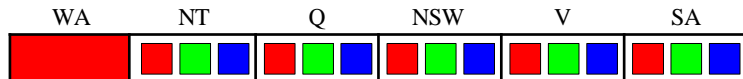
Consistency of A Single Arc

- An arc $X \rightarrow Y$ is **consistent** iff for every x in the tail there is *some* y in the head which could be assigned without violating a constraint



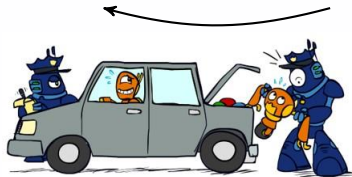
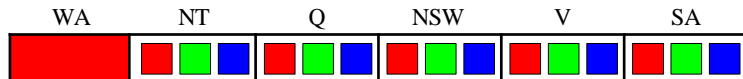
Consistency of A Single Arc

- An arc $X \rightarrow Y$ is **consistent** iff for every x in the tail there is *some* y in the head which could be assigned without violating a constraint



Consistency of A Single Arc

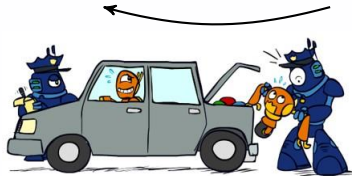
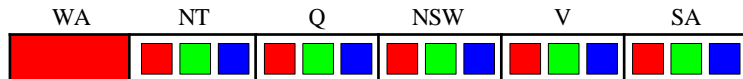
- An arc $X \rightarrow Y$ is **consistent** iff for every x in the tail there is *some* y in the head which could be assigned without violating a constraint



Delete from the tail!

Consistency of A Single Arc

- An arc $X \rightarrow Y$ is **consistent** iff for every x in the tail there is *some* y in the head which could be assigned without violating a constraint

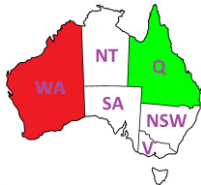


Delete from the tail!

- Forward checking: Enforcing consistency of arcs pointing to each new assignment

Arc Consistency of an Entire CSP

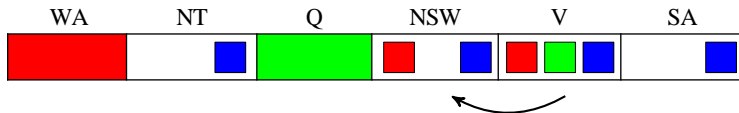
- A simple form of propagation makes sure **all** arcs are consistent:



*Remember:
Delete from the
tail!*

Arc Consistency of an Entire CSP

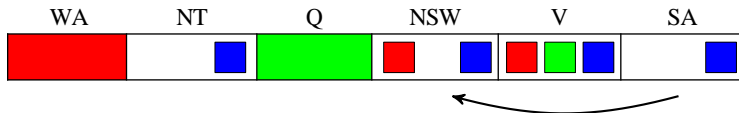
- A simple form of propagation makes sure **all** arcs are consistent:



*Remember:
Delete from the
tail!*

Arc Consistency of an Entire CSP

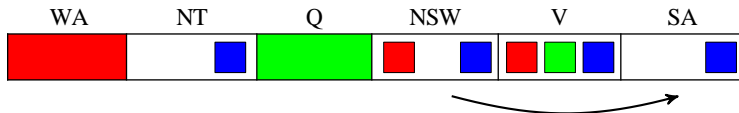
- A simple form of propagation makes sure **all** arcs are consistent:



*Remember:
Delete from the
tail!*

Arc Consistency of an Entire CSP

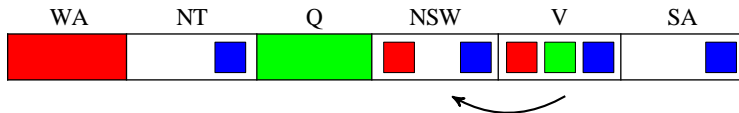
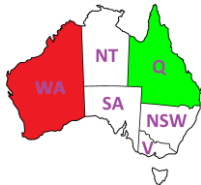
- A simple form of propagation makes sure **all** arcs are consistent:



*Remember:
Delete from the
tail!*

Arc Consistency of an Entire CSP

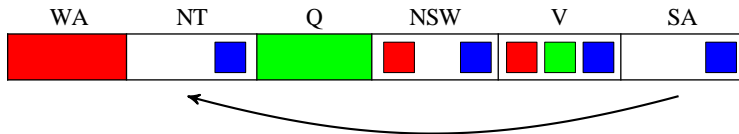
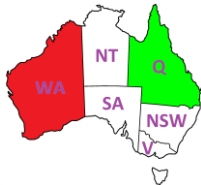
- A simple form of propagation makes sure **all** arcs are consistent:



*Remember:
Delete from the
tail!*

Arc Consistency of an Entire CSP

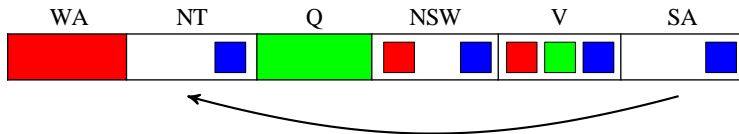
- A simple form of propagation makes sure **all** arcs are consistent:



*Remember:
Delete from the
tail!*

Arc Consistency of an Entire CSP

- A simple form of propagation makes sure **all** arcs are consistent:



- Important: 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 or after each assignment
- What's the downside of enforcing arc consistency?

*Remember:
Delete from the
tail!*

Enforcing Arc Consistency in a CSP

function AC-3(*csp*) returns the CSP, possibly with reduced domains

inputs: *csp*, a binary CSP with variables $\{X_1, X_2, \dots, X_N\}$

local variables: *queue*, a queue of arcs, initially all the arcs in *csp*

while *queue* is not empty do

$(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\textit{queue})$

 if REMOVE-INCONSISTENT-VALUES(X_i, X_j) then

 for each X_k in NEIGHBORS[X_j] do

 add (X_k, X_i) to *queue*

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) returns true iff succeeds

removed \leftarrow false

 for each x in DOMAIN[X_i] do

 if no value y in DOMAIN[X_j] allows (x, y) to satisfy the constraint $X_i \leftrightarrow X_j$

 then delete x from DOMAIN[X_i]; *removed* \leftarrow true

 return *removed*

Applet: CSP - fiveQueens

Suggested Reading

- Russell & Norvig: Chapter 6.1