

# Constraint Satisfaction Problems: Generalising Goal Search II

---

CS3243: Introduction to Artificial Intelligence – Lecture 6

13 February 2023

# Contents

---

1. Administrative Matters
2. Recap on Constraint Satisfaction Problem (CSP) Formulation
3. Variable-Order Heuristics
4. Value-Order Heuristics
5. Inference in CSPs

# Administrative Matters

---

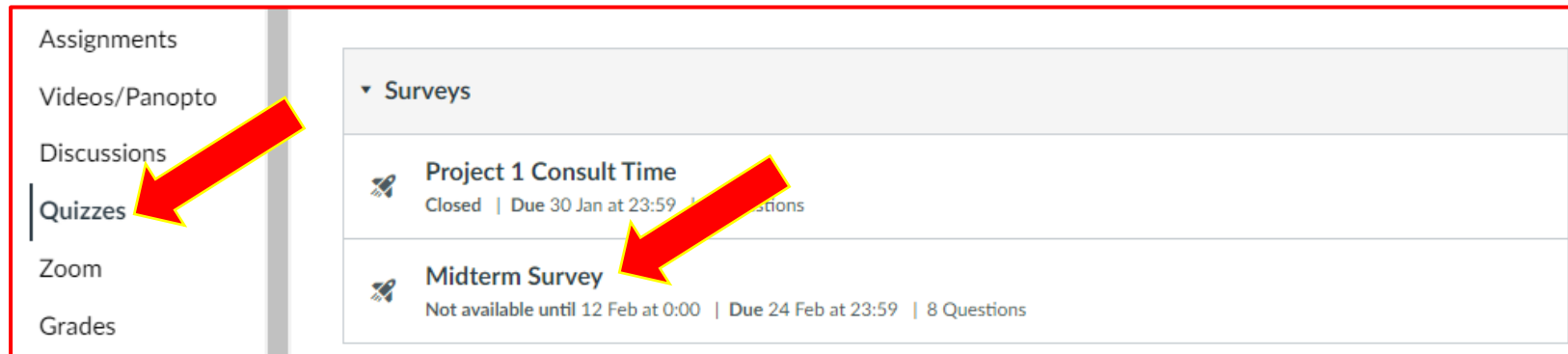
# Project 2 & Midterm Examination

---

- Project 2
  - Released at the end of this week
  - Due in Week 9
  - Implement goal search via local search and CSP
  - Consultation TBA
- Midterm Examination
  - Week 7 Lecture Slot
    - 27 February (Monday), 1030-1130 hrs
    - Conducted in-person at the MPSH1a
  - Duration: 60 minutes
  - Topics: Lectures 1-5 (i.e., everything up to local search – excludes CSPs)

# Midterm Survey

- Particulars
  - Now Open
    - Closes next Friday (24 February), 2359 hrs
  - Anonymous Survey
  - Access at Canvas > CS3243 > Quizzes > Midterm Survey



- Please help us to improve the course by providing feedback

# Upcoming...

---

- Deadlines
  - TA4 (released last week)
    - *Due in your Week 6 tutorial session*
    - *Submit the a physical copy (more instructions on the Tutorial Worksheet)*
  - Prepare for the tutorial!
    - Participation marks = 5%
  - Project 1 (released Week 3 – 25 January)
    - *Due this Sunday (19 February), 2359 hrs*

# Recap on CSPs

---

# Formulating CSPs

---

- State representation
  - Variables:  $X = \{x_1, \dots, x_n\}$
  - Domains:  $D = \{d_1, \dots, d_k\}$ 
    - Such that  $x_i$  has a domain  $d_i$
  - Initial state: all variables unassigned
  - Intermediate state: partial assignment
- Goal test
  - Constraints:  $C = \{c_1, \dots, c_m\}$ 
    - Defined via a constraint language
      - Algebra, Logic, Sets
    - Each  $c_i$  corresponds to a requirement on some subset of  $X$
  - Objective is a **complete** and **consistent** assignment
    - Find a legal assignment  $(y_1, \dots, y_n)$ 
      - $y_i \in d_i$  for all  $i \in [n]$
    - Complete: all variables assigned values
    - Consistent: all constraints  $C$  satisfied
- Actions, costs and transition
  - Assignment of values (within domain) to variables
  - Costs are not utilised

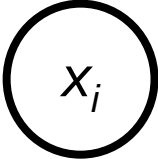



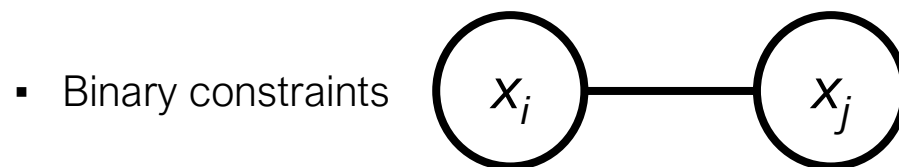
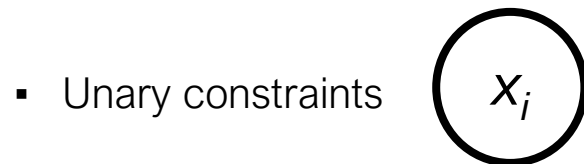
# More on Constraints

- A language is necessary to express the constraints
  - Arithmetic
  - Sets (of legal values)
  - Logic
- For example,  $x_1$  greater than  $x_2$  given  $d = \{1, 2, 3\}$  may be written
  - $\langle (x_1, x_2), x_1 > x_2 \rangle$
  - $\langle (x_1, x_2), \{ (2, 1), (3, 1), (3, 2) \} \rangle$
- Each constraint,  $c_i$ ,
  - Describes the necessary relationship, **rel**, between a set of variables, **scope**
    - For the example above, **scope** =  $(x_1, x_2)$ . **rel** =  $x_1 > x_2$
- Types of constraints
  - Unary:  $|\text{scope}| = 1$
  - Binary:  $|\text{scope}| = 2$
  - Global:  $|\text{scope}| > 2$  (i.e., higher-order constraints)

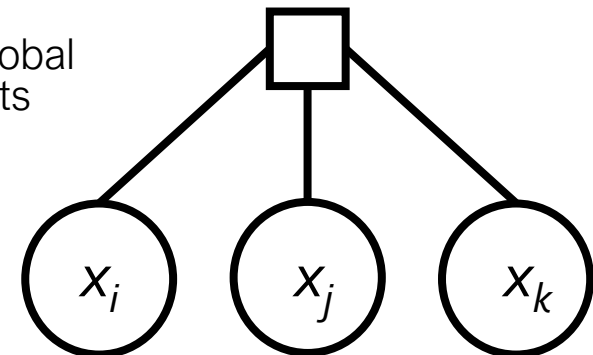
# Drawing Constraint Graphs and Hypergraphs

- Constraint graphs represent the constraints in a CSP

- Simple Vertex: variable 
- Linking Vertex: for global constraints 
- Edge: links all variables in the scope of a constraint (*rel*)

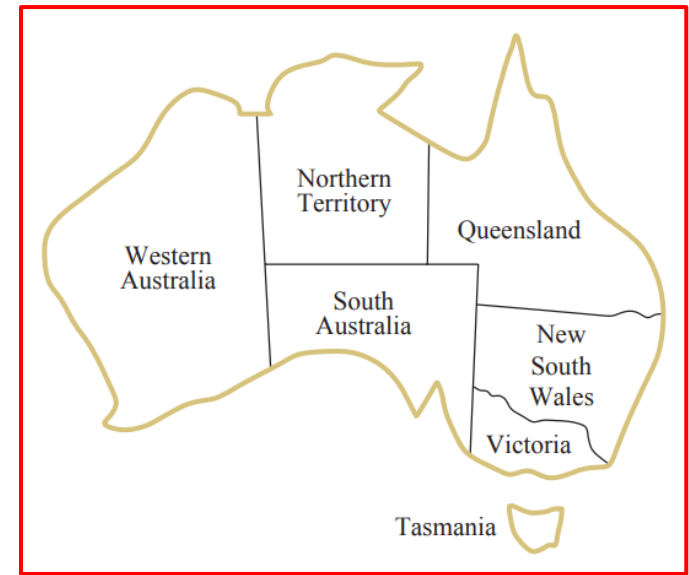


- Binary/Global constraints

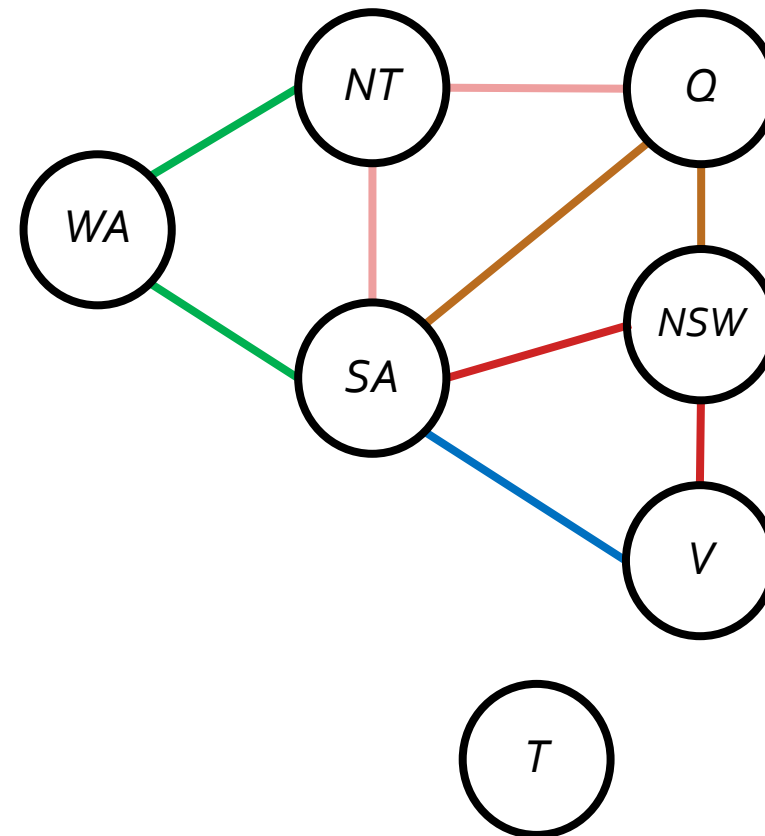
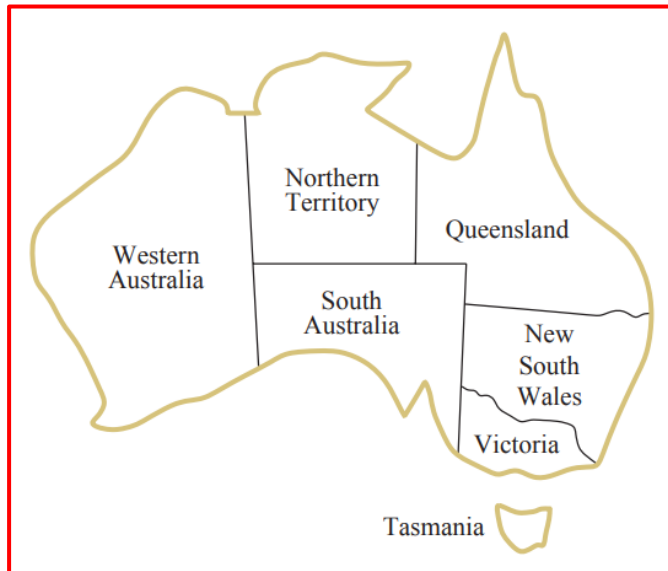


# CSP Formulation Example: Graph Colouring

- Colour each state of Australia such that no two adjacent states share the same colour
- Variables
  - $X = \{ WA, NT, Q, NSW, V, SA, T \}$
- Domains
  - $d_i = \{ \text{Red, Green, Blue} \}$
- Constraints
  - $\forall (x_i, x_j) \in E, \text{colour}(x_i) \neq \text{colour}(x_j)$



# Constraint Graph for Example: Graph Colouring



# Backtracking Algorithm for CSPs

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

```
function BACKTRACK(csp, assignment) returns a solution or failure
```

```
  if assignment is complete then return assignment
```

```
  var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
```

```
  for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
```

```
    if value is consistent with assignment then
```

```
      add {var = value} to assignment
```

```
      inferences ← INFERENCE(csp, var, assignment)
```

```
      if inferences ≠ failure then
```

```
        add inferences to csp
```

```
        result ← BACKTRACK(csp, assignment)
```

```
        if result ≠ failure then return result
```

```
        remove inferences from csp
```

```
        remove {var = value} from assignment
```

```
  return failure
```

Determine the variable to assign to

1

Determine the value to assign

2

Trying to determine if the chosen assignment will lead to a terminal state

3

Continues recursively as long as the *assignment* is *viable*

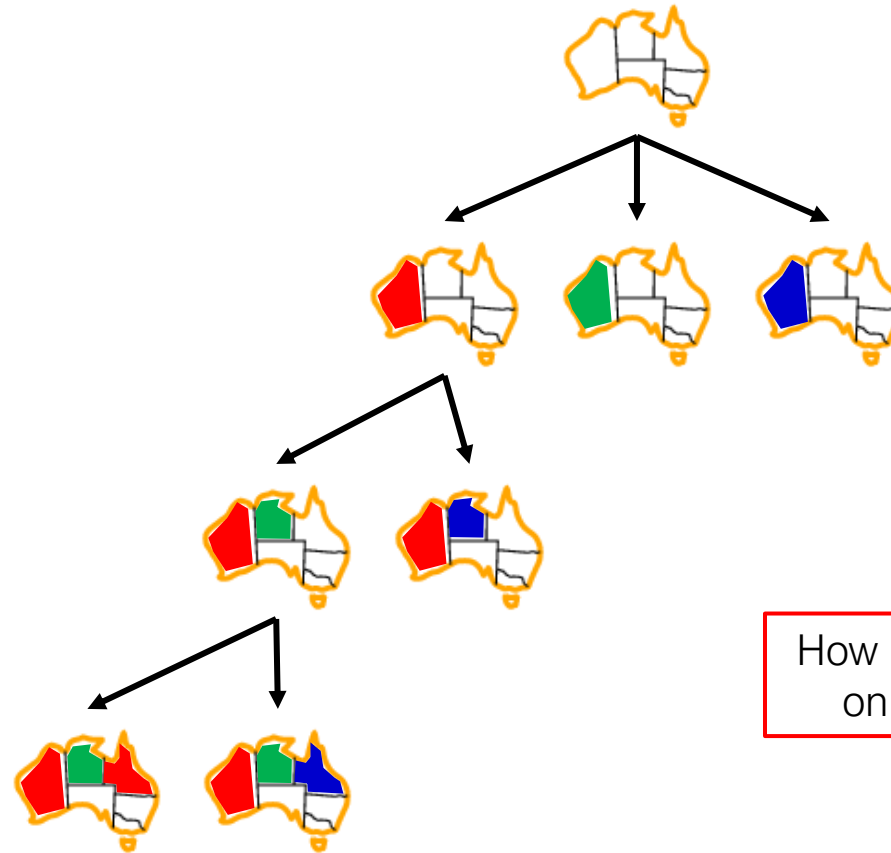
*General purpose* heuristics for 1, 2 and 3 can lead to improved search efficiency

# Variable-Order Heuristics

---

# Backtracking Example: Graph Colouring

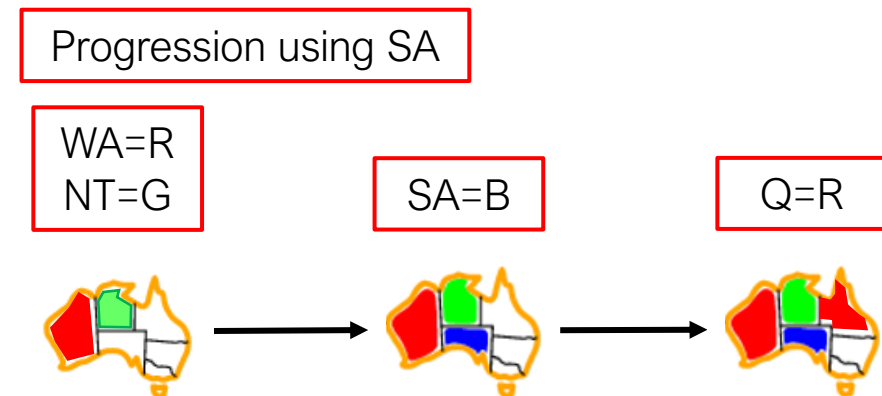
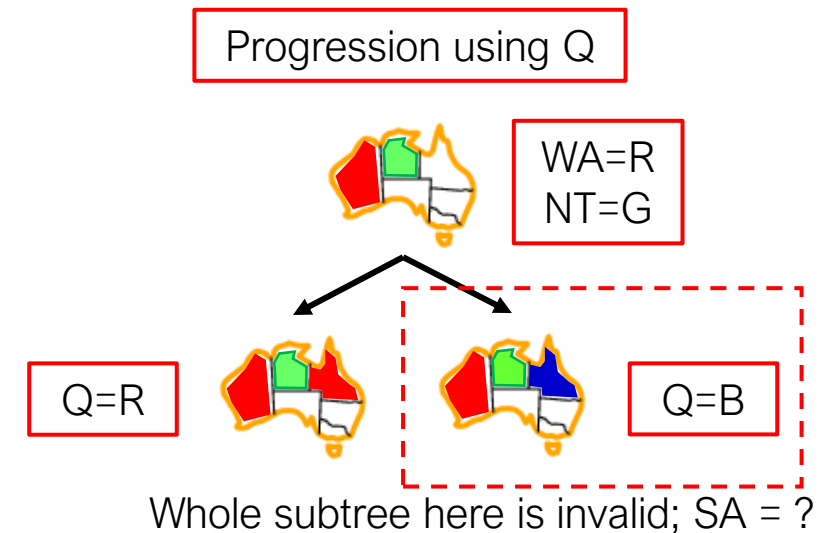
- During backtracking search
  - Assign variables in some order
    - For example:
      - WA
      - NT
      - Q
      - etc ...



How do we decide  
on the order?

# Minimum-Remaining-Values Heuristic

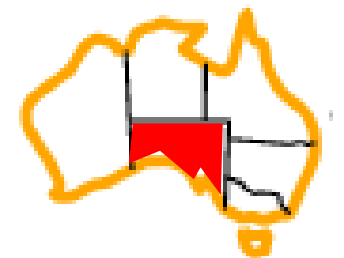
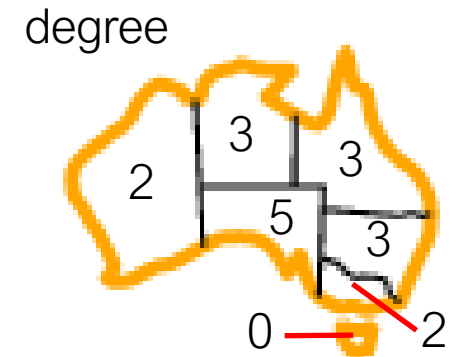
- Choose the variable with fewest legal values
  - Minimum-remaining-values (MRV) heuristic
  - Also considered the most constrained variable
  - Smallest consistent domain size among unassigned variables
- General idea
  - Places larger subtrees closer to the root
    - Any invalid state found prunes a larger subtree
  - Eliminate larger subtrees earlier
- Consider the Australia Colouring problem
  - Suppose we start with WA = R and NT = G
    - Remaining values: SA = 1, Q = 2, Rest = 3
    - MRV suggest selecting SA, what if we use Q?
- MRV usually performs better than static or random ordering





# Degree Heuristic

- MRV heuristic requires tie-breaking
  - E.g., at initial state all variables have same RV
- Tie-break with degree heuristic
  - Picks variable with most constraints relative to unassigned variables
- General idea
  - By selecting variable that restricts the most number of other variables, we reduce b
- Consider the Australia Colouring problem
  - Using MRV with degree for tie-breaking
    - Initially state
      - all states have same RV; tie-break with degree
      - chosen variable is SA
    - After assigning to SA, any assignment leads to a solution without any backtracking



Recommended variable selection:  
MRV, then degree, then random

# Backtracking Algorithm for CSPs

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

```
function BACKTRACK(csp, assignment) returns a solution or failure
```

```
  if assignment is complete then return assignment
```

```
  var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
```

```
  for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
```

```
    if value is consistent with assignment then
```

```
      add {var = value} to assignment
```

```
      inferences ← INFERENCE(csp, var, assignment)
```

```
      if inferences ≠ failure then
```

```
        add inferences to csp
```

```
        result ← BACKTRACK(csp, assignment)
```

```
        if result ≠ failure then return result
```

```
        remove inferences from csp
```

```
        remove {var = value} from assignment
```

```
  return failure
```

Determine the variable to assign to

1

Determine the value to assign

2

Trying to determine if the chosen assignment will lead to a terminal state

3

Continues recursively as long as the *assignment* is *viable*

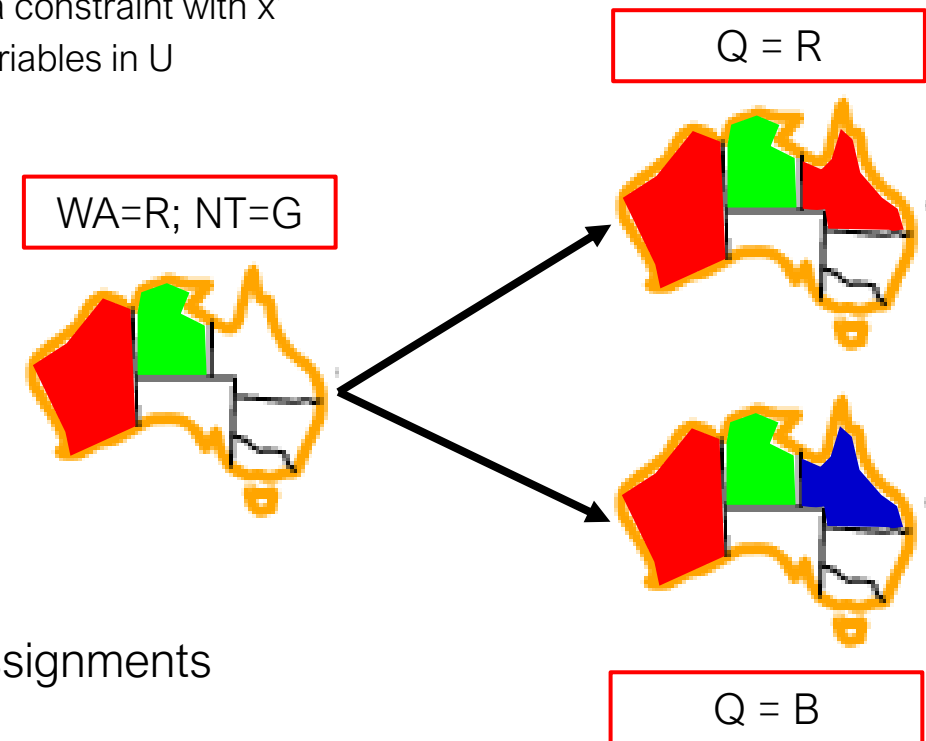
*General purpose* heuristics for 1, 2 and 3 can lead to improved search efficiency

# Value-Order Heuristics

---

# Least-Constraining-Value Heuristic

- Choose the value that rules out the fewest choices
  - Least-constraining-value (LCV) heuristic
    - Given assignment of value  $v$  to variable  $x'$
    - Determine unassigned variables  $U = \{x_a, x_b, \dots\}$  that share a constraint with  $x'$
    - Pick  $v$  that maximises sum of consistent domain sizes of variables in  $U$
- General idea
  - Avoid failure (i.e., avoid empty domains)
- Consider the Australia Colouring problem
  - Suppose we start with  $WA = R$  and  $NT = G$
  - Suppose our next choice is  $Q$ 
    - $Q = R$  leaves  $SA = B$
    - $Q = B$  leaves  $SA = \text{None}$
    - Select  $Q = R$  (since it constrains 1 less than  $Q = B$ )
- Tries to leave maximum flexibility for subsequent assignments



# Why Different Strategies with Variables & Values?

---

- With variables: fail-first
  - Every variable must be assigned to arrive at a solution
  - Must look at all variables
  - Fail-first strategy on average leads to fewer successful assignments to backtrack over
- With values: fail-last
  - Only one solution required
  - May not have to look at some values
  - If all solutions required, then value-ordering irrelevant

# Backtracking Algorithm for CSPs

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

```
function BACKTRACK(csp, assignment) returns a solution or failure
```

```
  if assignment is complete then return assignment
```

```
  var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
```

```
  for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
```

```
    if value is consistent with assignment then
```

```
      add {var = value} to assignment
```

```
      inferences ← INFERENCE(csp, var, assignment)
```

```
      if inferences ≠ failure then
```

```
        add inferences to csp
```

```
        result ← BACKTRACK(csp, assignment)
```

```
        if result ≠ failure then return result
```

```
        remove inferences from csp
```

```
        remove {var = value} from assignment
```

```
  return failure
```

Determine the variable to assign to

1

Determine the value to assign

2

Trying to determine if the chosen assignment will lead to a terminal state

3

Continues recursively as long as the *assignment* is *viable*

*General purpose* heuristics for 1, 2 and 3 can lead to improved search efficiency

# Inference in CSPs

---

# Avoiding Failure

---

- With certain states, we know we are heading for failure



- Searching such subtrees is a waste of computation
- How can we detect these as early as possible?



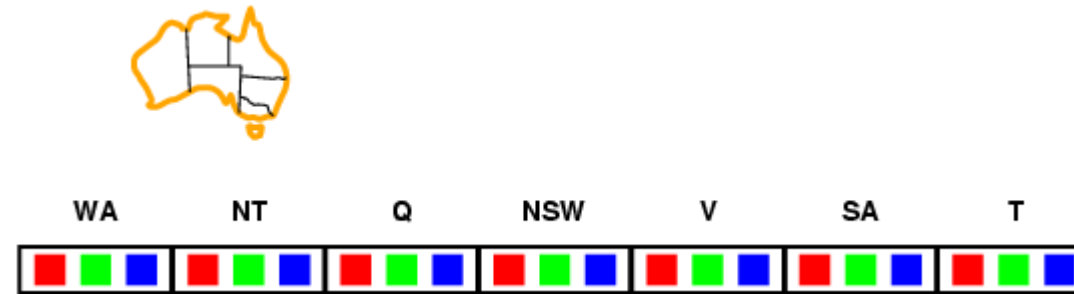
# Forward Checking

---

# Forward Checking

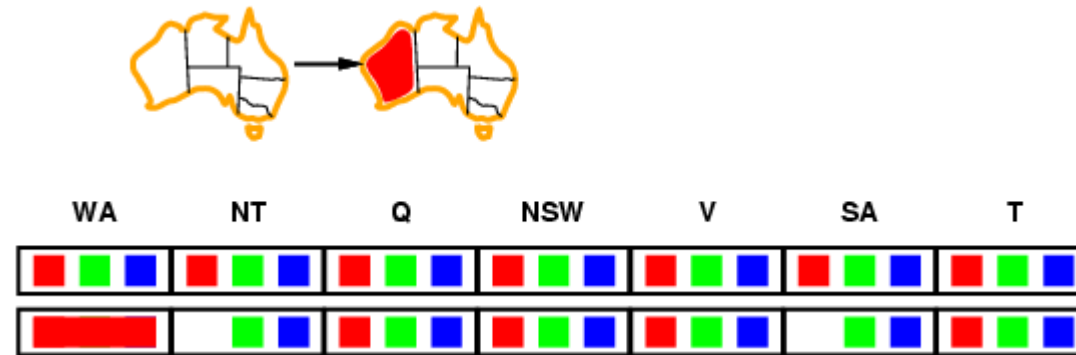
---

- Track remaining legal values for unassigned variables
- Terminate search when any variable has no legal values



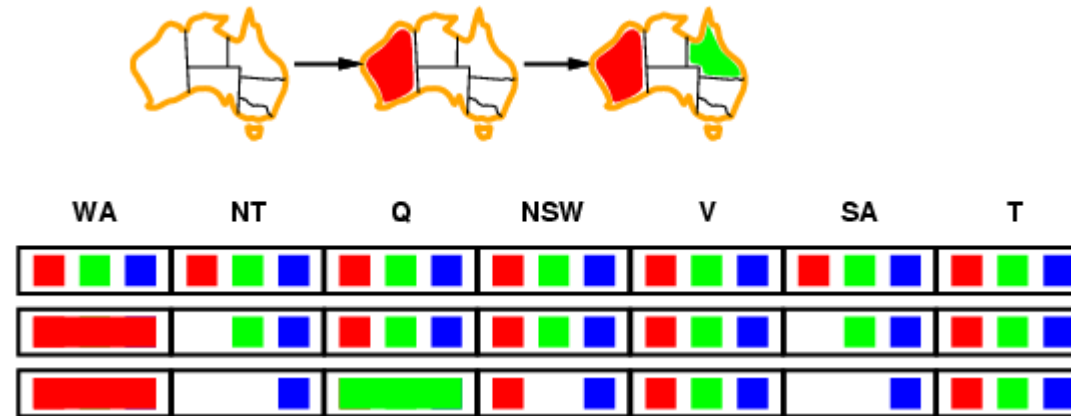
# Forward Checking

- Track remaining legal values for unassigned variables
- Terminate search when any variable has no legal values



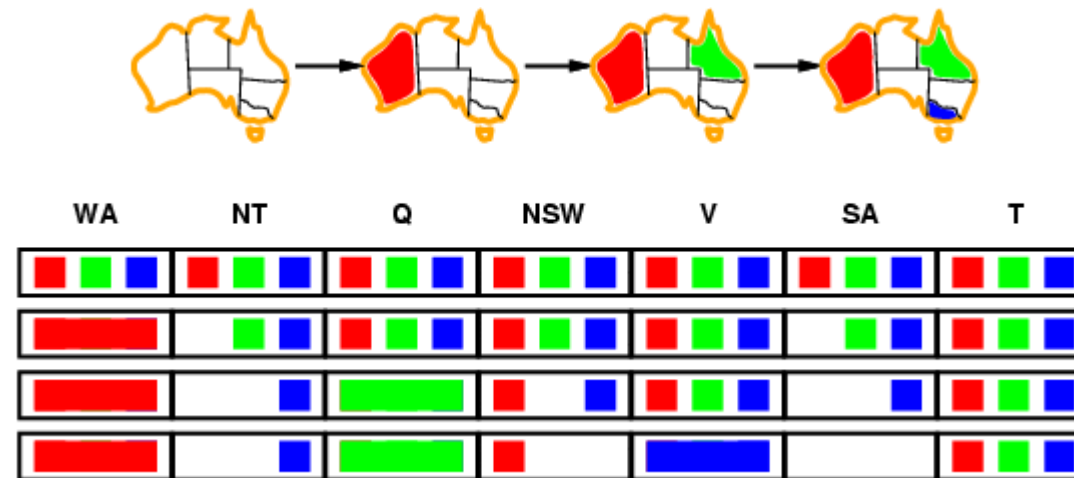
# Forward Checking

- Track remaining legal values for unassigned variables
- Terminate search when any variable has no legal values



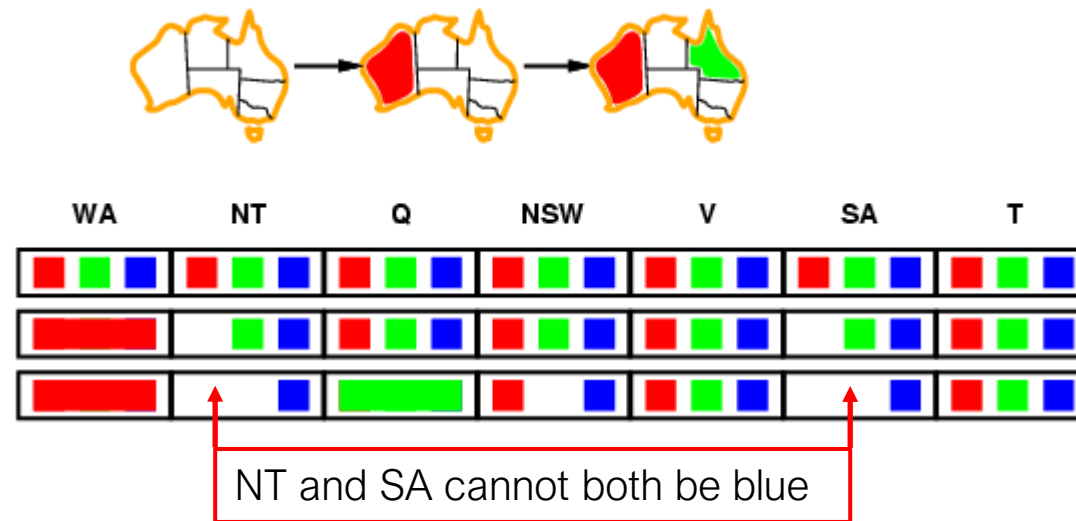
# Forward Checking

- Track remaining legal values for unassigned variables
- Terminate search when any variable has no legal values



# The Issue with Forward Checking

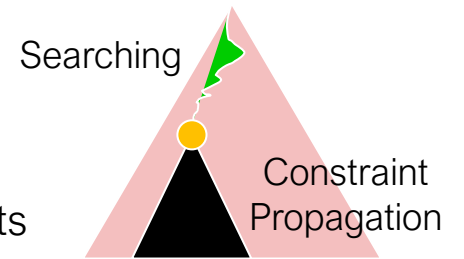
- Problem: forward checking propagates information from assigned to unassigned variables, but *does not provide early detection for all failures*



Solution: ensuring consistency via constraint propagation

# Constraint Propagation

- Inference step to ensure local consistency of ALL variables
  - Traverse constraint graph to ensure variable at each node is consistent
    - Eliminate all values in variable's domain that are not consistent with linked constraints
- Node-consistent
  - A single variable is node-consistent if its domain is consistent with related unary constraints
- Arc-consistent (i.e., edge-consistent)
  - A single variable is arc-consistent if its domain is consistent with related binary constraints



All global constraints may be transformed into binary constraints via hidden variable encoding (use a new variable whose domain contains legal n-tuples domain values for variables in global constraint) or dual encoding (general idea described in AIMA pp.168-169)

For CS3243: assume all questions requiring algorithm traces done over unary/binary constraints only (global constraints only present in CSP formulation questions)

# Questions about the Lecture?

---

- Was anything unclear?
- Do you need to clarify anything?
- Ask on Archipelago
  - Specify a question
  - Upvote someone else's question



Invitation Link (Use NUS Email --- starts with E)  
<https://archipelago.rocks/app/resend-invite/91207333004>



# Node Consistency

---

# Node Consistency

---

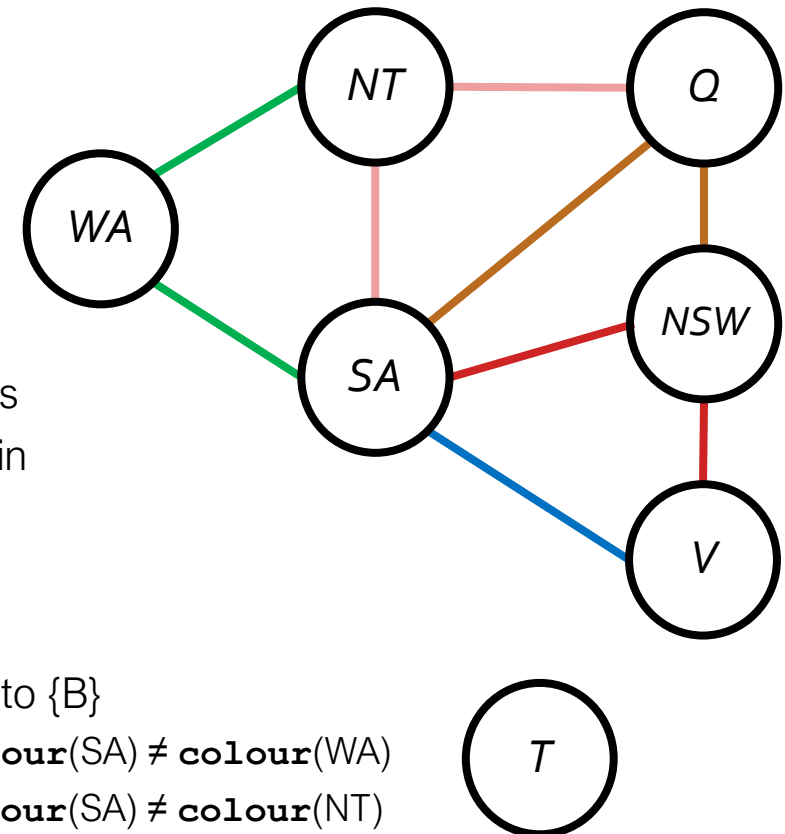
- Node-consistent
  - A single variable is node-consistent if its domain is consistent with related unary constraints
- Trivial to ensure node consistency
  - Only concerned with unary constraints
  - For each variable
    - Eliminate domain values inconsistent with unary constraints
  - Perform the above as a pre-processing step (i.e., before application of backtracking)

# Arc Consistency

---

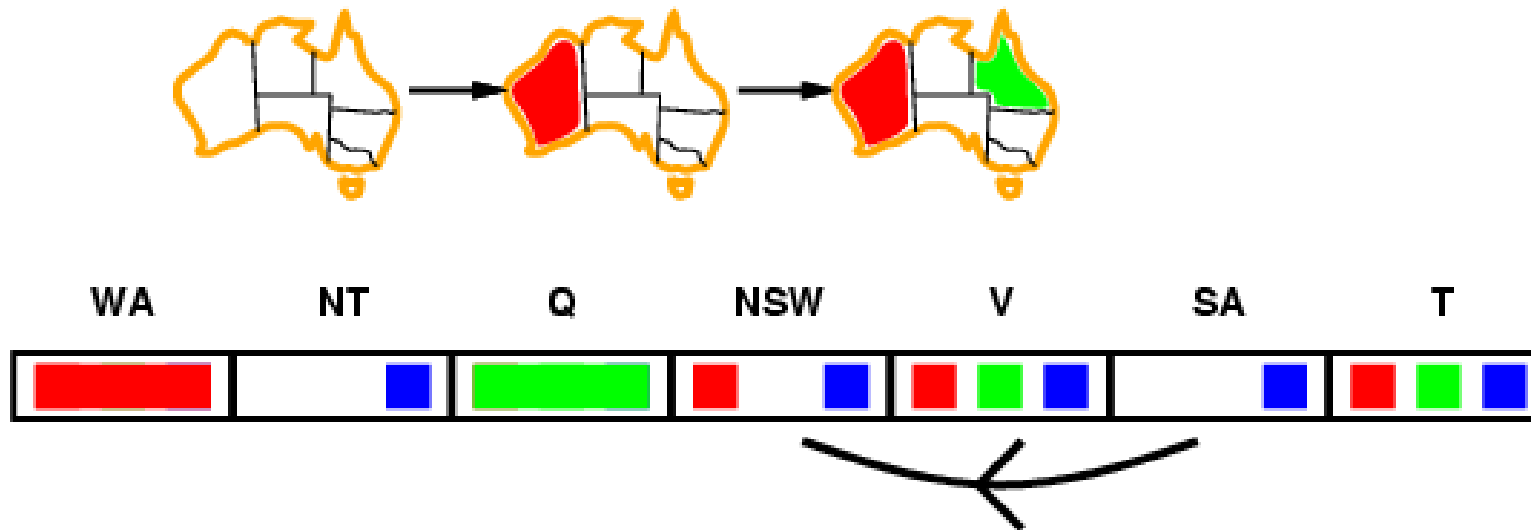
# Arc Consistency

- Arc-consistent (i.e., edge-consistent)
  - A single variable is arc-consistent if its domain is consistent with related binary constraints
- Ensuring arc consistency (AC)
  - For each variable
    - Eliminate domain values inconsistent with binary constraints
    - Variable domain value must have partnering domain value in other variable that will satisfy constraint
  - Consider the Australia Colouring problem
    - Assume  $D_{WA} = \{R\}$ ,  $D_{NT} = \{G\}$ , domains of rest =  $\{R, G, B\}$
    - To ensure arc consistency at SA, reduce  $D_{SA}$  from  $\{R, G, B\}$  to  $\{B\}$ 
      - Eliminate  $SA = R$  since there is no value in  $D_{WA}$  that satisfies  $\text{colour}(SA) \neq \text{colour}(WA)$
      - Eliminate  $SA = G$  since there is no value in  $D_{NT}$  that satisfies  $\text{colour}(SA) \neq \text{colour}(NT)$



# Arc Consistency

$X_i$  is arc-consistent wrt  $X_j$  (i.e., the arc  $(X_i, X_j)$  is consistent) iff for every value  $x \in D_i$  there exists some value  $y \in D_j$  that satisfies the binary constraint on the arc  $(X_i, X_j)$

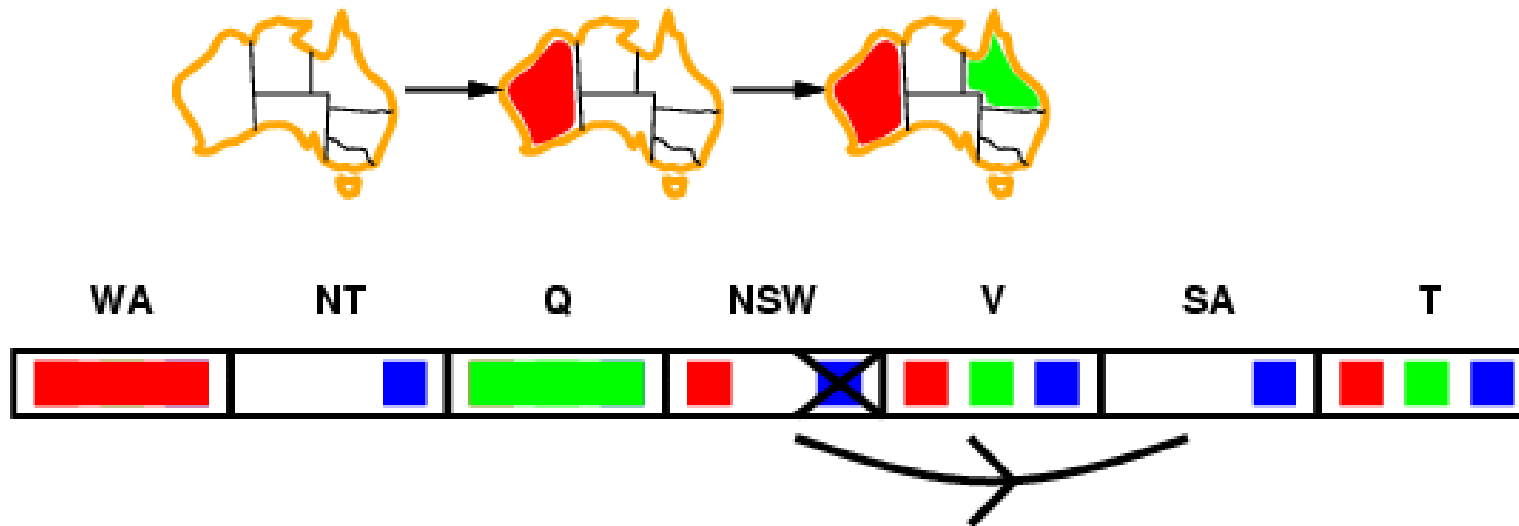


To maintain AC, remove any value from the target variable if it makes a constraint impossible to satisfy

Arc (SA, NSW) is consistent since the constraint linking SA and NSW is still satisfied when NSW = R

# Arc Consistency

$X_i$  is arc-consistent wrt  $X_j$  (i.e., the arc  $(X_i, X_j)$  is consistent) iff for every value  $x \in D_i$  there exists some value  $y \in D_j$  that satisfies the binary constraint on the arc  $(X_i, X_j)$

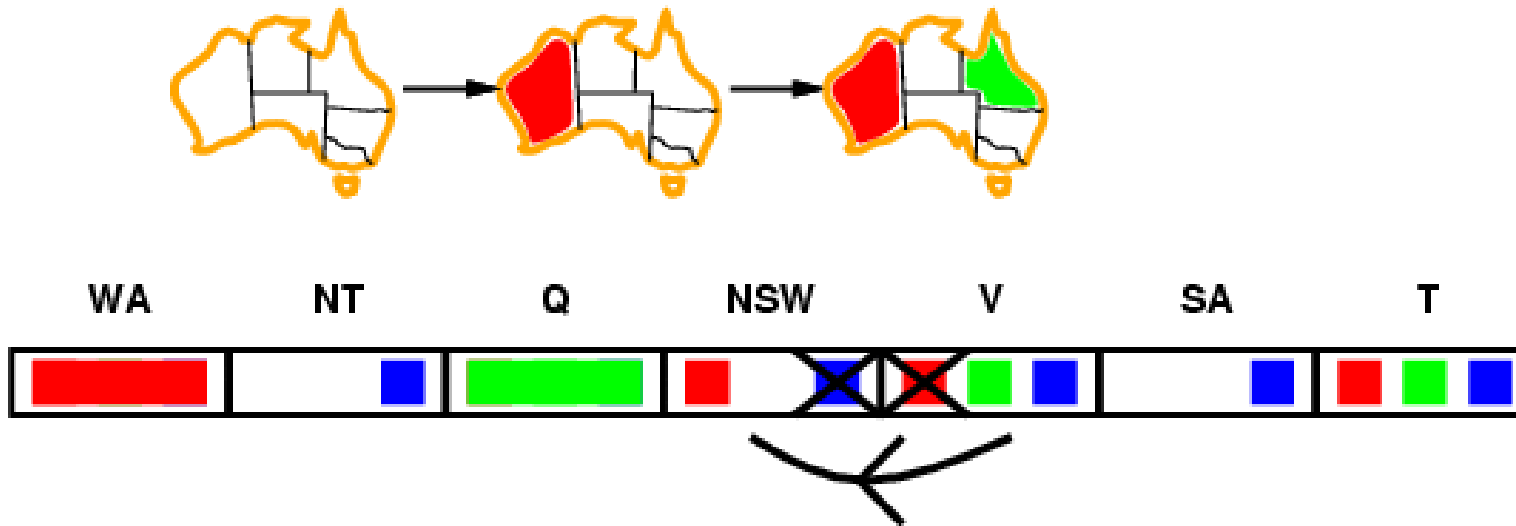


Arcs are directed  
A binary constraint becomes two arcs

Arc (NSW, SA) is originally not consistent  
It becomes consistent after deleting NSW = B

# Arc Consistency

$X_i$  is arc-consistent wrt  $X_j$  (i.e., the arc  $(X_i, X_j)$  is consistent) iff for every value  $x \in D_i$  there exists some value  $y \in D_j$  that satisfies the binary constraint on the arc  $(X_i, X_j)$

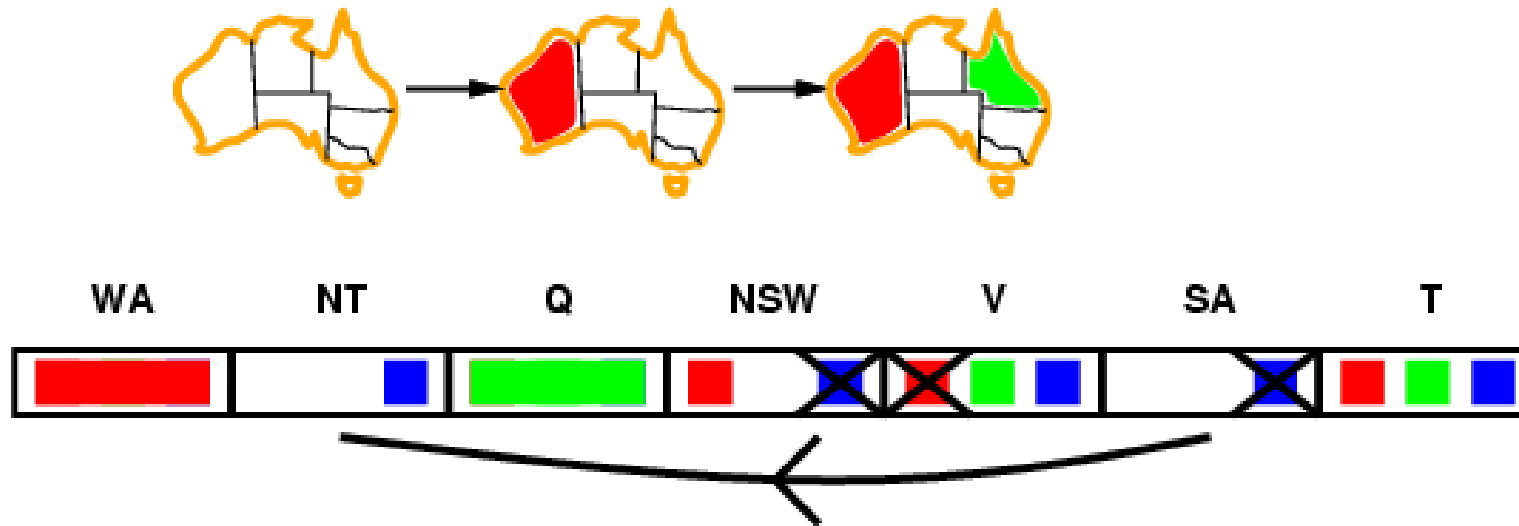


If  $X_i$  loses a value, neighbours of  $X_i$  need to be (re)checked

After deleting NSW = B, Arc (V, NSW) is originally not consistent  
It becomes consistent after deleting V = R

# Arc Consistency

$X_i$  is arc-consistent wrt  $X_j$  (i.e., the arc  $(X_i, X_j)$  is consistent) iff for every value  $x \in D_i$  there exists some value  $y \in D_j$  that satisfies the binary constraint on the arc  $(X_i, X_j)$



Arc consistency propagation detects failure earlier than forward checking

Terminal state detection requires the check on ALL arcs

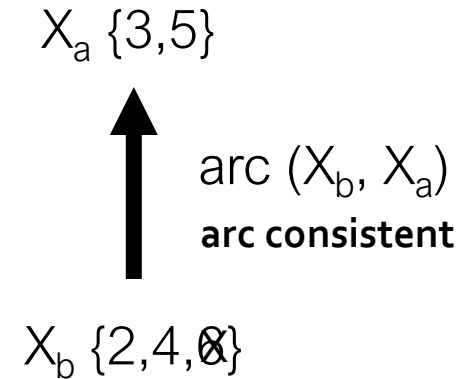
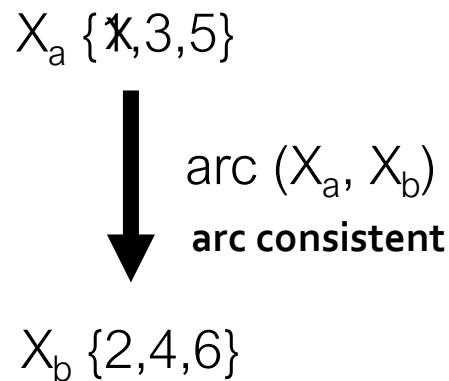
Can be run within backtracking (after each variable assignment), or as a pre-processing step before backtracking begins



# Arc Consistency Example 1

- Example

- $D_a = \{1, 3, 5\}$
- $D_b = \{2, 4, 6\}$
- $X_a > X_b$

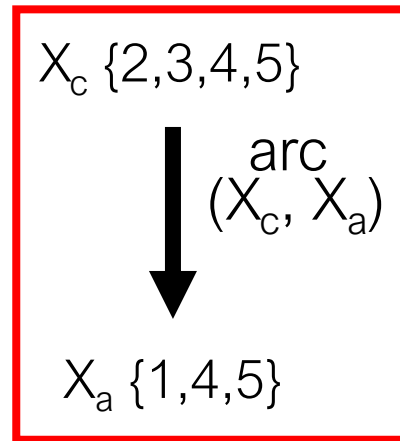


- Should arc  $(X_a, X_b)$  be checked again after the update to  $X_b$ ?
  - No
  - Notice that 1 in  $D_a$  was not required to satisfy any value (2,4,6) from  $D_b$
  - Notice that 6 in  $D_b$  was not required to satisfy any value (1,3,5) from  $D_a$

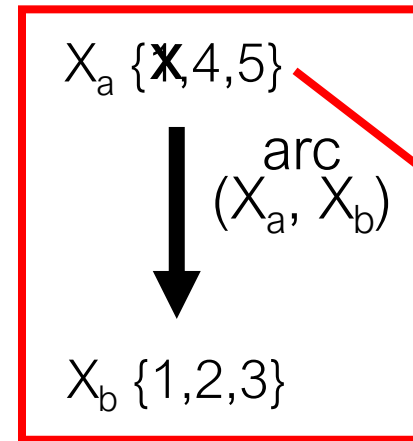
# Arc Consistency Example 2

- Example

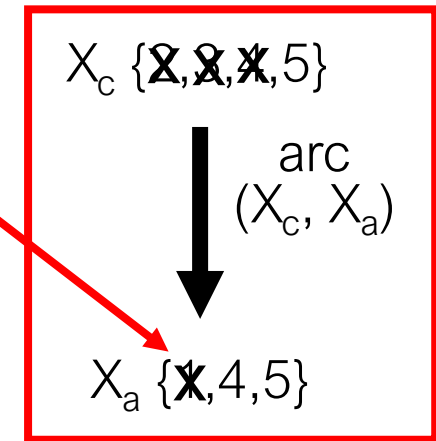
- $D_a = \{1,4,5\}$
- $D_b = \{1,2,3\}$
- $D_c = \{2,3,4,5\}$
- $X_c > X_a$
- $X_a > X_b$



arc consistent



arc consistent



arc consistent

- Constraint propagation may result in a chain-reaction of domain reductions

# Sudoku Chain Reaction

- Consider the following Sudoku puzzle
  - **Alldiff** constraint on middle box reduces domain of red square  $\{3,4,5,6,9\}$
  - Column constraint further reduces it to  $\{4\}$
  - Box and column constraints on the orange square reduce its domain to  $\{4, 7\}$
  - Red square further reduces it to  $\{7\}$
  - Blue box now has domain  $\{1\}$  since the rest of the column is defined

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

# AC-3 Algorithm

**function** AC-3(*csp*) **returns** false if an inconsistency is found and true otherwise

*queue*  $\leftarrow$  a queue of arcs, initially all the arcs in *csp*

Initialise a queue containing all arcs (both directions for each binary constraint)

**while** *queue* is not empty **do**

    (*X<sub>i</sub>*, *X<sub>j</sub>*)  $\leftarrow$  POP(*queue*)

**if** REVISE(*csp*, *X<sub>i</sub>*, *X<sub>j</sub>*) **then**

**if** size of *D<sub>i</sub>* = 0 **then return** false

**for each** *X<sub>k</sub>* **in** *X<sub>i</sub>*.NEIGHBORS - {*X<sub>j</sub>*} **do**

            add (*X<sub>k</sub>*, *X<sub>i</sub>*) to *queue*

Each time a variable *X<sub>i</sub>*'s domain is updated add all arcs corresponding to binary constraints with other variable (not *X<sub>i</sub>*) as target (except the one that just caused the revision)

**return** true

**function** REVISE(*csp*, *X<sub>i</sub>*, *X<sub>j</sub>*) **returns** true iff we revise the domain of *X<sub>i</sub>*

*revised*  $\leftarrow$  false

**for each** *x* **in** *D<sub>i</sub>* **do**

**if** no value *y* in *D<sub>j</sub>* allows (*x*,*y*) to satisfy the constraint between *X<sub>i</sub>* and *X<sub>j</sub>* **then**

            delete *x* from *D<sub>i</sub>*

Eliminating domain values of the target variable *X<sub>i</sub>* relative to the other variable *X<sub>j</sub>* in the binary constraint

*revised*  $\leftarrow$  true

**return** *revised*

# Time Complexity of AC-3

- CSPs have at most  $2^n C_2$  or  $O(n^2)$  directed arcs (given  $n$  variables)
- Each arc  $(X_i, X_j)$  can be inserted at most  $d$  times because  $X_i$  has at most  $d$  values to delete (given domain size  $d$ )
  - Checking consistency of an arc (REVISE function) takes  $O(d^2)$  time
- AC-3 time complexity:
  - $O(n^2 \times d \times d^2) = O(n^2 d^3)$

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
    queue  $\leftarrow$  a queue of arcs, initially all the arcs in csp

    while queue is not empty do
        (Xi, Xj)  $\leftarrow$  POP(queue)
        if REVISE(csp, Xi, Xj) then
            if size of Di = 0 then return false
            for each Xk in Xi.NEIGHBORS - {Xj} do
                add (Xk, Xi) to queue
    return true

function REVISE(csp, Xi, Xj) returns true iff we revise the domain of Xi
    revised  $\leftarrow$  false
    for each x in Di do
        if no value y in Dj allows (x,y) to satisfy the constraint between Xi and Xj then
            delete x from Di
            revised  $\leftarrow$  true
    return revised
```

# Maintaining Arc Consistency (MAC)

---

- AC usage
  - Pre-processing step before backtracking begins
    - Reduces domain sizes, so reduces size of search tree
  - After each variable assignment within backtracking
    - Inference to update domains
    - Checks for terminal state (i.e., if any domain empty)
      - Backtrack from terminal states
    - Only initialise queue with arcs of neighbouring unassigned variables (relative to current node)

# Questions about the Lecture?

---

- Was anything unclear?
- Do you need to clarify anything?
- Ask on Archipelago
  - Specify a question
  - Upvote someone else's question



Invitation Link (Use NUS Email --- starts with E)  
<https://archipelago.rocks/app/resend-invite/91207333004>

# Appendix

---



# AC-3 Algorithm

**function** AC-3(*csp*) **returns** false if an inconsistency is found and true otherwise

*queue*  $\leftarrow$  a queue of arcs, initially all the arcs in *csp*

**while** *queue* is not empty **do**

    (*X<sub>i</sub>*, *X<sub>j</sub>*)  $\leftarrow$  POP(*queue*)

**if** REVISE(*csp*, *X<sub>i</sub>*, *X<sub>j</sub>*) **then**

**if** size of *D<sub>i</sub>* = 0 **then return** false

**for each** *X<sub>k</sub>* **in** *X<sub>i</sub>*.NEIGHBORS - {*X<sub>j</sub>*} **do**

            add (*X<sub>k</sub>*, *X<sub>i</sub>*) to *queue*

**return** true

**function** REVISE(*csp*, *X<sub>i</sub>*, *X<sub>j</sub>*) **returns** true iff we revise the domain of *X<sub>i</sub>*

*revised*  $\leftarrow$  false

**for each** *x* **in** *D<sub>i</sub>* **do**

**if** no value *y* in *D<sub>j</sub>* allows (*x*,*y*) to satisfy the constraint between *X<sub>i</sub>* and *X<sub>j</sub>* **then**

        delete *x* from *D<sub>i</sub>*

*revised*  $\leftarrow$  true

**return** *revised*

Why not enqueue arc (*X<sub>j</sub>*, *X<sub>i</sub>*) ?

On initialisation, we enqueue all arcs

So we have either already checked arc (*X<sub>j</sub>*, *X<sub>i</sub>*) or else it is in the queue

If it is on the queue, then we would check it anyway, so no need to enqueue it again

If we have already checked it, then arc (*X<sub>j</sub>*, *X<sub>i</sub>*) was consistent, and we need to know if any change of the domain of *X<sub>i</sub>* could cause arc (*X<sub>j</sub>*, *X<sub>i</sub>*) to become inconsistent

However, we know that all values removed from *X<sub>j</sub>* did not have values in *X<sub>i</sub>* that satisfied the constraint; therefore, there must not be any value in *X<sub>i</sub>* that requires that value in *X<sub>j</sub>* either

Once both arcs corresponding to a binary constraint have been checked, there is no need to propagate the checking further