Constraint Satisfaction Problems: Generalising Goal Search II

CS3243: Introduction to Artificial Intelligence – Lecture 6

NUS | Computing
National University of Singapore

1 Administrative Matters

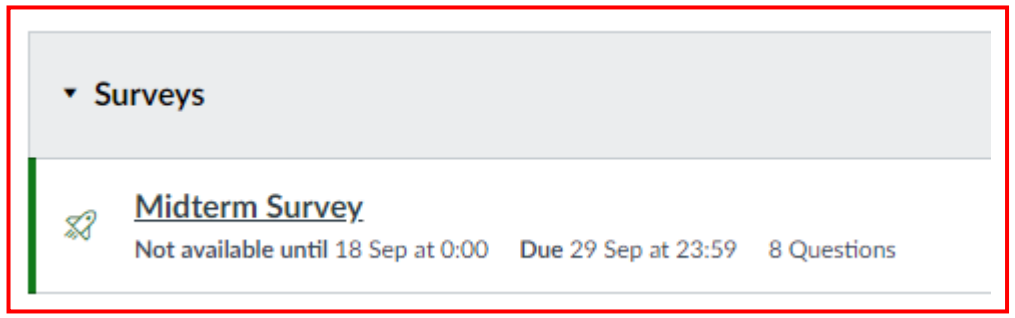
Midterm Assessment

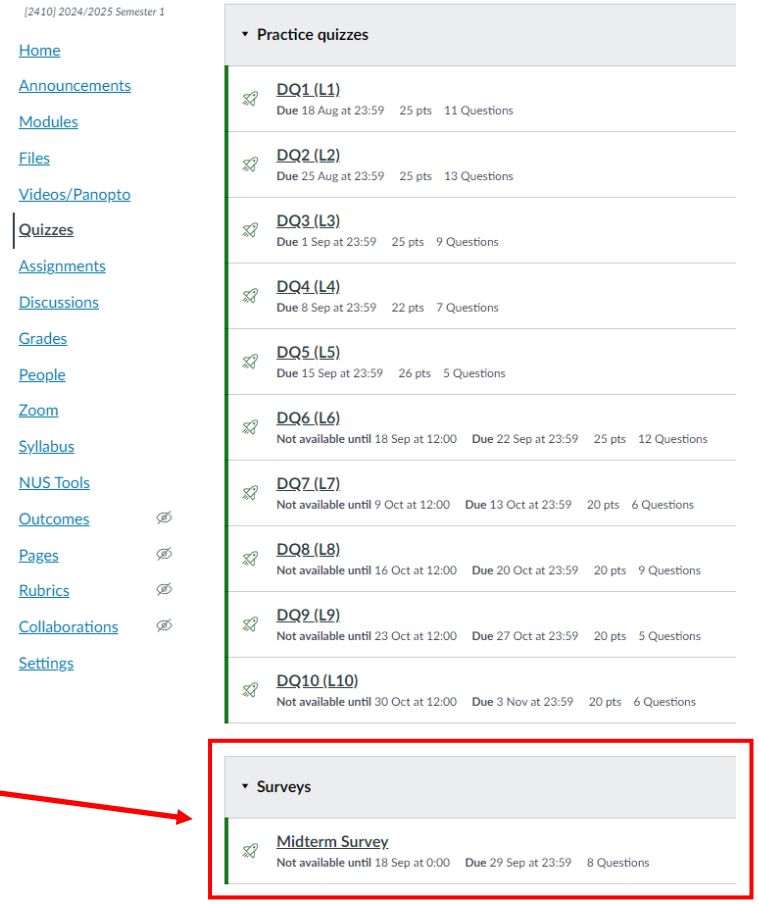
- Week 7 Lecture Slot
  - Date: 2 October (Wednesday)
  - Time: 1005-1135 hrs
  - Venue: LT16 (COM2)
- Format
  - Duration: 1 hour and 30 minutes
  - Total: 30 marks
  - Closed-book + Cheat Sheet (1 × Double-sided A4 Sheet)
- Topics
  - Everything up to Lecture 5a (i.e., up to and including Local Search)

Practice Papers: Canvas > CS3243 > Files > Past Papers > Midterm

Midterm Survey

- Particulars
  - Now Open
    - Closes on 3 MAR 2359 hrs
    - Sunday of Recess Week
  - Anonymous Survey
- Please help us to improve the course by providing feedback
- Canvas > CS3243 > Quizzes > Midterm Survey





Upcoming...

- Deadlines
  - Tutorial Assignment 4 (released last week)
    - Post-tutorial submission: Due this Friday!
  - Tutorial Assignment 5 (released today)
    - Pre-tutorial Submission: Due this Sunday!
    - Post-tutorial Submission: Due next Friday!
  - Project 1.2 (released 26 August)
    - Due THIS Sunday!

      22 September 2024
  - Project 2 (released 16 September)
    - 2.1 due Week 8; 2.2 due Week 9

Late Penalties
- < deadline +24 hours = 80% of score
- < deadline +48 hours = 50% of score
- ≥ deadline +48 hours = 0% of score

Project Consultations:
Thursdays via Zoom
Canvas > Announcements

Contents

- Recap on CSP Formulation
- Variable-Order Heuristics
- Value-Order Heuristics
- Inference in CSPs

2 Recap on CSPs

Formulating CSPs

- State representation
  - Variables: $\mathbf{X} = \{x_1, ..., x_n\}$
  - Domains: $\mathbf{D} = \{d_1, ..., d_k\}$
    - Such that x_i has a domain d_i
  - Initial state: all variables unassigned
  - Intermediate state: partial assignment
- Goal test
  - Constraints: $\mathbf{C} = \{c_1, ..., c_m\}$
    - Defined via a constraint language
      - Algebra, Logic, Sets
    - Each c_i corresponds to a requirement on some subset of $\mathbf{X}$

- Actions, costs and transition
  - Assignment of values (within domain) to variables
  - Costs are unnecessary

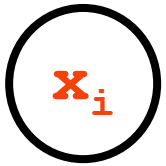
Requires ordering of variables during search!

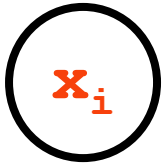
  - Objective is a complete and consistent assignment
    - Find a legal assignment $(y_1, ..., y_n)$
      - $y_i \in D_i$ for all $i \in [1, n]$
  - Complete: all variables assigned values
  - Consistent: all constraints in $\mathbf{C}$ satisfied

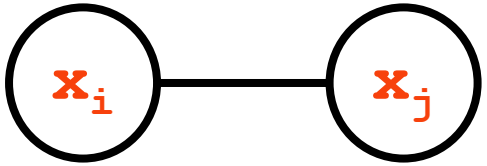
More on Constraints

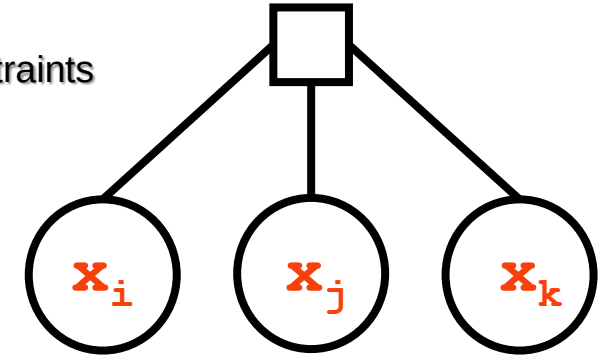
- A language is necessary to express the constraints (based on constraint type)
  - Algebraic for arithmetic
    - e.g., ⟨ (x_1, x_2), $x_1 > x_2$ ⟩ → x_1 greater than x_2 given $D = \{1, 2, 3\}$
  - Propositional logic for logical
  - Sets (of legal values) for extensional
    - e.g., ⟨ (x_1, x_2), { (2, 1), (3, 1), (3, 2) } ⟩ with the same example as above
- 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: | `scope` | = 1
  - Binary: | `scope` | = 2
  - Global: | `scope` | > 2 (i.e., higher-order constraints)

Drawing Constraint Graphs and Hypergraphs

- Constraint graphs represent the constraints in a CSP
  - Simple Vertex: variable 

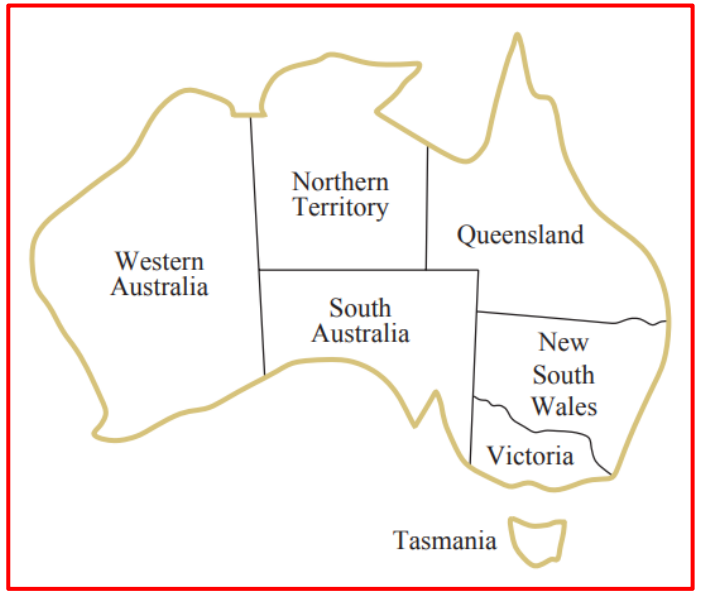
  - Linking Vertex: for global constraints 
  - Edge: links all variables in the `scope` of a constraint (`rel`)
    - Unary constraints 

    - Binary constraints 

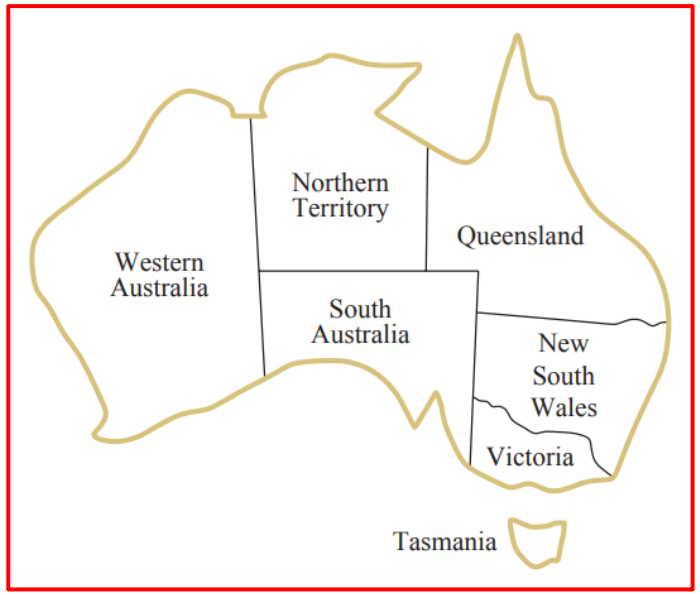
    - Binary/Global constraints 

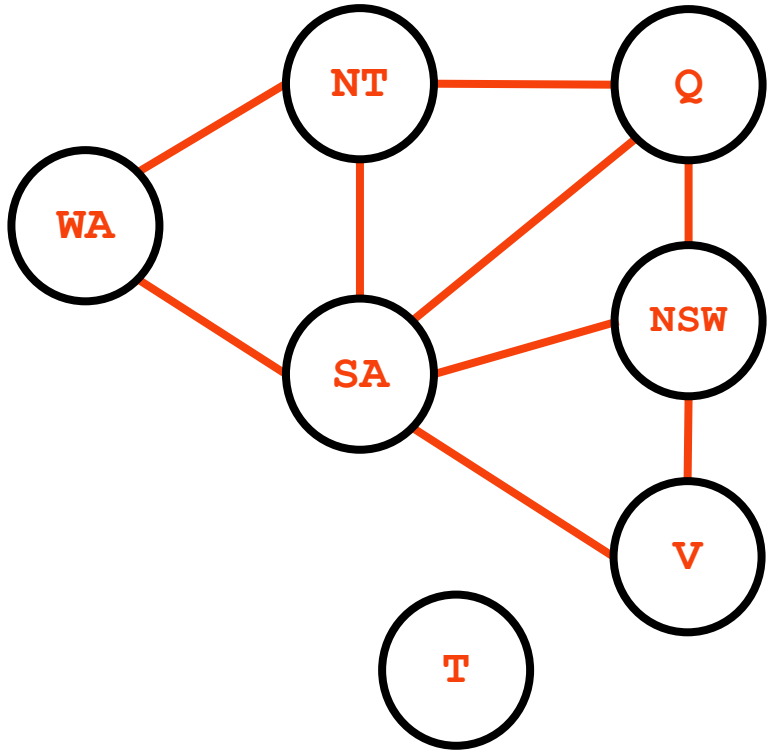
CSP Formulation Example 1: Graph Colouring

- 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
    - $d_i = \{ Red, Green, Blue \}$
  - Constraints
    - $\forall (x_i, x_j) \in E, colour(x_i) \neq colour(x_j)$



Constraint Graph for Example 1: Graph Colouring





- Constraints
  - $\forall(x_i, x_j) \in E, \text{colour}(x_i) \neq \text{colour}(x_j)$

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
```

(A) Determine the variable to assign to

(B) Determine the value to assign

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

General purpose heuristics for (A), (B), and (C) can lead to improved search efficiency

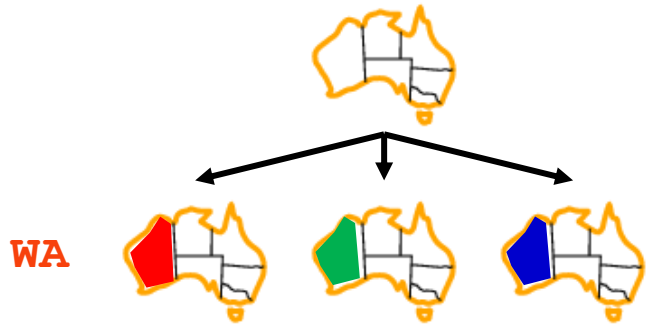
3 Variable-Order Heuristics

Backtracking Example: Graph Colouring

- During backtracking search
  - Assign variables in some order
  - i.e., one specific variable per level
  - Consider the example on the right

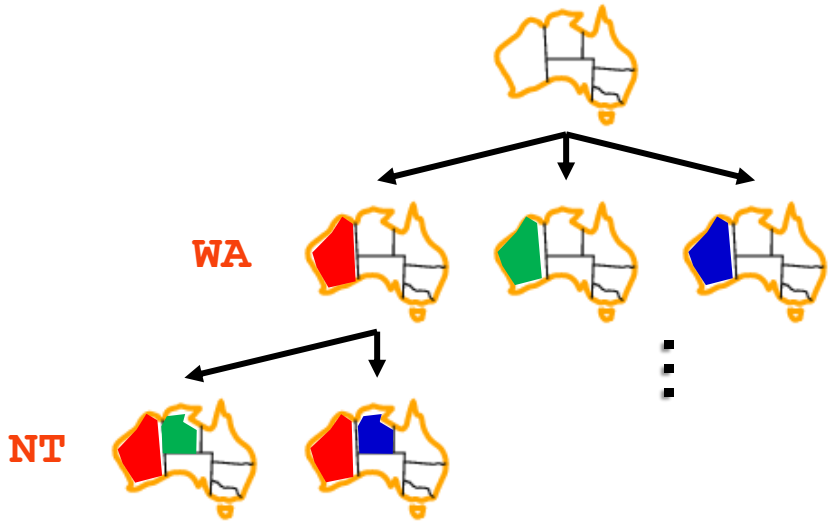
Backtracking Example: Graph Colouring

- During backtracking search
  - Assign variables in some order
  - i.e., one specific variable per level
  - Consider the example on the right



Backtracking Example: Graph Colouring

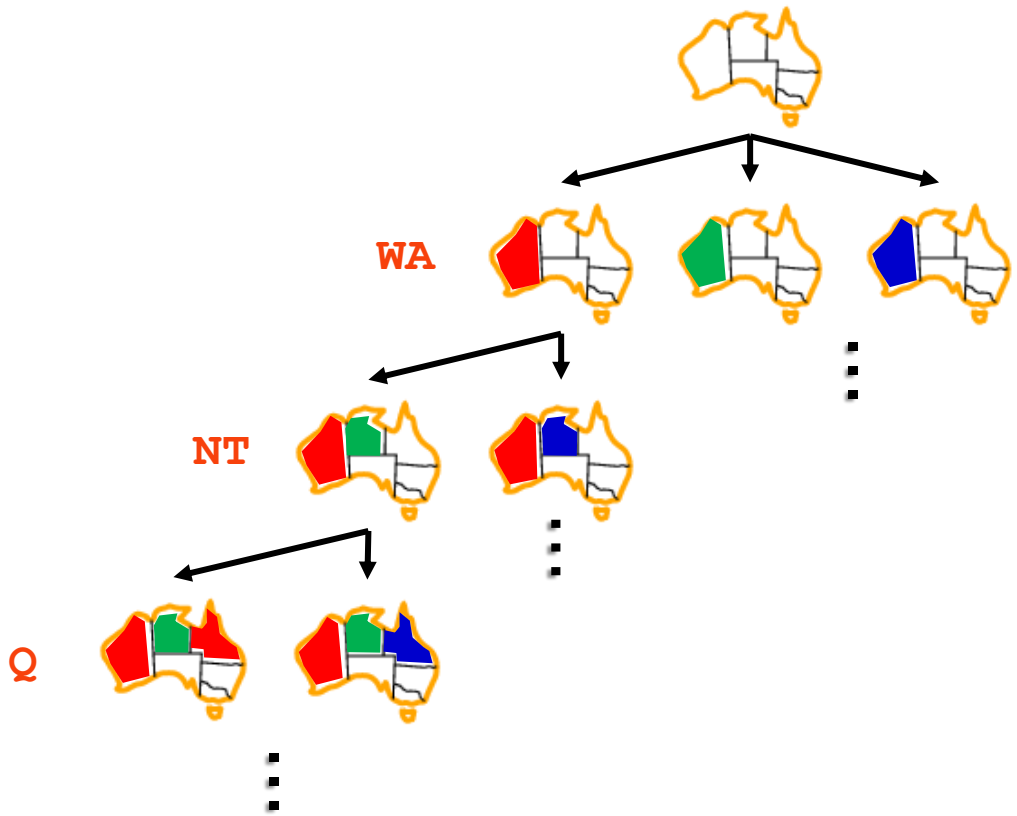
- During backtracking search
  - Assign variables in some order
  - i.e., one specific variable per level
  - Consider the example on the right



Backtracking Example: Graph Colouring

- During backtracking search
  - Assign variables in some order
  - i.e., one specific variable per level
  - Consider the example on the right

How do we decide on the variable order?
Does it matter?



Minimum-Remaining-Values Heuristic

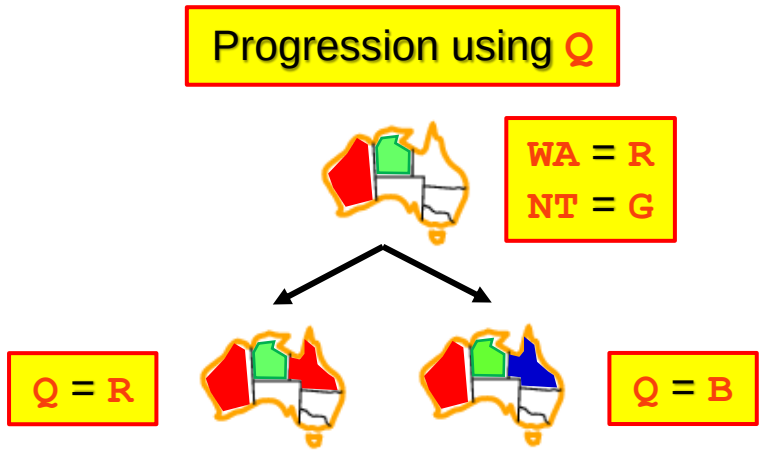
- Choose the variable with fewest legal values
  - Minimum-remaining-values (MRV) heuristic
  - Also considered the most constrained variable
  - Smallest consistent domain size among unassigned variables
- General idea
  - Places larger subtrees closer to the root
    - Any invalid state found prunes 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 suggests selecting `SA`, what if we select `Q` instead?

Progression using `Q`

`WA` = `R`
`NT` = `G`

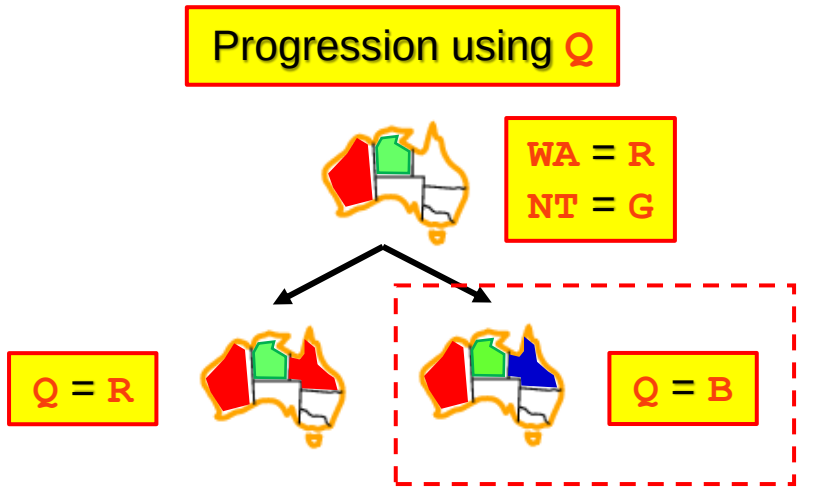
Minimum-Remaining-Values Heuristic

- 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 suggests selecting SA, what if we select Q instead?



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 suggests selecting `SA`, what if we select `Q` instead?



Whole subtree here is invalid; `SA` = ?
But this is not strictly inconsistent yet!
So, we would search this subtree needlessly!

Minimum-Remaining-Values Heuristic

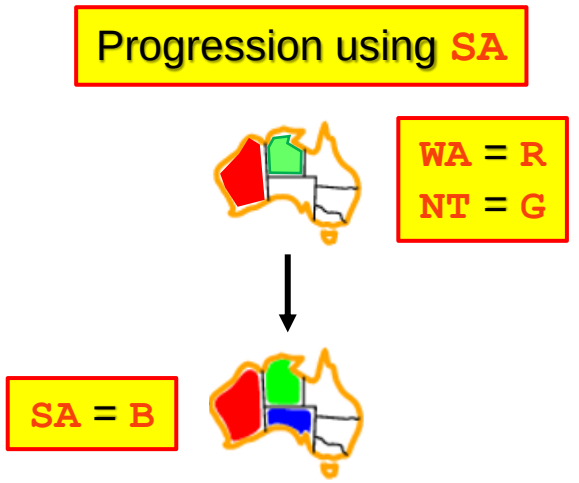
- Choose the variable with fewest legal values
  - Minimum-remaining-values (MRV) heuristic
  - Also considered the most constrained variable
  - Smallest consistent domain size among unassigned variables
- General idea
  - Places larger subtrees closer to the root
    - Any invalid state found prunes 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`; let's trace the selection of `SA`

Progression using `SA`

`WA` = `R`
`NT` = `G`

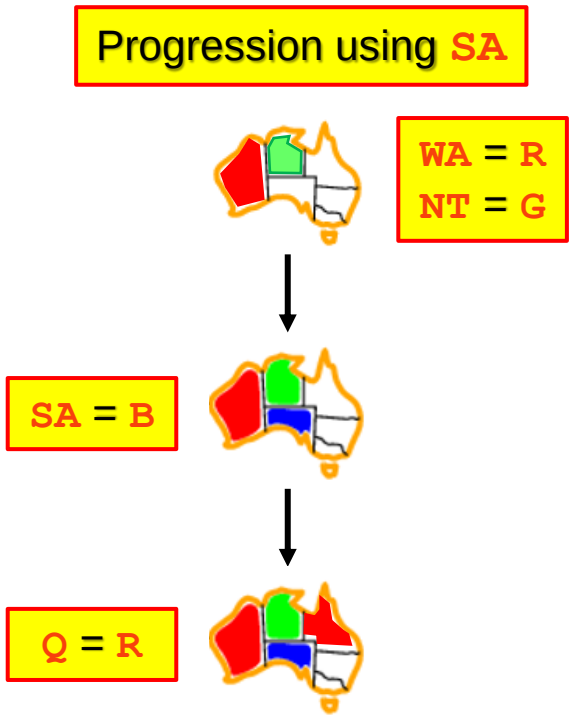
Minimum-Remaining-Values Heuristic

- 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`; let's trace the selection of `SA`



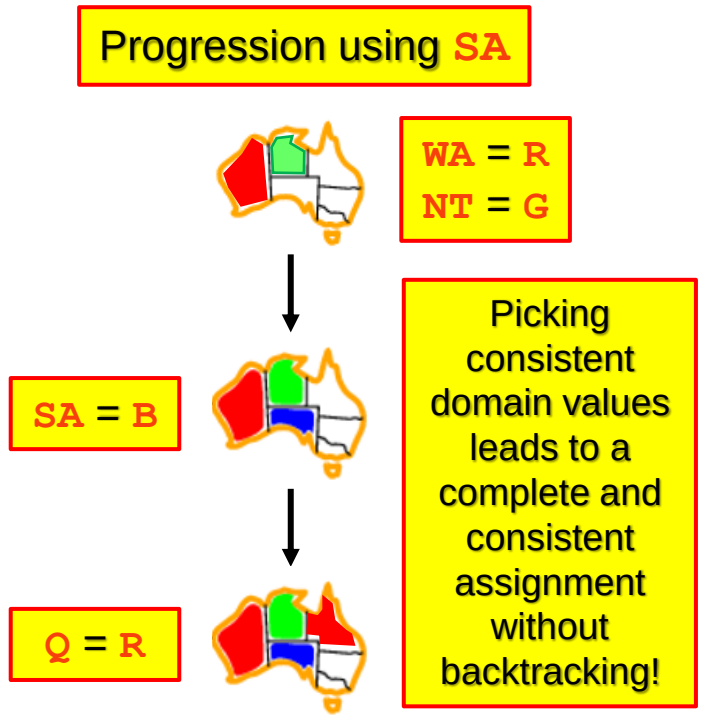
Minimum-Remaining-Values Heuristic

- 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`; let's trace the selection of `SA`



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`; let's trace the selection of `SA`



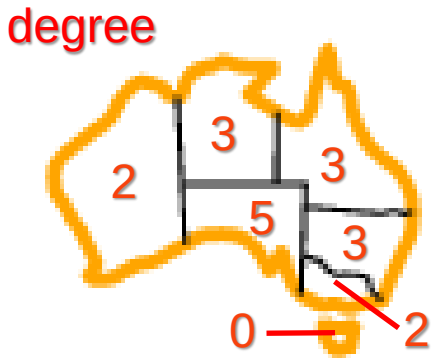
MRV usually performs better than static or random ordering

Degree Heuristic

- MRV heuristic requires tie-breaking
  - e.g., at initial state, all variables have same RV
- Tie-break with (constraint graph) degree heuristic
  - Picks unassigned variable with most constraints relative to unassigned variables
- General idea
  - By selecting variable that restricts the greatest number of other variables, we reduce `b`
- Consider the Australia Colouring problem
  - Using MRV with degree for tie-breaking

Degree Heuristic

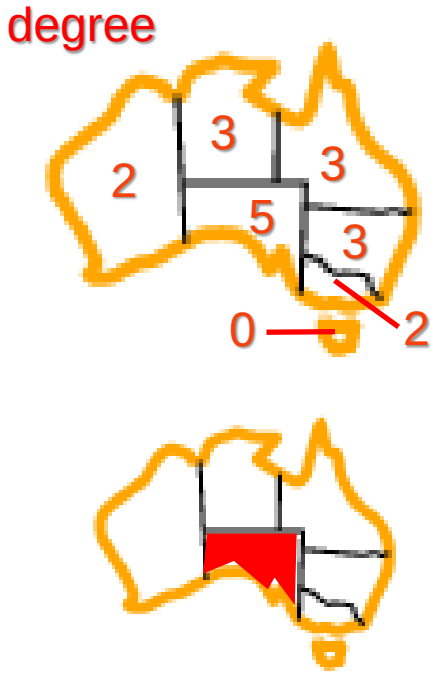
- MRV heuristic requires tie-breaking
  - e.g., at initial state, all variables have same RV
- Tie-break with (constraint graph) degree heuristic
  - Picks unassigned variable with most constraints relative to unassigned variables
- General idea
  - By selecting variable that restricts the greatest number of other variables, we reduce `b`
- Consider the Australia Colouring problem
  - Using MRV with degree for tie-breaking
    - At the initial state (no assignments), all states have same MRV
      - Tie-break with degree



Degree Heuristic

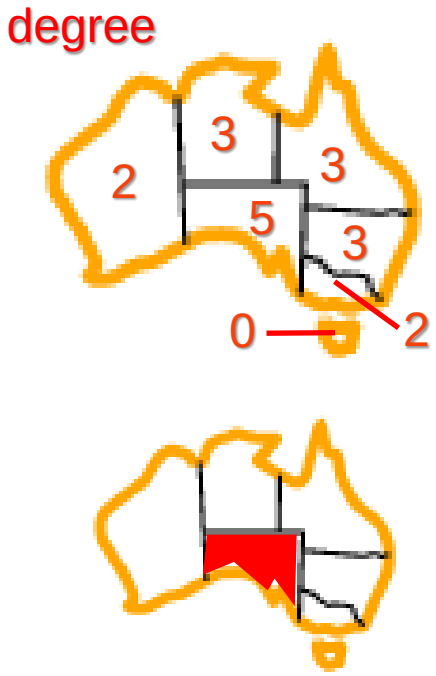
- MRV heuristic requires tie-breaking
  - e.g., at initial state, all variables have same RV
- Tie-break with (constraint graph) degree heuristic
  - Picks unassigned variable with most constraints relative to unassigned variables
- General idea
  - By selecting variable that restricts the greatest number of other variables, we reduce `b`
- Consider the Australia Colouring problem
  - Using MRV with degree for tie-breaking
    - At the initial state (no assignments), all states have same MRV
      - Tie-break with degree

      → chosen variable is `SA`



Degree Heuristic

- MRV heuristic requires tie-breaking
  - e.g., at initial state, all variables have same RV
- Tie-break with (constraint graph) degree heuristic
  - Picks unassigned variable with most constraints relative to unassigned variables
- General idea
  - By selecting variable that restricts the greatest number of other variables, we reduce `b`
- Consider the Australia Colouring problem
  - Using MRV with degree for tie-breaking
    - At the initial state (no assignments), all states have same MRV
      - 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
```

(A) Determine the variable to assign to

(B) Determine the value to assign

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

4 Value-Order Heuristics

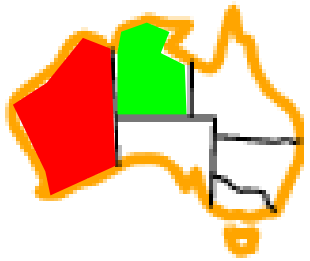
Least-Constraining-Value Heuristic

- Choose the value that rules out the fewest choices (from remaining domain values)
  - Least-constraining-value (LCV) heuristic
  - Given assignment of value v to variable x'
  - Determine set of unassigned variables $U = \{ x_a, x_b, ... \}$ 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

Least-Constraining-Value Heuristic

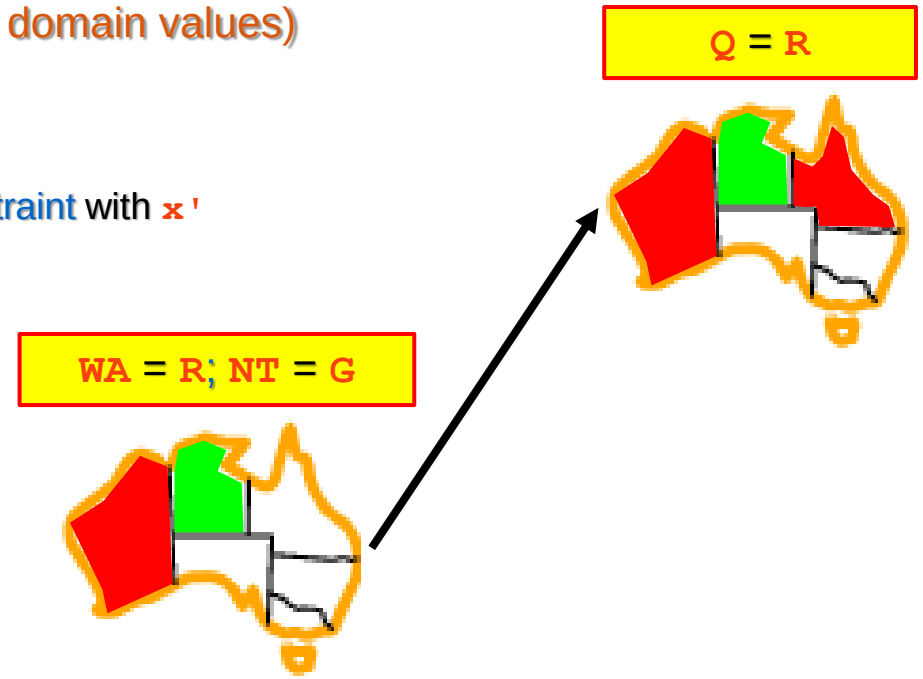
- Choose the value that rules out the fewest choices (from remaining domain values)
  - Least-constraining-value (LCV) heuristic
  - Given assignment of value v to variable x
  - Determine set of unassigned variables $U = \{ x_a, x_b, \ldots \}$ that share a constraint with x
  - Pick v that maximises 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

WA = R; NT = G



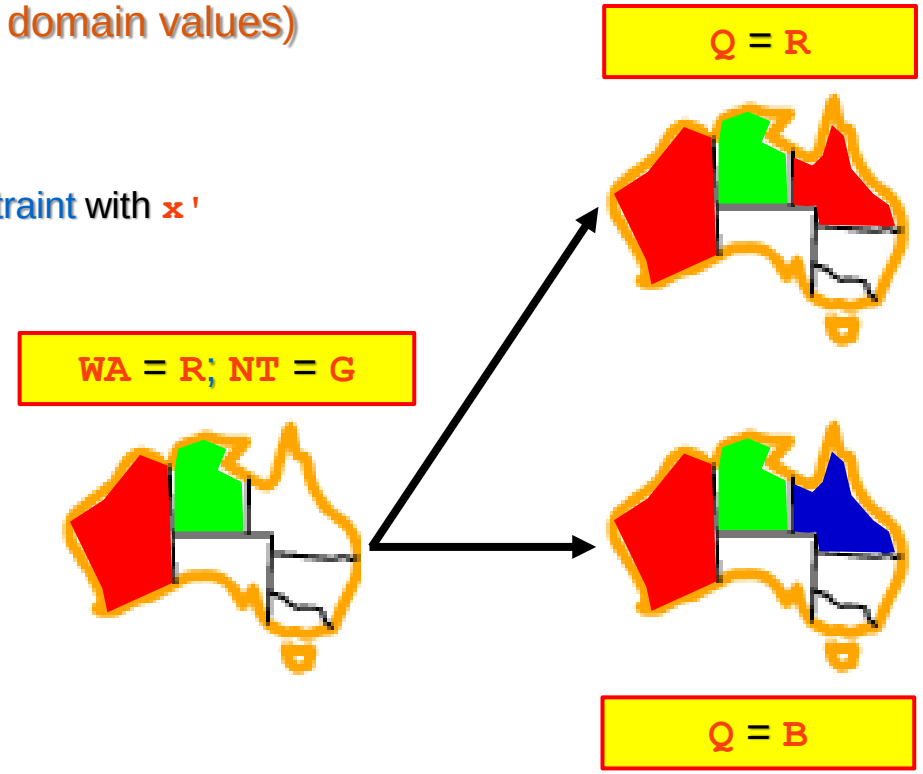
Least-Constraining-Value Heuristic

- Choose the value that rules out the fewest choices (from remaining domain values)
  - Least-constraining-value (LCV) heuristic
  - Given assignment of value v to variable x'
  - Determine set of unassigned variables $U = \{ x_a, x_b, \ldots \}$ 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



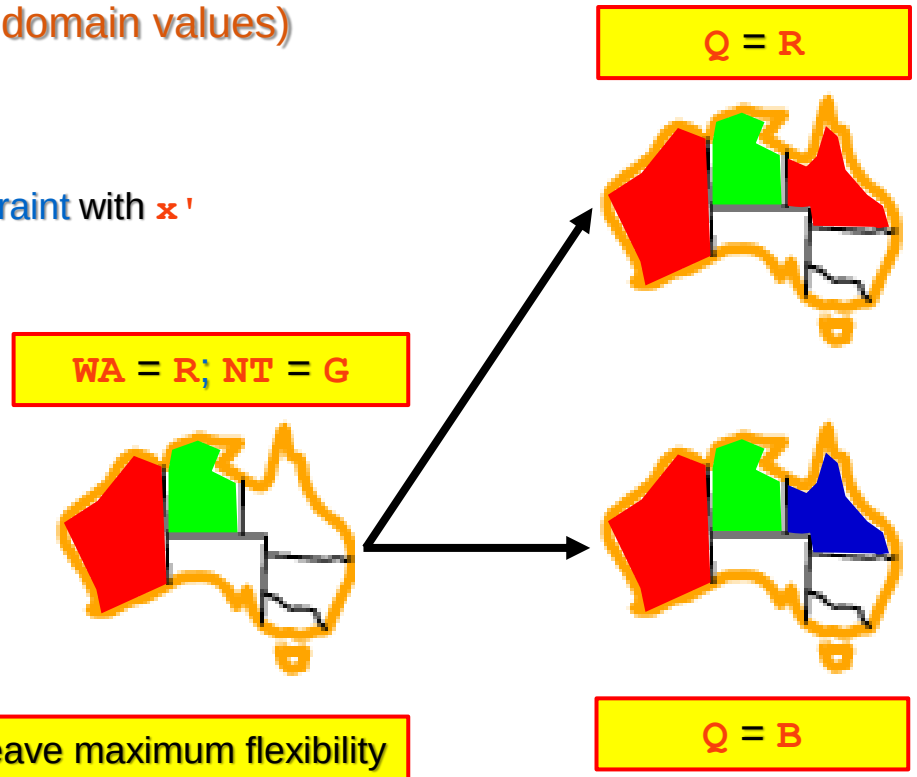
Least-Constraining-Value Heuristic

- Choose the value that rules out the fewest choices (from remaining domain values)
  - Least-constraining-value (LCV) heuristic
  - Given assignment of value `v` to variable `x'`
  - Determine set of unassigned variables `U` = { x_a, x_b, ... } 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` = `None`



Least-Constraining-Value Heuristic

- Choose the value that rules out the fewest choices (from remaining domain values)
  - Least-constraining-value (LCV) heuristic
  - Given assignment of value `v` to variable `x'`
  - Determine set of unassigned variables `U` = { x_a, x_b, ... } 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` = `None`
    - Select `Q` = `R` (since it constrains `1` less than `Q` = `B`)



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
```

(A) Determine the variable to assign to

(B) Determine the value to assign

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

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/90507063206

5 Inference in CSPs

Avoiding Failure

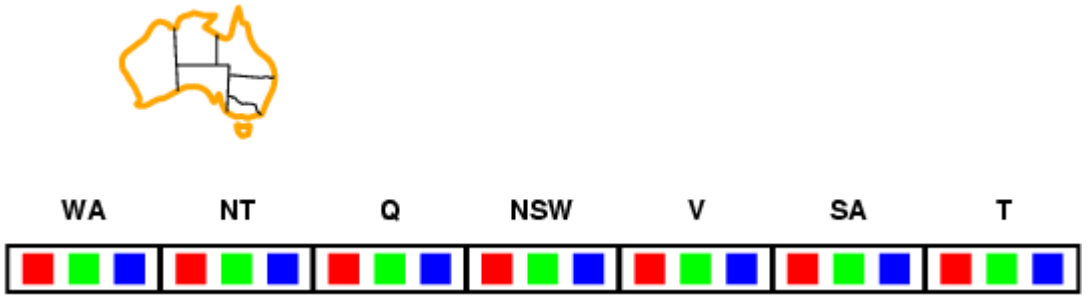
- With certain states, we know we are heading for failure

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

6 Forward Checking

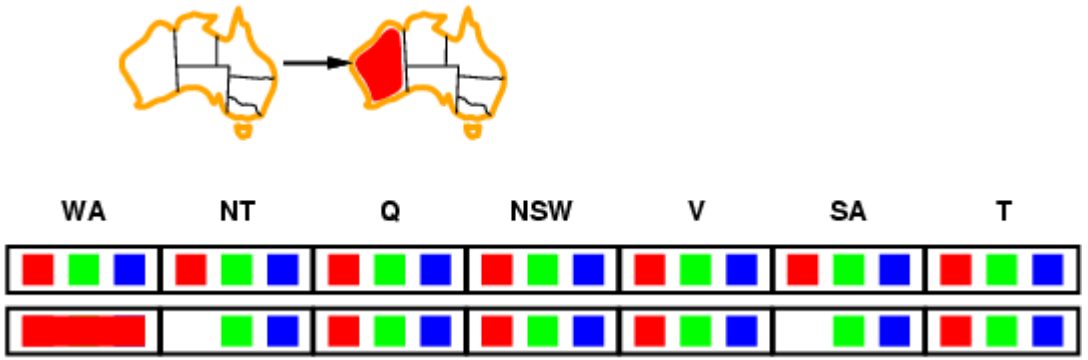
Forward Checking

- Track remaining legal values for unassigned variables
- Terminate search when any variable has no legal values (based on constraints with recently assigned variable)



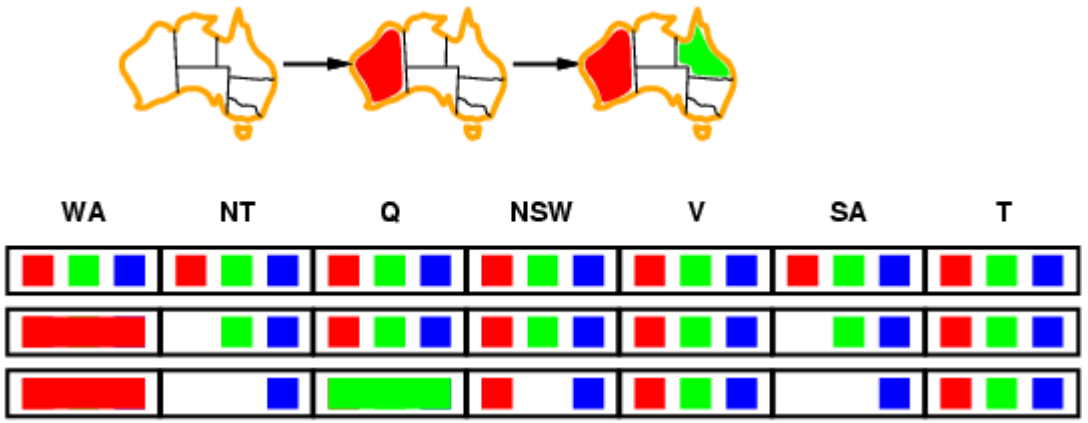
Forward Checking

- Track remaining legal values for unassigned variables
- Terminate search when any variable has no legal values (based on constraints with recently assigned variable)



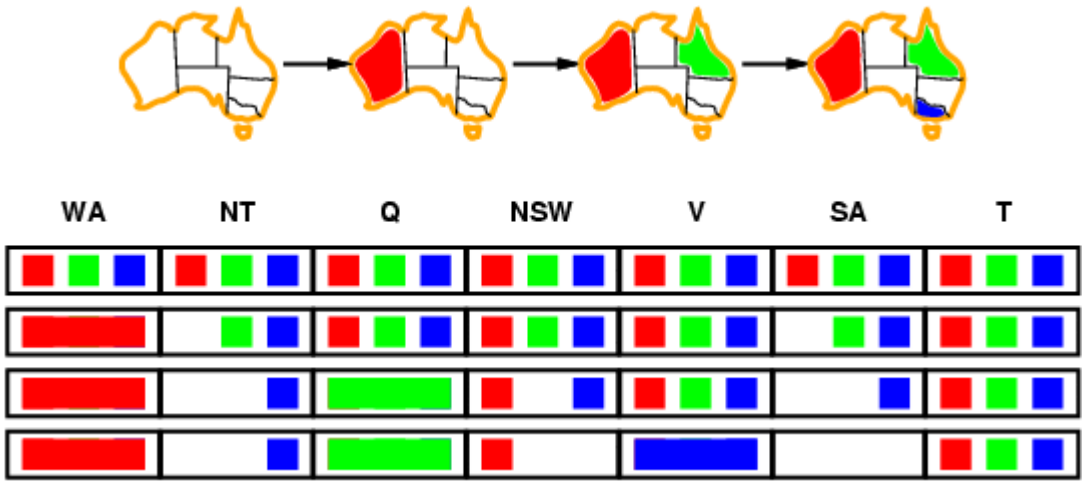
Forward Checking

- Track remaining legal values for unassigned variables
- Terminate search when any variable has no legal values (based on constraints with recently assigned variable)



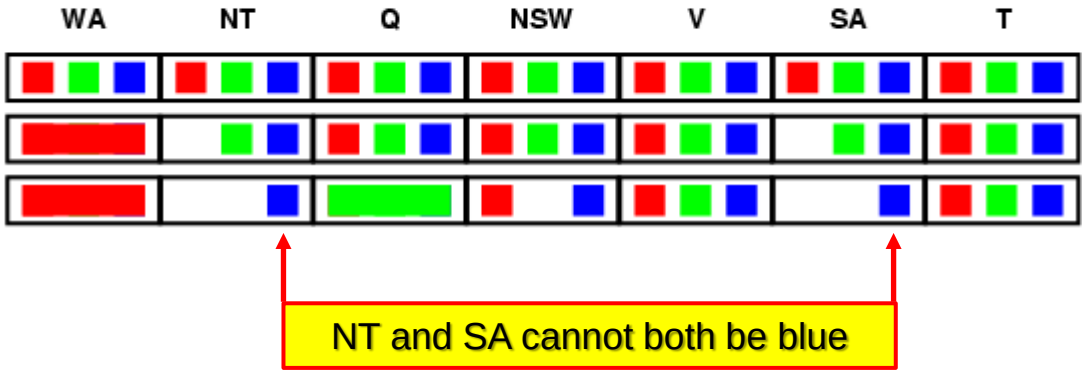
Forward Checking

- Track remaining legal values for unassigned variables
- Terminate search when any variable has no legal values (based on constraints with recently assigned variable)



Forward Checking

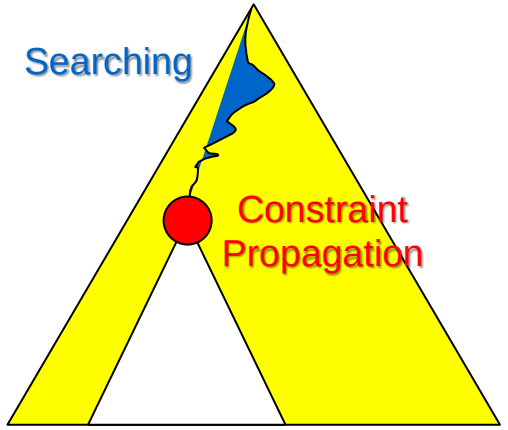
- Problem
  - Forward checking propagates information from assigned to unassigned variables, but does not provide earliest detection for all failures



Solution:
Ensure 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 (i.e., vertex-consistent)
  - A single variable is node-consistent if its domain is consistent with its related unary constraints
- Arc-consistent (i.e., edge-consistent)
  - A single variable is arc-consistent if its domain is consistent with its 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 CSP trace questions are done only over unary/binary constraints (global constraints only in CSP formulation questions)

7 Node Consistency

Node Consistency

- Node-consistent (i.e., edge-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)

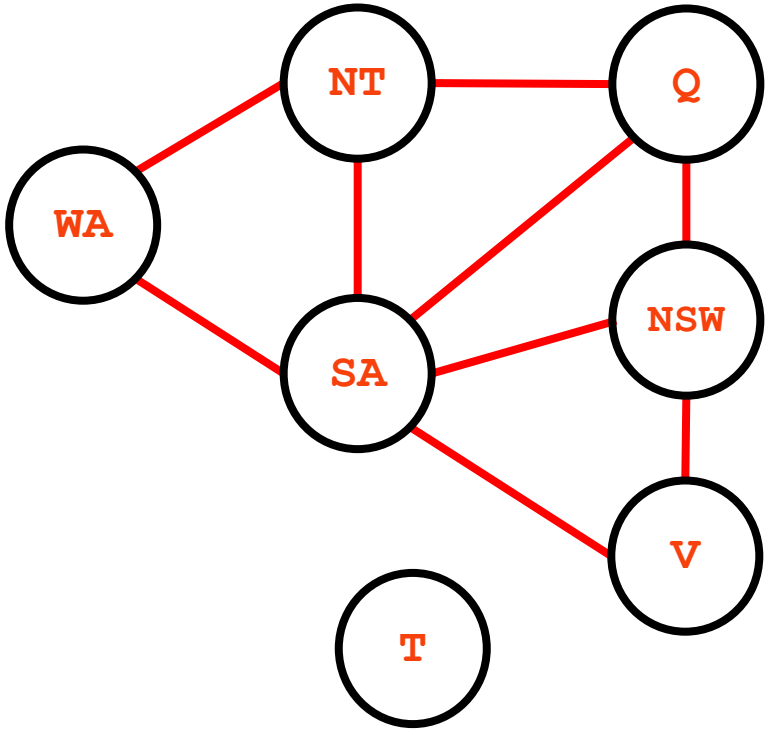
8 Arc Consistency

Arc Consistency

- Arc-consistent (i.e., edge-consistent)
  - A single variable is arc-consistent if its domain is consistent with its related binary constraints
- Ensuring arc consistency (AC)
  - For each variable
    - Eliminate domain values inconsistent with binary constraints
    - As arc-consistency is measured against
    - i.e., the variable's domain value must have a partnering domain value in other variable that will satisfy the binary constraint

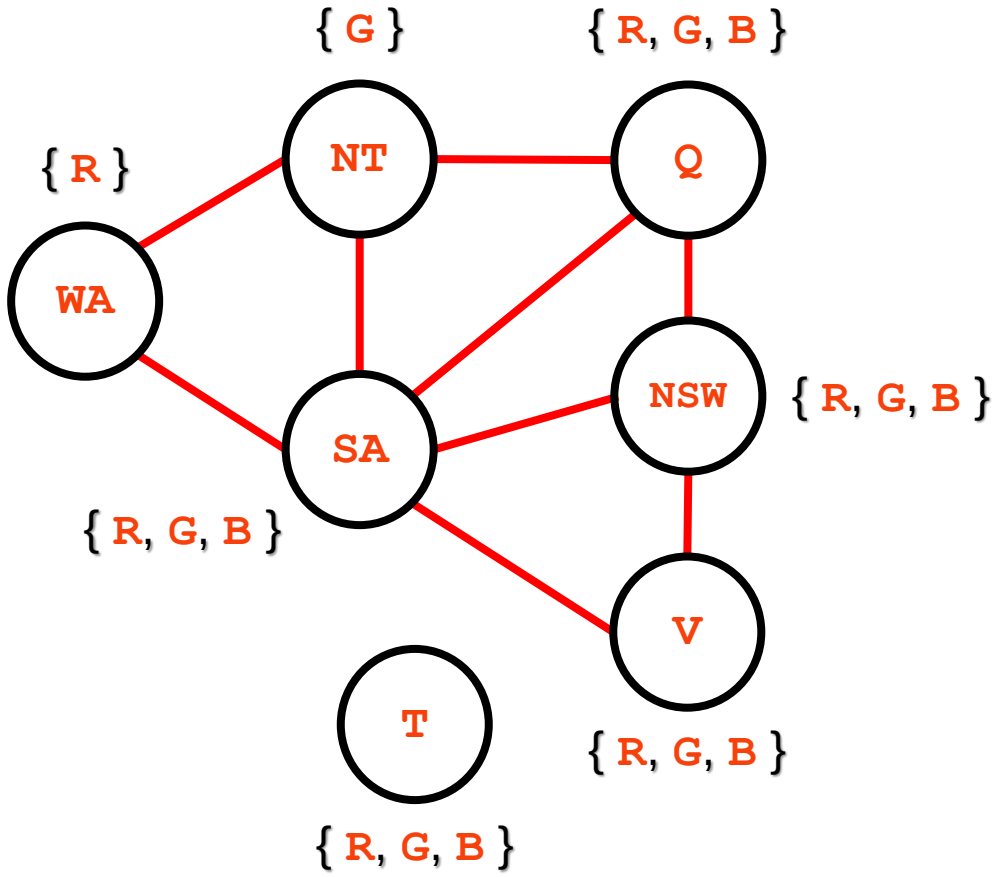
Arc Consistency

- Arc-consistent (i.e., edge-consistent)
  - A single variable is arc-consistent if its domain is consistent with its related binary constraints
- Ensuring arc consistency (AC)
  - For each variable
    - Eliminate domain values inconsistent with binary constraints
    - As arc-consistency is measured against
    - i.e., the variable's domain value must have a partnering domain value in other variable that will satisfy the binary constraint
  - Consider the Australia Colouring problem



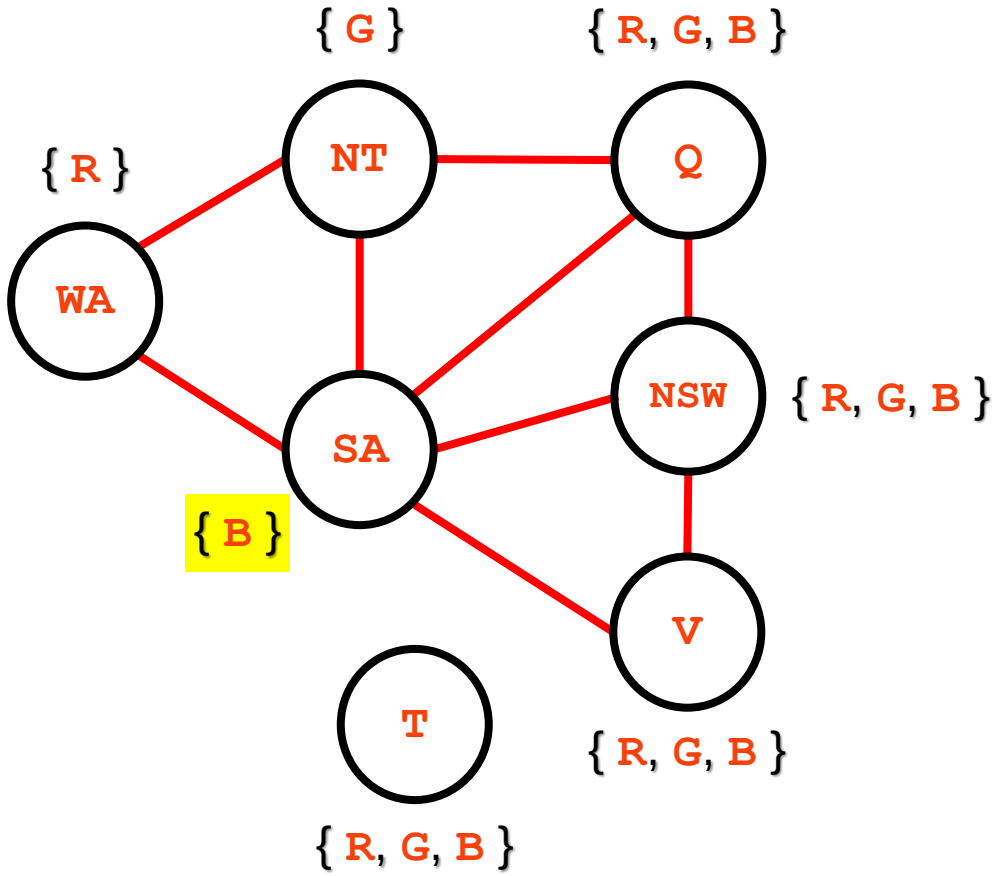
Arc Consistency

- Arc-consistent (i.e., edge-consistent)
  - A single variable is arc-consistent if its domain is consistent with its related binary constraints
- Ensuring arc consistency (AC)
  - For each variable
    - Eliminate domain values inconsistent with binary constraints
    - As arc-consistency is measured against
    - i.e., the variable's domain value must have a partnering domain value in other variable that will satisfy the binary constraint
  - Consider the Australia Colouring problem
    - Assume D_{WA} = { R }, D_{NT} = { G }, domains of rest = { R, G, B }



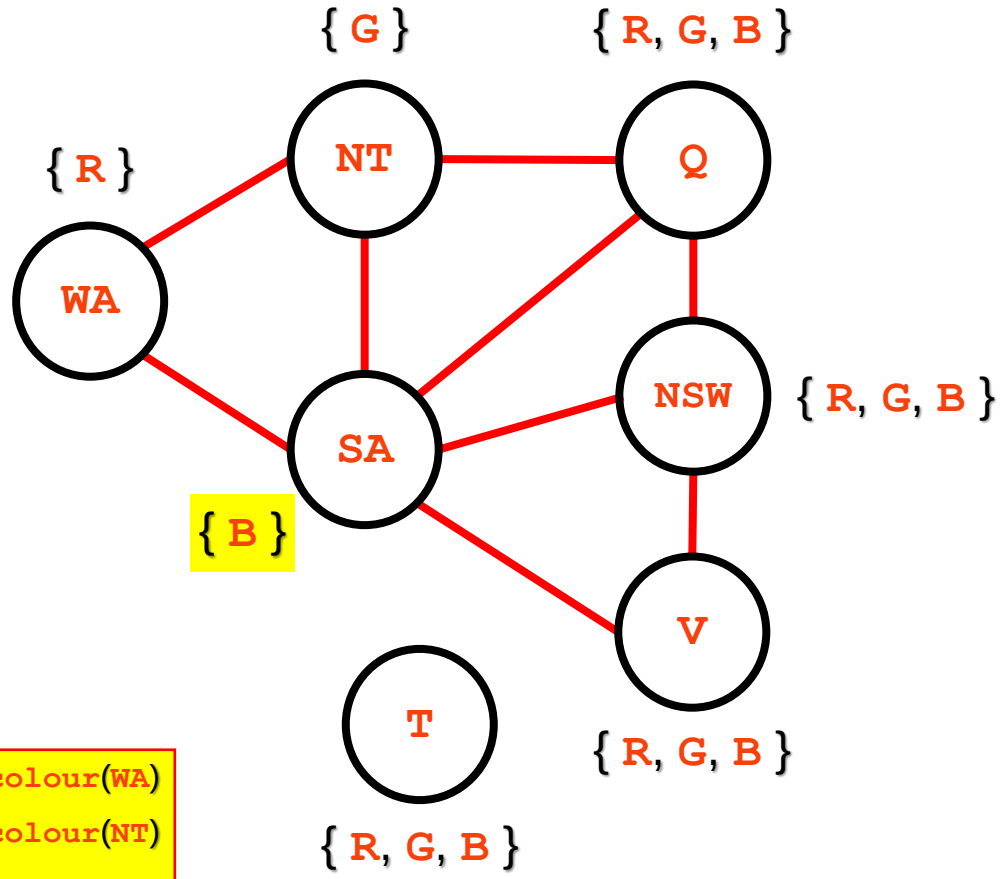
Arc Consistency

- Arc-consistent (i.e., edge-consistent)
  - A single variable is arc-consistent if its domain is consistent with its related binary constraints
- Ensuring arc consistency (AC)
  - For each variable
    - Eliminate domain values inconsistent with binary constraints
    - As arc-consistency is measured against
    - i.e., the variable's domain value must have a partnering domain value in other variable that will satisfy the binary 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 }



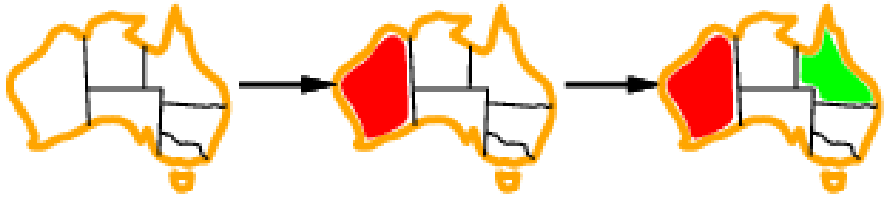
Arc Consistency

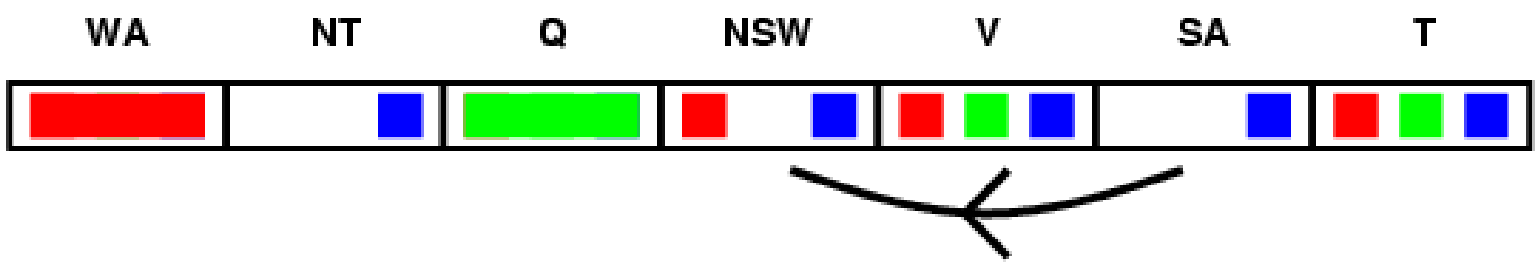
- Arc-consistent (i.e., edge-consistent)
  - A single variable is arc-consistent if its domain is consistent with its related binary constraints
- Ensuring arc consistency (AC)
  - For each variable
    - Eliminate domain values inconsistent with binary constraints
    - As arc-consistency is measured against
    - i.e., the variable's domain value must have a partnering domain value in other variable that will satisfy the binary 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 colour(SA) ≠ colour(WA)
      - Eliminate SA = G since there is no value in D_{NT} that satisfies colour(SA) ≠ colour(NT)
      - Repeat for other binary constraints that include SA



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)



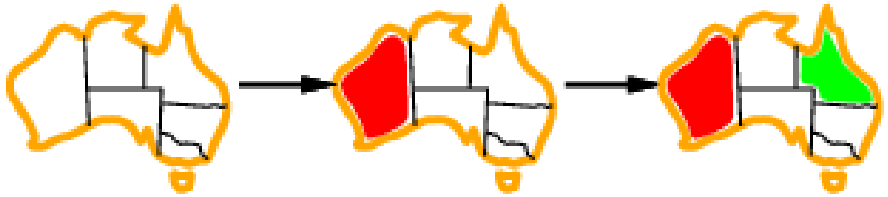


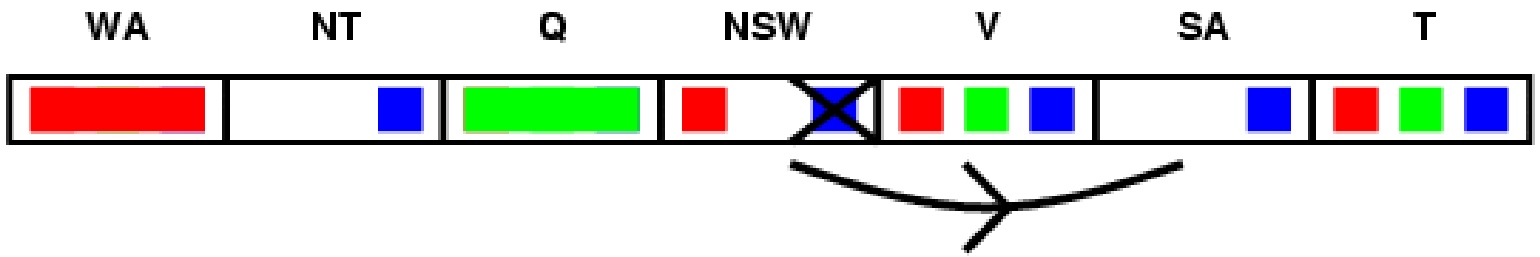
To maintain AC, remove any value from the first variable if it makes a constraint impossible to satisfy

Arc (SA, NSW) is consistent as 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)





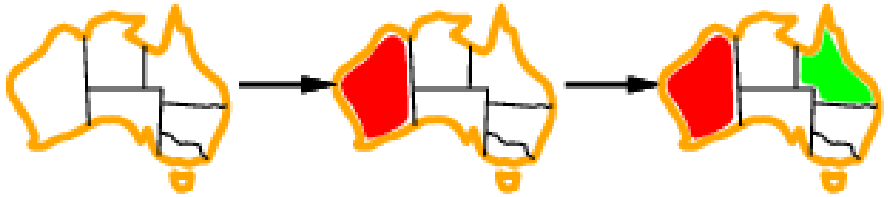
Arcs are directed

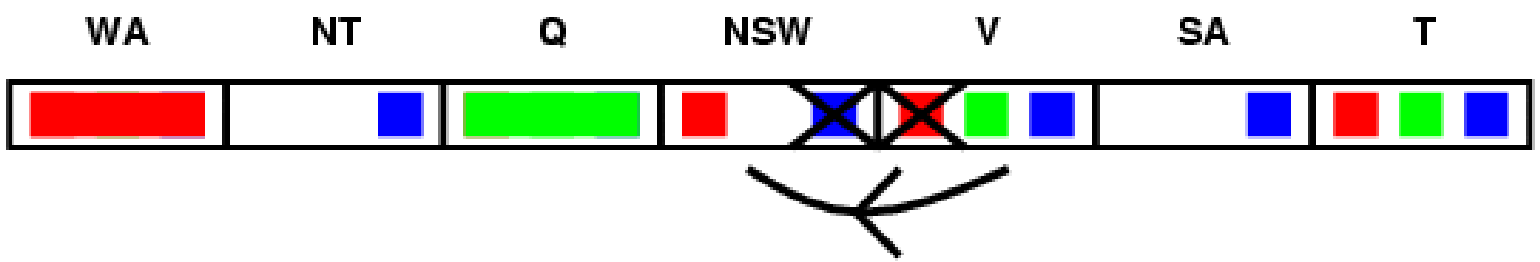
A binary constraint becomes two arcs

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)



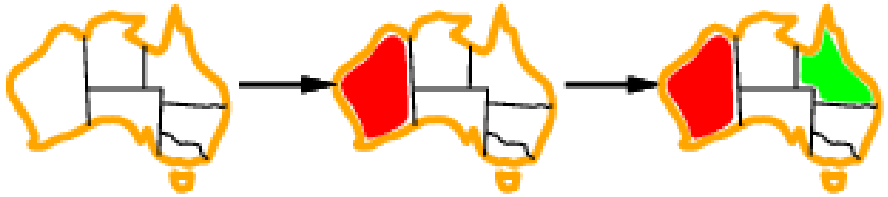


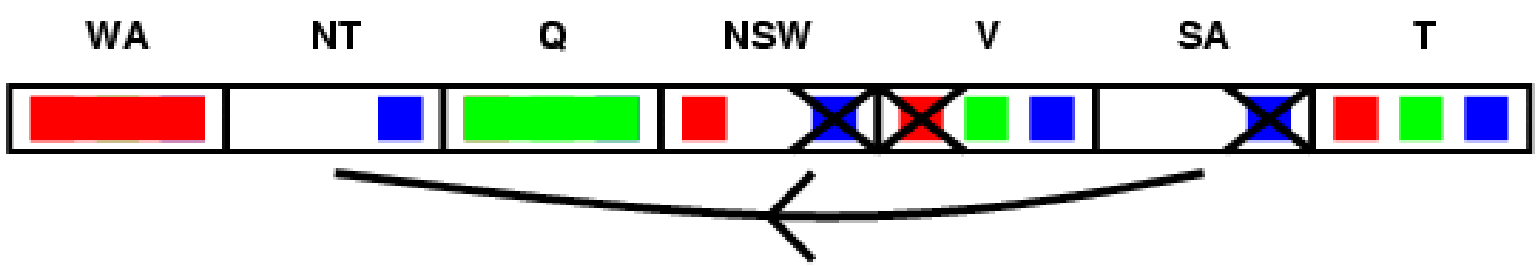
If X_i loses a value, neighbours of X_i need to be (re)checked

Arc (V, NSW) not consistent after deleting NSW = B; it regains consistency 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

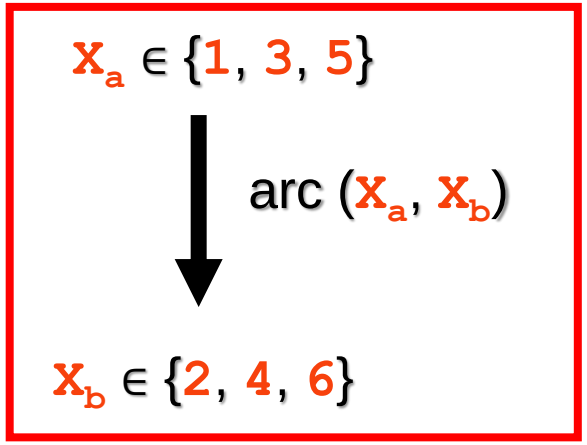
Run with backtracking (after variable assignment); OR as a pre-processing step (before search)

Arc Consistency Example 1

- Example
  - $D_a = \{ 1, 3, 5 \}$
  - $D_b = \{ 2, 4, 6 \}$
  - $X_a > X_b$
- Check
  - arc(X_a, X_b) – i.e., $X_a > X_b$
  - arc(X_b, X_a) – i.e., $X_a > X_b$

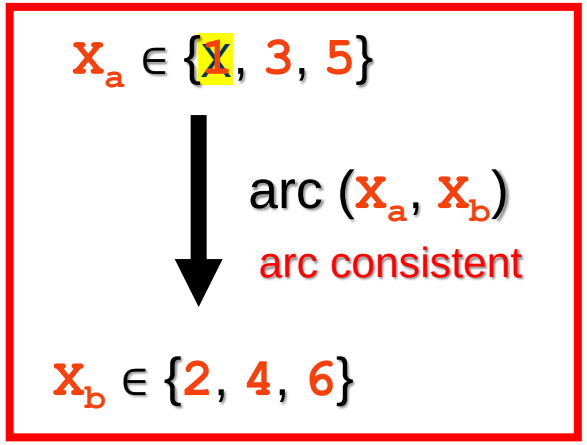
Arc Consistency Example 1

- Example
  - $D_a = \{ 1, 3, 5 \}$
  - $D_b = \{ 2, 4, 6 \}$
  - $X_a > X_b$
- Check
  - arc(X_a, X_b) – i.e., $X_a > X_b$
  - arc(X_b, X_a) – i.e., $X_a > X_b$



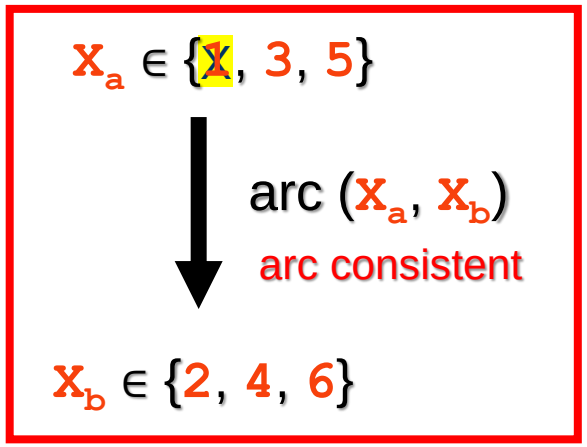
Arc Consistency Example 1

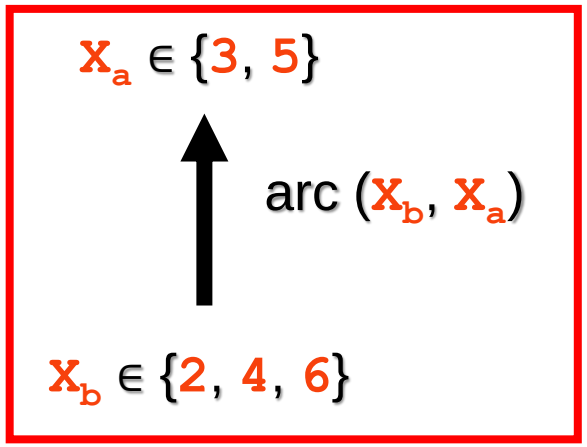
- Example
  - $D_a = \{ 1, 3, 5 \}$
  - $D_b = \{ 2, 4, 6 \}$
  - $X_a > X_b$
- Check
  - arc(X_a, X_b) – i.e., $X_a > X_b$
  - arc(X_b, X_a) – i.e., $X_a > X_b$



Arc Consistency Example 1

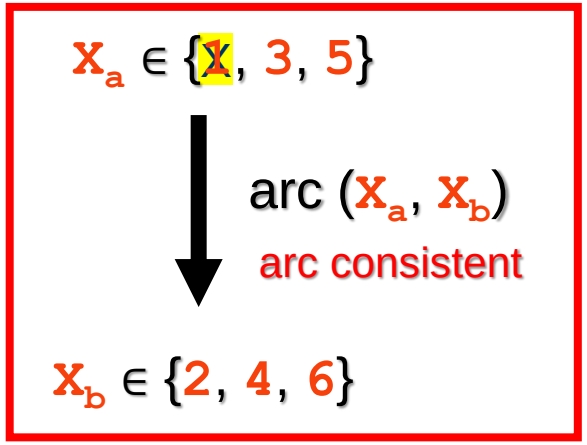
- Example
  - $D_a = \{ 1, 3, 5 \}$
  - $D_b = \{ 2, 4, 6 \}$
  - $X_a > X_b$
- Check
  - arc(X_a, X_b) – i.e., $X_a > X_b$
  - arc(X_b, X_a) – i.e., $X_a > X_b$

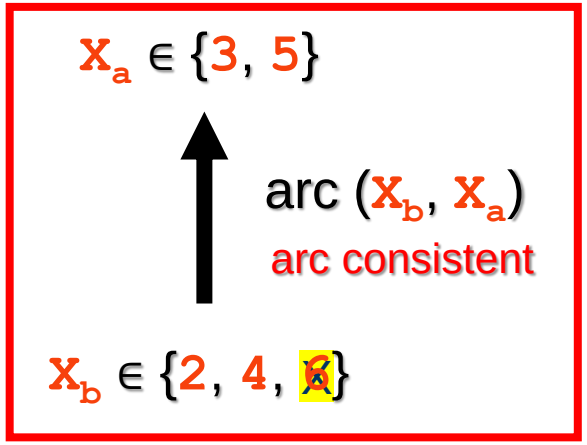




Arc Consistency Example 1

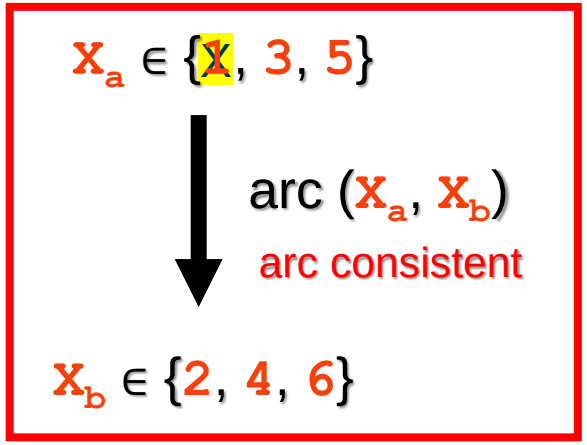
- Example
  - $D_a = \{ 1, 3, 5 \}$
  - $D_b = \{ 2, 4, 6 \}$
  - $X_a > X_b$
- Check
  - arc(X_a, X_b) – i.e., $X_a > X_b$
  - arc(X_b, X_a) – i.e., $X_a > X_b$

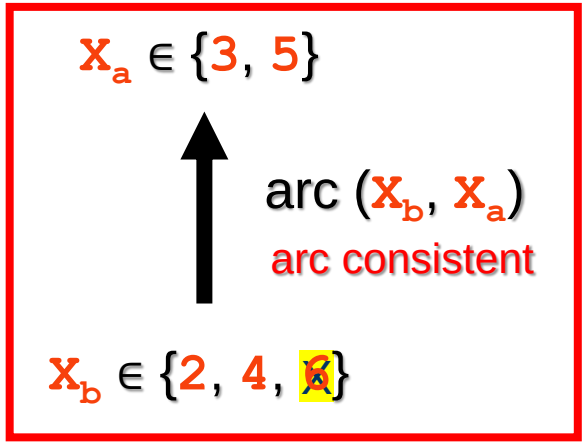




Arc Consistency Example 1

- Example
  - $D_a = \{ 1, 3, 5 \}$
  - $D_b = \{ 2, 4, 6 \}$
  - $X_a > X_b$
- Check
  - arc(X_a, X_b) – i.e., $X_a > X_b$
  - arc(X_b, X_a) – i.e., $X_a > X_b$





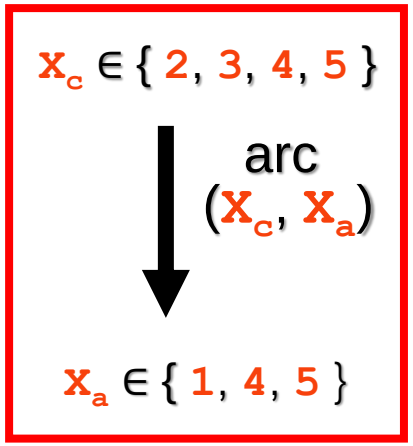
- Should arc (X_a, X_b) be checked again after the update to X_b?
  - No
  - Notice that 1 in D_a was not required to satisfy any value (2, 4, 6) from D_b
  - Notice that 6 in D_b was not required to satisfy any value (1, 3, 5) from D_a

Arc Consistency Example 2

- Example
  - $D_a = \{ 1, 4, 5 \}$
  - $D_b = \{ 1, 2, 3 \}$
  - $D_c = \{ 2, 3, 4, 5 \}$
  - $X_c > X_a$
  - $X_a > X_b$
- Check
  - arc(X_c, X_a) – i.e., $X_c > X_a$
  - arc(X_a, X_b) – i.e., $X_a > X_b$

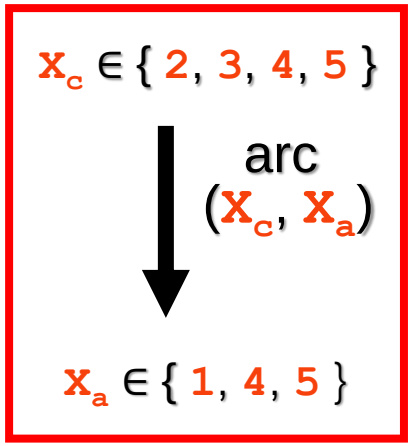
Arc Consistency Example 2

- Example
  - $D_a = \{ 1, 4, 5 \}$
  - $D_b = \{ 1, 2, 3 \}$
  - $D_c = \{ 2, 3, 4, 5 \}$
  - $X_c > X_a$
  - $X_a > X_b$
- Check
  - arc(X_c, X_a) – i.e., $X_c > X_a$
  - arc(X_a, X_b) – i.e., $X_a > X_b$



Arc Consistency Example 2

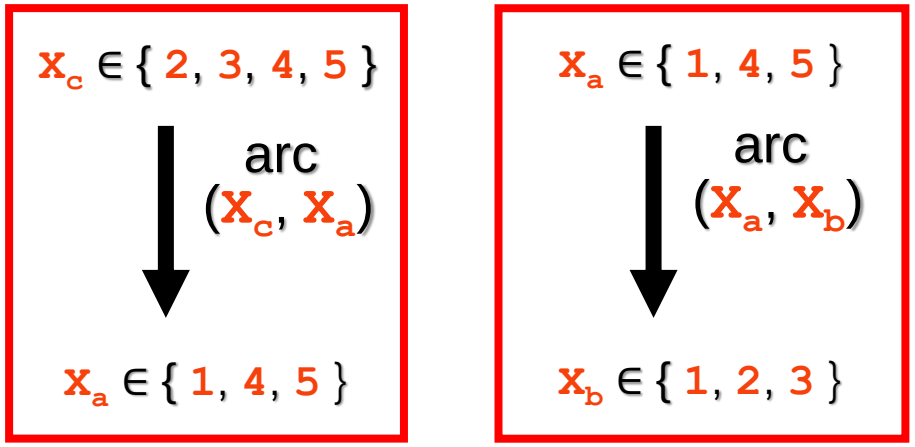
- Example
  - $D_a = \{ 1, 4, 5 \}$
  - $D_b = \{ 1, 2, 3 \}$
  - $D_c = \{ 2, 3, 4, 5 \}$
  - $X_c > X_a$
  - $X_a > X_b$
- Check
  - arc(X_c, X_a) – i.e., $X_c > X_a$
  - arc(X_a, X_b) – i.e., $X_a > X_b$



arc consistent

Arc Consistency Example 2

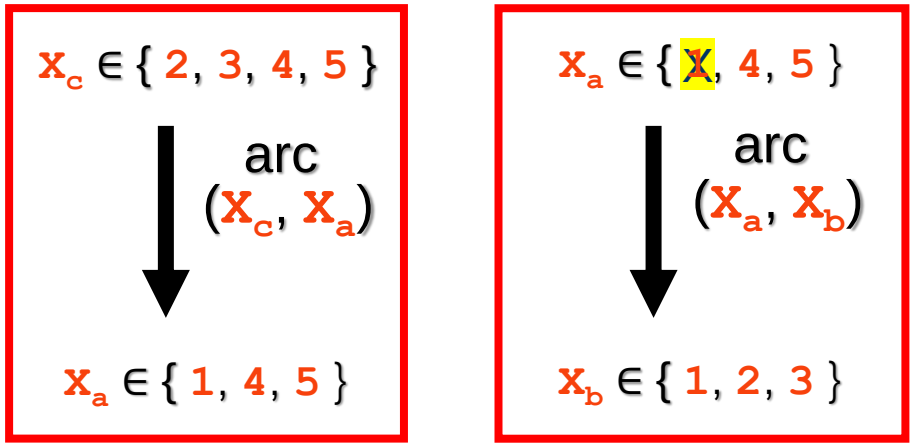
- Example
  - $D_a = \{ 1, 4, 5 \}$
  - $D_b = \{ 1, 2, 3 \}$
  - $D_c = \{ 2, 3, 4, 5 \}$
  - $X_c > X_a$
  - $X_a > X_b$
- Check
  - arc(X_c, X_a) – i.e., $X_c > X_a$
  - arc(X_a, X_b) – i.e., $X_a > X_b$



arc consistent

Arc Consistency Example 2

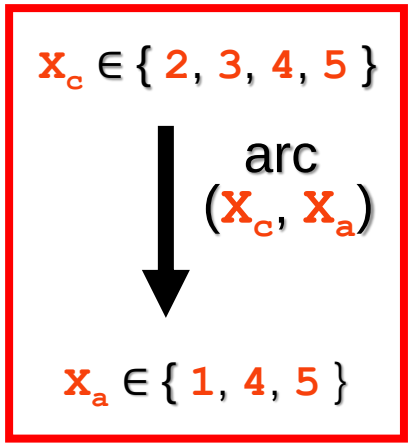
- Example
  - $D_a = \{ 1, 4, 5 \}$
  - $D_b = \{ 1, 2, 3 \}$
  - $D_c = \{ 2, 3, 4, 5 \}$
  - $X_c > X_a$
  - $X_a > X_b$
- Check
  - arc(X_c, X_a) – i.e., $X_c > X_a$
  - arc(X_a, X_b) – i.e., $X_a > X_b$



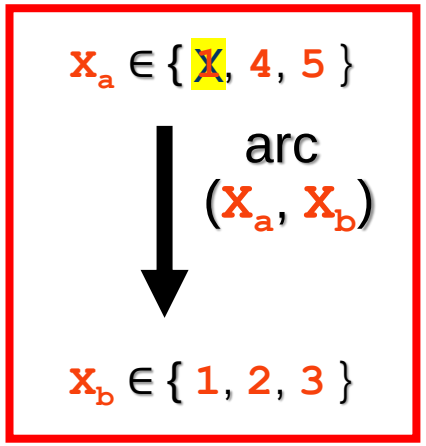
arc consistent

Arc Consistency Example 2

- Example
  - $D_a = \{ 1, 4, 5 \}$
  - $D_b = \{ 1, 2, 3 \}$
  - $D_c = \{ 2, 3, 4, 5 \}$
  - $X_c > X_a$
  - $X_a > X_b$
- Check
  - arc(X_c, X_a) – i.e., $X_c > X_a$
  - arc(X_a, X_b) – i.e., $X_a > X_b$



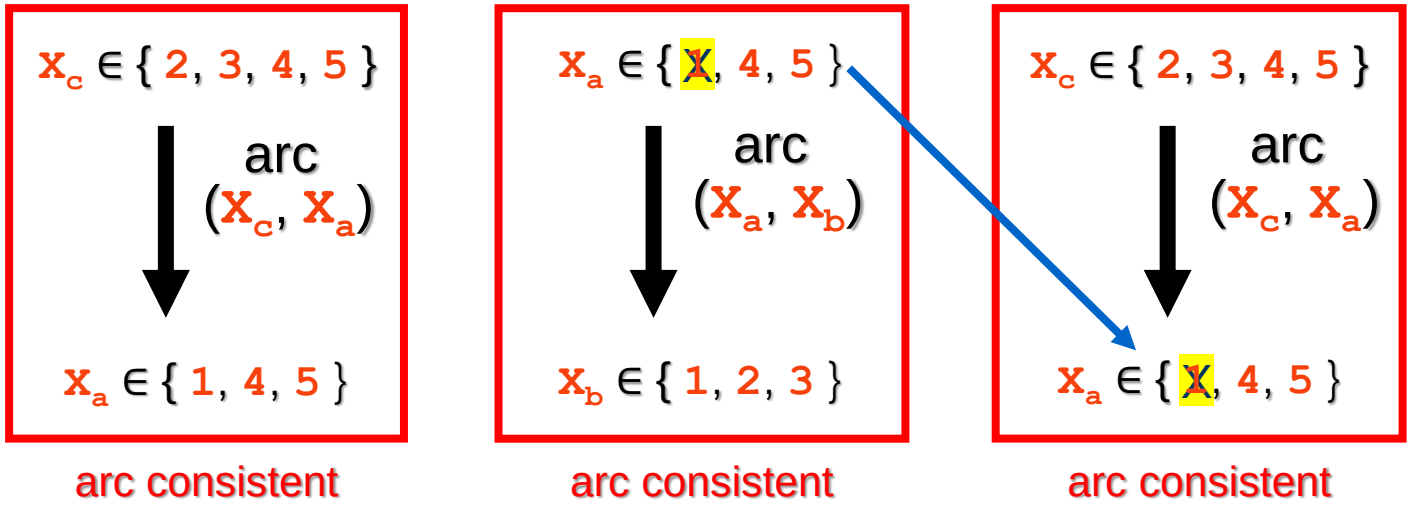
arc consistent



arc consistent

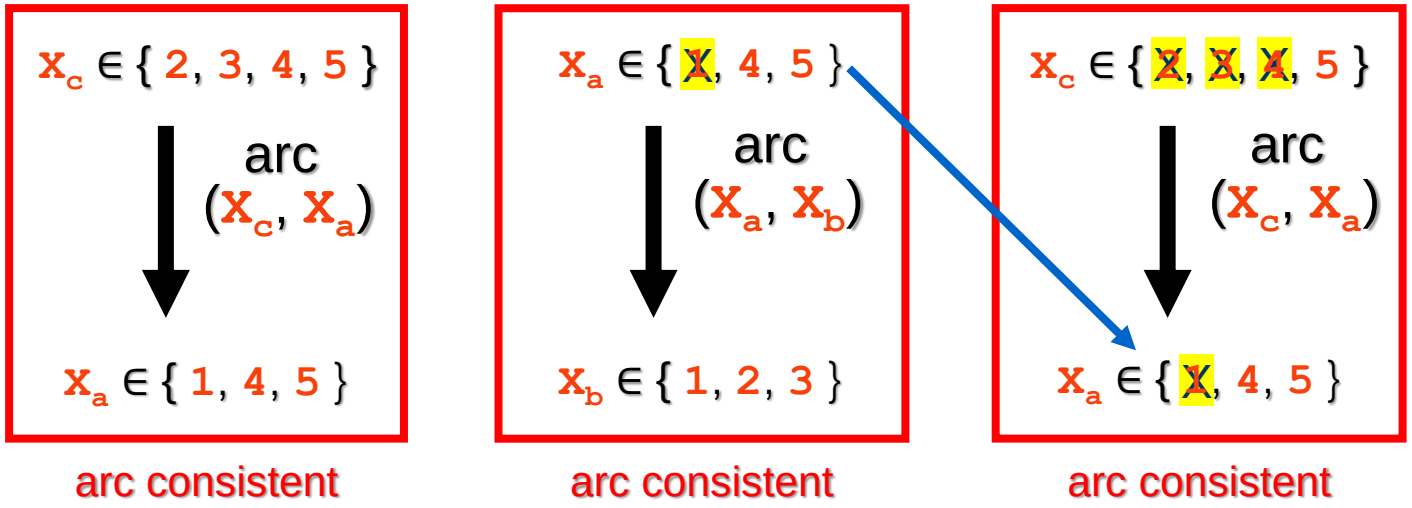
Arc Consistency Example 2

- Example
  - $D_a = \{ 1, 4, 5 \}$
  - $D_b = \{ 1, 2, 3 \}$
  - $D_c = \{ 2, 3, 4, 5 \}$
  - $X_c > X_a$
  - $X_a > X_b$
- Check
  - arc(X_c, X_a) – i.e., $X_c > X_a$
  - arc(X_a, X_b) – i.e., $X_a > X_b$



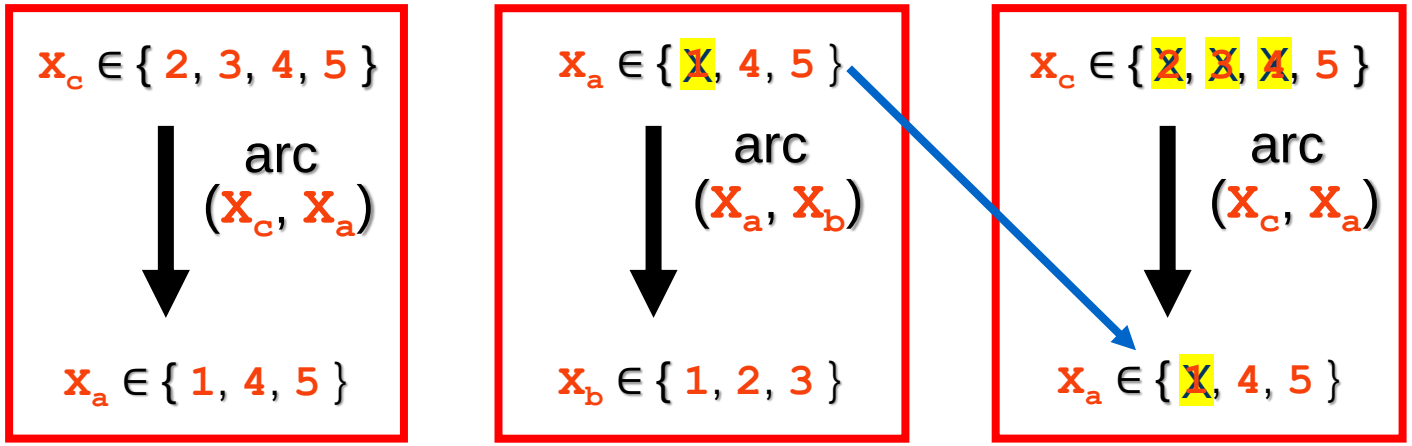
Arc Consistency Example 2

- Example
  - $D_a = \{ 1, 4, 5 \}$
  - $D_b = \{ 1, 2, 3 \}$
  - $D_c = \{ 2, 3, 4, 5 \}$
  - $X_c > X_a$
  - $X_a > X_b$
- Check
  - arc(X_c, X_a) – i.e., $X_c > X_a$
  - arc(X_a, X_b) – i.e., $X_a > X_b$



Arc Consistency Example 2

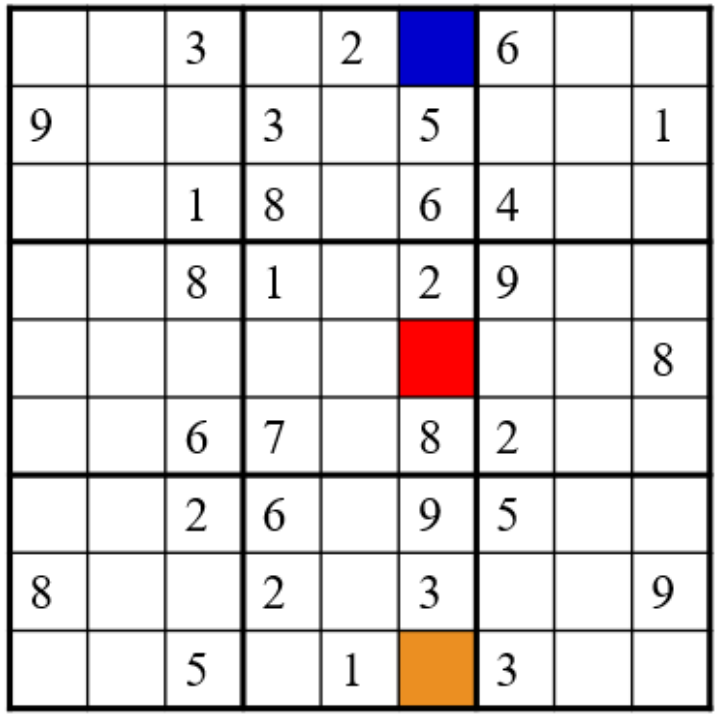
- Example
  - $D_a = \{ 1, 4, 5 \}$
  - $D_b = \{ 1, 2, 3 \}$
  - $D_c = \{ 2, 3, 4, 5 \}$
  - $X_c > X_a$
  - $X_a > X_b$
- Check
  - arc(X_c, X_a) – i.e., $X_c > X_a$
  - arc(X_a, X_b) – i.e., $X_a > X_b$



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

Sudoku Example - Chain Reaction

- Consider the following Sudoku puzzle
  - The `Alldiff` constraint on middle box reduces domain of red square to { `3`, `4`, `5`, `6`, `9` }
  - The column constraint further reduces it to { `4` }
  - Box and column constraints on the orange square reduce its domain to { `4`, `7` }
  - The red square further reduces it to { `7` }
  - the blue square 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 ← a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← POP(queue)
    if REVISE(csp, X_i, X_j) then
      if size of D_i = 0 then return false
      for each X_k in X_i.NEIGHBORS - {X_j} do
        add (X_k, X_i) to queue
  return true

function REVISE(csp, X_i, X_j) returns true iff we revise the domain of X_i
  revised ← false
  for each x in D_i do
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
      delete x from D_i
      revised ← true
  return revised
```

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

Each time a variable x_i's domain is updated add all arcs corresponding to binary constraints with other variable (not x_i) as target (except the one that just caused the revision)

Eliminating domain values of the target variable x_i relative to the other variable x_j in the binary constraint

Time Complexity of AC-3

- CSPs have at most $2 \cdot {}^{n}C_2$ 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
- Thus, the time complexity of AC-3 is: $O(n^2 \times d \times d^2) = O(n^2 \cdot d^3)$

function AC-3(*csp*) **returns** false if an inconsistency is found and true otherwise
  queue ← a queue of arcs, initially all the arcs in *csp*
  while *queue* is not empty **do**
    (X_i, X_j) ← POP(*queue*)
    if REVISE(*csp*, X_i, X_j) **then**
      if size of $D_i = 0$ **then return** *false*
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
        add (X_k, X_i) to *queue*
  return *true*

function REVISE(*csp*, X_i, X_j) **returns** true iff we revise the domain of X_i
  revised ← *false*
  for each x **in** D_i **do**
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j **then**
      delete x from D_i
      revised ← *true*
  return *revised*

Maintaining Arc Consistency

- 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/90507063206

9 Appendix

AC-3 Algorithm

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

  while *queue* is not empty **do**
    (X_i, X_j) ← POP(*queue*)
    if REVISE(*csp*, X_i, X_j) **then**
      if size of $D_i = 0$ **then return** *false*
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
        add (X_k, X_i) to *queue*
  return *true*

function REVISE(*csp*, X_i, X_j) **returns** true iff we revise the domain of X_i
  revised ← *false*
  for each x **in** D_i **do**
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j **then**
      delete x from D_i
      revised ← *true*
  return *revised*

Why not enqueue arc (x_j, x_i) ?

On initialisation, we enqueue all arcs

So, we have either already checked arc (x_j, x_i) 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_j, x_i) was consistent, and we need to know if any change of the domain of x_i could cause arc (x_j, x_i) to become inconsistent

However, we know that all values removed from x_j did not have values in x_i that satisfied the constraint; therefore, there must not be any value in x_i that requires that value in x_j either

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