Fundamentals of Artificial Intelligence

Lecture-6
Constraint Satisfaction Problems(CSP)

Debela Desalegn

April 11, 2025

Constraint Satisfaction Problems (CSP)

Variables: $\boldsymbol{X_1, X_2, \ldots, X_N}$

Domains: $\mathbf{Domain}_1, \ldots, \mathbf{Domain}_N$

$\boldsymbol{X_i}$ takes values in $\mathbf{Domain}_{\boldsymbol{i}}$

Constraints: specifying the relations between the variables

Solution: An assignment $\{\boldsymbol{X_1 : v_1, X_2 : v_2, \ldots, X_N : v_N}\}$ that satisfies all constraints

Example: Map Coloring

Variables: $\mathsf{WA,\ NT,\ Q,\ NSW,\ V,\ SA,\ T}$

Domains: $\mathsf{D} = \{\mathsf{red}, \mathsf{green}, \mathsf{blue}\}$

Constraints: adjacent regions must have different colors

Implicit: $\mathsf{WA} \neq \mathsf{NT}$

Explicit: $(\mathsf{WA}, \mathsf{NT}) \in \{(\mathsf{red}, \mathsf{green}), (\mathsf{red}, \mathsf{blue}), \ldots\}$

Solutions are assignments satisfying all constraints, e.g.:

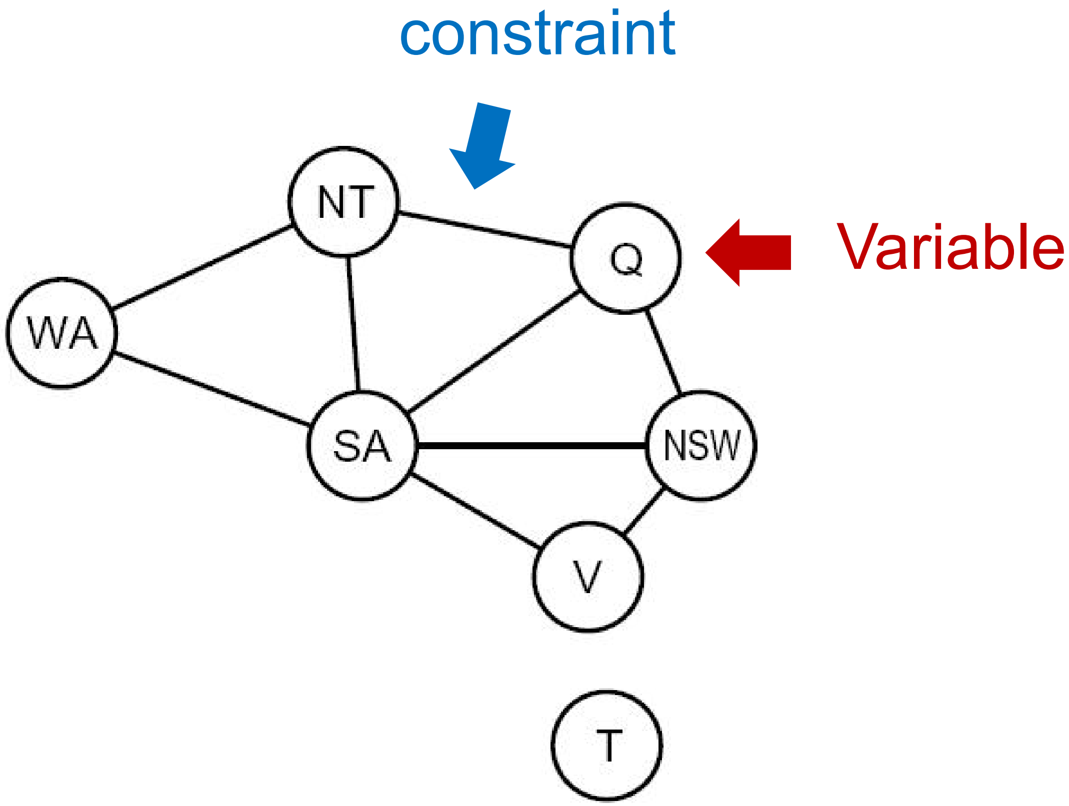
$\{\mathsf{WA{=}red,\ NT{=}green,\ Q{=}red,\ NSW{=}green,\ V{=}red,\ SA{=}blue,\ T{=}green}\}$

Real-World CSPs

- ✓ Assignment problems: e.g., who teaches what class
- ✓ Timetabling problems: e.g., which class is offered when and where?
- ✓ Hardware configuration
- ✓ Transportation scheduling
- ✓ Factory scheduling
- ✓ Circuit layout
- ✓ …

Many real-world problems involve real-valued variables.

Constraint Graph



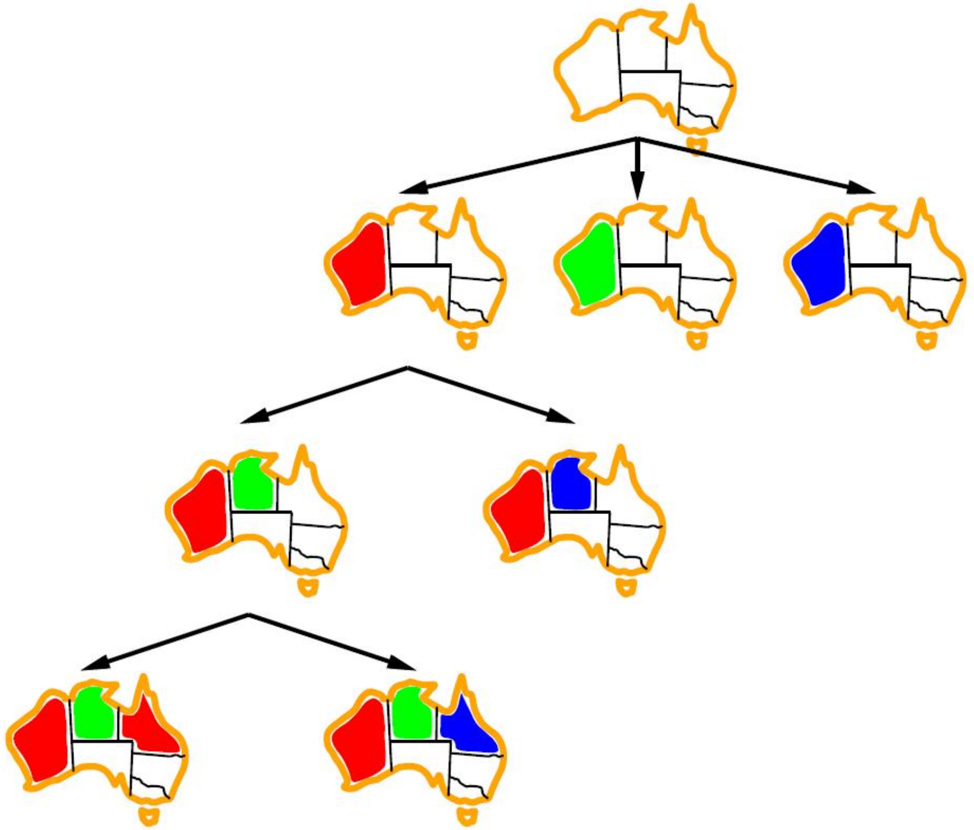
(more convenient for **binary constraint** CSPs)

Every constraint involves at most 2 variables

How to Solve CSP?

Treat it as a search problem

- ✓ Assign one variable at a time
- ✓ **State:** A partial assignment
- ✓ **Action:** Assign value to an unassigned variable
- ✓ **Goal test:** check whether all constraint are satisfied



But there's more structure to leverage

- Variable ordering doesn't matter
- Variables are interdependent in a local way

We will start from known search algorithms, and try to speed it up.

Backtracking Search

Backtracking Search

Backtracking search = DFS + failure-on-violation

BacktrackingSearch({ }, Domain) returns an assignment or reports failure.

BacktrackingSearch(x, Domain):

  If x is a complete assignment: **return** x.

  Let X_i be the next unassigned variable.

  For each value $v \in \mathrm{Domain}_i$:

    $x' \leftarrow x \cup \{X_i : v\}$

    If x' violates constraints: **continue**

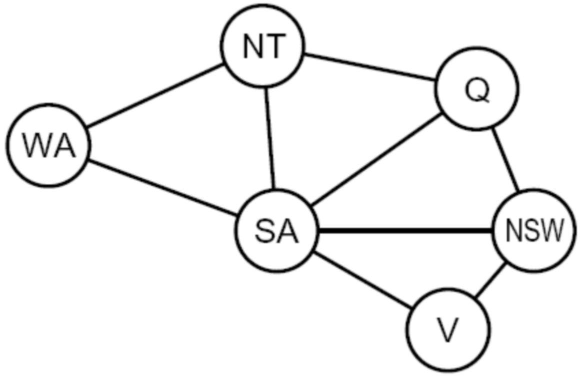
    return **BacktrackingSearch**(x', Domain)

  return failure

Improving Backtracking Search

- ✓ Forward checking
- ✓ Maintaining arc consistency (more powerful than forward checking)
- ✓ Dynamic ordering

Vanilla Backtracking Search

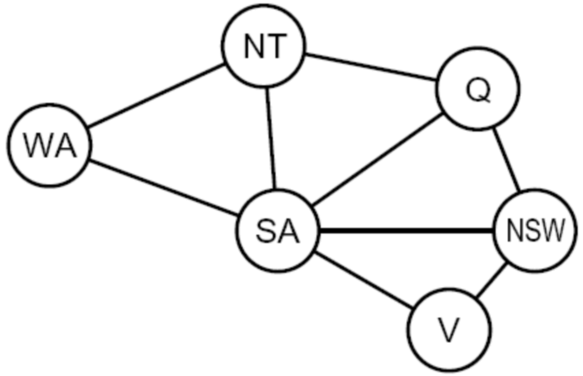


Suppose we assign the variables in the order of

WA, Q, V, NT, NSW, SA

No valid assignment for **SA**.
Then the algorithm backtracks to try other assignments…

Forward Checking



Cross off values that violate a constraint when added to the existing assignment

Inconsistency found for **SA** (even though we haven't reached the layer of SA).

Forward Checking

BacktrackingSearch(x, Domain):

  If x is a complete assignment: **return** x.

  Let X_i be the next unassigned variable.

  For each value $v \in \mathrm{Domain}_i$:

    $x' \leftarrow x \cup \{X_i : v\}$

    If x' violates constraints: **continue**

    return BacktrackingSearch(x', Domain')

  return failure

Forward Checking

BacktrackingSearch(x, Domain):

If x is a complete assignment: **return** x.

Let X_i be the next unassigned variable.

For each value $v \in \text{Domain}_i$:

$x' \leftarrow x \cup \{X_i: v\}$

$\text{Domain}', \ \text{Consistent} =$ **ForwardChecking**$(x', X_i, v, \text{Domain})$

If not Consistent: **continue**

return **BacktrackingSearch**(x', Domain')

return failure

ForwardChecking $(x', X_i, v, \text{Domain})$:

$\text{Domain}' \leftarrow \text{Domain}$

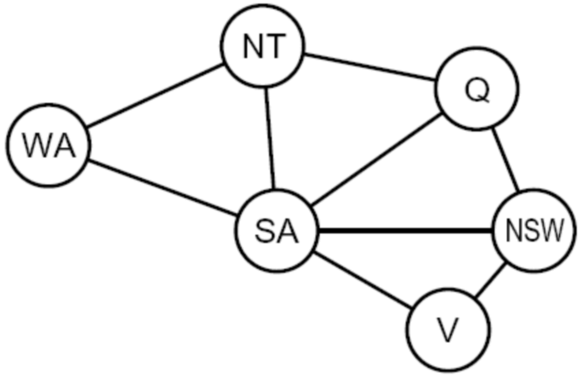
For all X_j that is unassigned in x' and connected to X_i:

Delete values in Domain'_j that are inconsistent with $\{X_i: v\}$

If Domain'_j is empty: return $\text{Domain}', False$

return $\text{Domain}', True$

Can We Prune Even More?



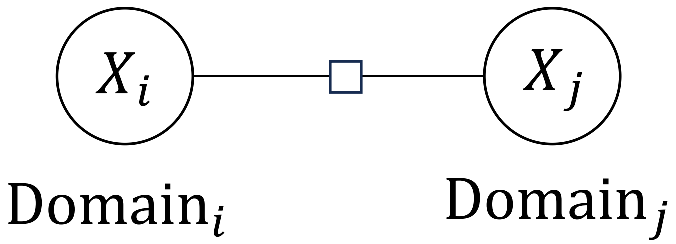
With forward checking:

After assigning **Q** with green, **NT** and **SA**’s domains are left with only blue.

But **NT** and **SA** are neighbors, so there is no consistent assignment from here.

How can we detect such inconsistency at this step?

Arc Consistency



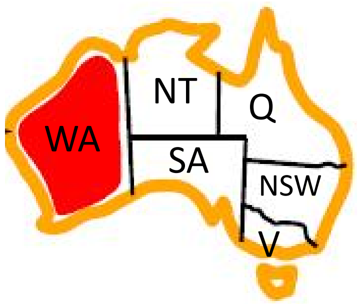
Fact. Let $v \in \text{Domain}_i$ be such that for all $w \in \text{Domain}_j$, $\{X_i: v, \quad X_j: w\}$ violates the constraint on (X_i, X_j). Then we can remove v from Domain_i.

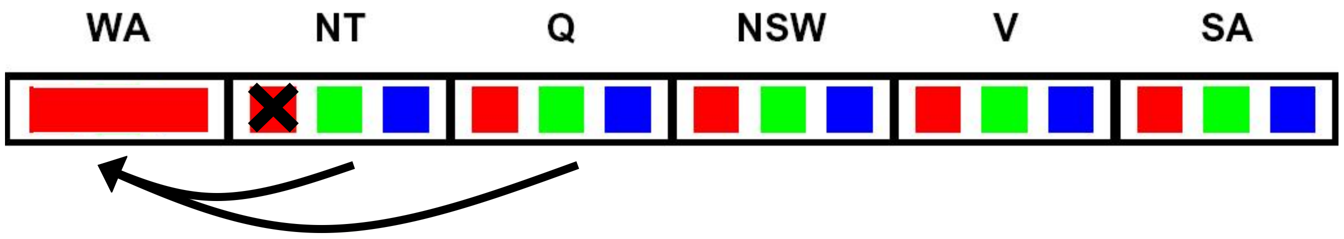
Definition (Arc Consistency on $X_i \rightarrow X_j$). For all $v \in \text{Domain}_i$, there is some $w \in \text{Domain}_j$ such that $\{X_i: v, \quad X_j: w\}$ satisfies the constraint on (X_i, X_j).

Idea to prune more: keep checking whether we can remove elements from any Domain using the fact above. (i.e., always maintaining arc consistency).

Maintaining Arc Consistency (MAC)

Forward checking: maintaining arc consistency from unassigned variables to newly assigned variables.

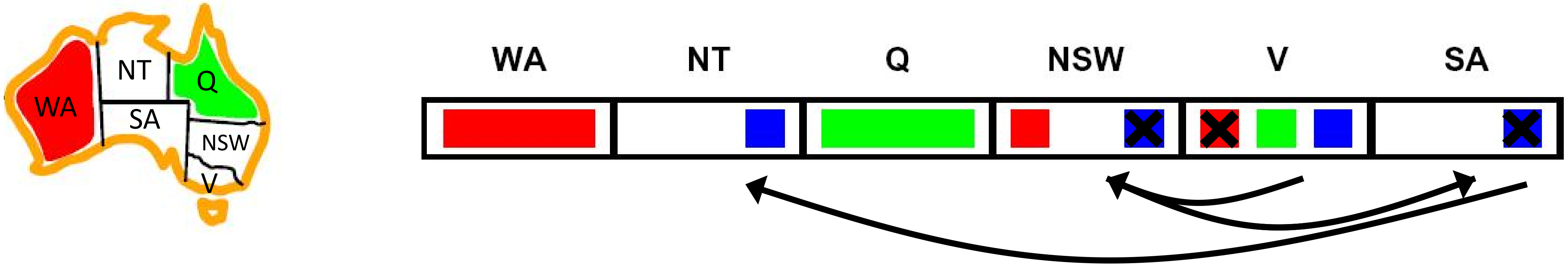




Maintaining Arc Consistency (MAC)

We can prune more if we ensure arc consistency for all arcs.

Remember: Delete from the tail!



If X's domain changes, neighbors of X need to be rechecked!

Maintaining Arc Consistency (MAC)

AC3:

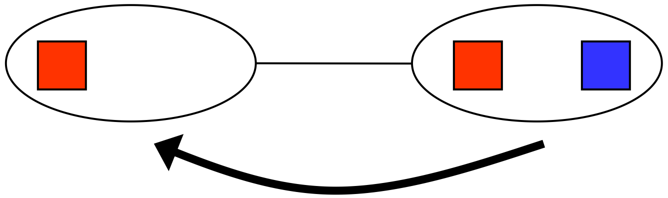
queue ← initial queue
while queue not empty:
  (X_i, X_j) ← POP(queue)
  if arc $X_i \rightarrow X_j$ is not consistent:
    Revise Domain_i to make it consistent
    if Domain_i is empty: **return** $False$
    for each X_k connected to X_i:
      add (X_k, X_i) to queue

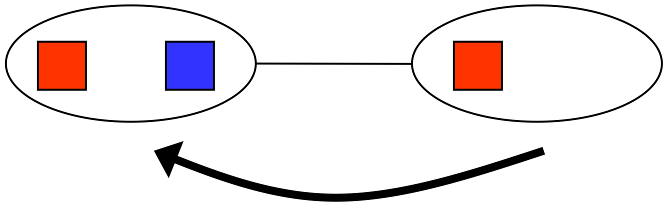
return $True$

Arc Consistency in Map Coloring Problems

Useful when there is only one color left at the arc head

Actually, this is also the only useful case in the map coloring problem





Arc Consistency in Other Problems

$X_i \in \text{Domain}_i = \{1,2,3,4,5\}$

$X_j \in \text{Domain}_j = \{1,2\}$

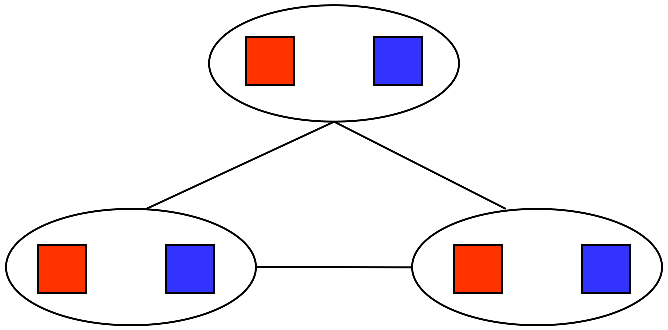
Constraint: $X_i + X_j = 4$

$X_i \longrightarrow X_j$

X_i	X_j
~~1~~	1
2	2
3	
~~4~~	
~~5~~	

Limitations of Arc Consistency

- ✓ Some failure modes cannot be detected by arc consistency.
- ✓ Therefore, we still need "backtracking".



Maintaining Arc Consistency (MAC)

Combining AC3 with backtracking search:

BacktrackingSearch(x, Domain):
  If x is a complete assignment: **return** x.
  Let X_i be the next unassigned variable.
  For each value $v \in \text{Domain}_i$:
    $x' \leftarrow x \cup \{X_i : v\}$
    $\text{Domain}', \ \text{Consistent} =$ **AC3**($x', X_i, v, \text{Domain}$)
    If not Consistent: **continue**
    return **BacktrackingSearch**(x', Domain$'$)
  return failure

Maintaining Arc Consistency (MAC)

AC3$(x', X_i, v, \mathrm{Domain})$:

  $\mathrm{Domain}' \leftarrow \mathrm{Domain}$

  queue $\leftarrow \{(X_j, X_i)$ for all X_j that is unassigned in x' and connected to $X_i\}$

  while queue not empty:

    $(X_k, X_\ell) \leftarrow$ POP(queue)

    if arc $X_k \rightarrow X_\ell$ is not consistent:

      Revise Domain'_k to make it consistent

      if Domain'_k is empty: **return** $\mathrm{Domain}', False$

      for each X_m that is unassigned in x' and connected to X_i:

        add (X_m, X_k) to queue

  return $\mathrm{Domain}', True$

Maintaining Arc Consistency (MAC)

Drawbacks of backtracking search with arc-consistency check?

K-Consistency

2-Consistency (= Arc-Consistency)For any assignment $X_i = v_i$, there exists an assignment of $X_j = v_j$ such that $\{X_i = v_i,\ X_j = v_j\}$ satisfies the constraint on $(X_i,\ X_j)$.

3-Consistency

For any consistent assignment $\{X_i = v_i,\ X_j = v_j\}$, there exists an assignment $X_k = v_k$ such that $\{X_i = v_i,\ X_j = v_j,\ X_k = v_k\}$ satisfies constraints on $(X_i,\ X_k), (X_j,\ X_k),\ (X_i, X_j, X_k)$.

K-Consistency

For any consistent assignment $\{X^{(1)} = v^{(1)},\ \ldots,\ X^{(k-1)} = v^{(k-1)}\}$, there exists an assignment $X^{(k)} = v^{(k)}$ such that $\{X^{(1)} = v^{(1)},\ \ldots,\ X^{(k)} = v^{(k)}\}$ satisfies all constraints among $X^{(1)},\ \ldots,\ X^{(k)}$.

Improving Backtracking Search

Forward checking

Maintaining arc consistency (more powerful than forward checking)

Dynamic ordering

Ordering

BacktrackingSearch(x, Domain):

If x is a complete assignment: **return** x.

Let X_i be the next unassigned variable. Which **variable** should we pick first?

For each value $v \in \mathrm{Domain}_i$: Which **value** should we try first?

$x' \leftarrow x \cup \{X_i : v\}$

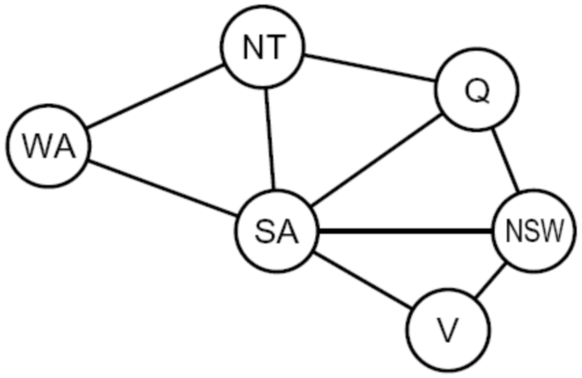
$\mathrm{Domain}', \ \mathrm{Consistent} =$ **AC3**(x', X_i, v, Domain)

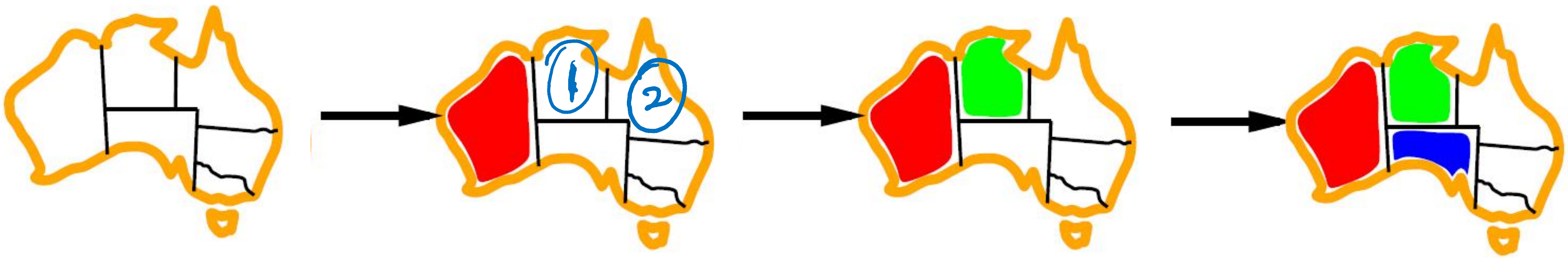
If not Consistent: **continue**

return BacktrackingSearch(x', Domain$'$)

return failure

Variable Ordering





Variable Ordering

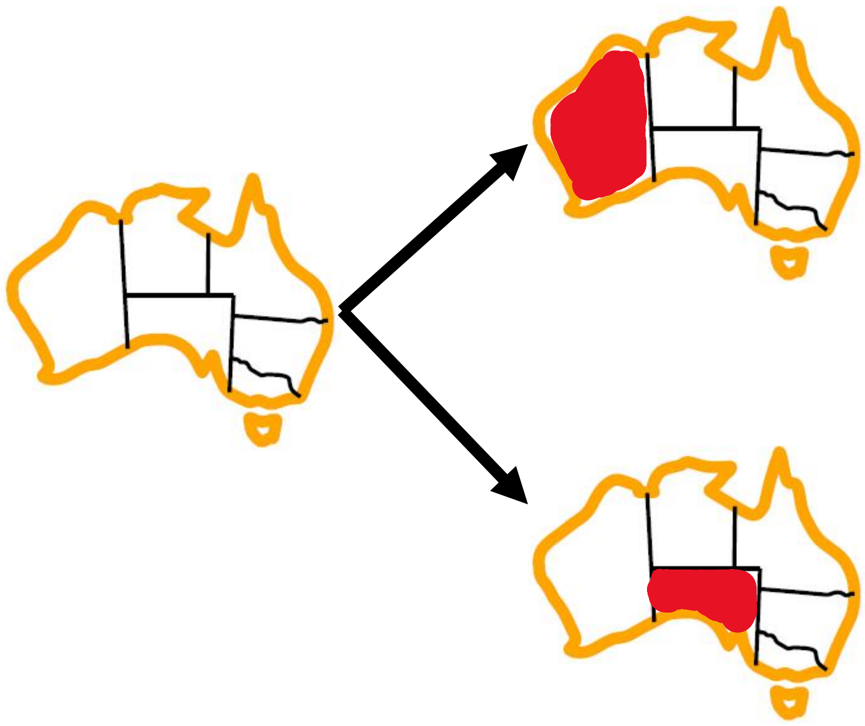
Minimum Remaining Value (MRV) heuristic

Choose variable that has the fewest left values in its domain.

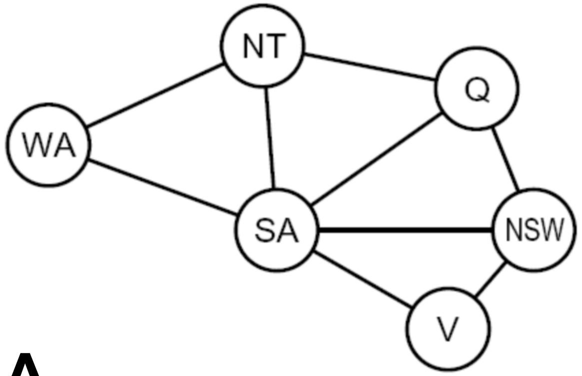
Why?

- Must assign **every** variable
- If going to fail, fail early → more pruning

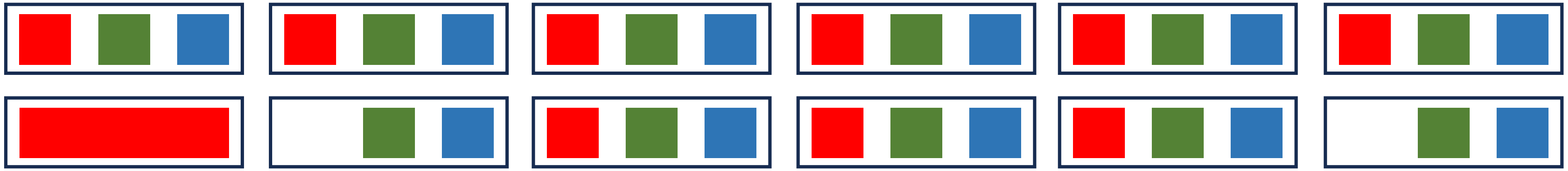
Variable Ordering



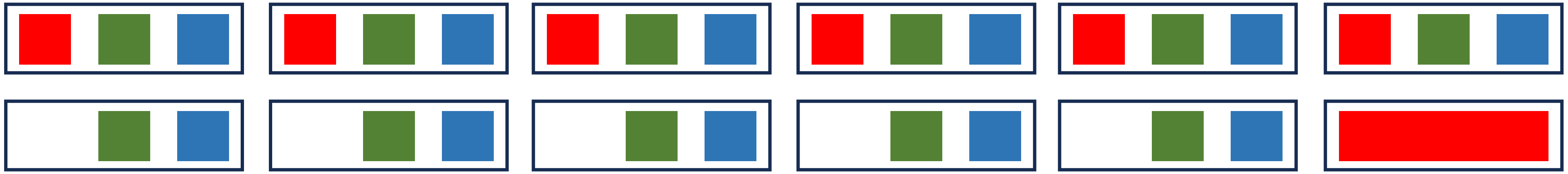
Variable Ordering



WA NT Q NSW V SA



WA NT Q NSW V SA



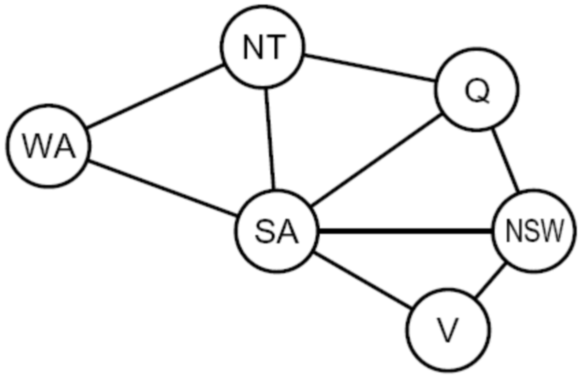
Variable Ordering

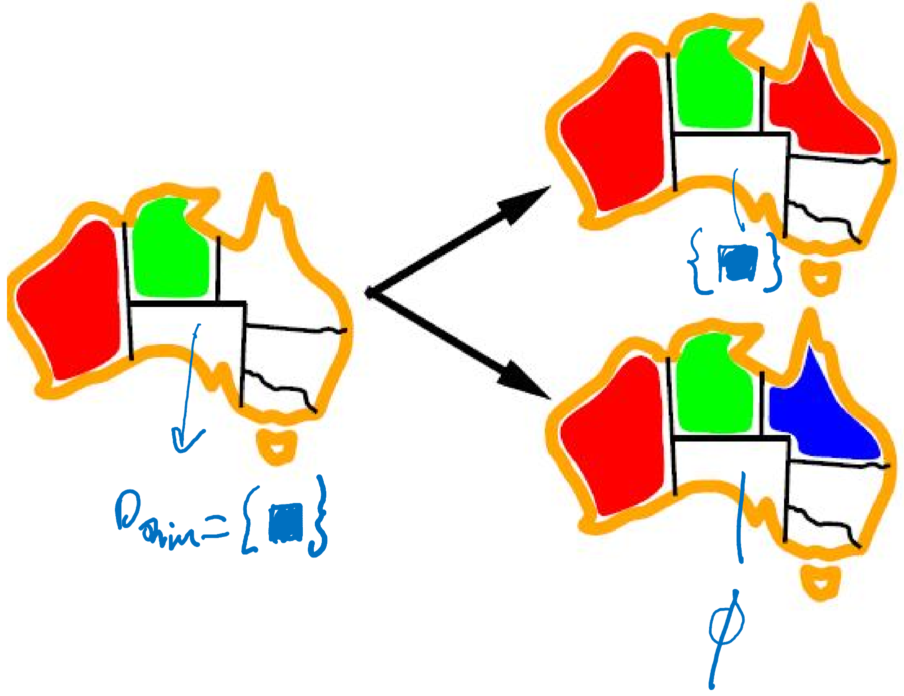
Degree heuristic

Choose variable that is involved in the largest number of constraint.

Could be a good tie-breaking strategy along with MRV.

Value Ordering





Value Ordering

Least Constrained Value (LCV) heuristic

Choose the value that rules out the fewest values in the remaining variables.

Can be estimated by forward checking or arc-consistency checking

Why?

- Needs to choose some value
- Choosing value most likely to lead to solution
- Unlike variable ordering where we have to consider all variables, there is no need to consider all values

Ordering

Minimum Remaining Value (MRV) heuristic

Choose variable that has the fewest left values in its domain.

Degree heuristic

Choose variable that is involved in the largest number of constraint.

Least Constrained Value (LCV) heuristic

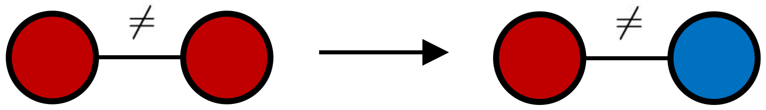
Choose the value that rules out the fewest values in the remaining variables.

Local Search

Iterative Improvement

Start from some complete assignment that may not satisfy all constraints

Modify the assignment, trying to resolve violations



Iterative Improvement

MinConflict (MaxSteps):

$x \leftarrow$ an initial complete assignment

for iter = 1 to MaxStep:

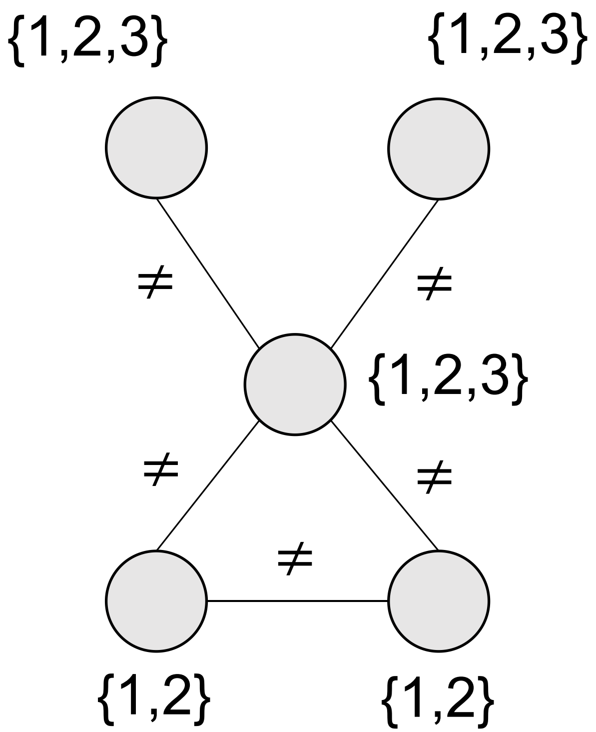
if x is a solution **then return** x

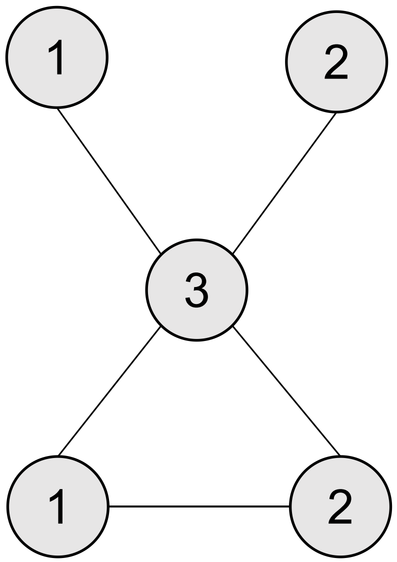
Let X_i be a randomly chosen conflicted variable

Let v be the value for X_i that minimizes conflicts

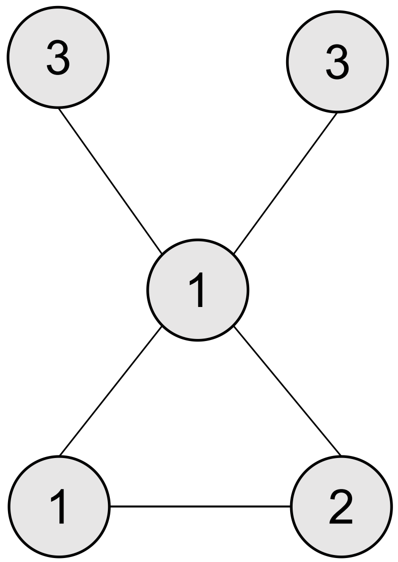
Reassign $X_i = v$ in x

Failure of MinConflict





A feasible solution



An unfortunate initialization for MinConflict

Rescue

For local search algorithms like MinConflict, we will randomly generate multiple initial assignments. For each initial assignment, we will only run it for MaxStep iterations before giving up.