



# COL333/671: Introduction to AI

Semester I, 2024-25

## Constraint Satisfaction

Rohan Paul

# Outline

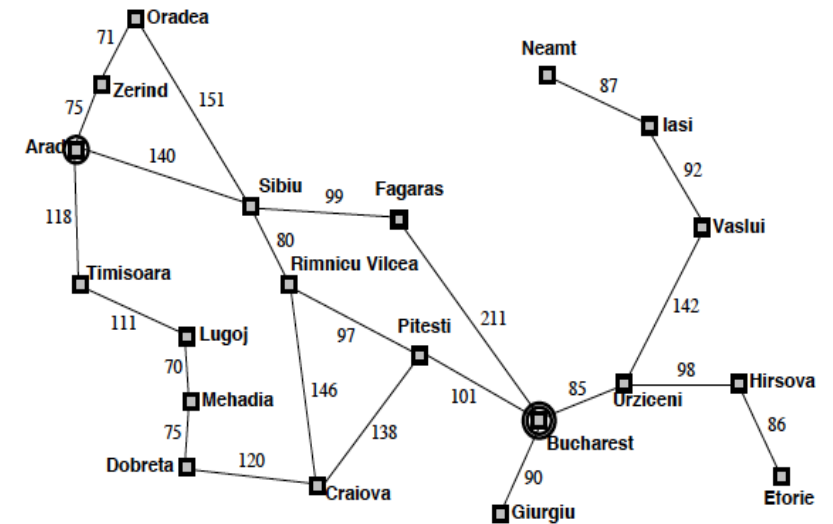
- Last Class
  - Local Search Algorithms
- This Class
  - Constraint Satisfaction Problems
- Reference Material
  - AIMA Ch. 6

# Acknowledgement

**These slides are intended for teaching purposes only. Some material has been used/adapted from web sources and from slides by Doina Precup, Dorsa Sadigh, Percy Liang, Mausam, Dan Klein, Nicholas Roy and others.**

# Standard Search Problems

- A path from the start to the goal state is the solution.
- Paths have costs (or depths).
- Heuristics provide **problem-specific** guidance.
- State is a “**black box**”, arbitrary data structure
- Goal test can be **any** function over states.



Route finding problem solved as a search problem

# Constraint Satisfaction Problems (CSPs)

- CSP
  - A set of **variables**  $\{X_1, X_2, \dots, X_n\}$  to which **values**  $(d_1, d_2, \dots, d_n)$  from a domain  $D$  can be assigned.
- Solution
  - A **complete** variable assignment that is **consistent** (satisfies all the given constraints).
- States
  - Explicitly represented as **variable assignments**
- Goal test:
  - The **set of constraints** specifying allowable combination of values for subset of variables.

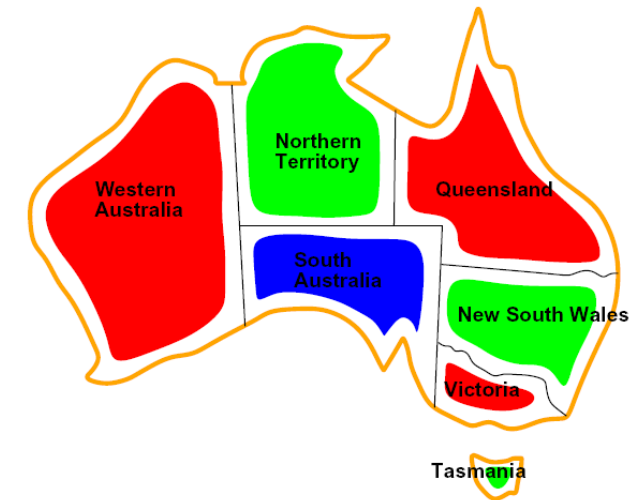
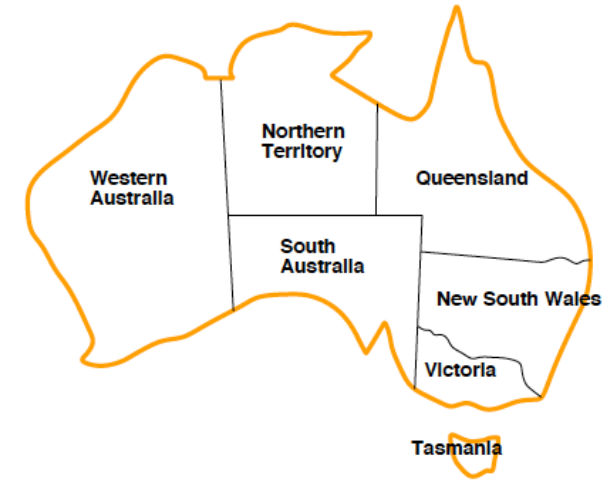
# Example: Map Coloring

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

Implicit:  $WA \neq NT$

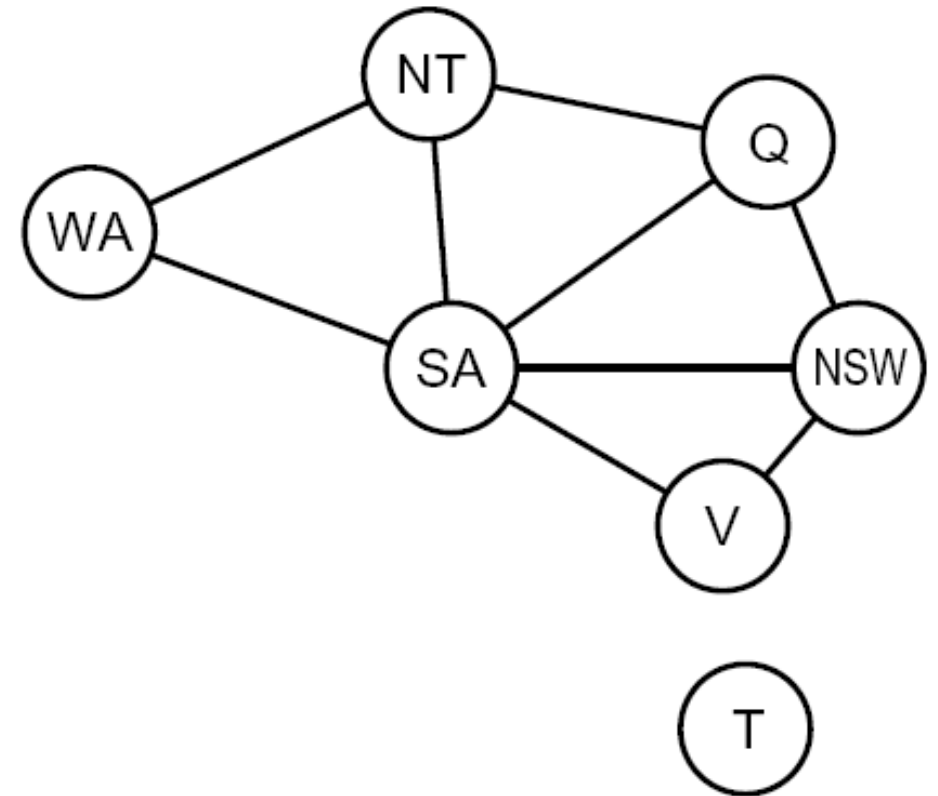
Explicit:  $(WA, NT) \in \{(\text{red, green}), (\text{red, blue}), \dots\}$

- Solutions are assignments satisfying all constraints, e.g.:  
 $\{WA=\text{red}, NT=\text{green}, Q=\text{red}, NSW=\text{green}, V=\text{red}, SA=\text{blue}, T=\text{green}\}$

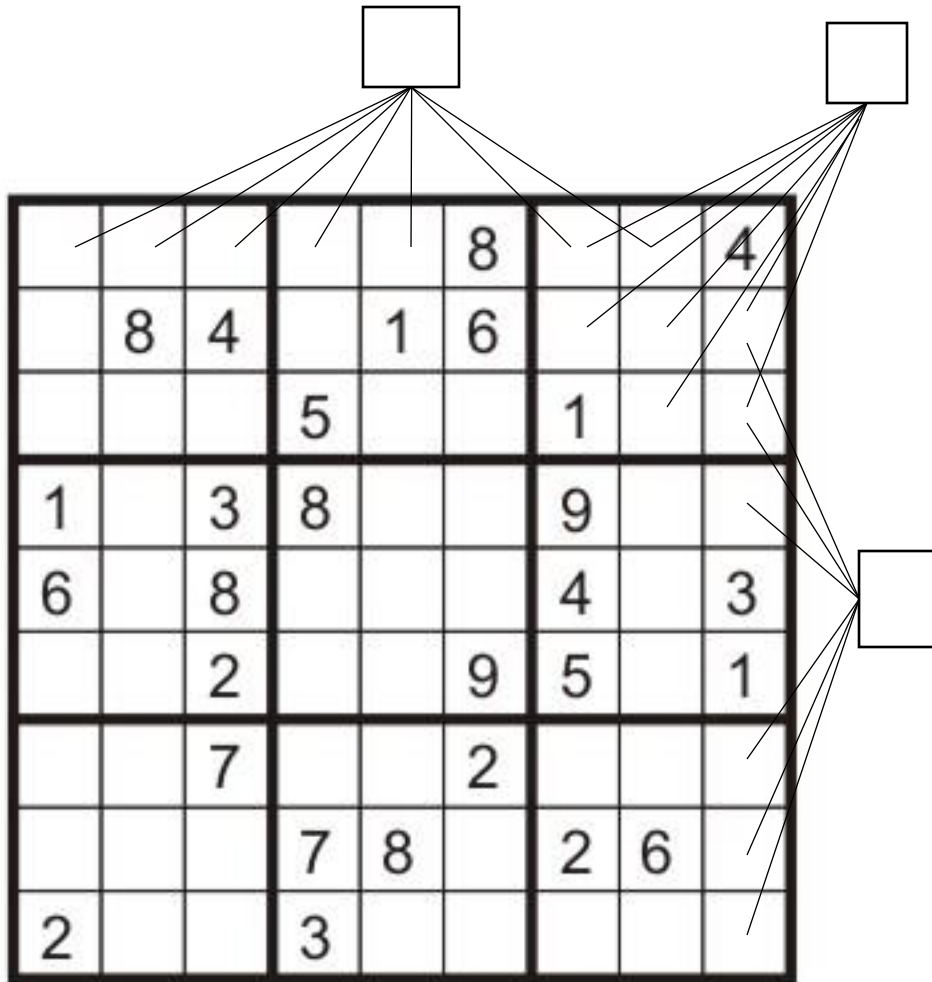


# Constraint Graph

- Binary constraint satisfaction problem
  - Each constraint relates (at most) two variables
- Binary constraint graph
  - Nodes are variables
  - Arcs show constraints
- General-purpose CSP solvers make use of the graph structure to speed up search.
  - E.g., Tasmania is an independent subproblem.



# Example: Sudoku



Variables: Each (open) square

Domains: {1, 2, ..., 9}

Constraints:

9-way alldiff for each column

9-way alldiff for each row

9-way alldiff for each region



# Example: Scheduling

ALL INDIA INSTITUTE OF MEDICAL SCIENCES, NEW DELHI  
DUTY ROSTER FOR THE MONTH OF Oct L.R / Mat. OT . OCTOBER 2011

Check your duty daily.

NAME OF STAFF	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Remarks
1. Ms. NARAY ANTONY	A	A	D	D	A	A	A	A	A	D	A	A	A	A	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	
2. Ms. PUNYA DALAL	A	A	A	A	A	A	A	A	A	D	A	A	A	A	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	
3. Ms. PUNAM SIDDHANT	A	A	A	A	A	A	A	A	A	D	A	A	A	A	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	
4. Ms. KUNJUN NAL RATH	A	A	A	A	A	A	A	A	A	D	A	A	A	A	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	
5. Ms. KHALIFA L. HAI	A	A	D	D	A	A	A	A	D	A	N	N	D	B	B	B	A	A	A	D	A	A	A	D	A	A	N	D	D	D	D	9
6. Ms. ABCHHA SINGH	A	A	D	D	A	A	A	A	D	A	N	N	D	B	B	B	A	A	A	D	A	A	A	D	A	A	N	D	D	D	D	9
7. Ms. MAYA VINOD	A	A	D	D	A	A	A	A	D	A	N	N	D	B	B	B	A	A	A	D	A	A	A	D	A	A	N	D	D	D	D	9
8. Ms. TERTY TERTY	A	A	D	D	A	A	A	A	D	A	N	N	D	B	B	B	A	A	A	D	A	A	A	D	A	A	N	D	D	D	D	9
9. Ms. TISHA MARYAM	A	A	D	D	A	A	A	A	D	A	N	N	D	B	B	B	A	A	A	D	A	A	A	D	A	A	N	D	D	D	D	9
10. Ms. KIRITA CHANDRA	A	A	D	D	A	A	A	A	D	A	N	N	D	B	B	B	A	A	A	D	A	A	A	D	A	A	N	D	D	D	D	9
11. Ms. SHABANA M. JALAL	A	A	D	D	A	A	A	A	D	A	N	N	D	B	B	B	A	A	A	D	A	A	A	D	A	A	N	D	D	D	D	9
12. Ms. MADHU SOLANKI	A	A	D	D	A	A	A	A	D	A	N	N	D	B	B	B	A	A	A	D	A	A	A	D	A	A	N	D	D	D	D	9
13. Ms. PARVATHY DAS	A	A	D	D	A	A	A	A	D	A	N	N	D	B	B	B	A	A	A	D	A	A	A	D	A	A	N	D	D	D	D	9
14. Ms. TINTA THOMAS	A	A	D	D	A	A	A	A	D	A	N	N	D	B	B	B	A	A	A	D	A	A	A	D	A	A	N	D	D	D	D	9
15. Ms. SHASHI	A	A	D	D	A	A	A	A	D	A	N	N	D	B	B	B	A	A	A	D	A	A	A	D	A	A	N	D	D	D	D	9
16. Ms. RENU CHANDAN	A	A	D	D	A	A	A	A	D	A	N	N	D	B	B	B	A	A	A	D	A	A	A	D	A	A	N	D	D	D	D	9

An example of a manually prepared roster of persons assigned to shifts/rooms.

## Nurse Scheduling Problem

- $I$  set of nurses
- $T$  discrete time horizon
- $A$  set of activities
- cover constraints  $C_t$  / day  $t$
- working rules  $R_i$  / nurse  $i$

28 days

$N$  night,  $M$  morning,  $E$  evening,  $R$  rest

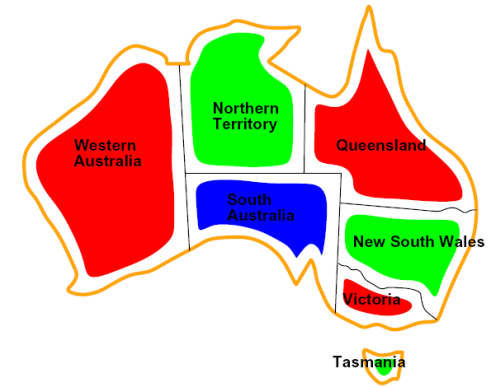
between 2 and 3 nurses at night

at least 2 mornings a week

	employees							
	N	N	E	$X_3^0$	M	M	R	
	R	M	N	$X_3^1$	N	R	R	
$\mathcal{R}_2$	$X_0^2$	$X_1^2$	$X_2^2$	$X_3^2$	$X_4^2$	$X_5^2$	$X_6^2$	
	E	N	N	$X_3^3$	R	R	M	
	N	R	R	$X_3^4$	E	N	N	
	R	R	M	$X_3^5$	N	N	N	
								time
								$\mathcal{C}_3$

# Types of Constraints

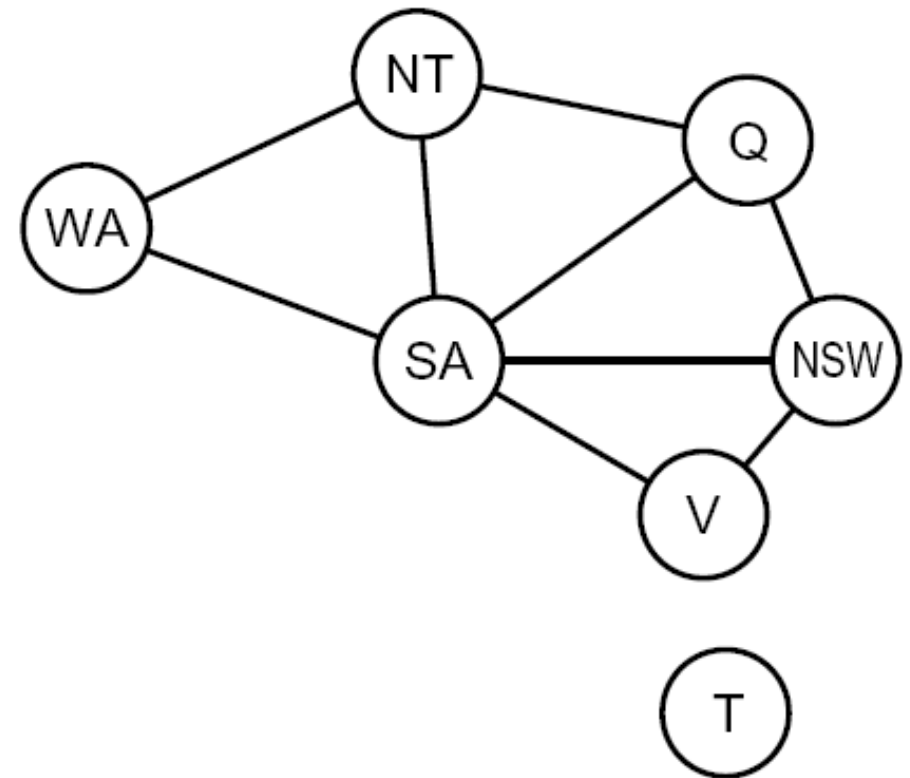
- Varieties of Constraints
  - Unary constraints involve a single variable (equivalent to reducing domains), e.g.:  
 $SA \neq \text{green}$
  - Binary constraints involve pairs of variables, e.g.:  
 $SA \neq WA$
  - Higher-order constraints involve 3 or more variables:  
e.g., sudoku constraints
- Preferences (soft constraints):
  - E.g., red is better than green
  - Often representable by a cost for each variable assignment
  - Gives constrained optimization problems



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

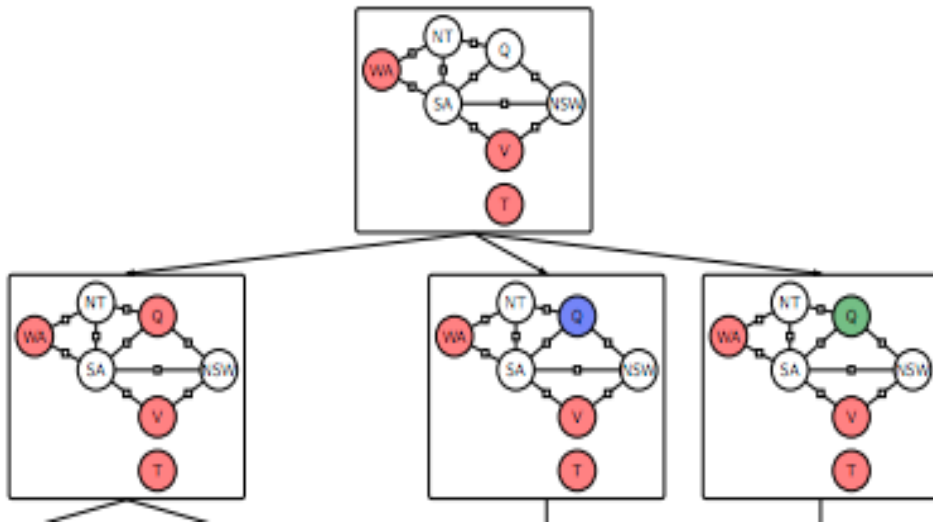
# Solving CSPs

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

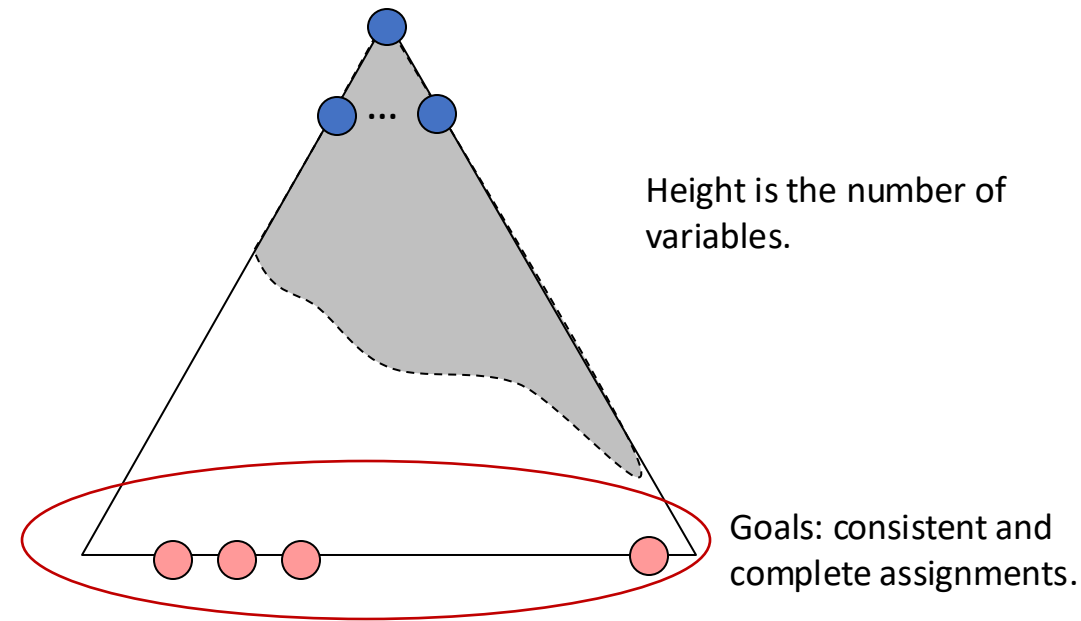


# Standard Search

One stage of successor generation. Select variable Q and try Red, Blue or Green values.



- Enumerate all assignments to variables. Create the entire tree.
- Check all the constraints at the end. Goal can be checked at the bottom of the tree.
- Can use a search method like DFS.



## Problem with a direct DFS search?

- Testing the constraints at the end, only then we know that the goal has been attained.
- *Do we need to wait till all variables assigned if we already know that the assignment is failing.*
- *Can we test incrementally and detect failures earlier than the complete assignment?*

# Backtracking Search over Assignments

- **Search component**

- At each step, consider assignments to a single variable
- Variable assignments are commutative (we can pick the order)
- I.e., [WA = red then NT = green] same as [NT = green then WA = red]

- **Inference or Constraint Checking**

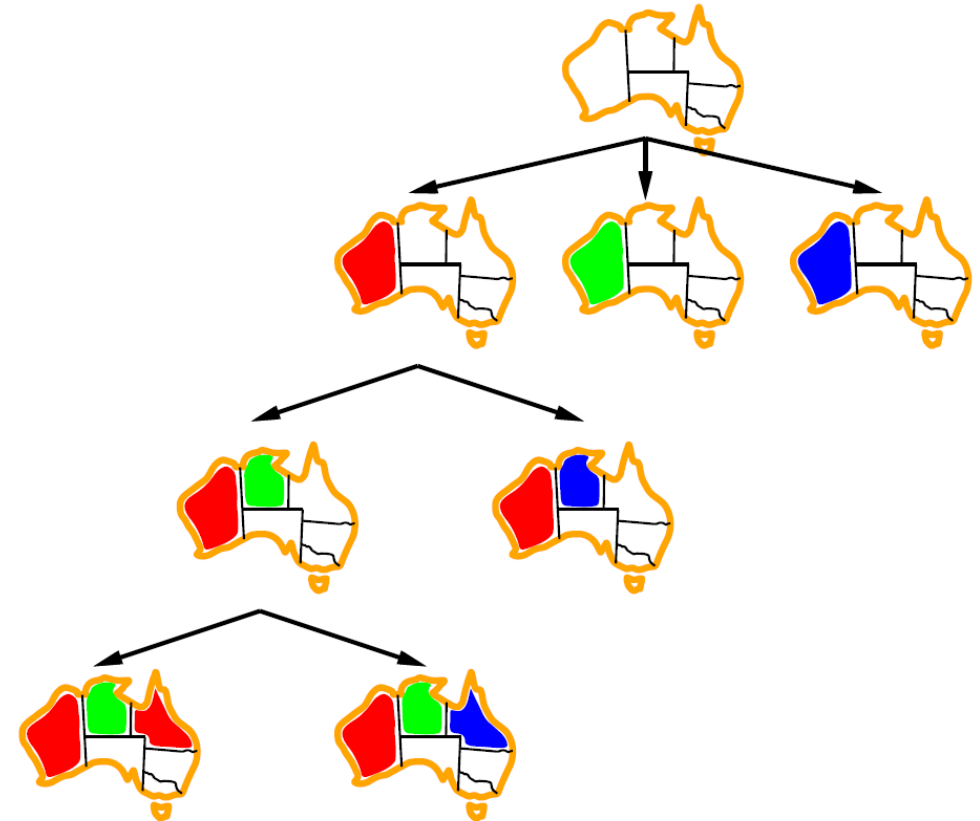
- **Can we check constraints incrementally instead of all at the end?**
  - Incremental "goal test". Check constraints as the variable is assigned.
  - I.e. only assign values to variables which do not conflict previous assignments.
- Some computation is involved in checking constraints.

- **Backtracking Search**

- Depth-first search with *incremental variable assignment* and *constraint checking on the go*.
- ***Back track as soon as a failure is detected.***

# Backtracking Search

- Informally,
  - Pick a variable to assign.
  - Pick an assignment for the variable.
  - Check if all the constraints are satisfied.
  - If the constraints are not satisfied, then try a different assignment.
  - If no assignments left, need to backtrack.
  - If the assignment is complete, then we have a solution.
  - .....



Generate successors by selecting variables and values. Incrementally check the violation of constraints (backtrack when necessary).

# Backtracking Search: Pseudocode

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

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

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

# Backtracking Search

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

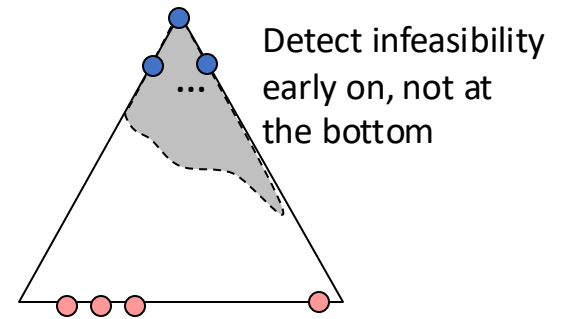
function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment given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

How to order the variables during backtracking search?

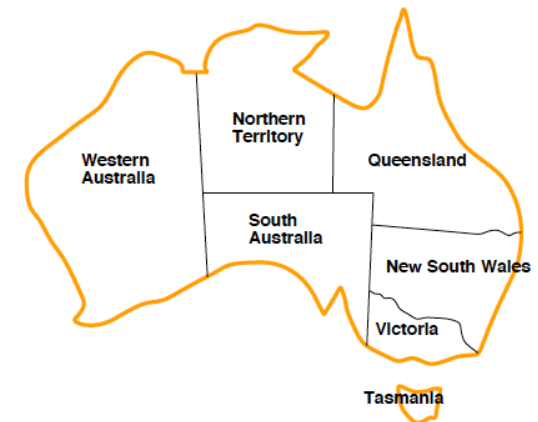


# Ordering Variables

- Most Constrained Variable (Minimum Remaining Values)
  - When you have multiple variables to assign, then choose the variable with the *fewest* remaining legal values in its domain
    - A CSP solution must have an assignment for all variables.
    - Try the variables likely to fail early rather than late. Fail fast.

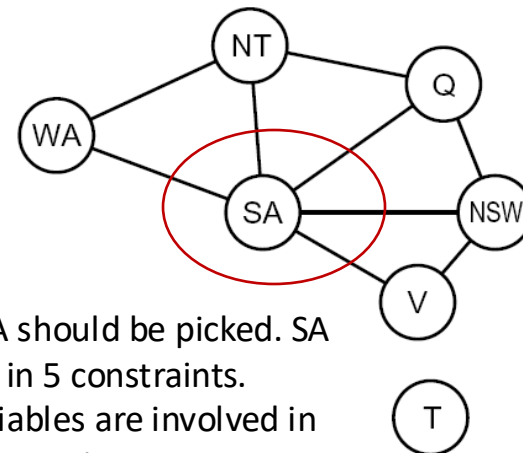
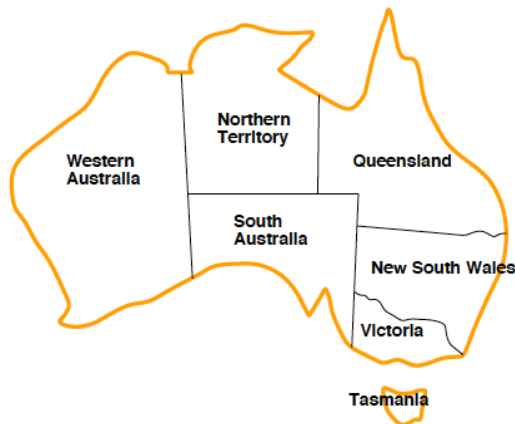


After assigning WA = R, the variables NT and SA have two legal values whereas Q, NSW, V and T have three legal values. Prefer selecting NT or SA over the other remaining variables.



# Degree Heuristic

- Take the case of picking the first variable to assign.
  - Minimum Remaining Values Heuristic does not help in the first variable. All have the same number of legal values in the domain.
  - In general, how to break ties among MRV variables?
- Degree Heuristic
  - Select the variable involved in the **largest number of constraints** on other unassigned variables.
  - Why?
    - This value reduces possible values for others. In effect, reduces branching factor.



Variable SA should be picked. SA is involved in 5 constraints. Others variables are involved in 3, 2 or 0 constraints.

# Backtracking Search

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

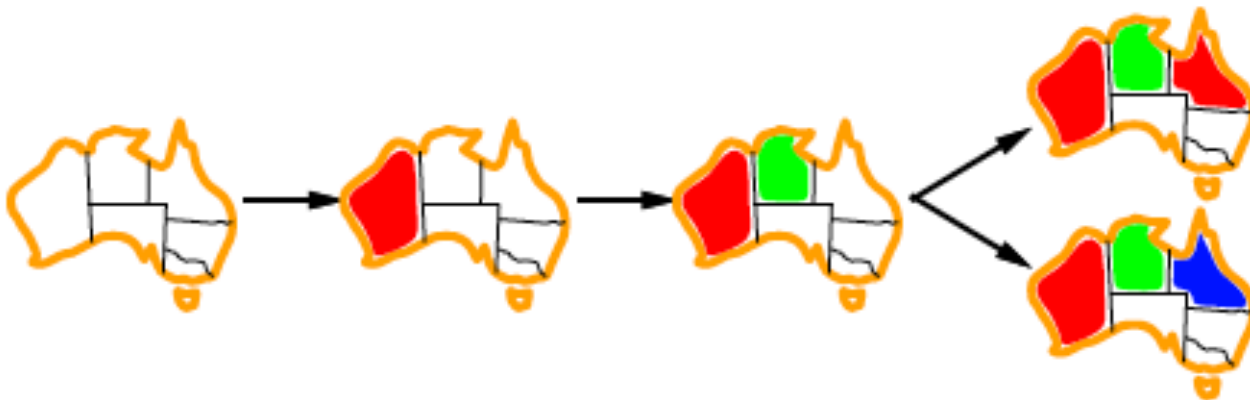
function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment given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

# Ordering Values

- Least Constrained Value

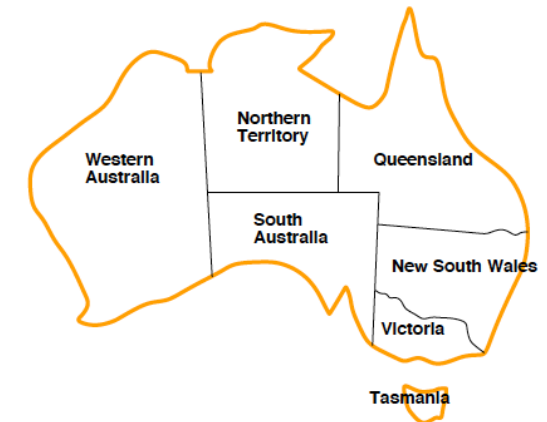
Given a variable choose a value that rules out the fewest values in the remaining unassigned variables.

- Leave maximum flexibility for subsequent assignments.
- We only need one value (assigned to a variable) so that the constraints are satisfied. Look for most likely solutions first. Fail last.



If **Q = Red** then, there is one possible assignment for SA = B

If Q = Blue then, there is no legal value left for SA.



Consider the case where: WA = R and NT = G.

Next, we pick Q for an assignment. Options are Q=R or Q=B.

Examine effect on SA (Q=Red is a better option) as it leaves a possible assignment instead of Q=Blue.

# Solving CSPs: Improving Efficiency

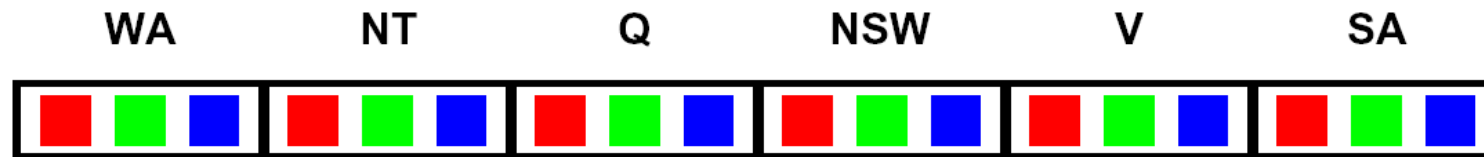
- Which variable should be assigned next?
- In what order should its values be tried?
- Can we detect inevitable failures early?
- Can we take advantage of the problem structure?

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

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add { var = value } to assignment
      inferences ← INFERENCE(csp, var, value)
      if inferences ≠ failure then
        add inferences to assignment
        result ← BACKTRACK(assignment, csp)
        if result ≠ failure then
          return result
      remove { var = value } and inferences from assignment
  return failure
```

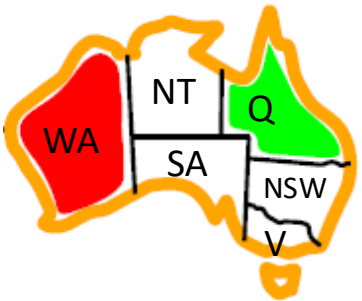
# Inference/Filtering: Forward Checking

- Basic Idea: Track domains for unassigned variables and eliminate values that violate constraints with existing assignments. Propagate information from assigned to unassigned variables linked with a constraint.
- Forward Checking: When a variable  $X$  is assigned, check the unassigned variable  $Y$  connected to  $X$  by a constraint. Delete from  $Y$  any value that is inconsistent with the value assigned for  $X$ .



# Problem with Forward Checking

- Forward Checking propagates information from assigned to unassigned variables. No propagation between unassigned variables.
- Only 1-step look ahead, does not examine all future implications of the current assignment.



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

NT and SA cannot be blue. This partial assignment could be extended. Still, we went ahead. No information propagated between two unassigned variables.

# Arc Consistency

*How to propagate information further?*

*First, we need a formal notion of arc consistency.*

- A directed arc  $X \rightarrow Y$  is “consistent” iff
  - for every value  $x$  of  $X$ , there exists a value  $y$  of  $Y$ , such that  $(x, y)$  satisfies the constraint between  $X$  and  $Y$
- Remove values from the domain of  $X$  to enforce arc-consistency

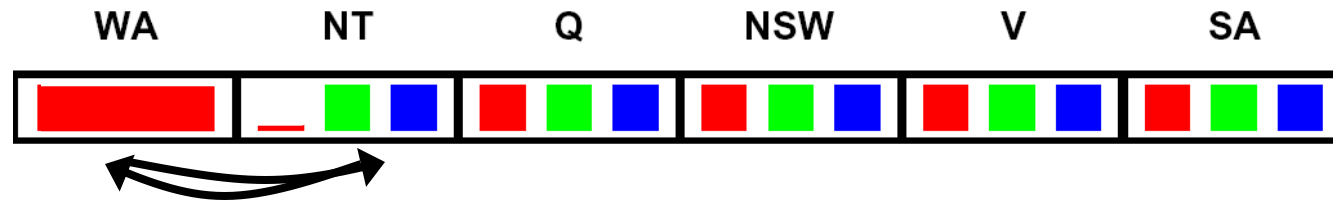
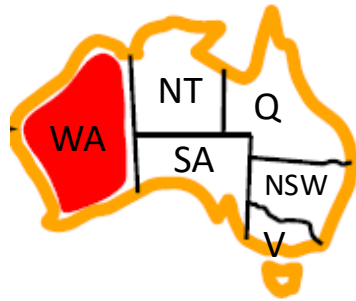
## Takeaway

- If the domain values at the head ( $Y$ ) change when assigned, then we need to check if the values in the tail ( $X$ ) are still consistent with the assignment to  $Y$ . If not, then remove the values for  $X$  that are inconsistent, thereby making the arc  $X \rightarrow Y$  “arc consistent”.



# Enforcing Consistency of a Single Arc

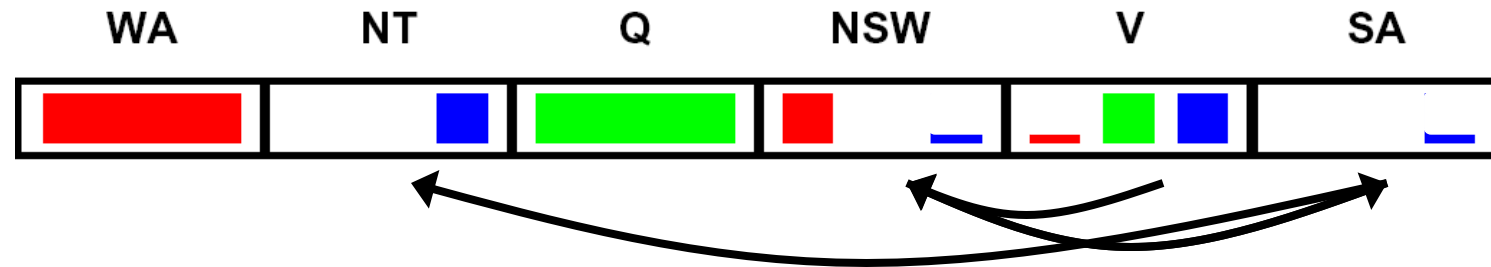
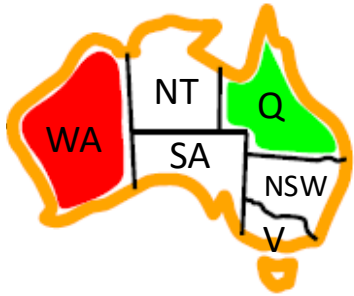
- An arc  $X \rightarrow Y$  is **consistent** iff for *every value* for  $x$  (the “tail” of the arc) there is *some* value  $y$  (the “head” of the arc) which could be assigned without violating a constraint.



**Remember: Always delete the domain value from the “tail” of the arc.**

# Enforcing Arc Consistency for the Entire CSP

- Ensure that **all** arcs in the constraint graph are consistent:



- Take away**

- If X loses a value in its domain, the neighbors of X (arcs coming in) need to be **re-examined** for consistency.
- Arc consistency detects failure **earlier** than forward checking.
- Forward checking was 1-step look ahead. Arc consistency further examines **implications**.
- If no values left in the domain of a variable, then do not continue and backtrack as the CSP does not have a solution

# AC-3: Enforcing Arc Consistency in a CSP

Mackworth, 1977

Maintain a queue of arcs

Obtain an arc

If the domain of  $X_i$  (tail) changes due to  $X_j$  (head) then, add all the edges coming into  $X_i$  from the  $X_k$ . (Done by inserting  $X_k, X_i$ )

Enforce arc consistency between  $X_i$  (tail)  $\rightarrow X_j$  (head). Checking if due to assignment for  $X_j$  is adjustment needed for  $X_i$ ?

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

**inputs:** *csp*, a binary CSP with components ( $X, D, C$ )

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

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

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

**if** REVISE(*csp*,  $X_i, X_j$ ) **then**

**if** size of  $D_i = 0$  **then return** false

**for each**  $X_k$  in  $X_i.\text{NEIGHBORS} - \{X_j\}$  **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*  $\leftarrow$  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*  $\leftarrow$  true

**return** *revised*


Runtime:  $O(n^2d^3)$  [ $n^2$  edges  $\times d^2$  time in consistency  $\times d$  arc insertions (only domain reduction triggers insertion of an edge in the queue)]

# Backtracking Search (with Inference)

- Run Forward Checking or Arc Consistency.
- Also called interleaving search and inference.
- Polynomial time
- We will still back track at times as AC-3 cannot detect all the inconsistencies (detecting all inconsistencies is NP-hard).

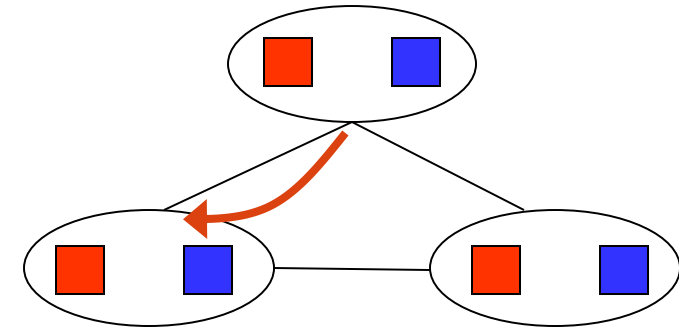
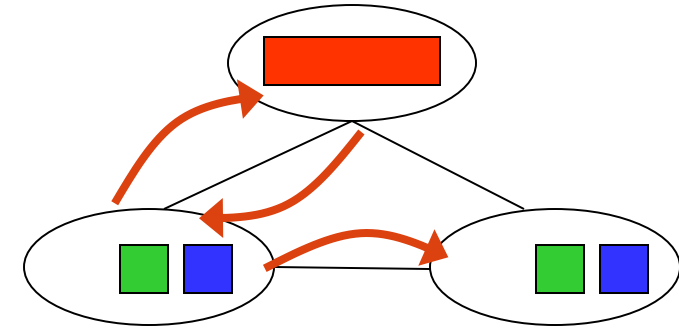
```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, value)
      if inferences ≠ failure then
        add inferences to assignment
        result ← BACKTRACK(assignment, csp)
        if result ≠ failure then
          return result
      remove {var = value} and inferences from assignment
  return failure
```



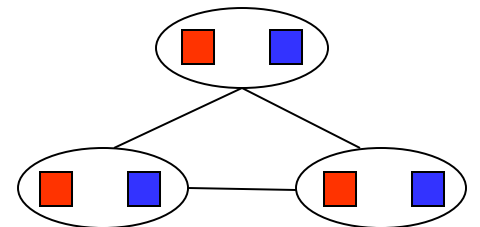
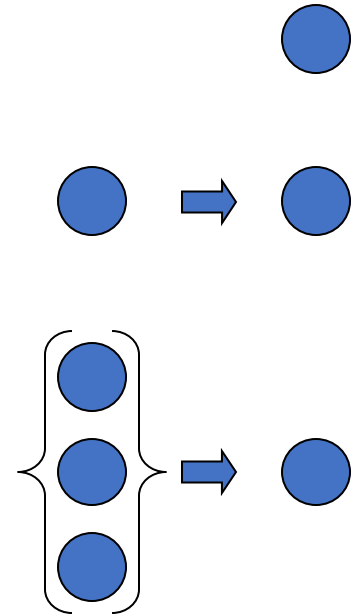
# Arc Consistency: Limitation

- After enforcing arc consistency:
  - Can have one solution left
  - Can have multiple solutions left
  - Can have no solutions left (and not know it)
- Example:
  - This CSP has a consistent assignment (see top figure).
  - Consider the second assignment, arc consistency is established but there is no overall solution in this case (see below figure).



# K-Consistency

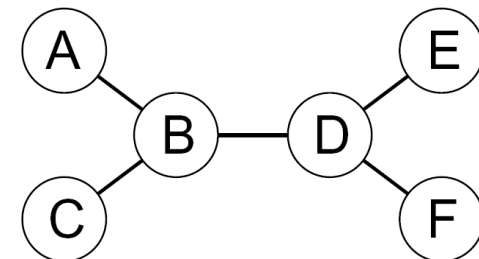
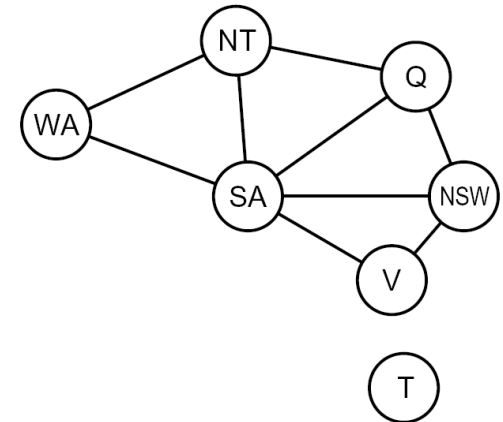
- Increasing degrees of consistency
  - 1-Consistency (Node Consistency): Each single node's domain has a value which meets that node's unary constraints
  - 2-Consistency (Arc Consistency): For each pair of nodes, any consistent assignment to one can be extended to the other
  - K-Consistency: For each k nodes, any consistent assignment to k-1 can be extended to the k<sup>th</sup> node.
  - Higher k more expensive to compute.
- k=2 case is arc consistency.
  - In our example, arc consistency was enforced (K=2) but path consistency (K=3) was not. The overall inconsistency could not be detected with K=2.



# Exploiting Problem Structure

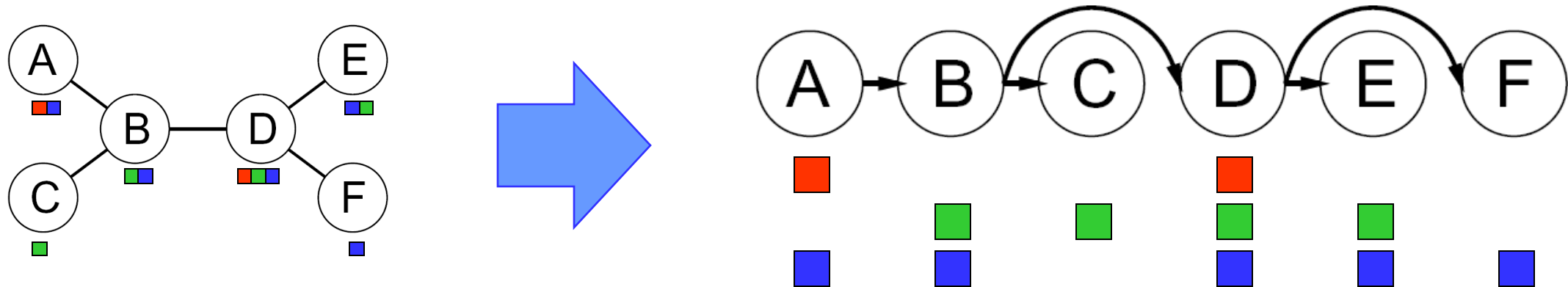
**General idea:** Some CSP structures are easy to solve. Either find and exploit that structure or perform reductions to simplify the problem.

- **Independent subproblems** are identifiable as connected components of constraint graph
  - Example: Tasmania and mainland do not interact
  - Decomposing a graph of  $n$  variables into subproblems of only  $c$  variables simplifies the problem.
- **Tree-structured CSPs**
  - Some CSP structures are easier to solve
  - Theorem: if the constraint graph has no loops, the CSP can be solved in  $O(n d^2)$  time
  - Compare to general CSPs, where worst-case time is  $O(d^n)$



# Tree-Structured CSPs

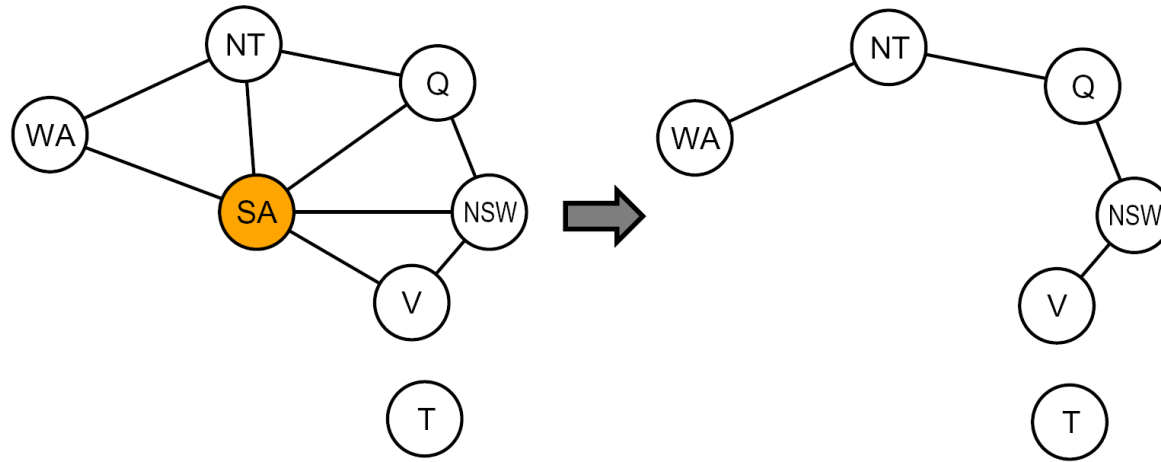
- Algorithm for tree-structured CSPs
  - Topological sort: Choose a root variable, order variables so that parents precede children



- Remove backward: For  $i = n : 2$ , apply Make-Arc-Consistent (Parent( $X_i$ ),  $X_i$ )
- Assign forward: For  $i = 1 : n$ , assign  $X_i$  consistently with Parent( $X_i$ ) by picking any consistent value from the domain.



# Improving Structure



- Overall idea
  - If we can handle the instantiation of certain variables, then the remaining problem can be simplified.
  - Conditioning: instantiate a variable, prune its neighbors' domains and solve the residual graph
  - The residual graph is easier to solve as it is tree structured.
- Cutset conditioning
  - Find a subset of variables  $S$ , such that the remaining constraint graph becomes a tree after the removal of  $S$  ( $S$  is a cycle cut set).
  - Instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree.
  - Remove from the domains of the remaining variables any values that are inconsistent with the assignment for  $S$

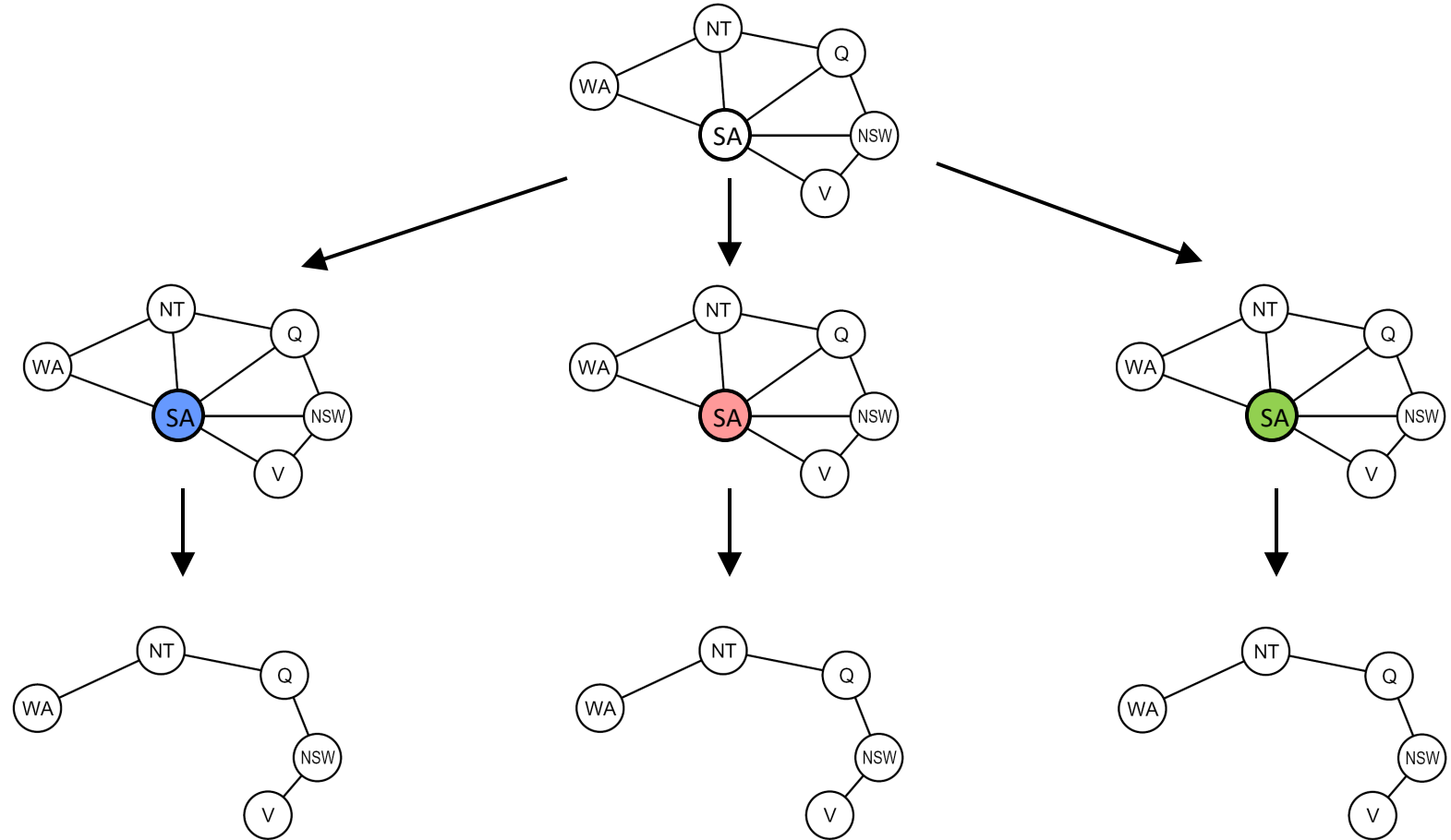
# Cutset Conditioning

Choose a cutset

Instantiate the cutset  
(all possible ways)

Compute residual CSP  
for each assignment

Solve the residual CSPs  
(tree structured)



Note: branching on the number of ways to instantiate the cut set variables.

Finding the “optimal” cutset is not easy (in general NP-hard). Easier if we know about the problem structure.

# Applications of CSPs

- Assignment problems
  - e.g., who teaches what class
- Timetable problems
  - e.g., which class is offered when and where?
- Scheduling problems
- VLSI or PCB layout problems
- Boolean satisfiability
- N-Queens
- Graph coloring
- Games: Minesweeper, Magic Squares, Sudoku, Crosswords
- Line-drawing labeling