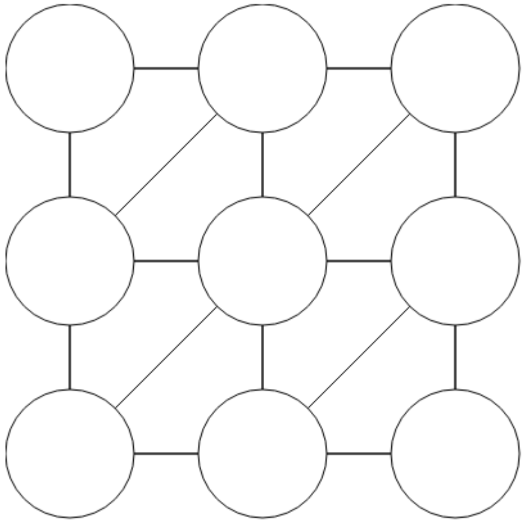
These are both Constraint Satisfaction Problems (CSP)

CSP Warm-up

Assign Red, Green, or Blue
Neighbors must be different



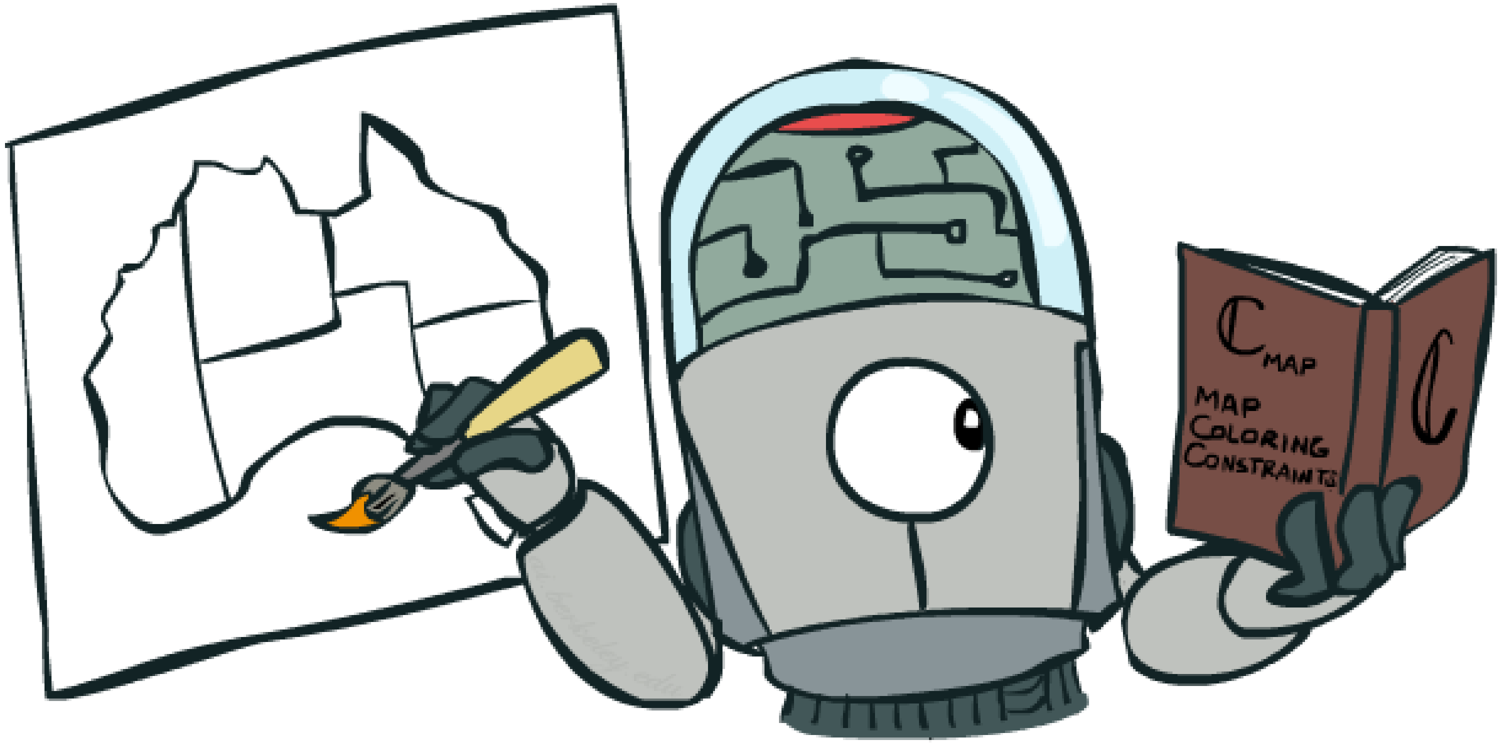
Sudoku

1			
	2	1	
		3	
			4

1) What is your brain doing to solve these?
2) How would you solve these with search (BFS, DFS, etc.)?

AI: Representation and Problem Solving

Constraint Satisfaction Problems (CSPs)



Instructors: Fei Fang & Pat Virtue

Slide credits: CMU AI, http://ai.berkeley.edu

Announcement

- Instructor for lectures
  - This week: Fei
  - Next week: Pat and Fei

- Reminder
  - HW2 (written) due 9/10 Tue, 10 pm
  - P1 due 9/12 Thu, 10 pm

Learning Objectives

- Describe definition of CSP problems and its connection with general search problems
- Formulate a real-world problem as a CSP
- Describe and implement backtracking algorithm
- Define arc consistency
- Describe and implement forward checking and AC-3
- Explain the differences between MRV and LCV heuristics
- Understand the complexity of general binary CSP and tree-structured binary CSP

Will cover in next lecture

What is Search For?

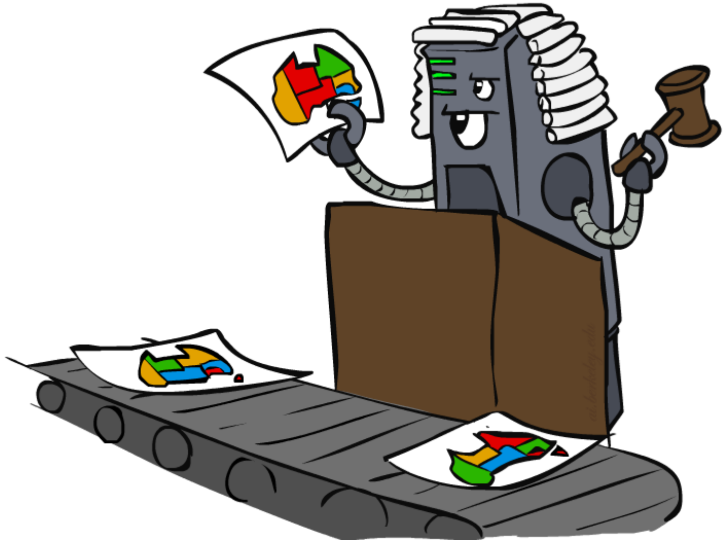
- Planning: sequences of actions
  - The path to the goal is the important thing
  - Paths have various costs, depths
  - Heuristics give problem-specific guidance

- Identification: assignments to variables
  - The goal itself is important, not the path
  - All paths at the same depth (for some formulations)

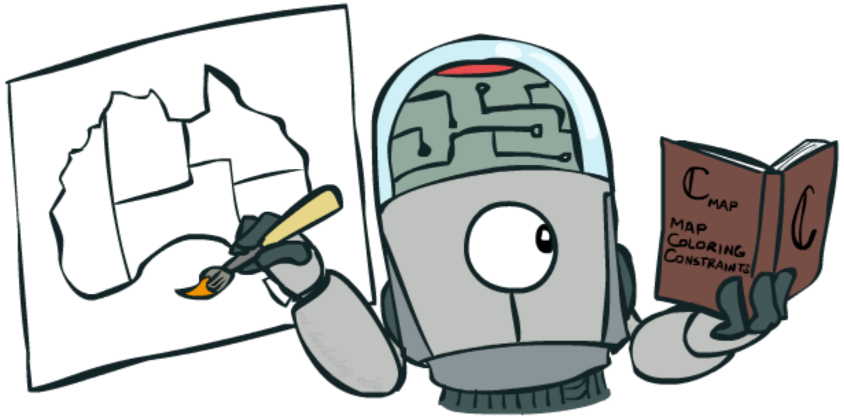
Are the warm-up assignments planning or identification problems?

Constraint Satisfaction Problems

- CSP is a special class of search problems
  - Mostly identification problems
  - Have specialized algorithms for them



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

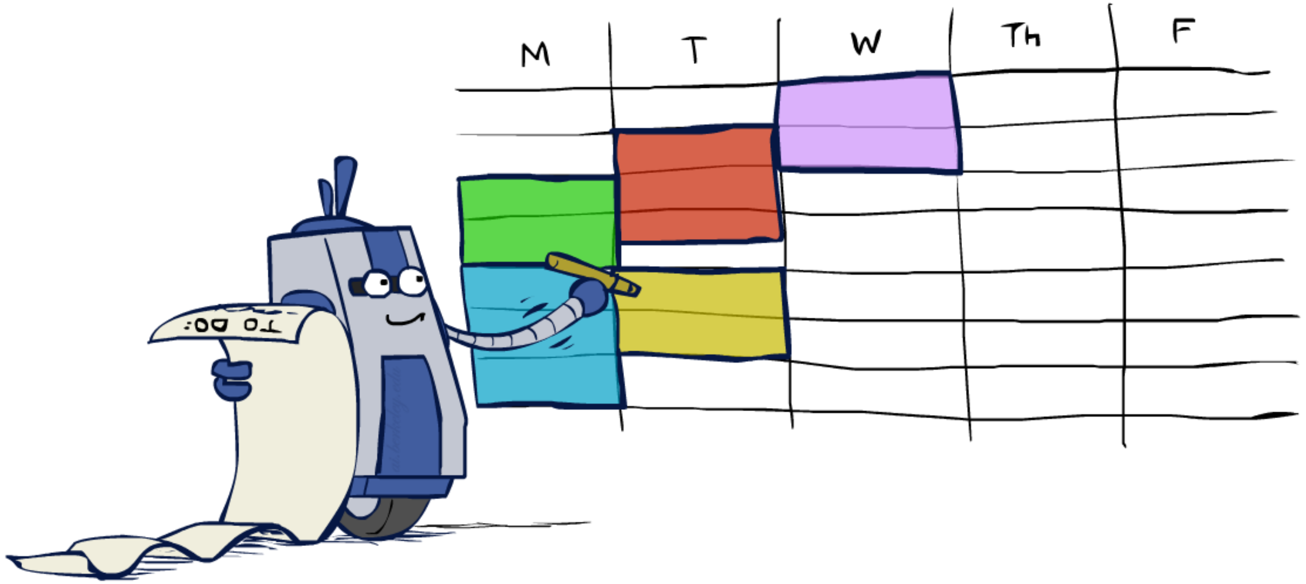


- Constraint satisfaction problems (CSPs):
  - State is defined by variables X_i with values from a domain D (sometimes D depends on i)
  - Goal test is a set of constraints specifying allowable combinations of values for subsets of variables

Why study CSPs?

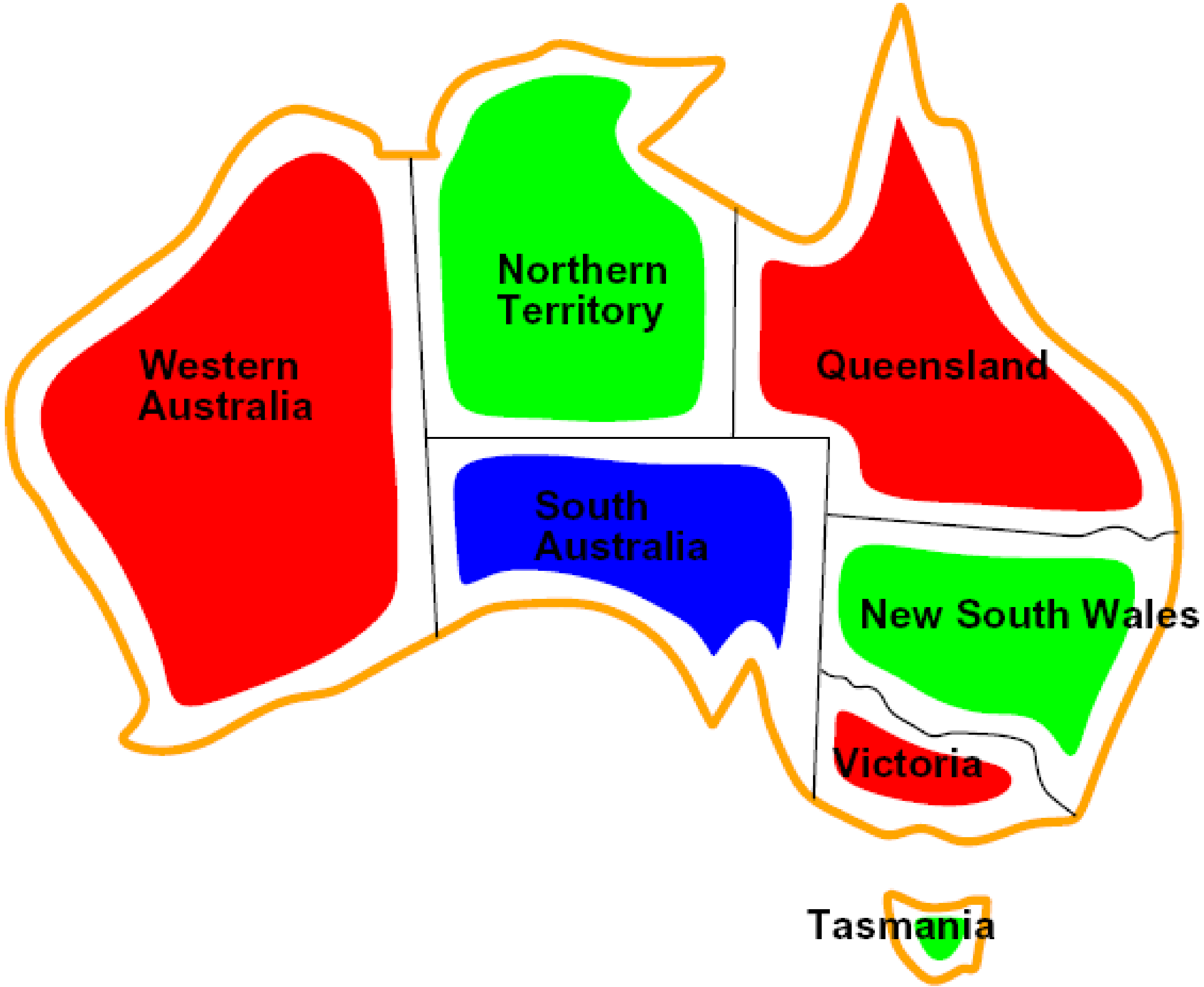
Many real-world problems can be formulated as CSPs

- 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
- Fault diagnosis
- … lots more!



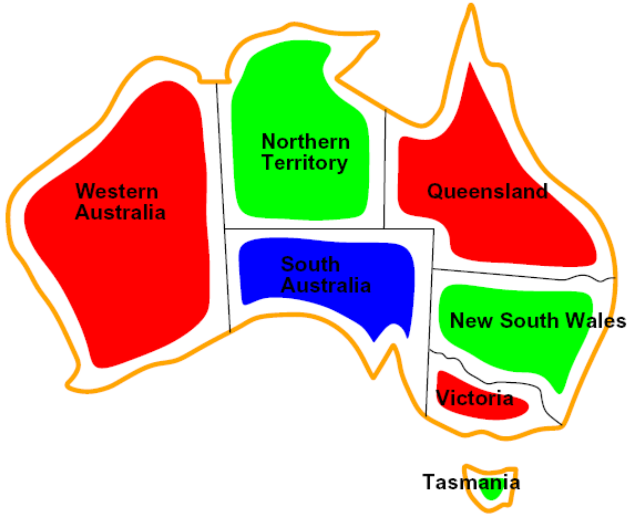
- Sometimes involve real-valued variables…

CSP Examples



Example: Map Coloring

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

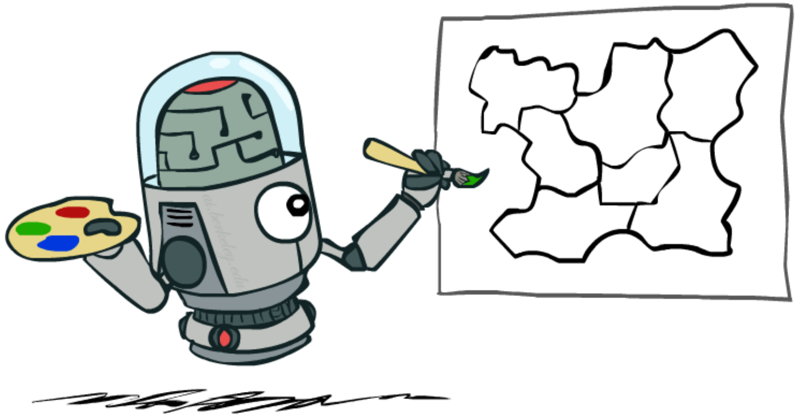


  Implicit: $WA \neq NT$

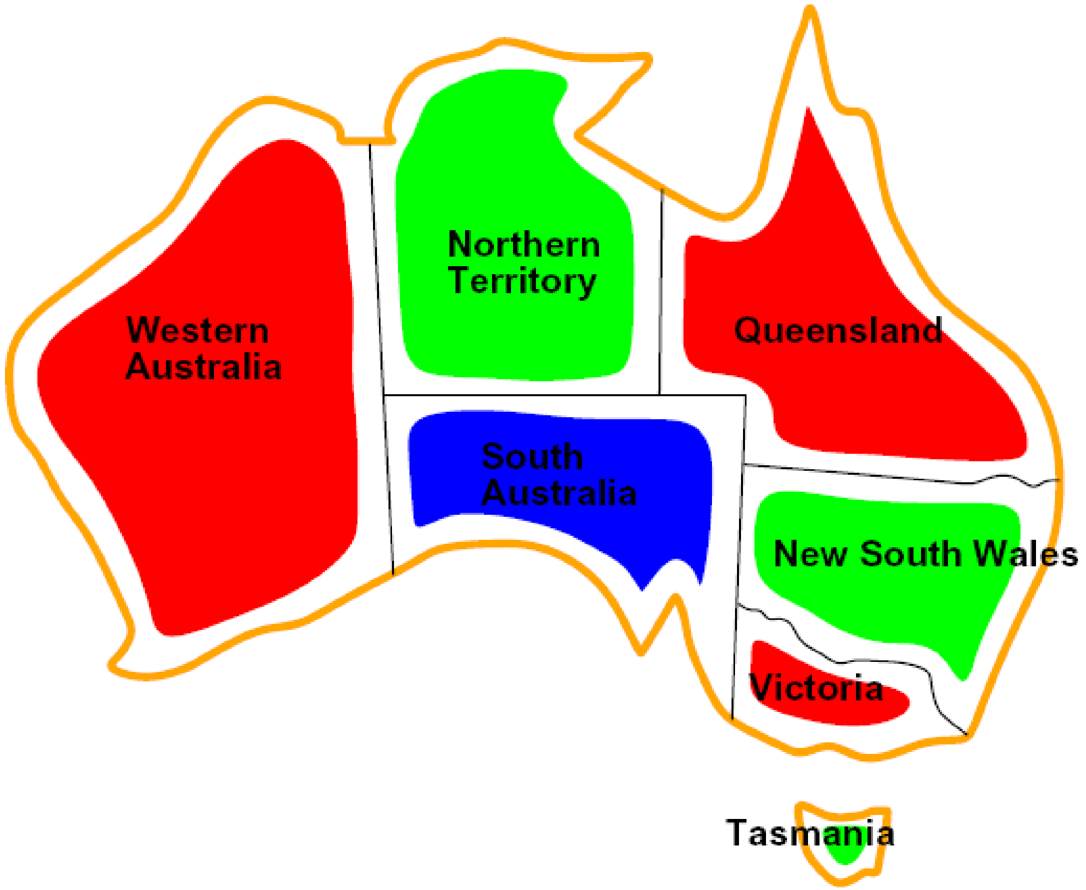
  Explicit: $(WA, NT) \in \{(red, green), (red, blue), \ldots\}$

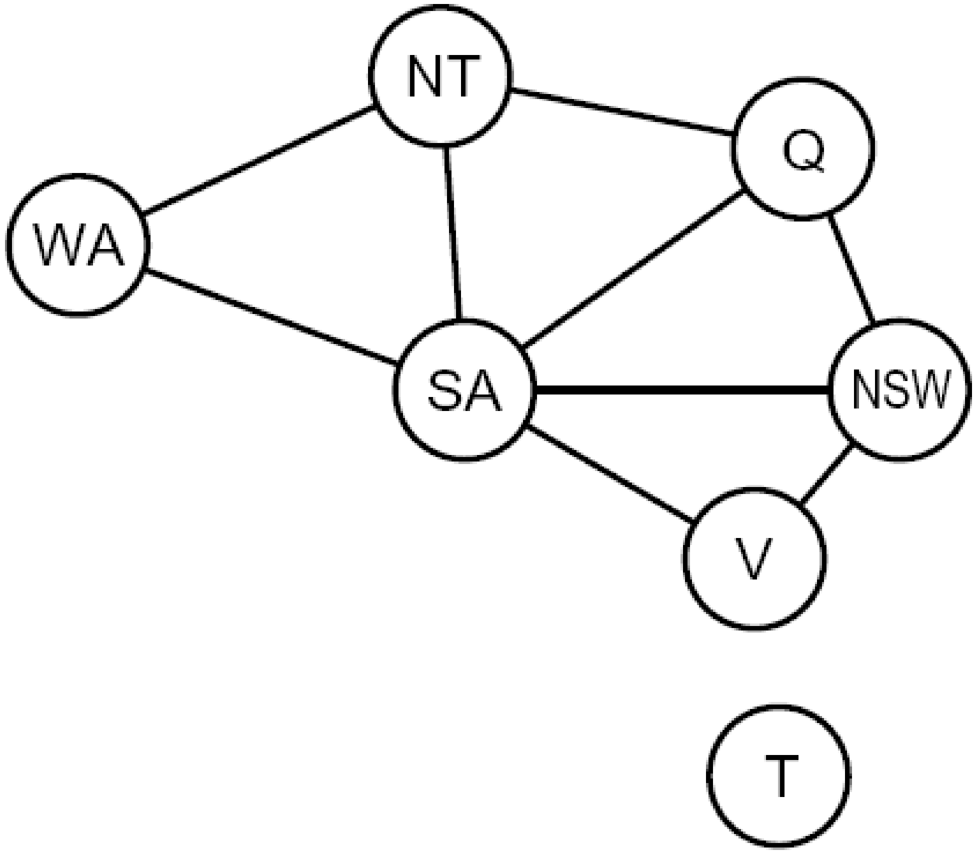
- Solutions are assignments satisfying all constraints, e.g.:

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



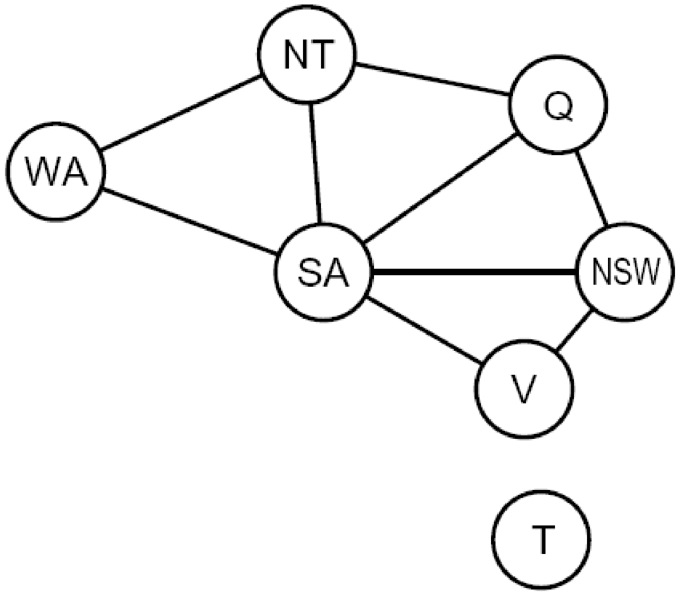
Constraint Graphs





Constraint Graphs

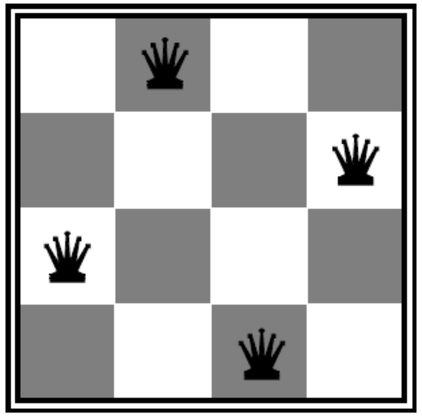
- Binary CSP: each constraint relates (at most) two variables
- Binary constraint graph: nodes are variables, arcs show constraints
- General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!

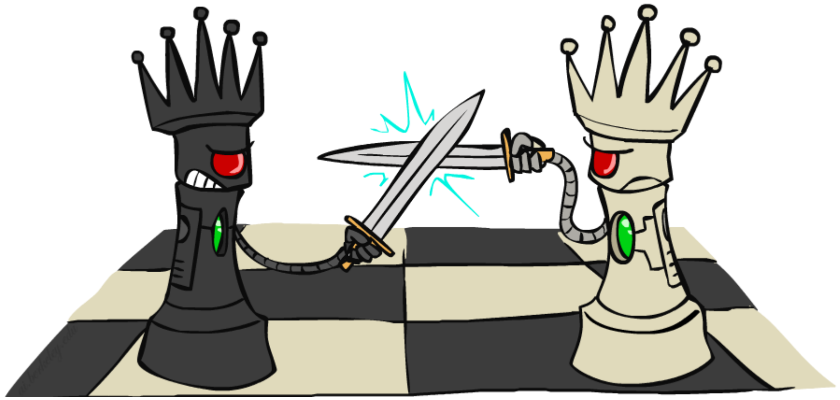


Varieties of CSPs and Constraints

Example: N-Queens

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



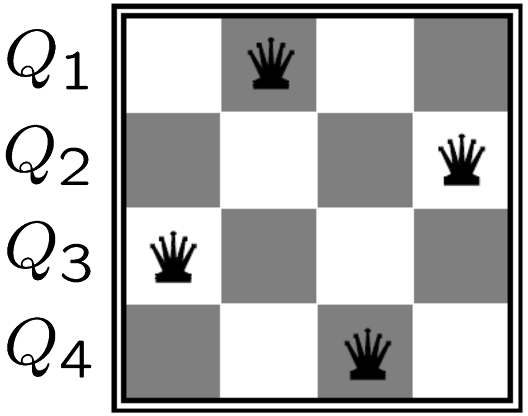


$$\forall i, j, k \quad (X_{ij}, X_{ik}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \quad (X_{ij}, X_{kj}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \quad (X_{ij}, X_{i+k,j+k}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \quad (X_{ij}, X_{i+k,j-k}) \in \{(0,0), (0,1), (1,0)\}$$

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

Example: N-Queens

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



Implicit: $\forall i, j \;\; \text{non-threatening}(Q_i, Q_j)$

Explicit: $(Q_1, Q_2) \in \{(1,3), (1,4), \ldots\}$

$\cdots$

Example: Cryptarithmetic

- Variables:

  $F \; T \; U \; W \; R \; O \; X_1 \; X_2 \; X_3$

- Domains:

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

- Constraints:

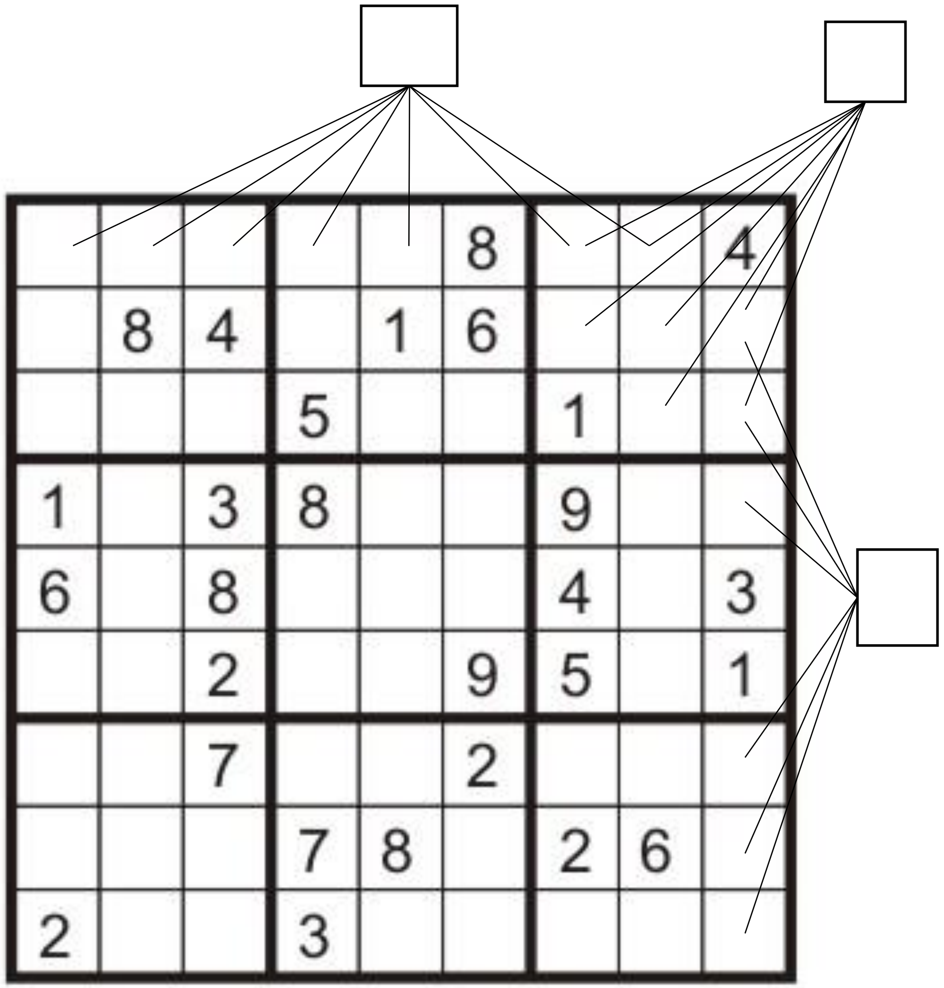
  $\text{alldiff}(F, T, U, W, R, O)$

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

  $\ldots$

    T W O
\+ T W O
F O U R

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

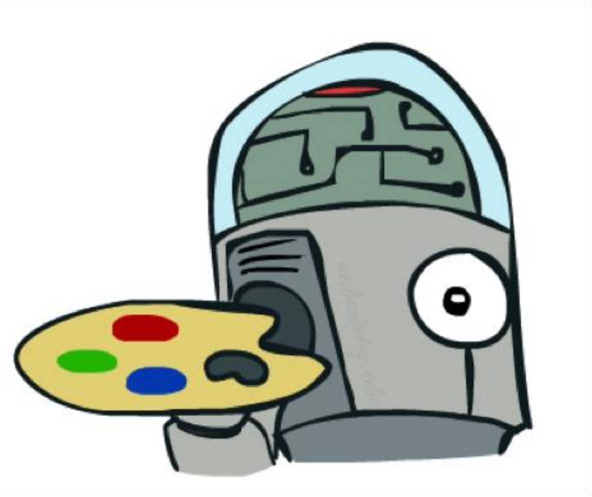
  (or can have a bunch
  of pairwise inequality
  constraints)

Varieties of CSPs

- Discrete Variables

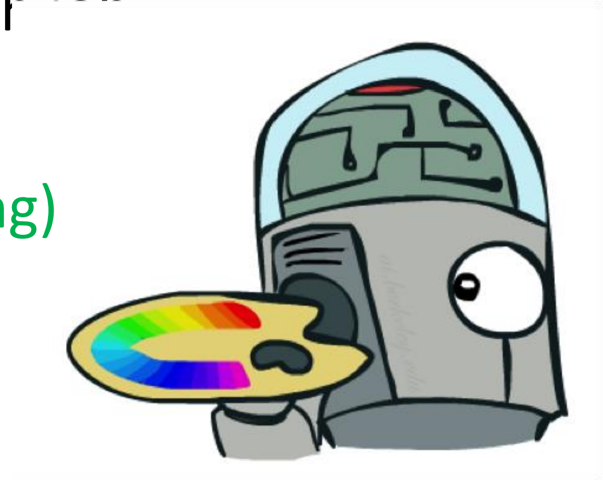
We will cover today

  - Finite domains
    - Size d means $O(d^n)$ complete assignments
    - E.g., Boolean CSPs, including Boolean satisfiability (NP-complete)
  - Infinite domains (integers, strings, etc.)
    - E.g., job scheduling, variables are start/end times for each
    - Linear constraints solvable, nonlinear undecidable



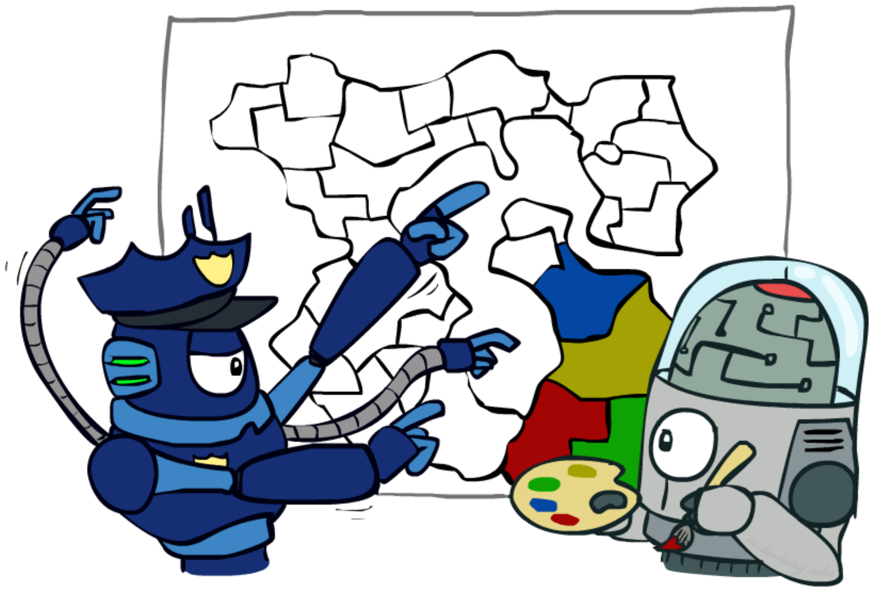
We will cover in later lecture (linear programming)

- Continuous variables
  - E.g., start/end times for Hubble Telescope observations
  - Linear constraints solvable in polynomial time



Varieties of Constraints

- Varieties of Constraints
  - Unary constraints involve a single variable (equivalent to reducing domains), e.g.:
    $$\text{SA} \neq \text{green}$$
    Focus of today
  - Binary constraints involve pairs of variables, e.g.:
    $$\text{SA} \neq \text{WA}$$
  - Higher-order constraints involve 3 or more variables:
    e.g., cryptarithmetic column constraints
    $$O + O = R + 10 \cdot X_1$$



- Preferences (soft constraints):
  - E.g., red is better than green
  - Often representable by a cost for each variable assignment
  - Gives constrained optimization problems

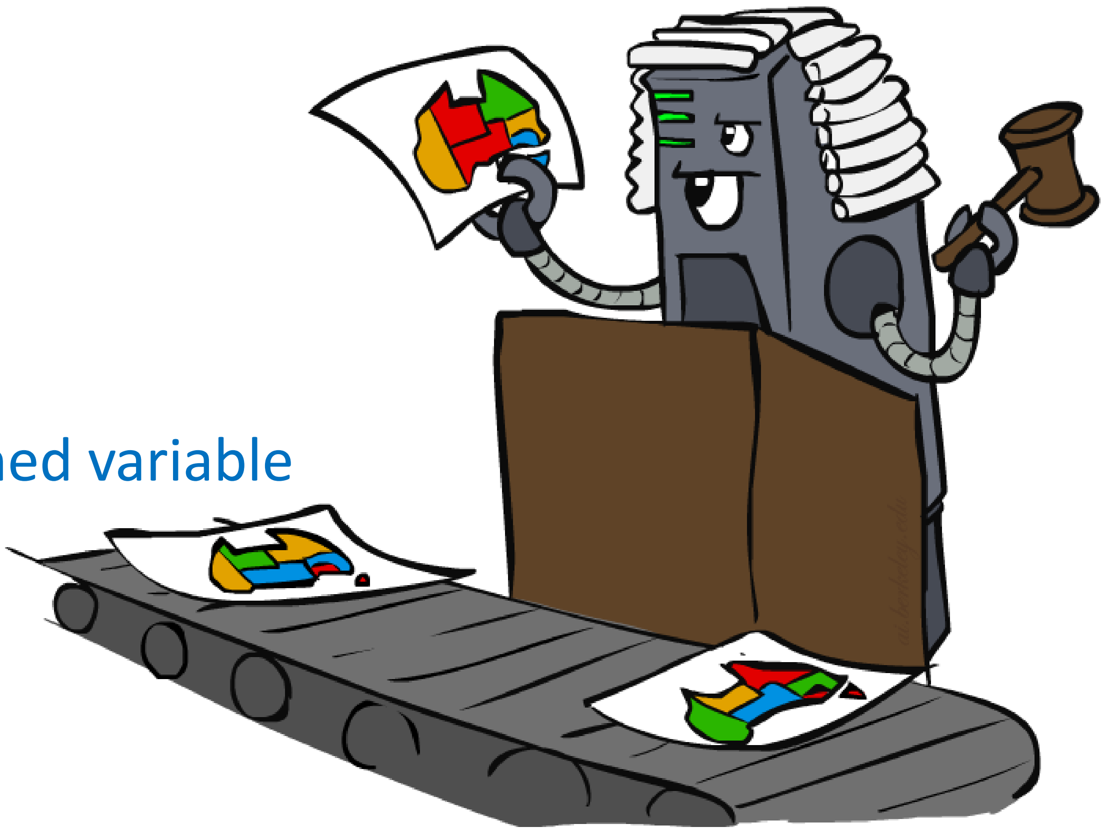
Solving CSPs

Standard Search Formulation

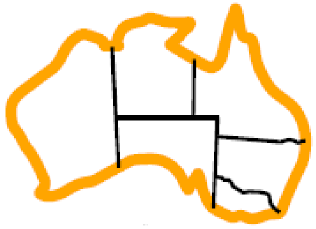
- Standard search formulation of CSPs

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

- We'll start with the straightforward, naïve approach, then improve it

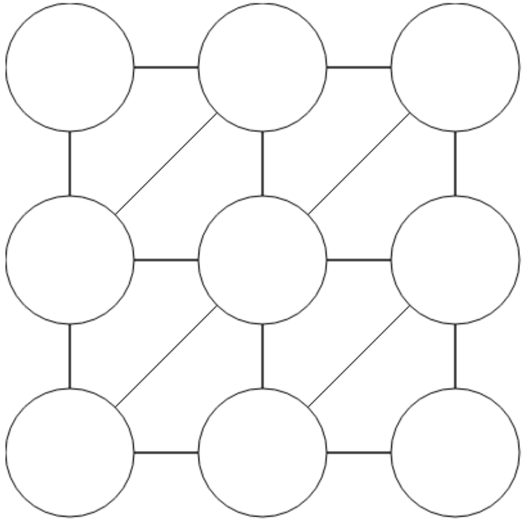


Depth First Search



- At each node, assign a value from the domain to the variable
- Check feasibility (constraints) when the assignment is complete

Demo – Naïve Search

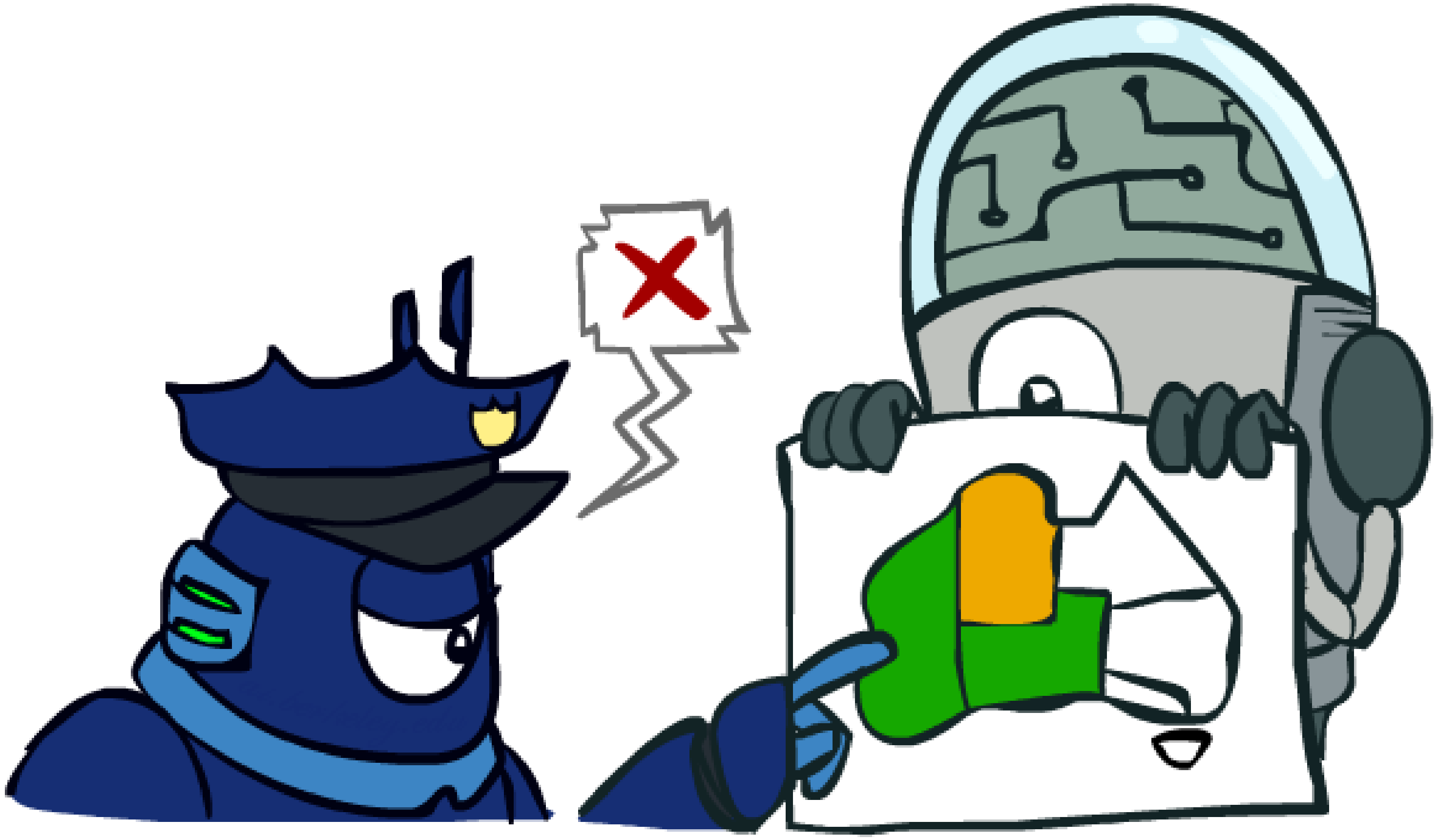


Keep these questions in mind:

Q1. How is the naïve search process in the demo different from the DFS process we just described?

Q2. Why the naïve search is “naïve”? How to make it more efficient?

Backtracking Search



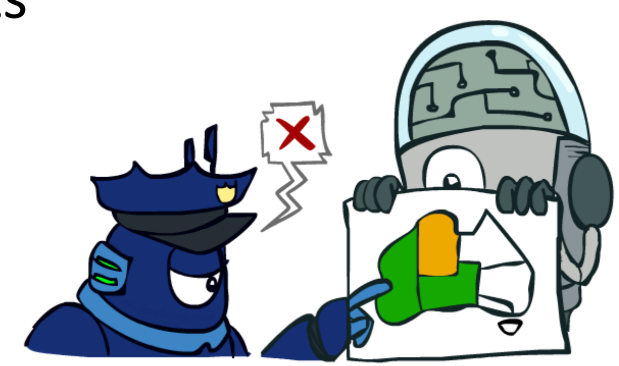
Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
- Backtracking search = DFS + two improvements

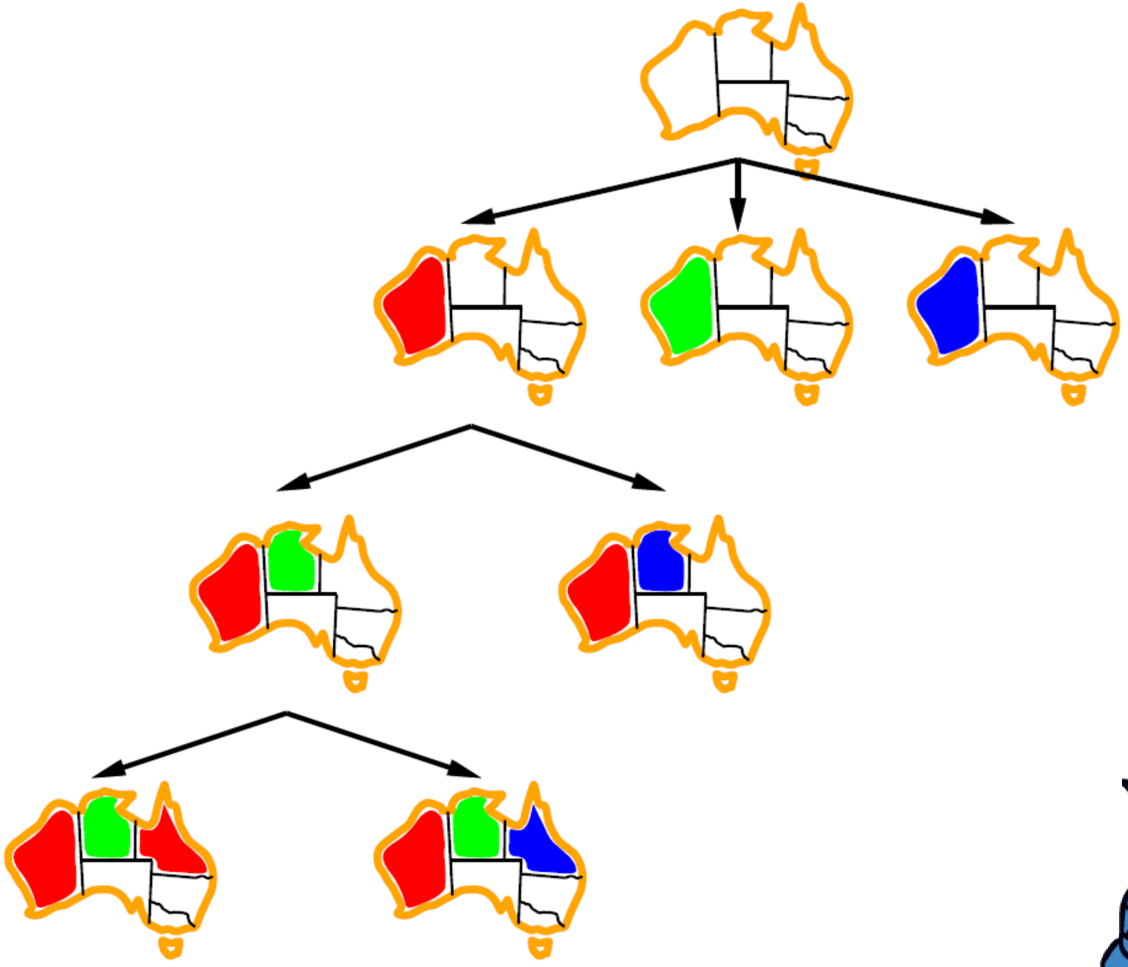
- Idea 1: One variable at a time
  - Variable assignments are commutative
    - [WA = red then NT = green] same as [NT = green then WA = red]
  - Only need to consider assign value to a single variable at each step

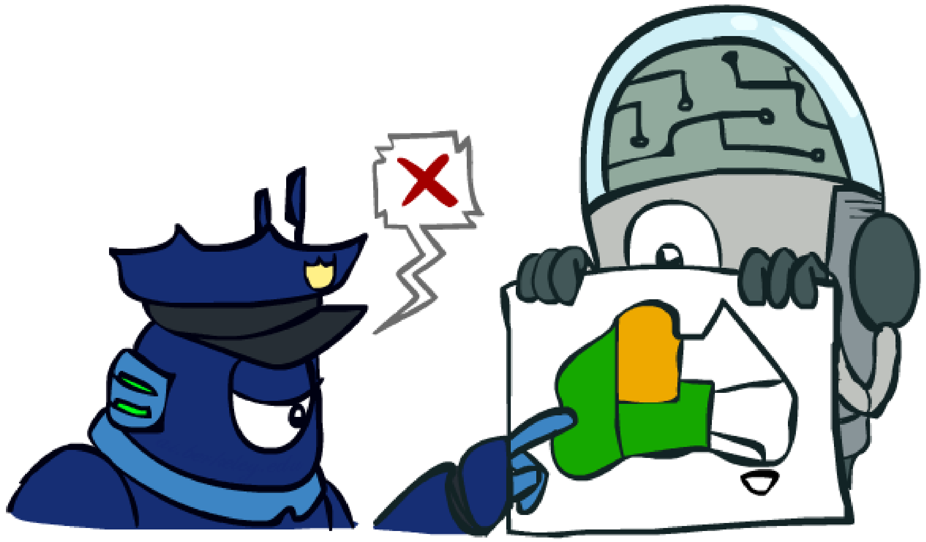
- Idea 2: Check constraints as you go
  - Consider only values which do not conflict previous assignments
  - May need some computation to check the constraints
  - "Incremental goal test"

- Can solve n-queens for $n \approx 25$



Backtracking Example





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
```

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
```

No need to check consistency for a complete assignment

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
```

Checks consistency at each assignment

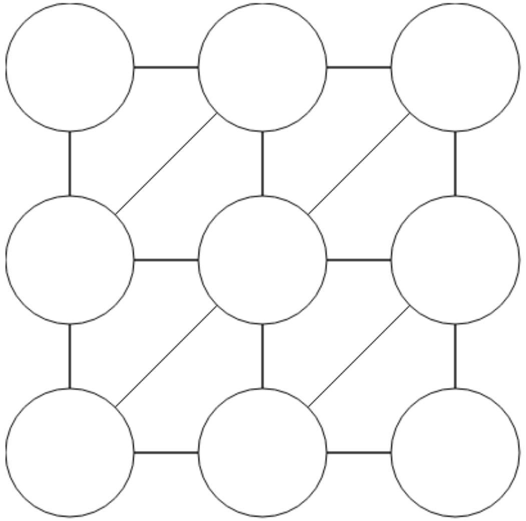
Backtracking Search

```
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
```

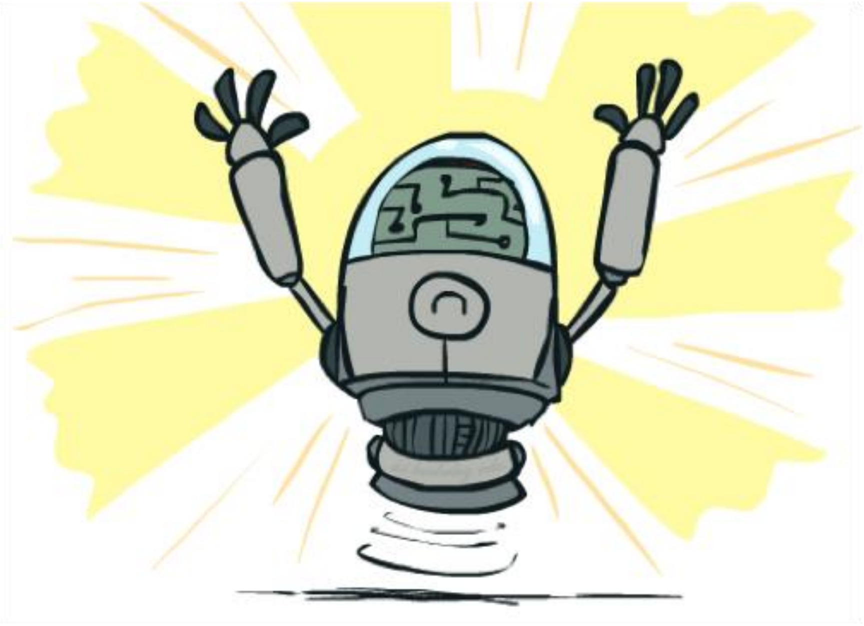
- Backtracking = DFS + variable-ordering + fail-on-violation
- What are the choice points?

Demo – Backtracking



Improving Backtracking

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



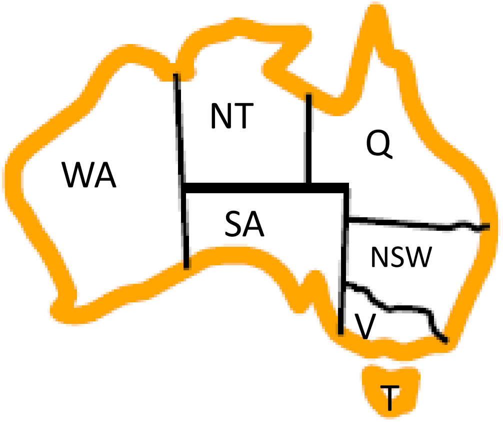
Filtering

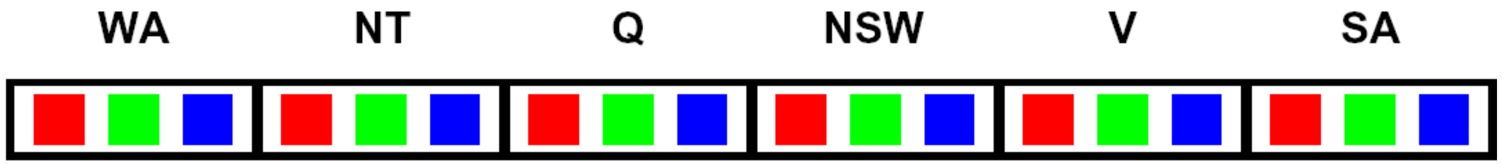
Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: A simple way for filtering
  - After a variable is assigned a value, check related constraints and cross off values of unassigned variables which violate the constraints
  - Failure detected if some variables have no values remaining

Filtering: Forward Checking

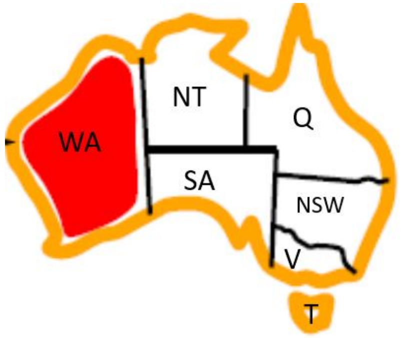
- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: A simple way for filtering
  - After a variable is assigned a value, check related constraints and cross off values of unassigned variables which violate the constraints
  - Failure detected if some variables have no values remaining

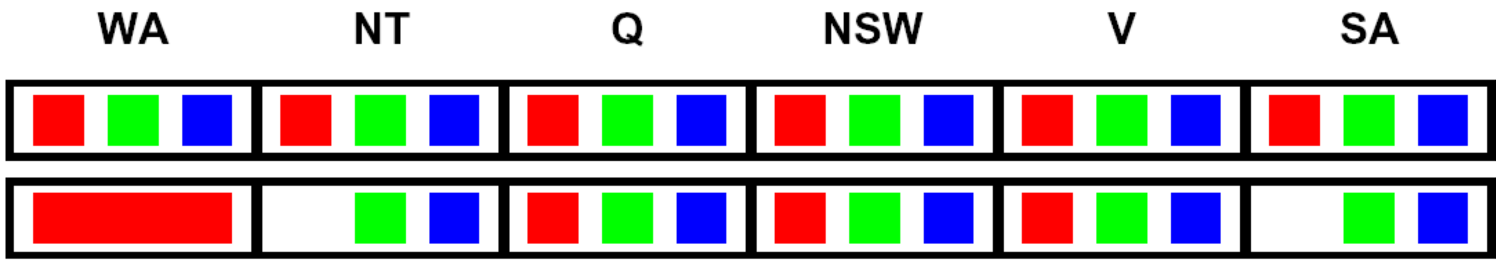


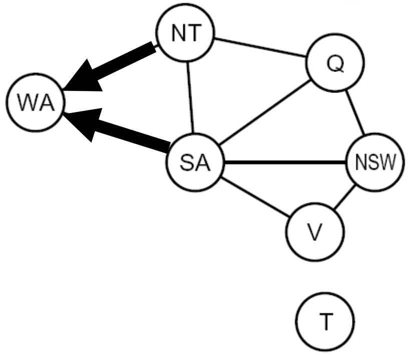


Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: A simple way for filtering
  - After a variable is assigned a value, check related constraints and cross off values of unassigned variables which violate the constraints
  - Failure detected if some variables have no values remaining



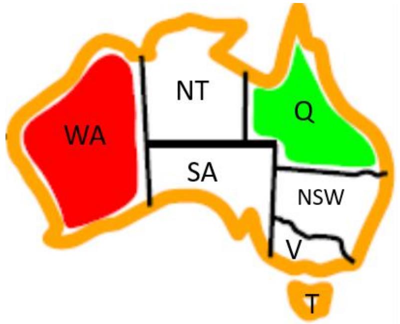


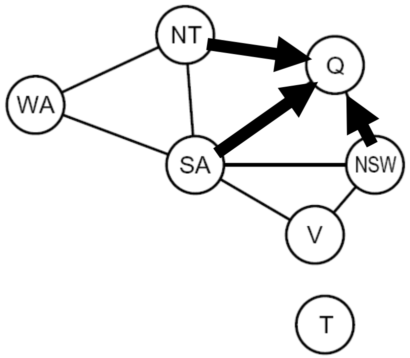


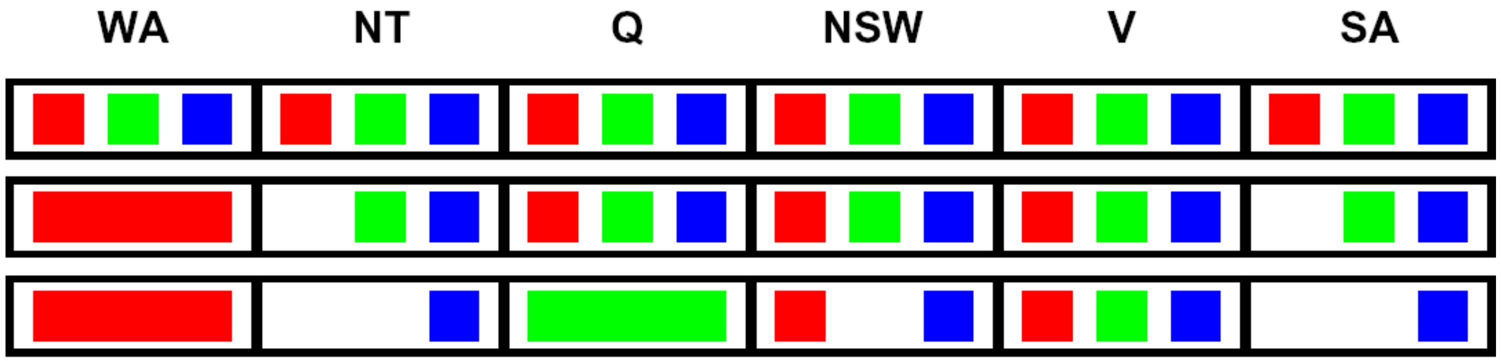
Recall: Binary constraint graph for a binary CSP (i.e., each constraint has most two variables): nodes are variables, edges show constraints

Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: A simple way for filtering
  - After a variable is assigned a value, check related constraints and cross off values of unassigned variables which violate the constraints
  - Failure detected if some variables have no values remaining

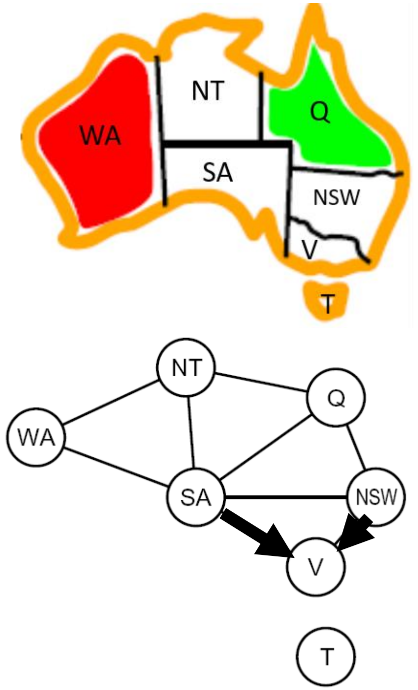


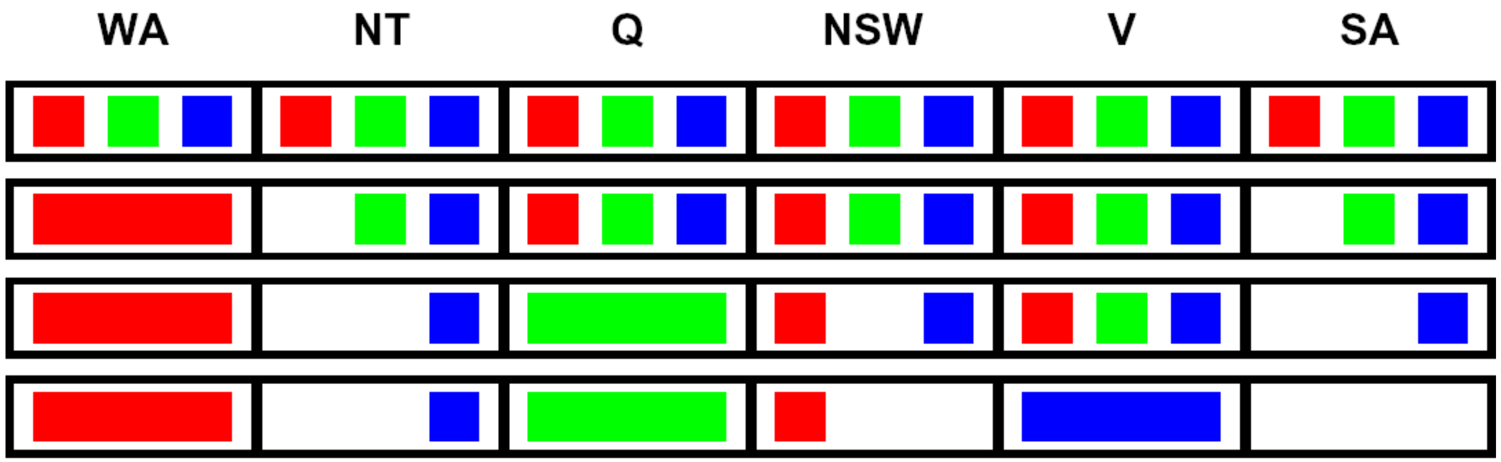




Filtering: Forward Checking

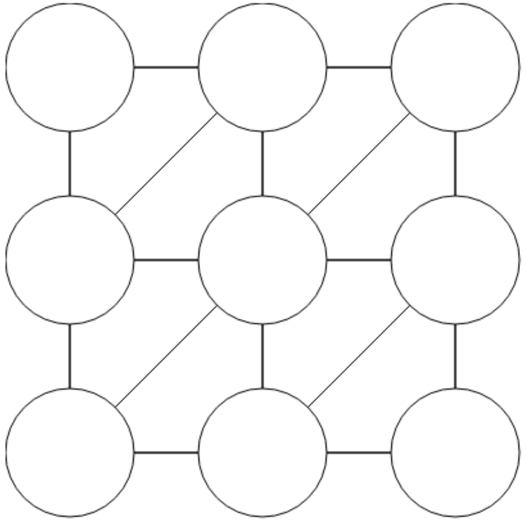
- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: A simple way for filtering
  - After a variable is assigned a value, check related constraints and cross off values of unassigned variables which violate the constraints
  - Failure detected if some variables have no values remaining





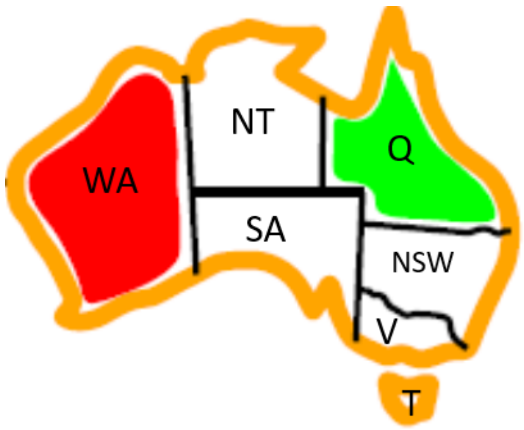
FAIL – variable with no possible values

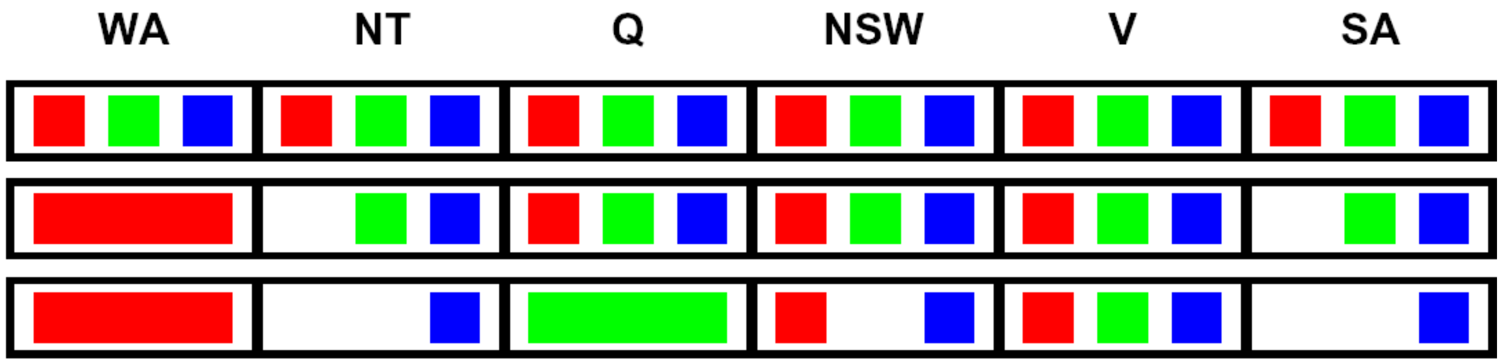
Demo – Backtracking with Forward Checking



Filtering: Constraint Propagation

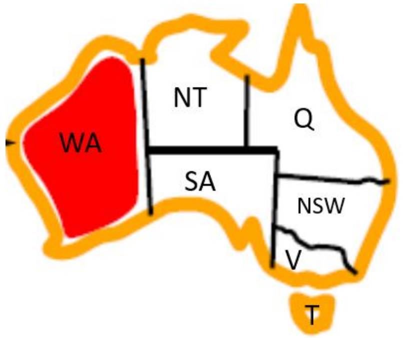
- Limitations of simple forward checking: propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures
  - NT and SA cannot both be blue! Why didn't we detect this yet?
- *Constraint propagation:* reason from constraint to constraint



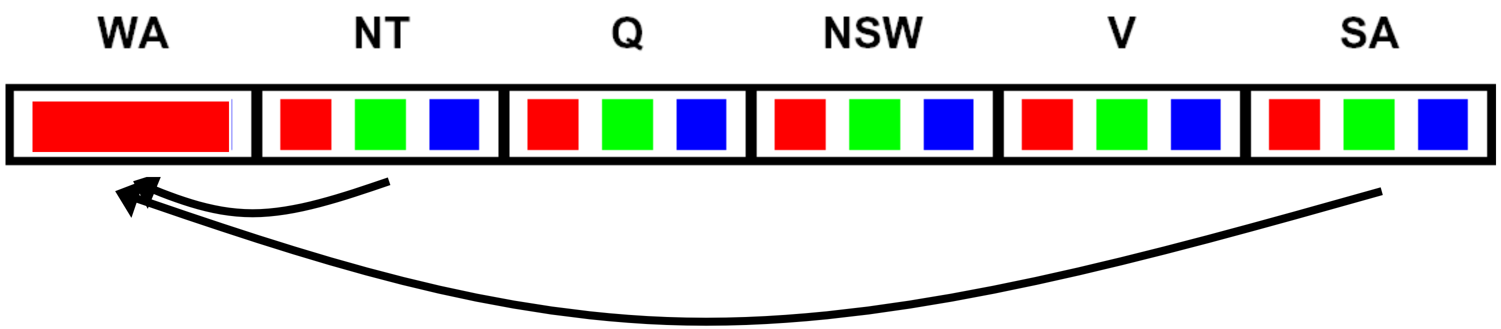


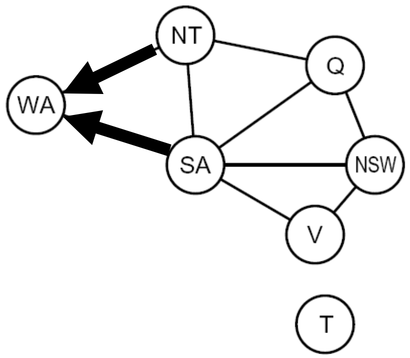
Consistency of A Single Arc

- An arc X → Y is consistent iff for every x in the tail there is some y in the head which could be assigned without violating a constraint
- Enforce arc consistency: Remove values in domain of X if no corresponding legal Y exists
- Forward checking: Only enforce $X \rightarrow Y, \forall (X,Y) \in E$ and Y newly assigned



(Remove values from the tail!)

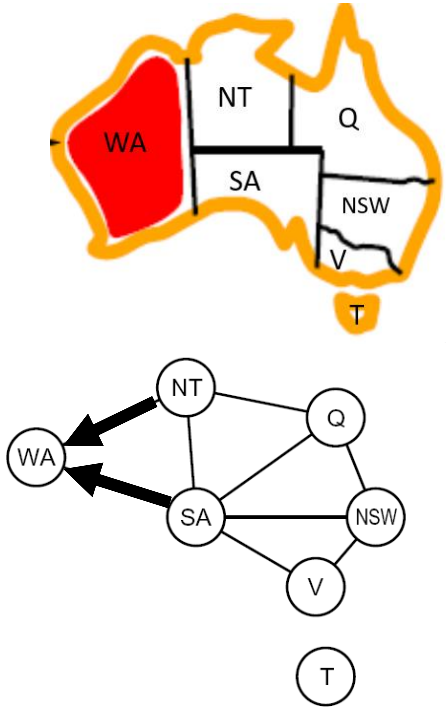


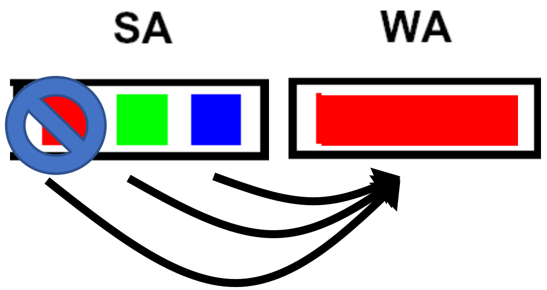


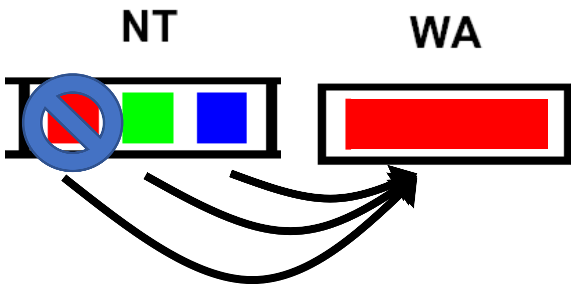
Recall: Binary constraint graph for a binary CSP (i.e., each constraint has most two variables): nodes are variables, edges show constraints

Consistency of A Single Arc

- An arc X $\rightarrow$ Y is consistent iff for every x in the tail there is some y in the head which could be assigned without violating a constraint
- Enforce arc consistency: Remove values in domain of X if no corresponding legal Y exists
- Forward checking: Only enforce $X \rightarrow Y, \forall (X, Y) \in E$ and Y newly assigned

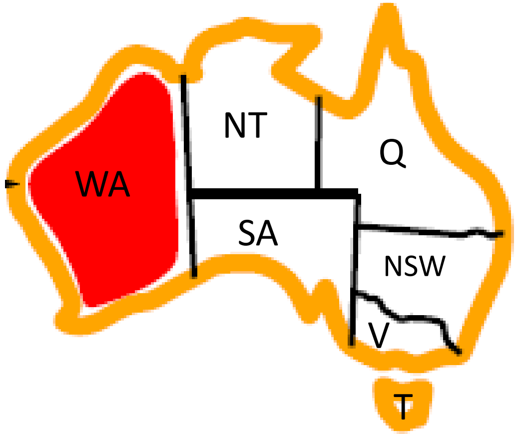


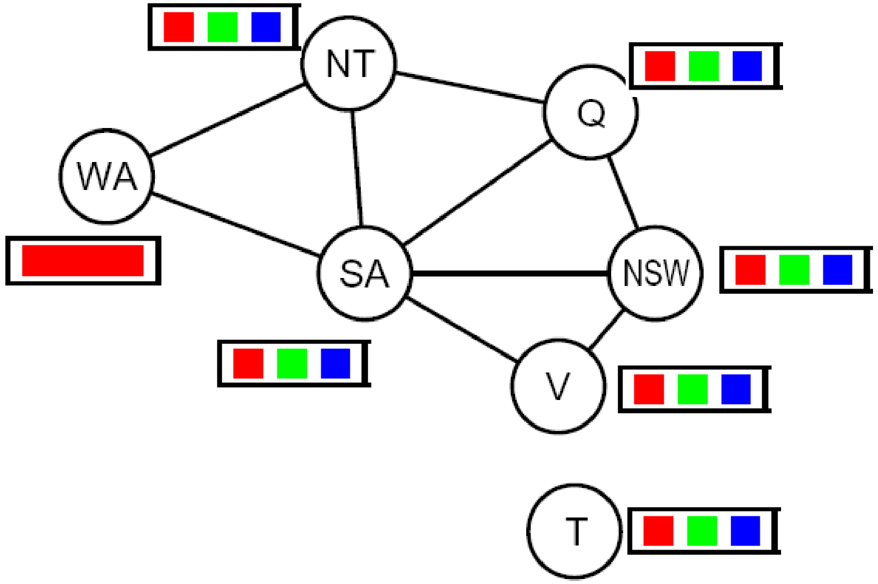




How to Enforce Arc Consistency of Entire CSP

- A simplistic algorithm: Cycle over the pairs of variables, enforcing arc-consistency, repeating the cycle until no domains change for a whole cycle
- AC-3 (short for Arc Consistency Algorithm #3): A more efficient algorithm ignoring constraints that have not been modified since they were last analyzed





AC-3: Enforce Arc Consistency of Entire CSP

function AC-3(*csp*) **returns** the CSP, possibly with reduced domains
  inputs: *csp*, a binary CSP with variables $\{X_1, X_2, \ldots, X_n\}$
  local variables: *queue*, a queue of arcs, initially all the arcs in *csp*

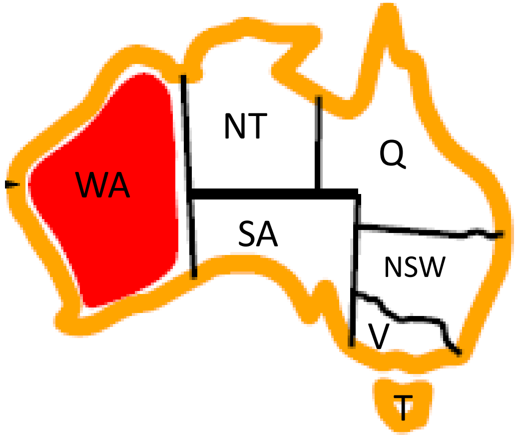
  while *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST(*queue*)
    if REMOVE-INCONSISTENT-VALUES(X_i, X_j) **then**
      for each X_k **in** NEIGHBORS[X_i] **do**
        add (X_k, X_i) to *queue*

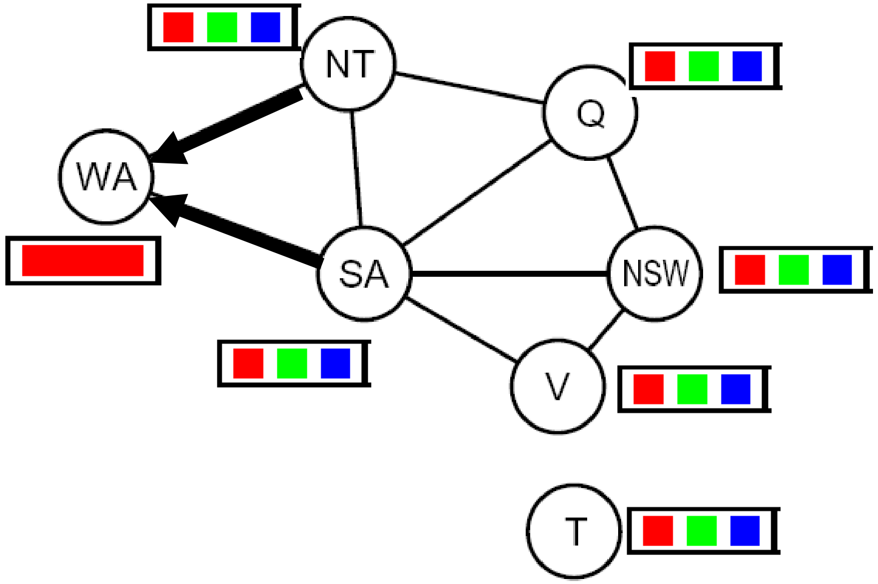
---

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) **returns** true iff succeeds
  removed $\leftarrow$ *false*
  for each x **in** DOMAIN[X_i] **do**
    if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint $X_i \leftrightarrow X_j$
      then delete x from DOMAIN[X_i]; *removed* $\leftarrow$ *true*
  return *removed*

Constraint Propagation!

AC-3: Enforce Arc Consistency of Entire CSP

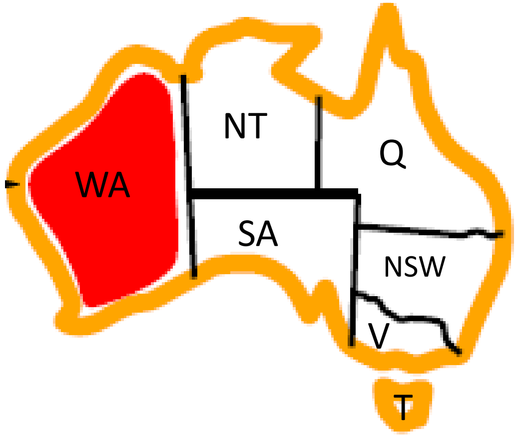


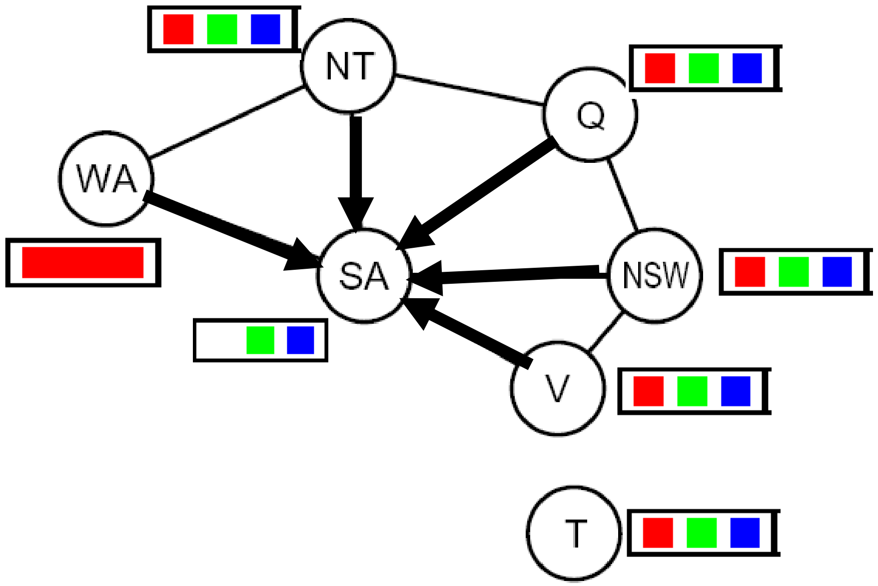


Queue:
SA->WA
NT->WA

Remember: Delete from the tail!

AC-3: Enforce Arc Consistency of Entire CSP

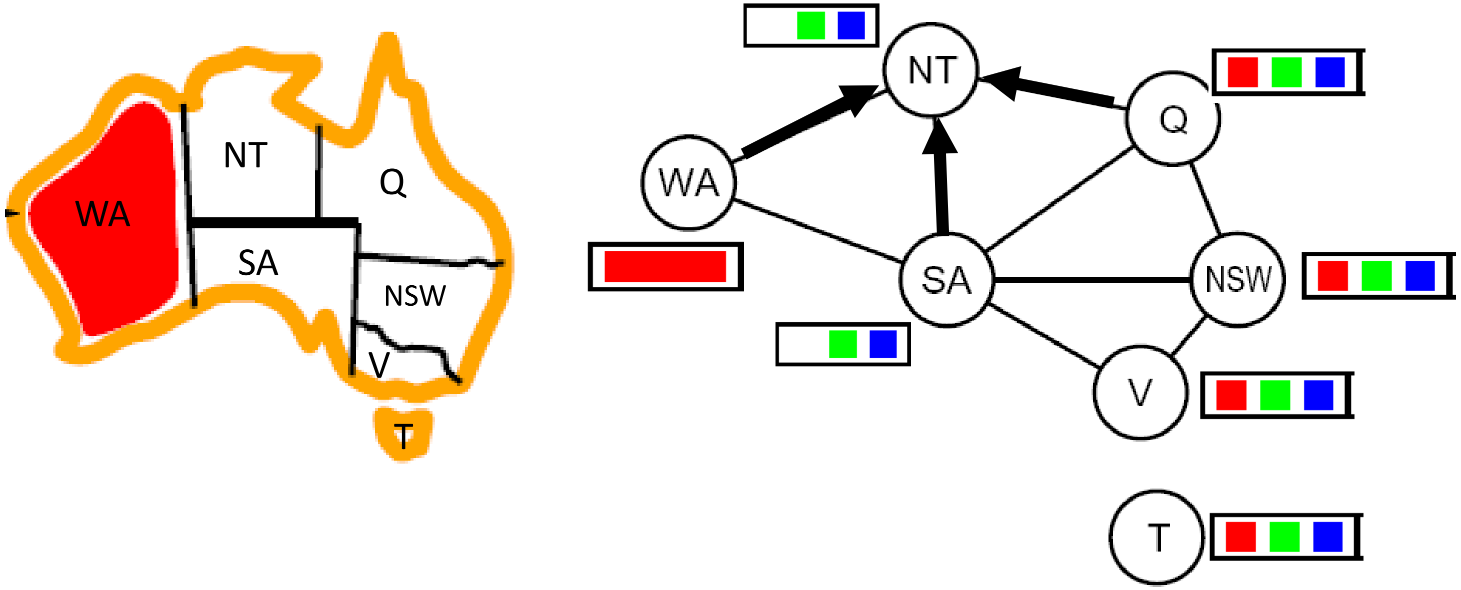




Queue:
NT->WA
WA->SA
NT->SA
Q->SA
NSW->SA
V->SA

Remember: Delete from the tail!

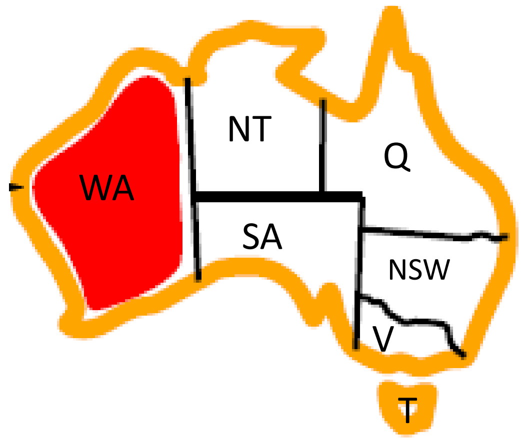
AC-3: Enforce Arc Consistency of Entire CSP

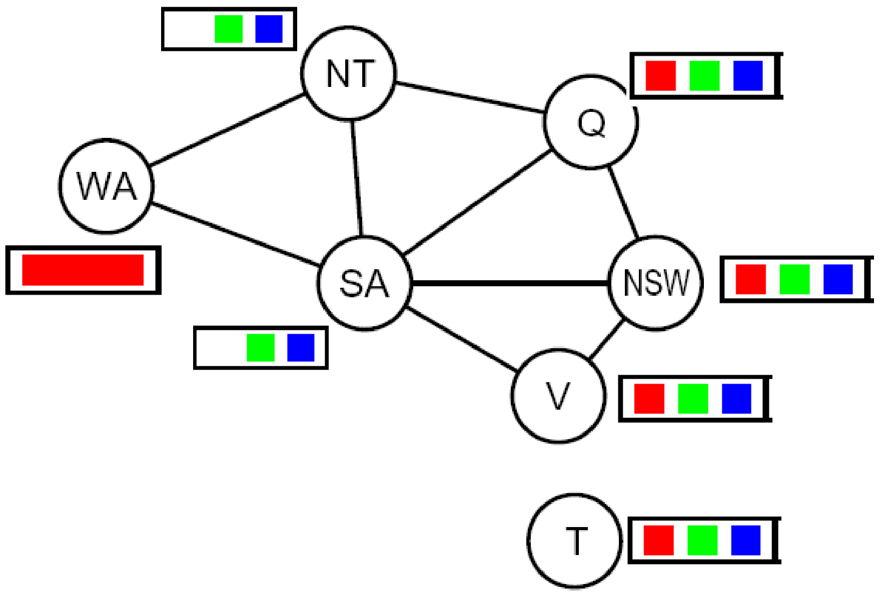


Queue:
WA->SA
NT->SA
Q->SA
NSW->SA
V->SA
WA->NT
SA->NT
Q->NT

Remember: Delete from the tail!

AC-3: Enforce Arc Consistency of Entire CSP

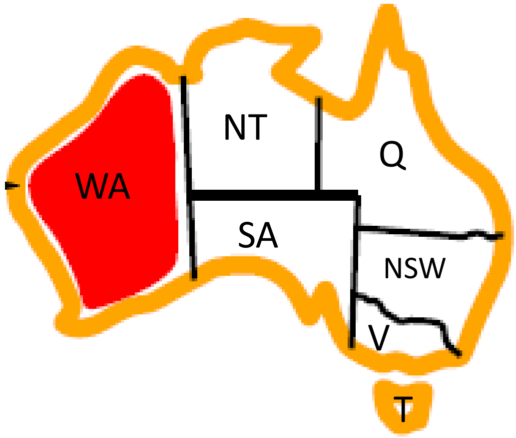


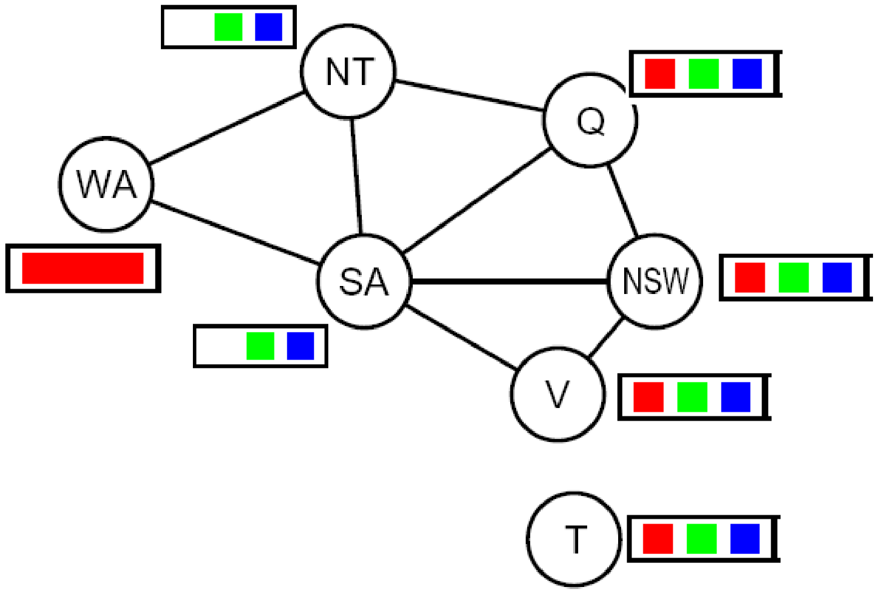


Queue:
WA->SA
NT->SA
Q->SA
NSW->SA
V->SA
WA->NT
SA->NT
Q->NT

Remember: Delete from the tail!

AC-3: Enforce Arc Consistency of Entire CSP

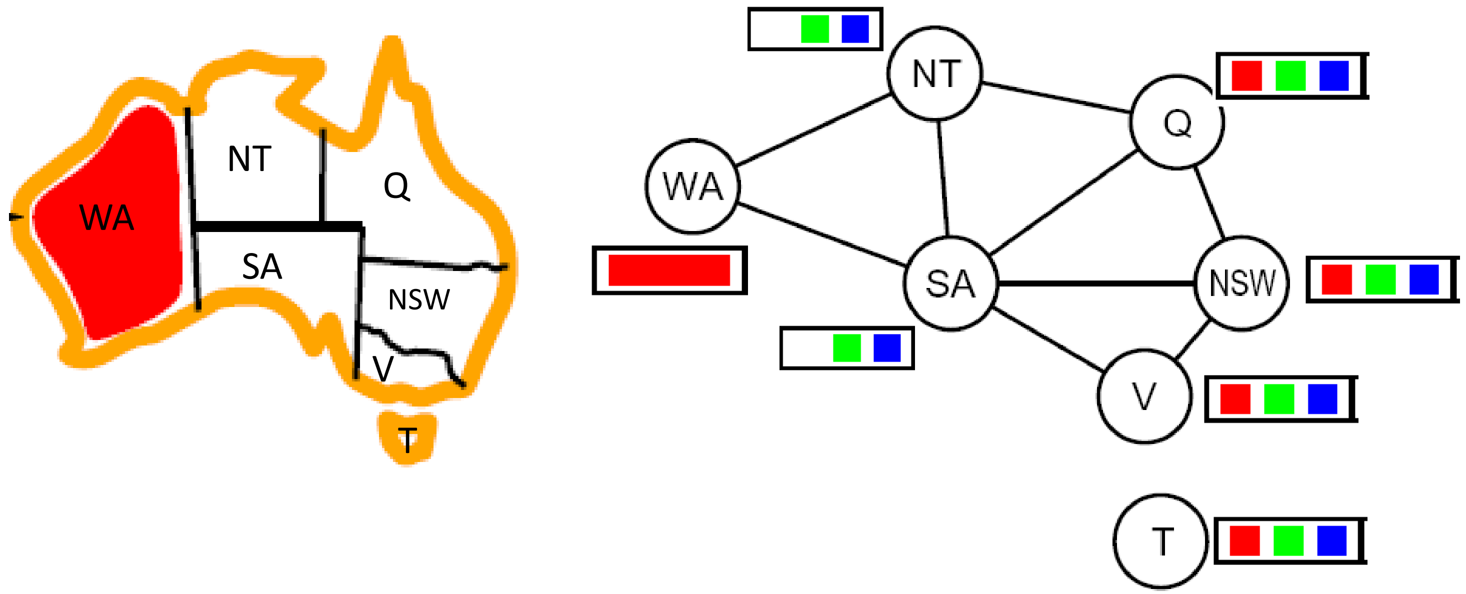




Queue:
NT->SA
Q->SA
NSW->SA
V->SA
WA->NT
SA->NT
Q->NT

Remember: Delete from the tail!

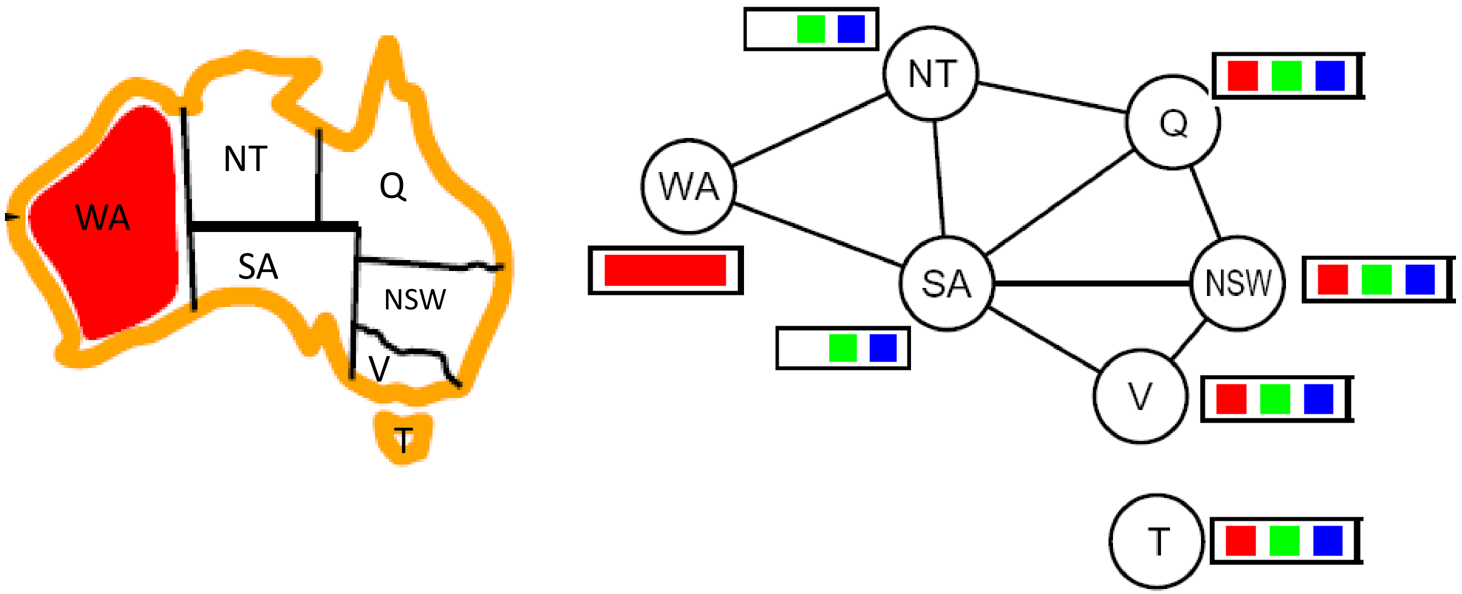
AC-3: Enforce Arc Consistency of Entire CSP



Queue:
Q->SA
NSW->SA
V->SA
WA->NT
SA->NT
Q->NT

Remember: Delete from the tail!

AC-3: Enforce Arc Consistency of Entire CSP



Queue:

NSW->SA

V->SA

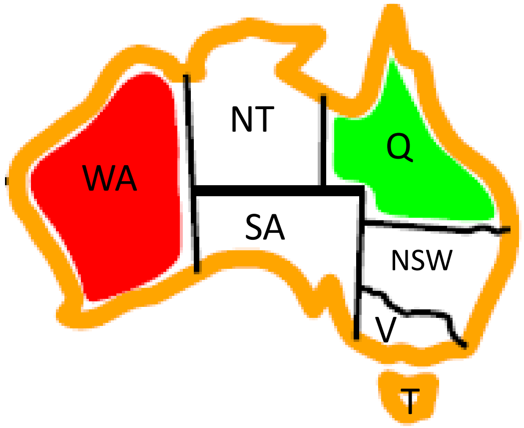
WA->NT

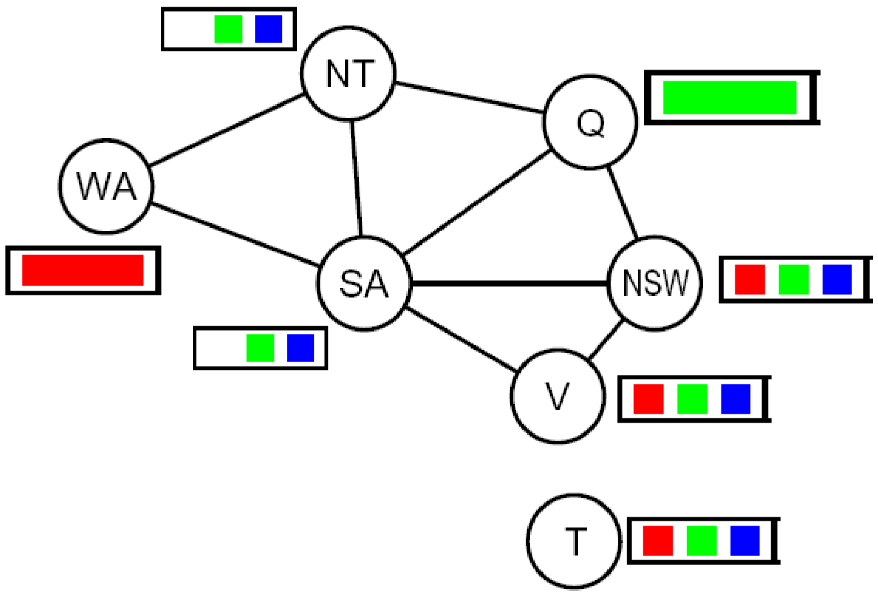
SA->NT

Q->NT

Remember: Delete from the tail!

AC-3: Enforce Arc Consistency of Entire CSP

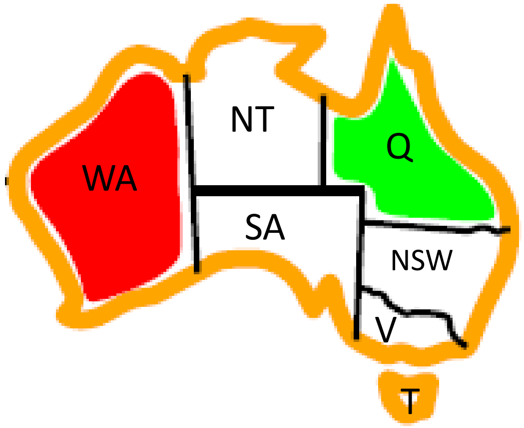


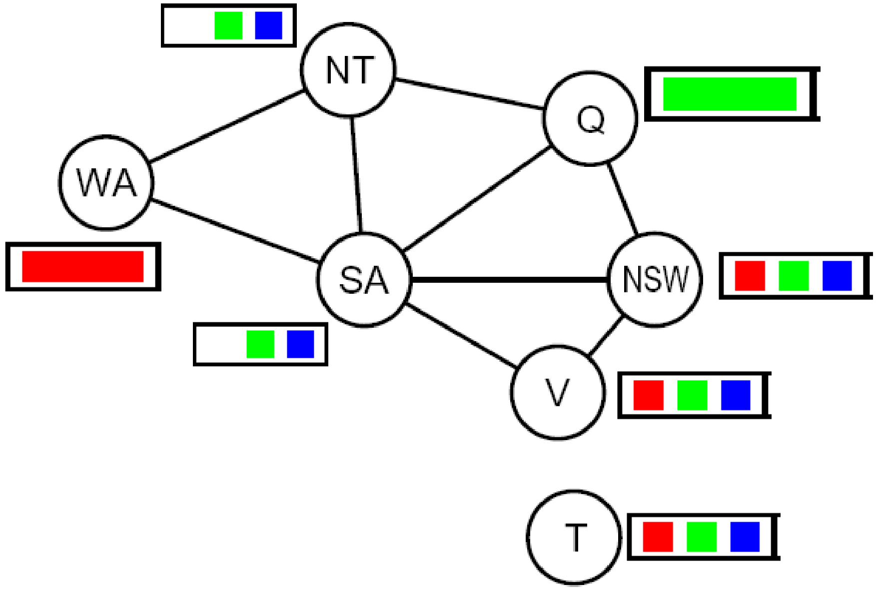


Queue:

Remember: Delete from the tail!

Piazza POLL: What gets added to the Queue?



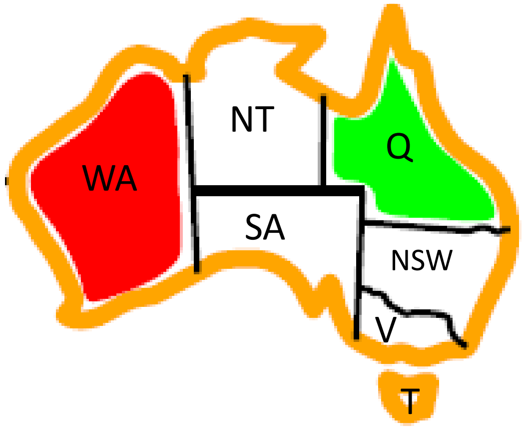


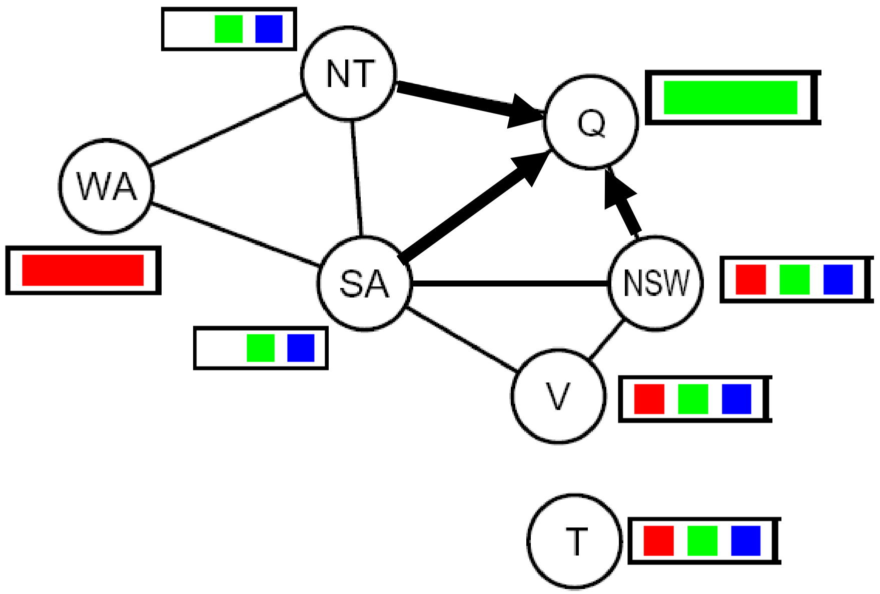
Queue:

A: NSW->Q, SA->Q, NT->Q

B: Q->NSW, Q->SA, Q->NT

AC-3: Enforce Arc Consistency of Entire CSP

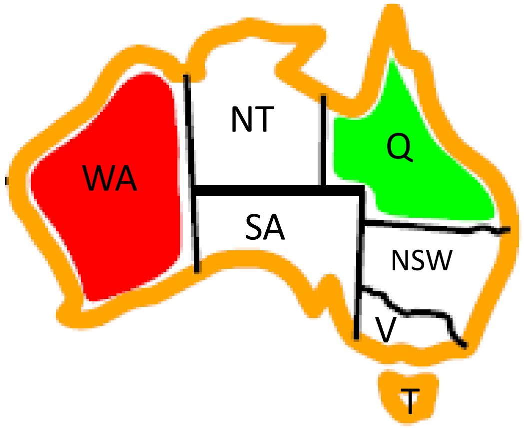


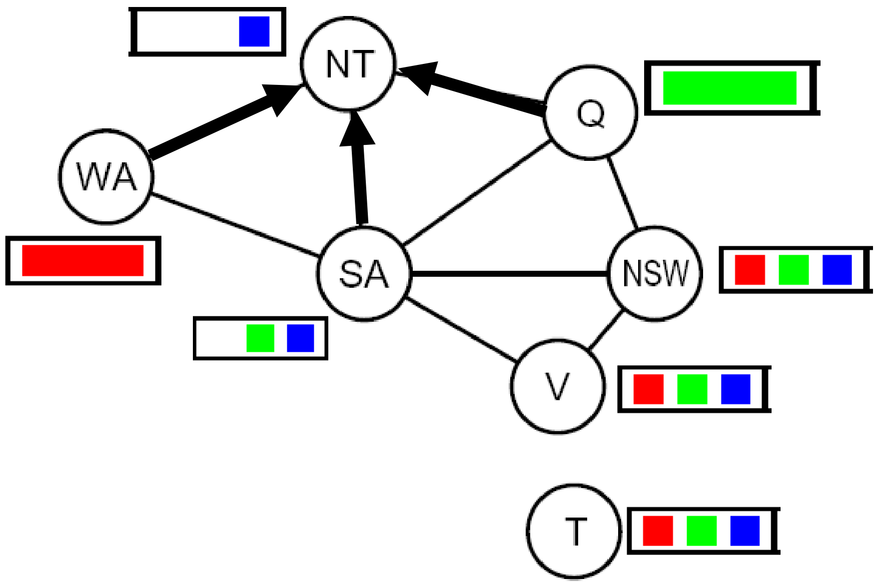


Queue:
NT->Q
SA->Q
NSW->Q

Remember: Delete from the tail!

AC-3: Enforce Arc Consistency of Entire CSP

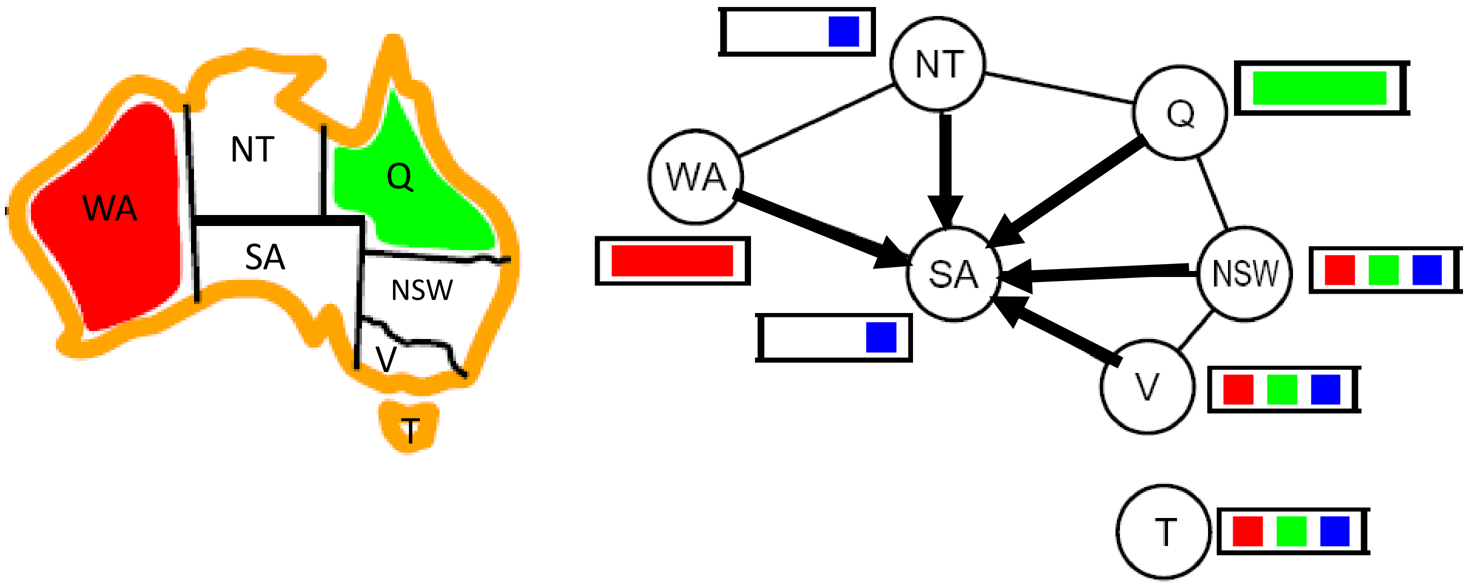




Queue:
SA->Q
NSW->Q
WA->NT
SA->NT
Q->NT

Remember: Delete from the tail!

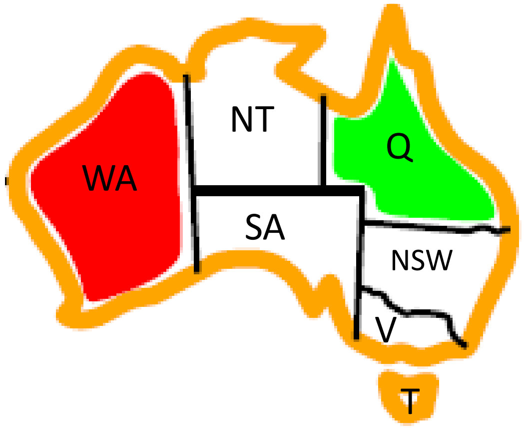
AC-3: Enforce Arc Consistency of Entire CSP

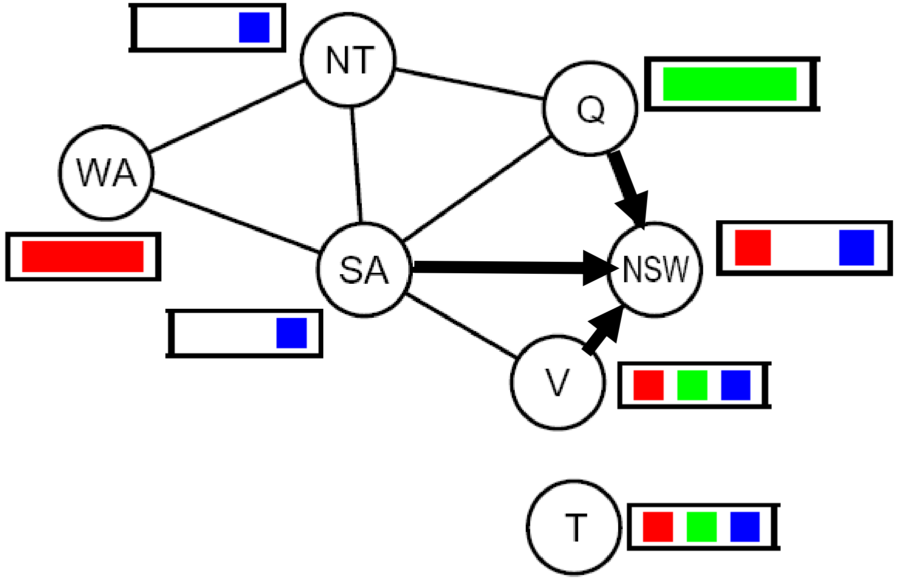


Queue:
NSW->Q
WA->NT
SA->NT
Q->NT
WA->SA
NT->SA
Q->SA
NSW->SA
V->SA

Remember: Delete from the tail!

AC-3: Enforce Arc Consistency of Entire CSP

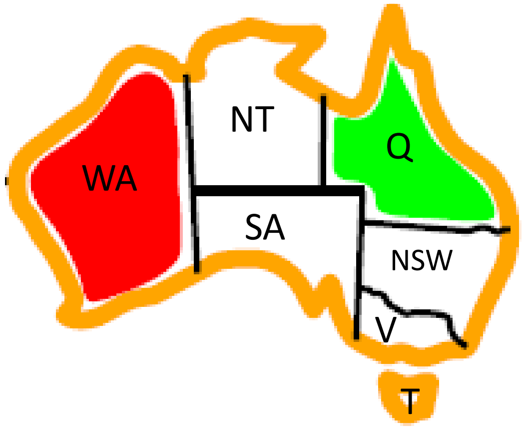


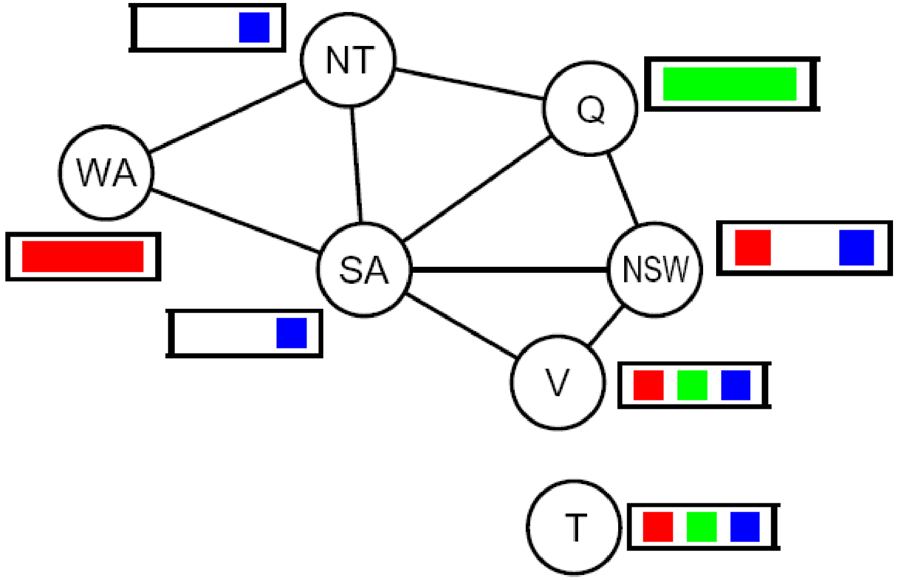


Queue:
WA->NT
SA->NT
Q->NT
WA->SA
NT->SA
Q->SA
NSW->SA
V->SA
V->NSW
Q->NSW
SA->NSW

Remember: Delete from the tail!

AC-3: Enforce Arc Consistency of Entire CSP





Queue:

WA->NT

SA->NT

Q->NT

WA->SA

NT->SA

Q->SA

NSW->SA

V->SA

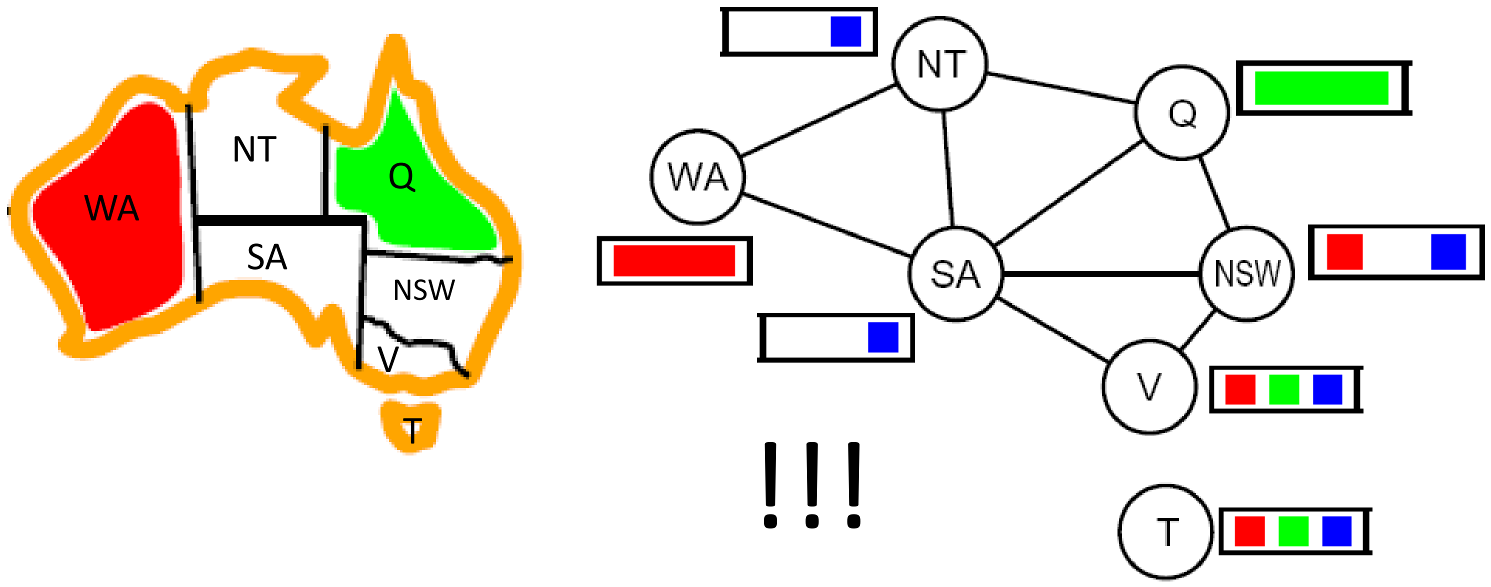
V->NSW

Q->NSW

SA->NSW

Remember: Delete from the tail!

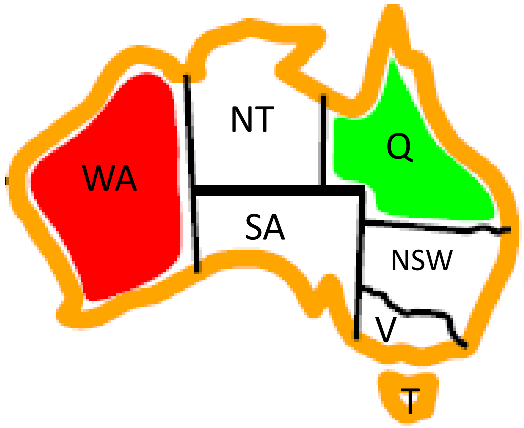
AC-3: Enforce Arc Consistency of Entire CSP

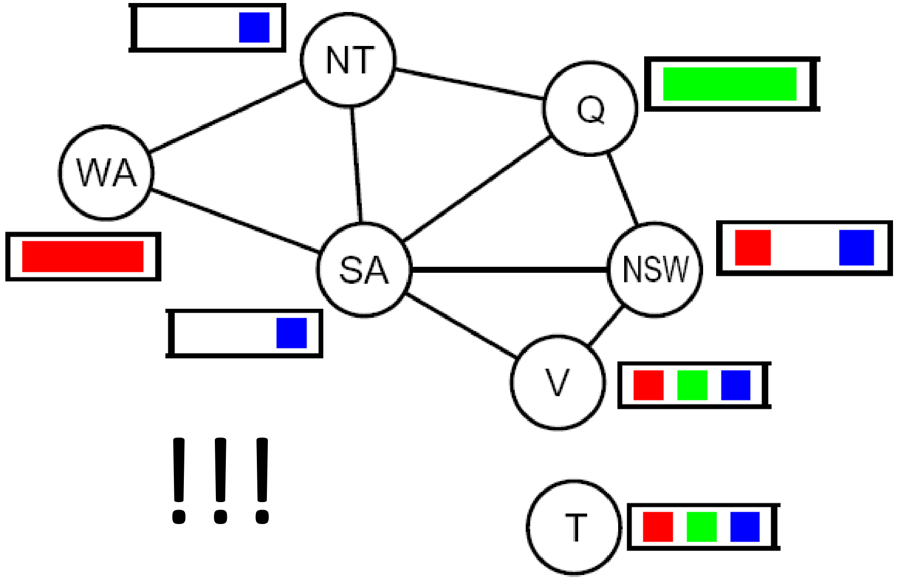


Queue:
SA->NT
Q->NT
WA->SA
NT->SA
Q->SA
NSW->SA
V->SA
V->NSW
Q->NSW
SA->NSW

Remember: Delete from the tail!

AC-3: Enforce Arc Consistency of Entire CSP





Queue:
SA->NT
Q->NT
WA->SA
NT->SA
Q->SA
NSW->SA
V->SA
V->NSW
Q->NSW
SA->NSW

- Backtrack on the assignment of Q
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment
- What's the downside of enforcing arc consistency?

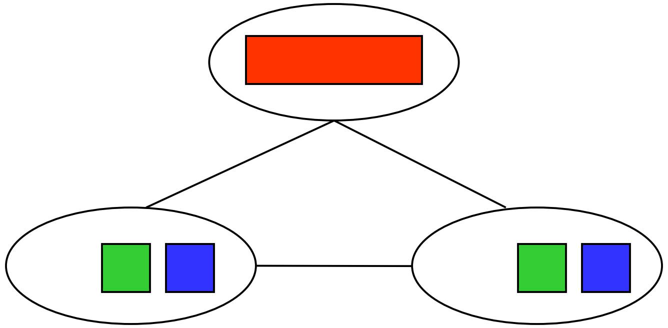
Remember: Delete from the tail!

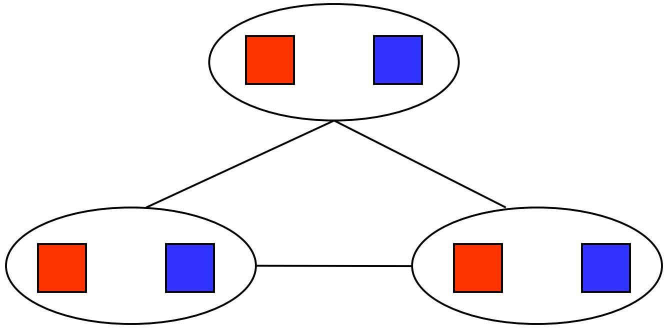
Limitations of Arc Consistency

- After enforcing arc consistency:
  - Can have one solution left
  - Can have multiple solutions left
  - Can have no solutions left (and not know it)

- Arc consistency only checks local consistency conditions

- Arc consistency still runs inside a backtracking search!





What went wrong here?

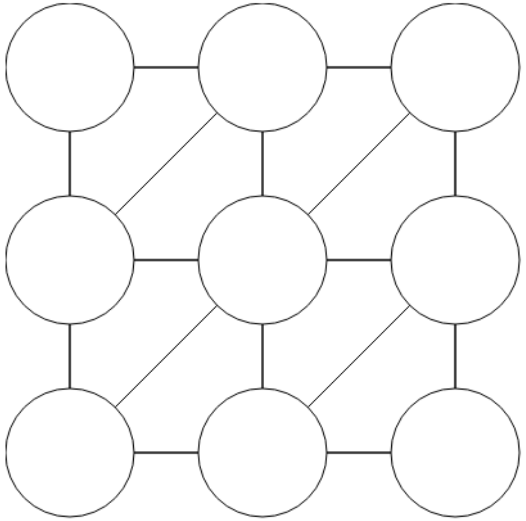
Backtracking Search with AC-3

```
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~~ AC-3(csp))
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

- Where do you run AC-3?

Demo – Backtracking with AC-3



Demo – Coloring with a Complex Graph

Compare

- Backtracking with Forward Checking
- Backtracking with AC-3

Complexity of a single run of AC-3

```
function AC-3( csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X_1, X_2, ..., X_n}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do
      (X_i, X_j) ← REMOVE-FIRST(queue)
      if REMOVE-INCONSISTENT-VALUES(X_i, X_j) then
         for each X_k in NEIGHBORS[X_i] do
            add (X_k, X_i) to queue
```

```
function REMOVE-INCONSISTENT-VALUES( X_i, X_j) returns true iff succeeds
   removed ← false
   for each x in DOMAIN[X_i] do
      if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint X_i ↔ X_j
         then delete x from DOMAIN[X_i]; removed ← true
   return removed
```

Recall that the whole backtracking algorithm with AC-3 will call AC-3 many times

Complexity of a single run of AC-3

```
function AC-3( csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do
      (Xi, Xj) ← REMOVE-FIRST(queue)
      if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
         for each Xk in NEIGHBORS[Xi] do
            add (Xk, Xi) to queue
```

```
function REMOVE-INCONSISTENT-VALUES( Xi, Xj) returns true iff succeeds
   removed ← false
   for each x in DOMAIN[Xi] do
      if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint Xi ↔ Xj
         then delete x from DOMAIN[Xi]; removed ← true
   return removed
```

- An arc is added after a removal of value at a node
- n node in total, each has $\leq d$ values
- Total times of removal: $O(nd)$

Complexity of a single run of AC-3

```
function AC-3( csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do
      (Xi, Xj) ← REMOVE-FIRST(queue)
      if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
         for each Xk in NEIGHBORS[Xi] do
            add (Xk, Xi) to queue
```

```
function REMOVE-INCONSISTENT-VALUES( Xi, Xj) returns true iff succeeds
   removed ← false
   for each x in DOMAIN[Xi] do
      if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint Xi ↔ Xj
         then delete x from DOMAIN[Xi];   removed ← true
   return removed
```

- An arc is added after a removal of value at a node
- n node in total, each has $\leq d$ values
- Total times of removal: $O(nd)$
- After a removal, $\leq n$ arcs added
- Total times of adding arcs: $O(n^2 d)$

Complexity of a single run of AC-3

```
function AC-3( csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X_1, X_2, ..., X_n}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do
      (X_i, X_j) ← REMOVE-FIRST(queue)
      if REMOVE-INCONSISTENT-VALUES(X_i, X_j) then
         for each X_k in NEIGHBORS[X_i] do
            add (X_k, X_i) to queue
```

```
function REMOVE-INCONSISTENT-VALUES( X_i, X_j) returns true iff succeeds
   removed ← false
   for each x in DOMAIN[X_i] do
      if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint X_i ↔ X_j
         then delete x from DOMAIN[X_i]; removed ← true
   return removed
```

- An arc is added after a removal of value at a node
- n node in total, each has $\leq d$ values
- Total times of removal: $O(nd)$
- After a removal, $\leq n$ arcs added
- Total times of adding arcs: $O(n^2 d)$
- Check arc consistency per arc: $O(d^2)$

Complexity of a single run of AC-3 is at most $O(n^2 d^3)$

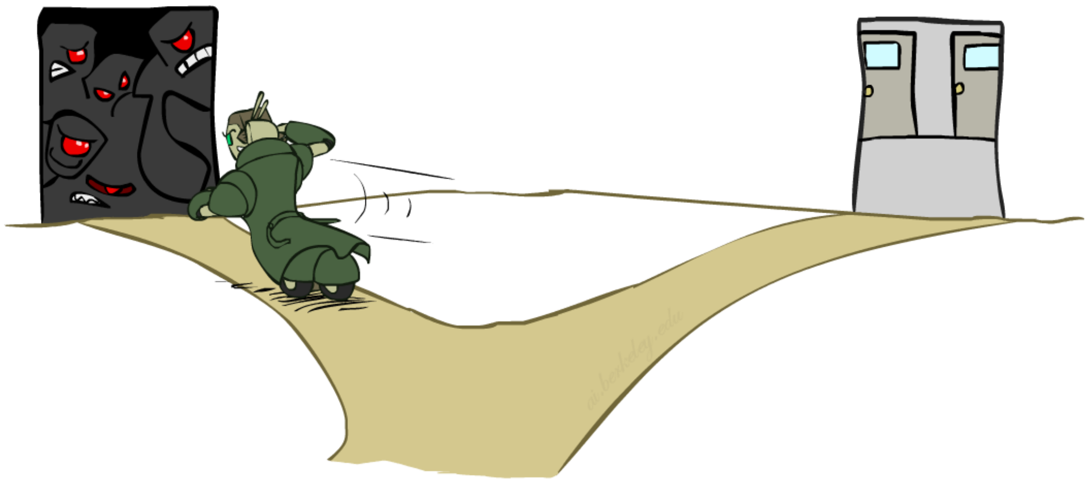
(Not required) Zhang&Yap (2001) show that its complexity is $O(n^2 d^2)$

Ordering

Ordering: Minimum Remaining Values

- Variable Ordering: Minimum remaining values (MRV):
    - Choose the variable with the fewest legal left values in its domain

- Why min rather than max?
- Also called "most constrained variable"
- "Fail-fast" ordering



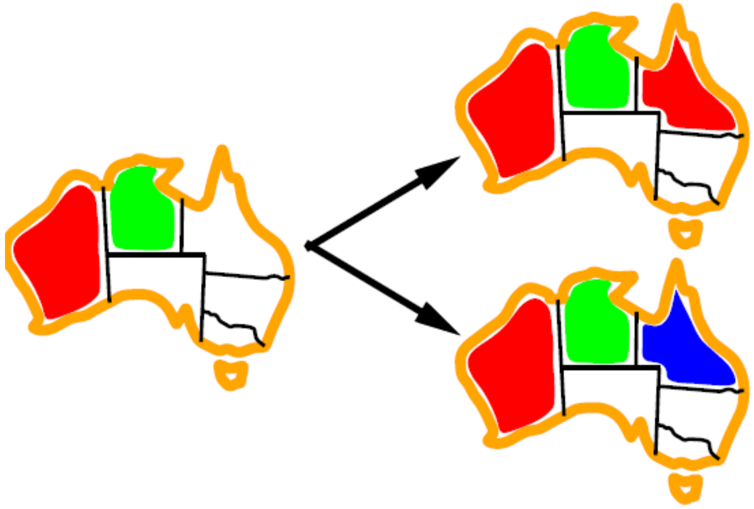
Demo – Coloring with a Complex Graph

Compare

- Backtracking with Forward Checking
- Backtracking with AC-3
- Backtracking + Forward Checking + Minimum Remaining Values (MRV)

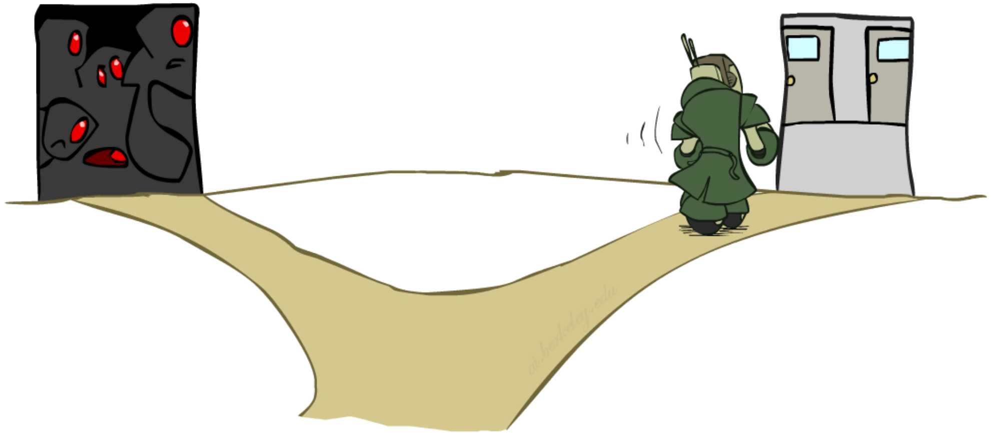
Ordering: Least Constraining Value

- Value Ordering: Least Constraining Value
  - Given a choice of variable, choose the *least constraining value*
  - i.e., the one that rules out the fewest values in the remaining variables
  - Note that it may take some computation to determine this! (E.g., rerunning filtering)



- Why least rather than most?

- Combining these ordering ideas makes 1000 queens feasible

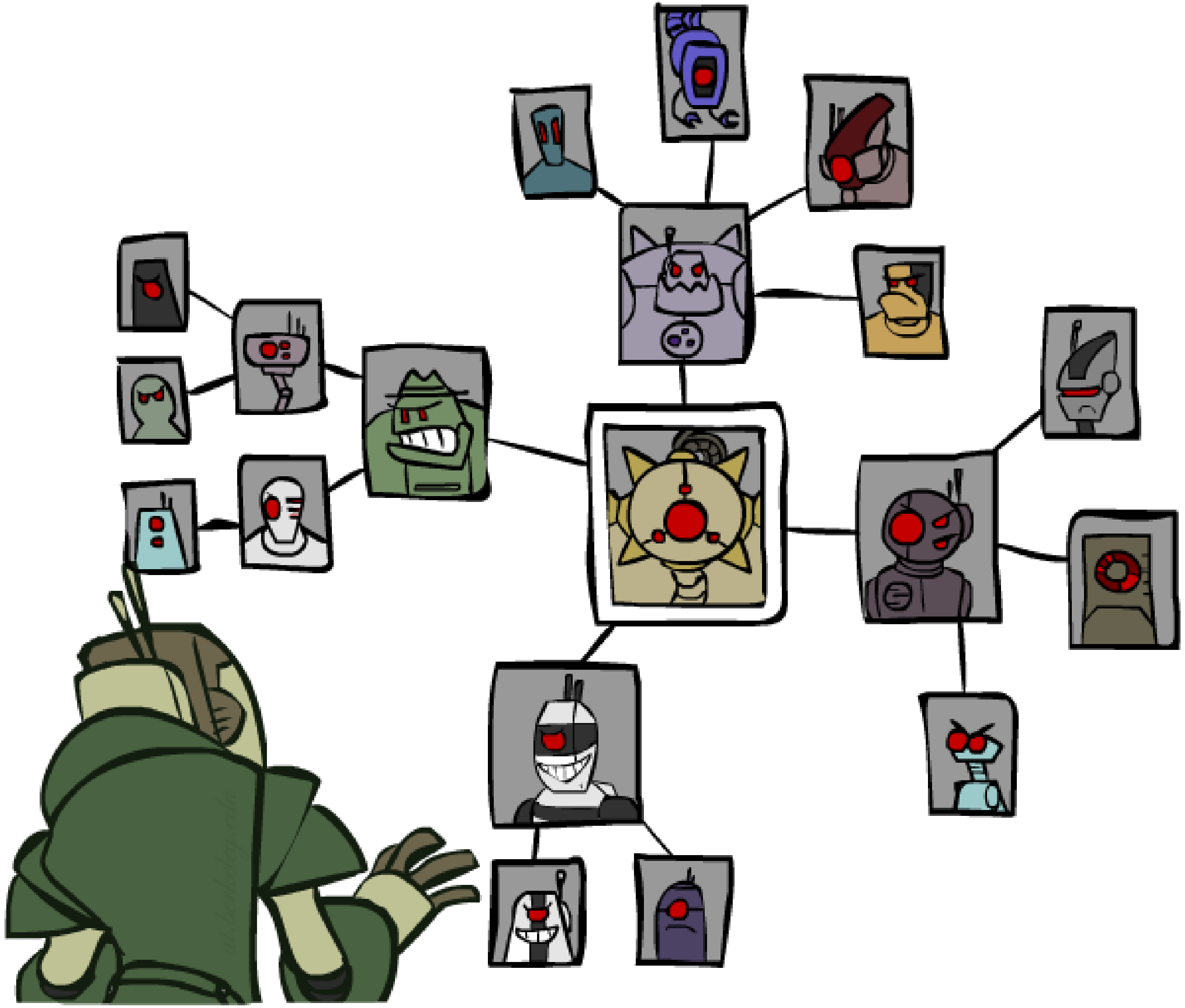


Demo – Coloring with a Complex Graph

Compare

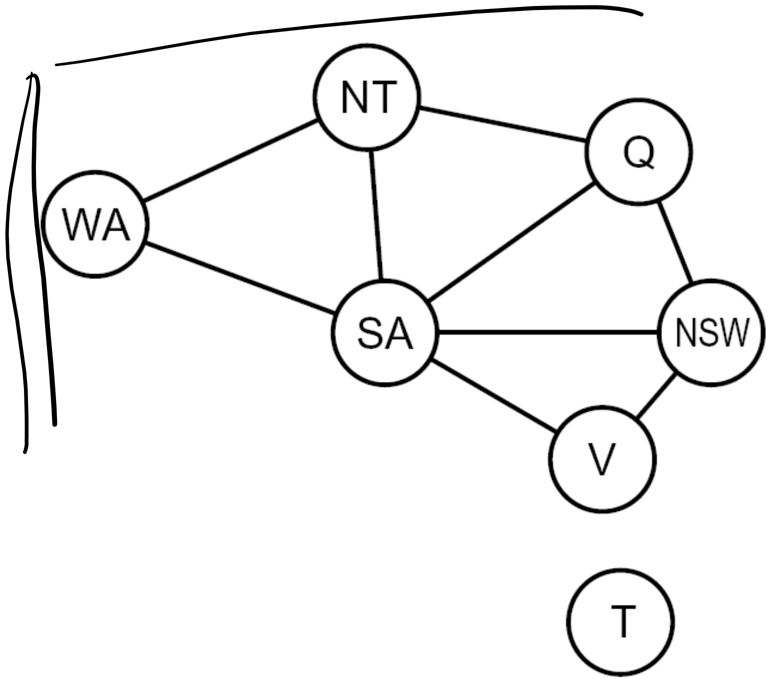
- Backtracking with Forward Checking
- Backtracking with AC-3
- Backtracking + Forward Checking + Minimum Remaining Values (MRV)
- Backtracking + AC-3 + MRV + LCV

Structure

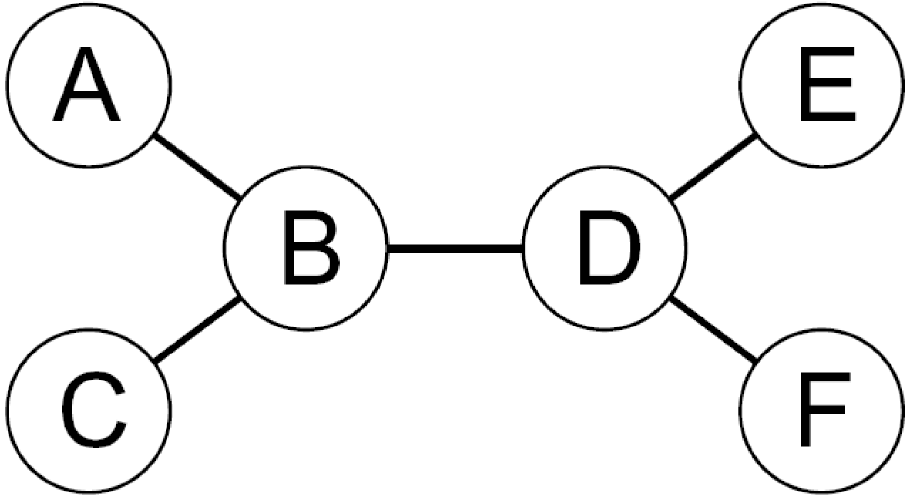


Problem Structure

- For general CSPs, worst-case complexity with backtracking algorithm is $O(d^n)$
- When the problem has special structure, we can often solve the problem more efficiently
- Special Structure 1: Independent subproblems
  - Example: Tasmania and mainland do not interact
  - Connected components of constraint graph
  - Suppose a graph of n variables can be broken into subproblems, each of only c variables:
    - Worst-case complexity is $O((n/c)(d^c))$, linear in n
    - E.g., n = 80, d = 2, c =20
    - 2^{80} = 4 billion years at 10 million nodes/sec
    - $(4)(2^{20})$ = 0.4 seconds at 10 million nodes/sec



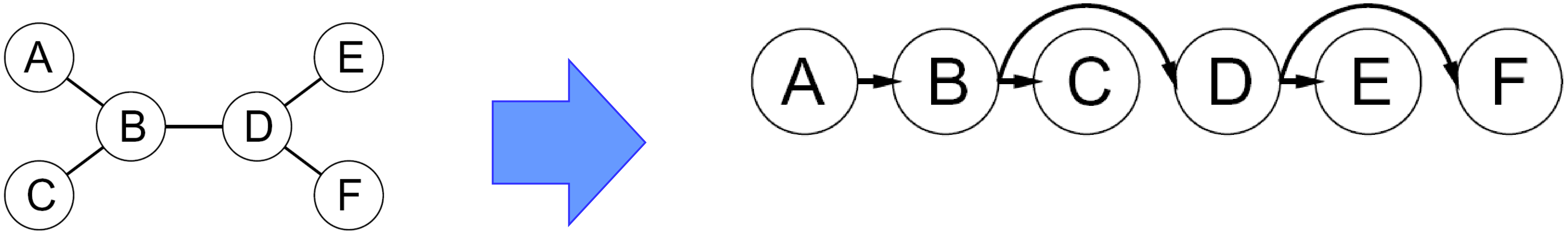
Tree-Structured CSPs



- Theorem: if the constraint graph has no loops, the CSP can be solved in $O(nd^2)$ time
  - Much smaller compare to general CSPs, where worst-case time is $O(d^n)$
  - How?

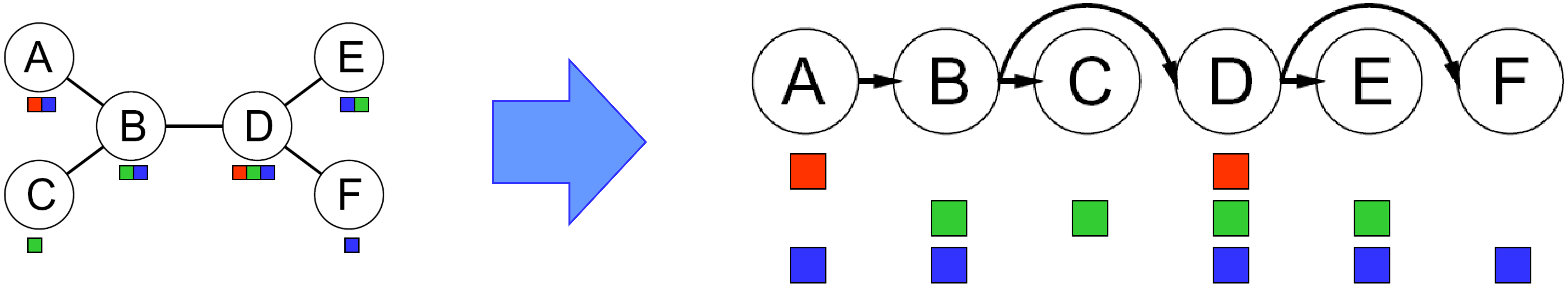
Tree-Structured CSPs

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



Tree-Structured CSPs

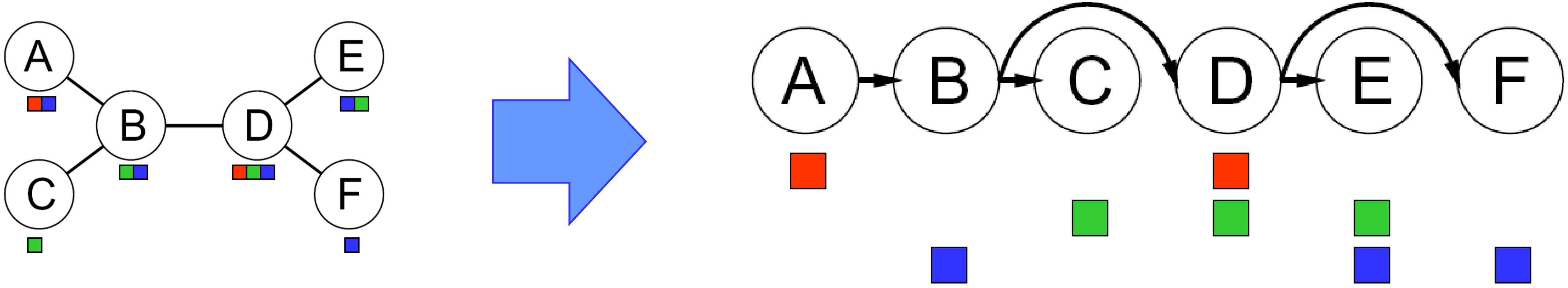
- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children



  - Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)

Tree-Structured CSPs

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

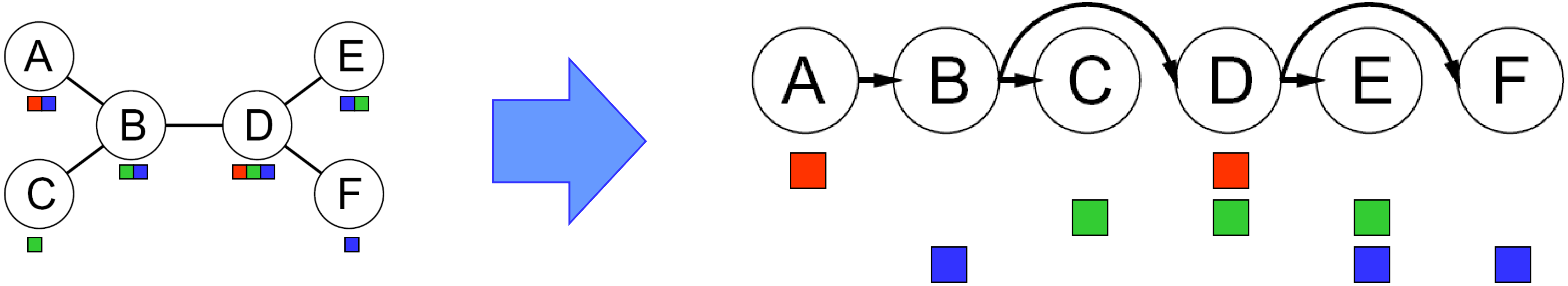


  - Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)
  - Assign forward: For i = 1 : n, assign X_i consistently with Parent(X_i)

- Runtime: $O(nd^2)$ (why?)
- Can always find a solution when there is one (why?)

Tree-Structured CSPs

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



  - Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)
  - Assign forward: For i = 1 : n, assign X_i consistently with Parent(X_i)
- Runtime: O(nd^2) (why?)

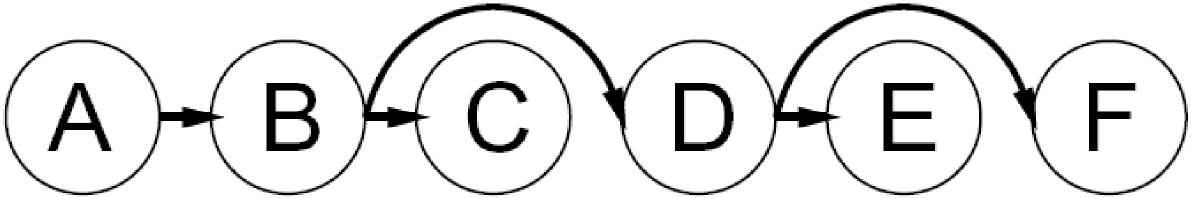
Remove backward $O(nd^2)$: $O(d^2)$ per arc and $O(n)$ arcs

Assign forward $O(nd)$: $O(d)$ per node and $O(n)$ nodes

- Can always find a solution when there is one (why?)

Tree-Structured CSPs

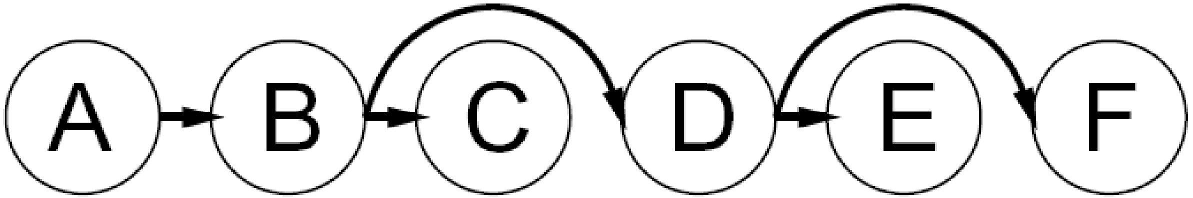
Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)



- Claim 1: After backward pass, all root-to-leaf arcs are consistent
- Proof: During backward pass, every node except the root node was "visited" once.
  - a. Parent$(X_i) \rightarrow X_i$ was made consistent when X_i was visited
  - b. After that, Parent$(X_i) \rightarrow X_i$ kept consistent until the end of the backward pass.

Tree-Structured CSPs

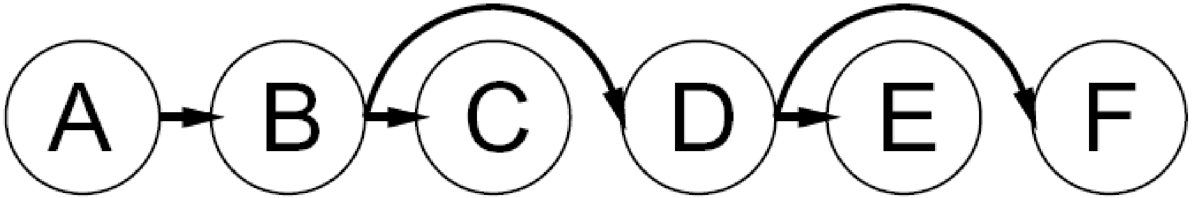
Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)



- Claim 1: After backward pass, all root-to-leaf arcs are consistent
- Proof: During backward pass, every node except the root node was "visited" once.
  - a. Parent$(X_i) \rightarrow X_i$ was made consistent when X_i was visited
    - When X_i was visited, we enforced arc consistency of Parent$(X_i) \rightarrow X_i$ by reducing the domain of Parent(X_i). By definition, for every value in the reduced domain of Parent(X_i), there was some x in the domain of X_i which could be assigned without violating the constraint involving Parent(X_i) and X_i
  - b. After that, Parent$(X_i) \rightarrow X_i$ kept consistent until the end of the backward pass.
    - Domain of X_i would not have been reduced after X_i is visited because X_i's children were visited before X_i. Domain of Parent(X_i) could have been reduced further. Arc consistency would still hold by definition.

Tree-Structured CSPs

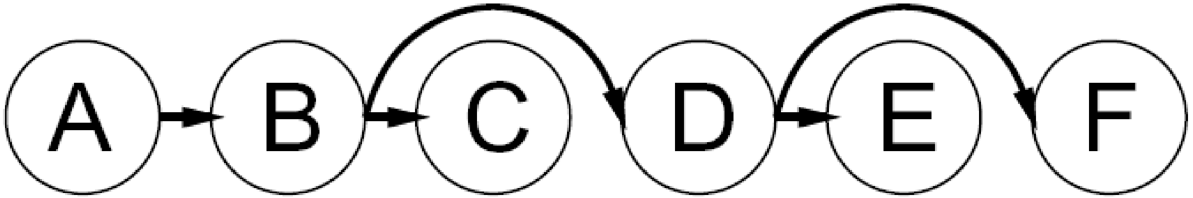
Assign forward: For i = 1 : n, assign X_i consistently with Parent(X_i)



- Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack.
- Proof: Follow the backtracking algorithm (on the reduced domains and with the same ordering). Induction on position.

Tree-Structured CSPs

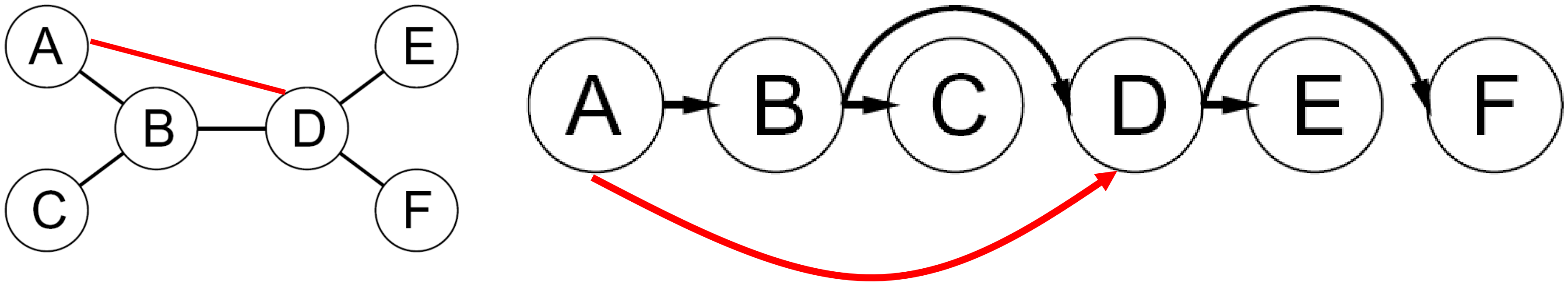
Assign forward: For i = 1 : n, assign X_i consistently with Parent(X_i)



- Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack.
- Proof: Follow the backtracking algorithm (on the reduced domains and with the same ordering). Induction on position. Suppose we have successfully reached node X_i. In the current step, the potential failure can only be caused by the constraint between X_i and Parent(X_i), since all other variables that are in a same constraint of X_i have not assigned a value yet. Due to the arc consistency of Parent$(X_i) \rightarrow X_i$, there exists a value x in the domain of X_i that does not violate the constraint. So we can successfully assign value to X_i and go to the next node. By induction, we can successfully assign a value to a variable in each step of the algorithm. A solution is found in the end.

Tree-Structured CSPs

- Why doesn't this algorithm work with cycles in the constraint graph?



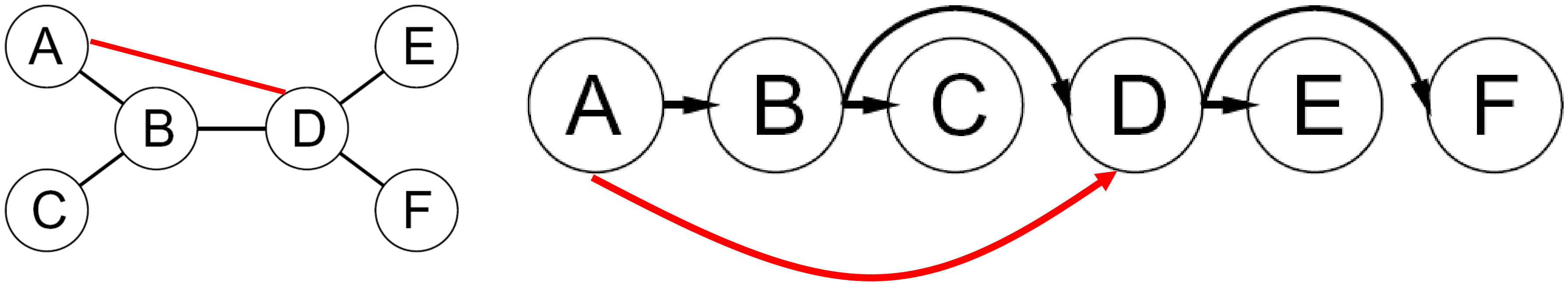
We can still apply the algorithm (choose an arbitrary order and draw "forward" arcs).
For remove backward, what would happen?

For assign forward, what would happen?

Note: We'll see a similar idea with Bayes' nets in later lectures

Tree-Structured CSPs

- Why doesn't this algorithm work with cycles in the constraint graph?



We can still apply the algorithm (choose an arbitrary order and draw "forward" arcs).

For remove backward, what would happen?

We can enforce all arcs pointing to X_i when X_i is visited. The complexity is $O(n^2d^2)$. After backward pass, the reduced domains do not exclude any solution and all the forward arcs are consistent

For assign forward, what would happen?

In a step of assigning values, we may encounter failure because we need to make sure the constraints involving the current node and any parent node is satisfied, which could be impossible. Therefore, we may need to backtrack.

How to deal with non-binary CSPs?

- Variables:

  $F \ T \ U \ W \ R \ O \ X_1 \ X_2 \ X_3$

- Domains:

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

- Constraints:

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

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

  $\cdots$

```
    T W O
  + T W O
  -------
  F O U R
```

Constraint graph for non-binary CSPs

- Variable nodes: nodes to represent the variables
- Constraint nodes: auxiliary nodes to represent the constraints
- Edges: connects a constraint node and its corresponding variables

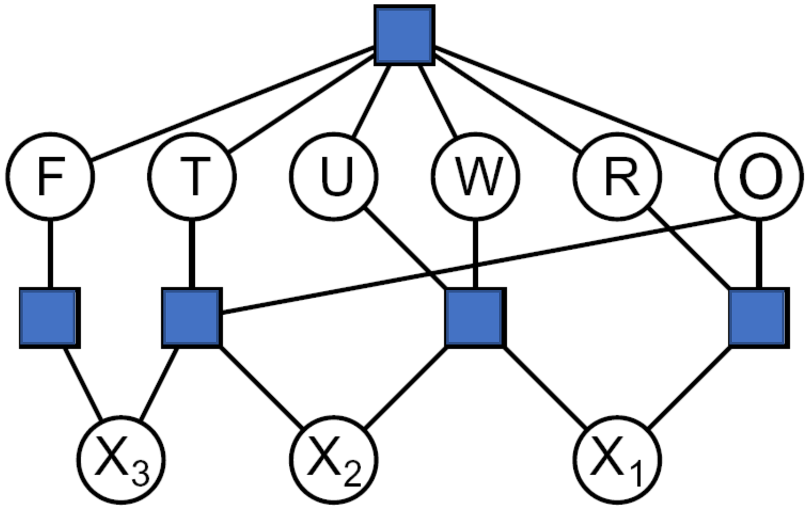
```
   T W O
+  T W O
---------
F  O U R
```

Constraints:

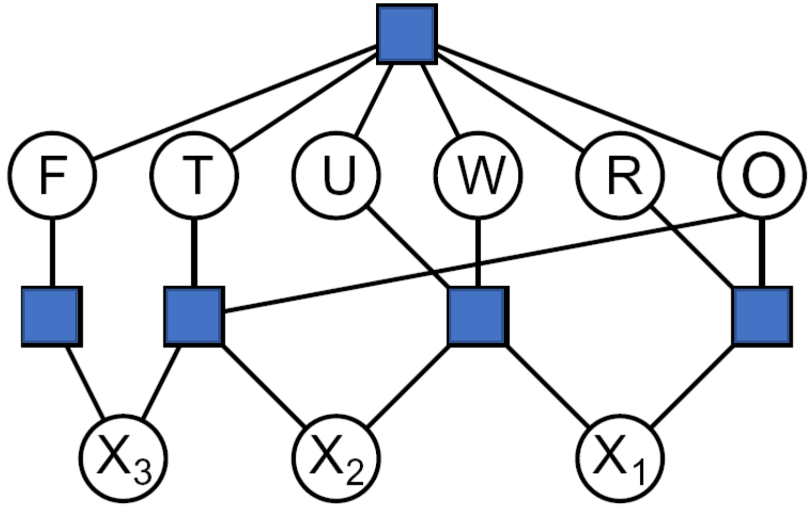
$$\text{alldiff}(F, T, U, W, R, O)$$

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

$$\cdots$$



Solve non-binary CSPs



- Naïve search?
  - Yes!
- Backtracking?
  - Yes!
- Forward Checking?
  - Need to generalize the original FC operation
  - (nFC0) After a variable is assigned a value, find all constraints with only one unassigned variable and cross off values of that unassigned variable which violate the constraint
  - There exist other ways to do generalized forward checking

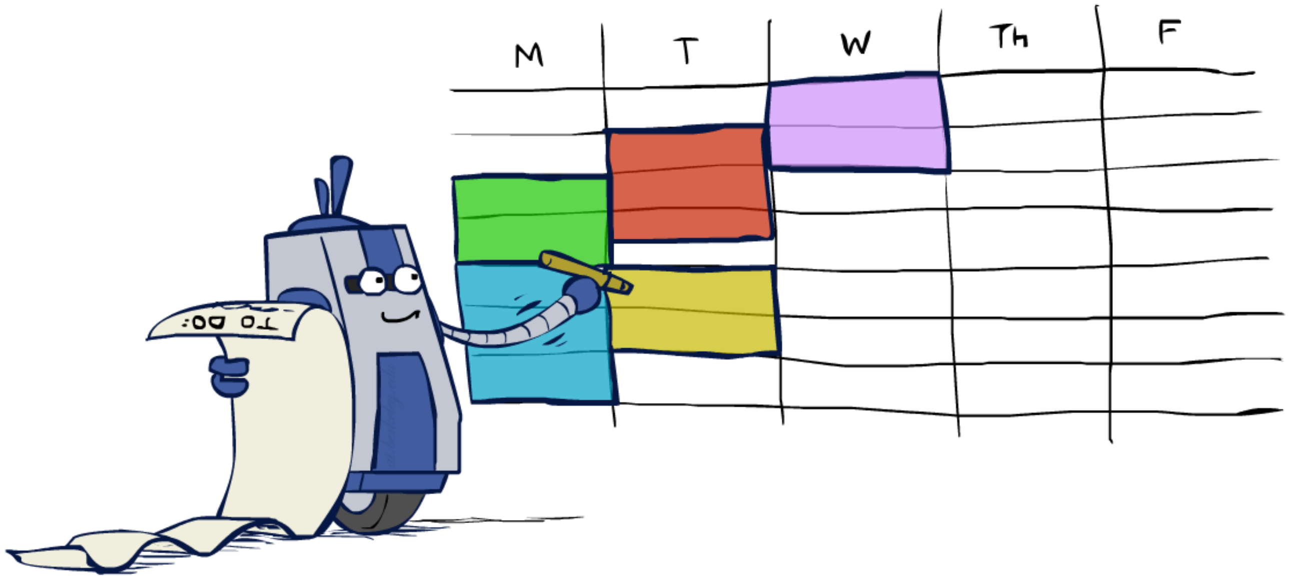
Solve non-binary CSPs

- (Bonus material, not required)
- AC-3? Need to generalize the definition of AC and enforcement of AC
- Generalized arc-consistency (GAC)
    - A non-binary constraint is GAC iff for every value for a variable there exist consistent value combinations for all other variables in the constraint
    - Reduced to AC for binary constraints
- Enforcing GAC
    - Simple schema: enumerate value combination for all other variables
    - O(d^k) on k-ary constraint on variables with domains of size d

- There are other algorithms for non-binary constraint propagation, e.g., (i,j)-consistency [Freuder, JACM 85]

Summary: CSPs

- CSPs are a special kind of search problem:
  - States are partial assignments
  - Goal test defined by constraints
- Basic solution: backtracking search
- Speed-ups:
  - Ordering
  - Filtering
  - Structure



Learning Objectives

- Describe definition of CSP problems and its connection with general search problems
- Formulate a real-world problem as a CSP
- Describe and implement backtracking algorithm
- Define arc consistency
- Describe and implement forward checking and AC-3
- Explain the differences between MRV and LCV heuristics
- Understand the complexity of general binary CSP and tree-structured binary CSP

Additional Resources (Not required)

- Demos, exercises: http://aispace.org/
- References

  - Zhang, Yuanlin, and Roland HC Yap. "Making AC-3 an optimal algorithm." In *IJCAI*, vol. 1, pp. 316-321. 2001.
  - Freuder, Eugene C. "A sufficient condition for backtrack-bounded search." *Journal of the ACM (JACM)* 32, no. 4 (1985): 755-761.
