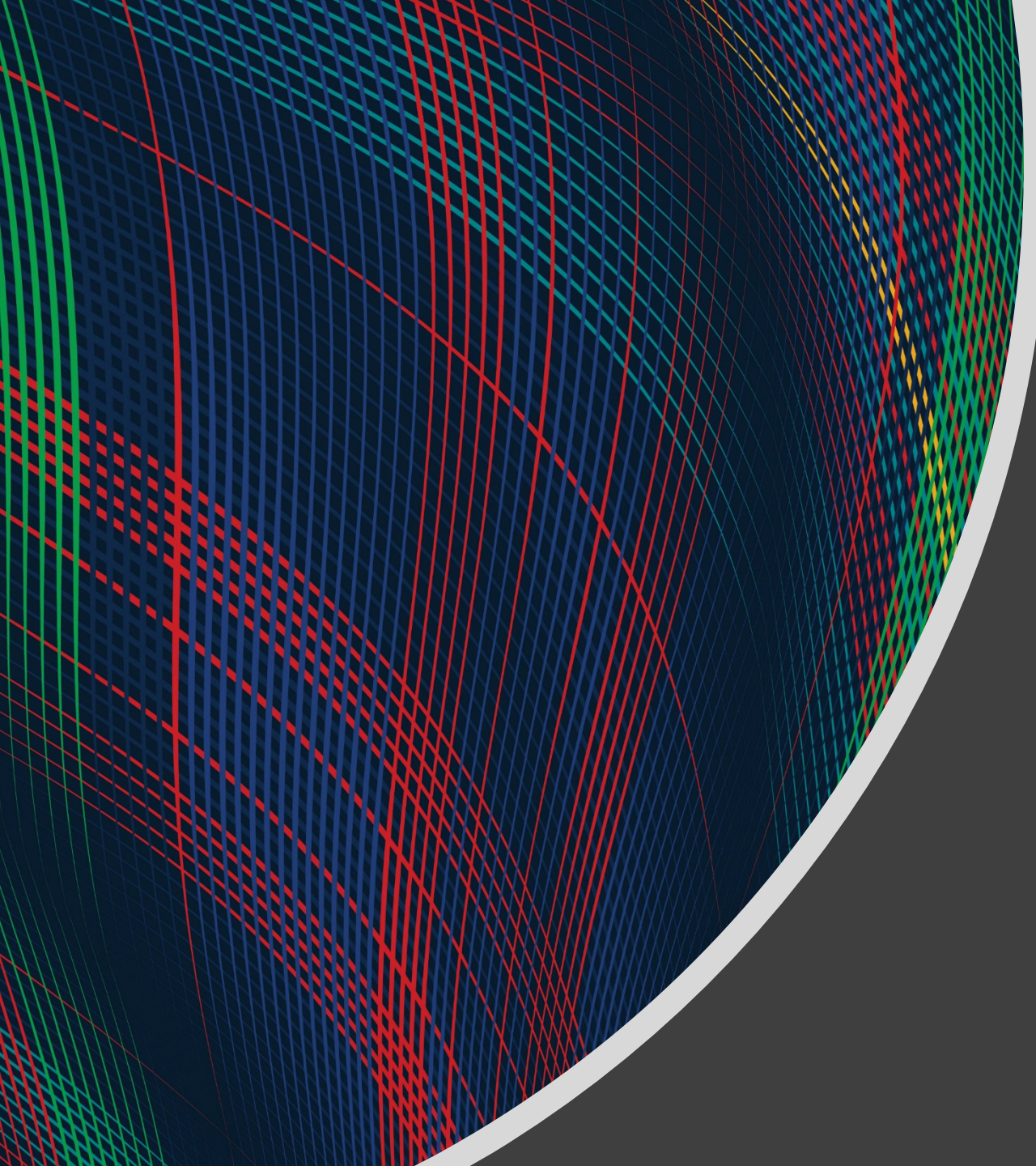


An abstract graphic on the left side of the slide, featuring a sphere-like shape composed of a dense grid of intersecting red, green, and blue lines. The lines are curved and follow the contours of the sphere, creating a complex, woven pattern. The sphere is set against a dark gray background.

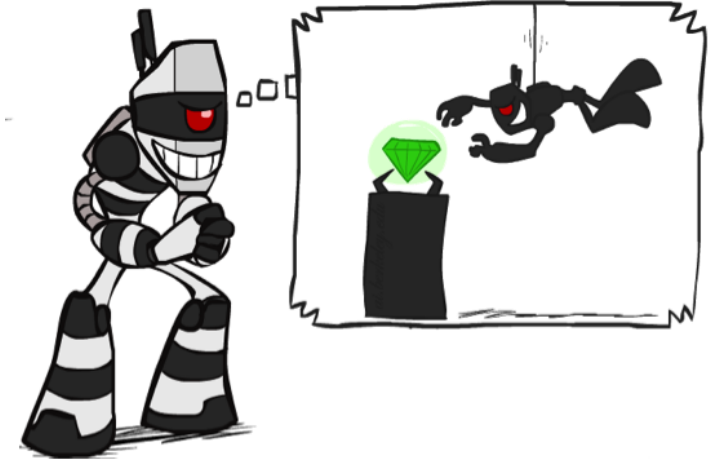
07-280 AI & ML I Constraint Satisfaction Problems (CSPs)

Instructors:
Pat Virtue & Nihar Shah

What is Search For?

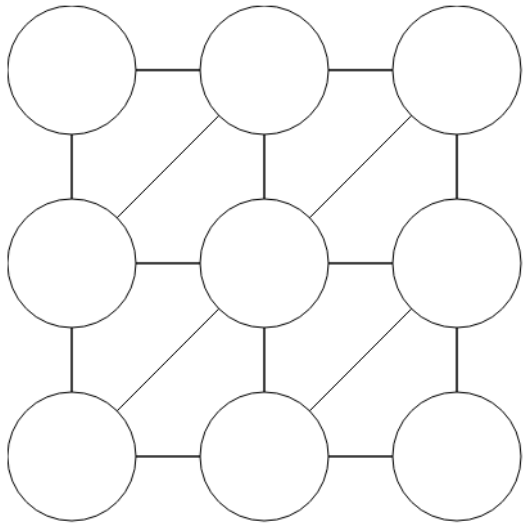
- Planning: sequences of actions
 - The path to the goal is the important thing
 - Paths have various costs, depths
 - Heuristics give problem-specific guidance
- Identification: assignments to variables
 - The goal itself is important, not the path
 - All paths at the same depth (for some formulations)

Are the warm-up assignments
planning or identification problems?



Warm-up as You Walk In

Assign Red, Green, or Blue
Neighbors must be different



Sudoku

1			
	2	1	
		3	
			4

Constraint Satisfaction Problems

CSP is a special class of search problems

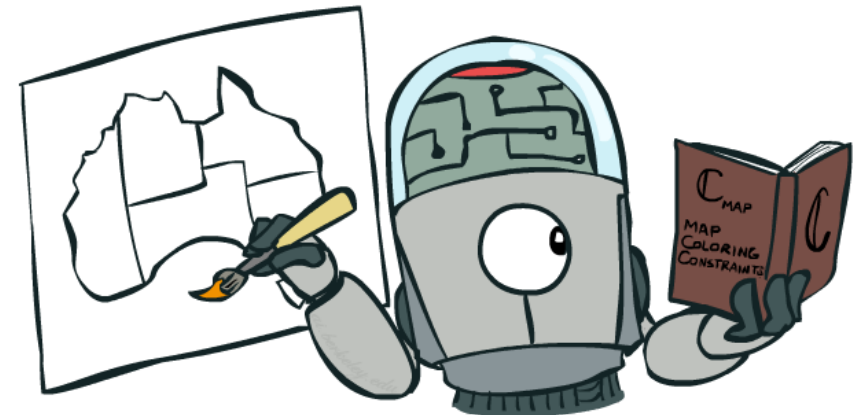
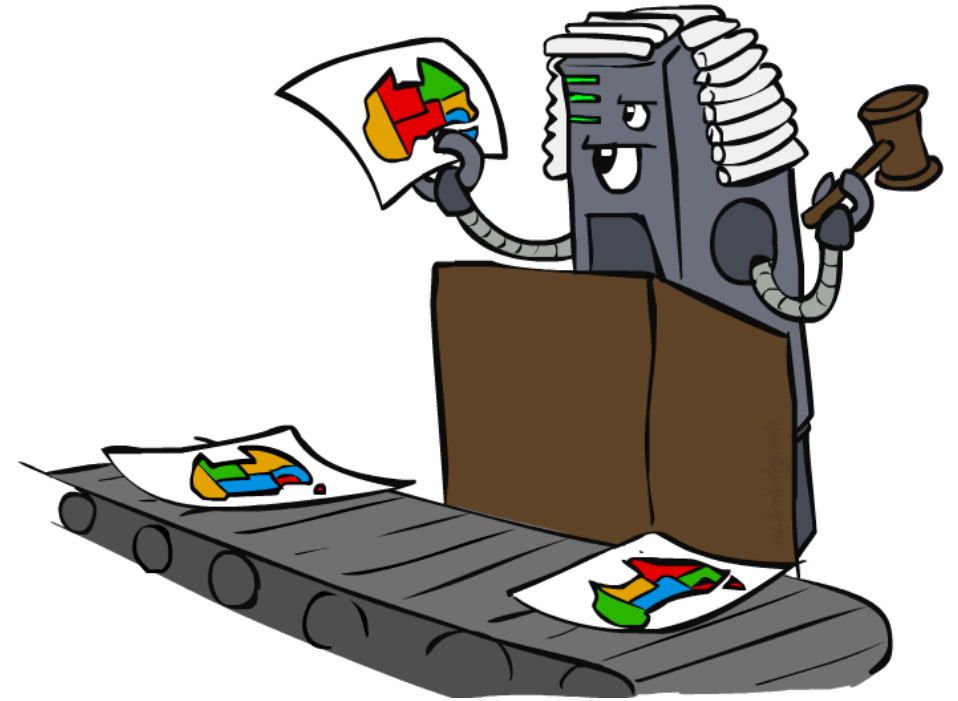
- Mostly identification problems
- Have specialized algorithms for them

Standard search problems:

- State is an arbitrary data structure
- Goal test can be any function over states

Constraint satisfaction problems (CSPs):

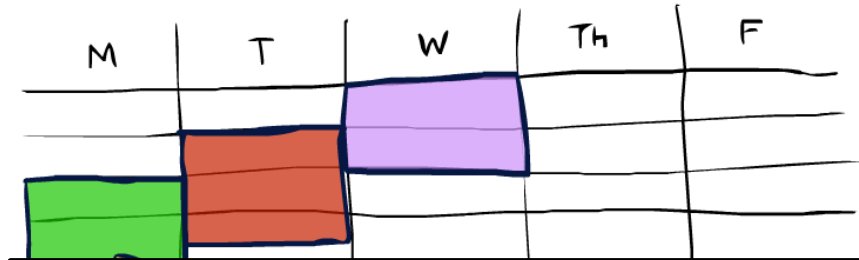
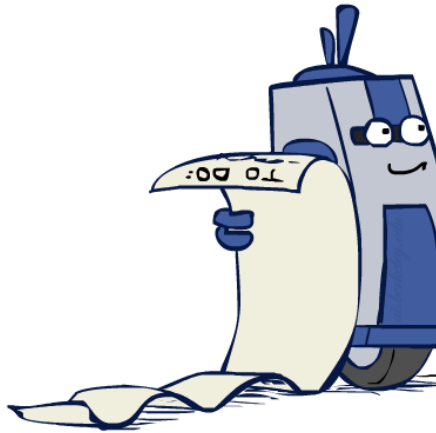
- State is defined by **variables X_i** with values from a **domain D** (sometimes D depends on i)
- Goal test is a **set of constraints** specifying allowable combinations of values for subsets of variables



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...



Stephanie Rosenthal (She/Her) • 1st

Artificial Intelligence and Human-Computer Interaction

5mo • 🌐

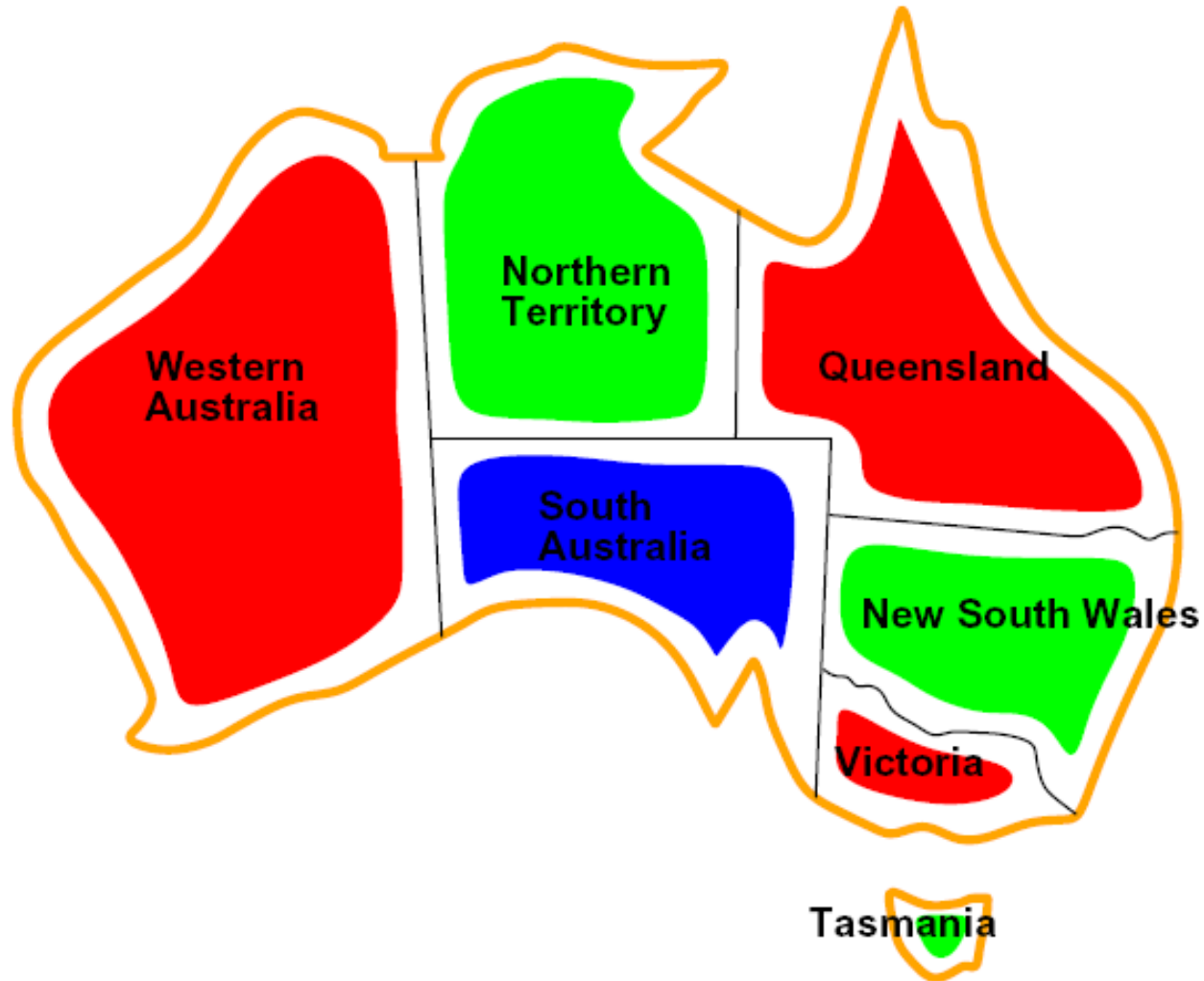
Hi Everyone. My amazing team at Microsoft is hiring a Scheduling/OR specialist to help with our efforts to create an automated scheduling service. Do you know anyone who fits the bill? Have them reach out to me for more information.

[Search Jobs | Microsoft Careers](#)

jobs.careers.microsoft.com



CSP Examples



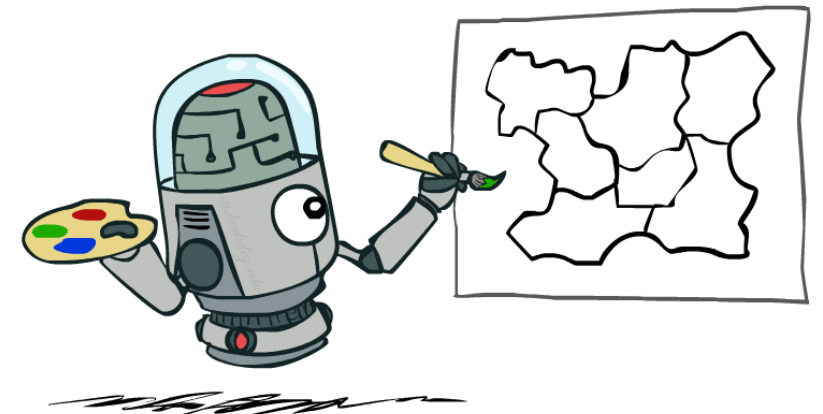
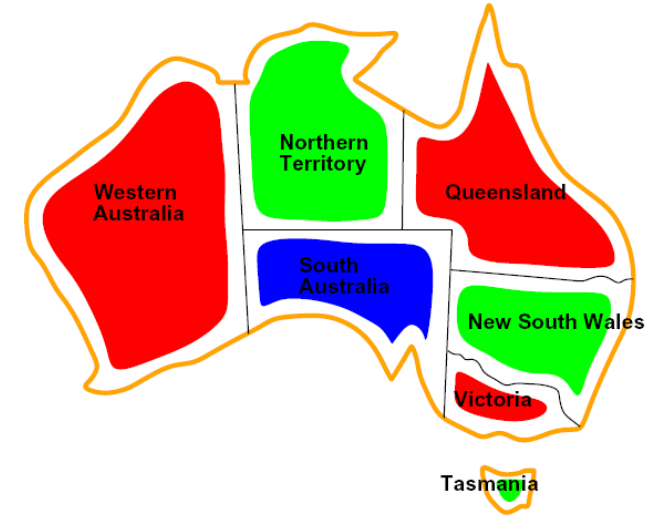
Example: Map Coloring

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

$$\text{WA} \neq \text{NT}$$

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

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



Example: Map Coloring

Google OR Tools

```
from ortools.sat.python import cp_model

model = cp_model.CpModel()

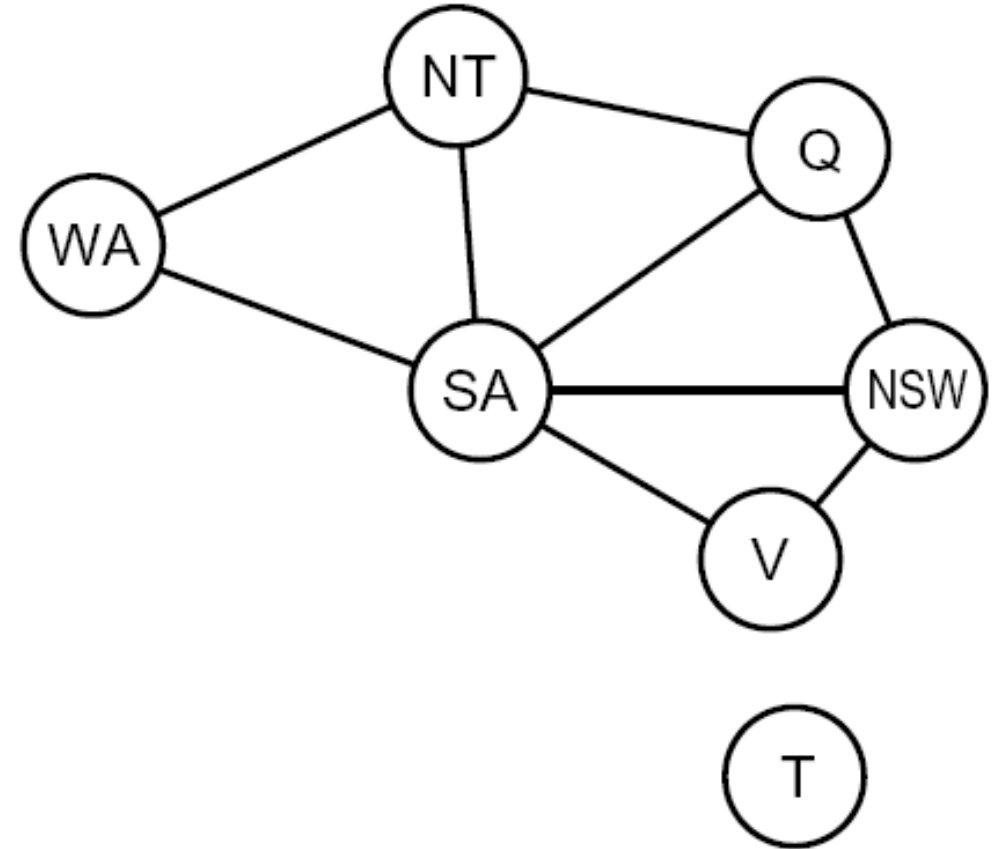
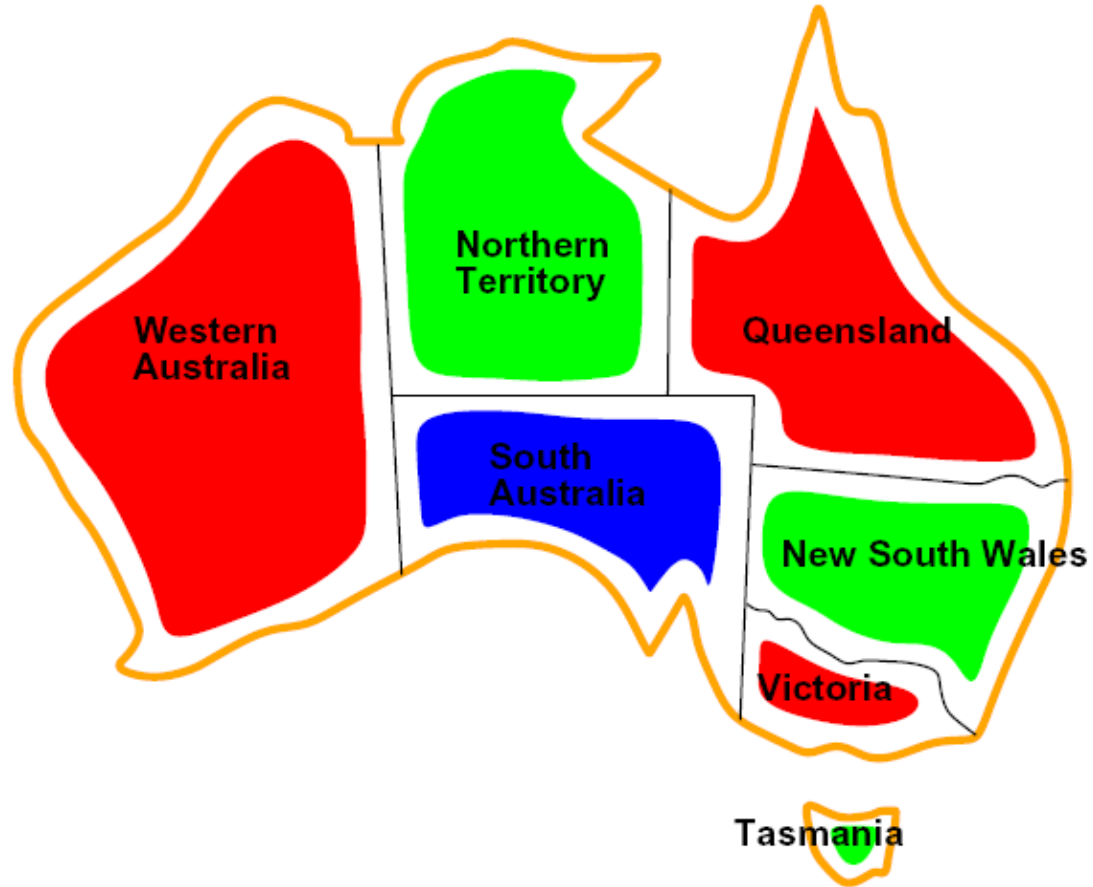
# All CSP variables:
# Domain: {0,1,2} representing three colors
variables = {
    "WA": model.NewIntVar(0, 2, "WA"),
    "NT": model.NewIntVar(0, 2, "NT"),
    "SA": model.NewIntVar(0, 2, "SA"),
    "Q": model.NewIntVar(0, 2, "Q"),
    "NSW": model.NewIntVar(0, 2, "NSW"),
    "V": model.NewIntVar(0, 2, "V"),
    "T": model.NewIntVar(0, 2, "T"),
}
```

```
# Edges (related to binary constraints)
constraints = [
    ("WA", "NT"),
    ("WA", "SA"),
    ("NT", "SA"),
    ("NT", "Q"),
    ("SA", "Q"),
    ("SA", "NSW"),
    ("SA", "V"),
    ("Q", "NSW"),
    ("NSW", "V"),
]

# Add binary inequality constraints:
# adjacent regions must differ in color
for v1, v2 in constraints:
    model.Add(variables[v1] != variables[v2])

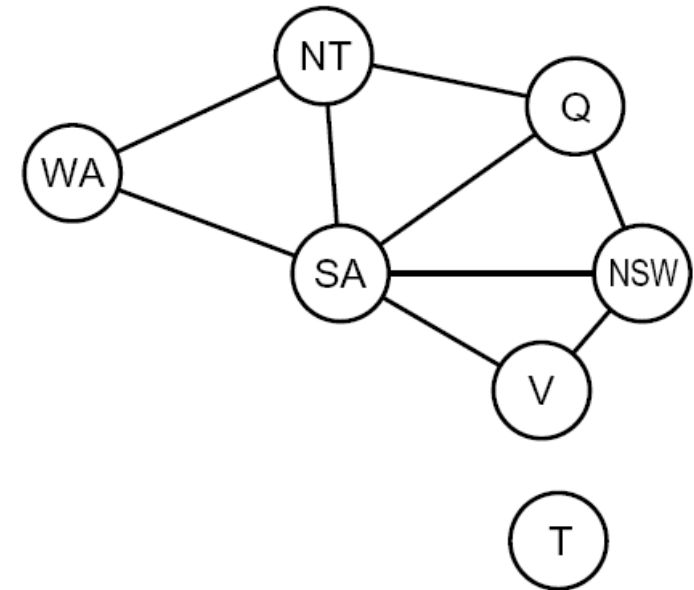
# Solve
solver = cp_model.CpSolver()
status = solver.Solve(model)
```

Constraint Graphs

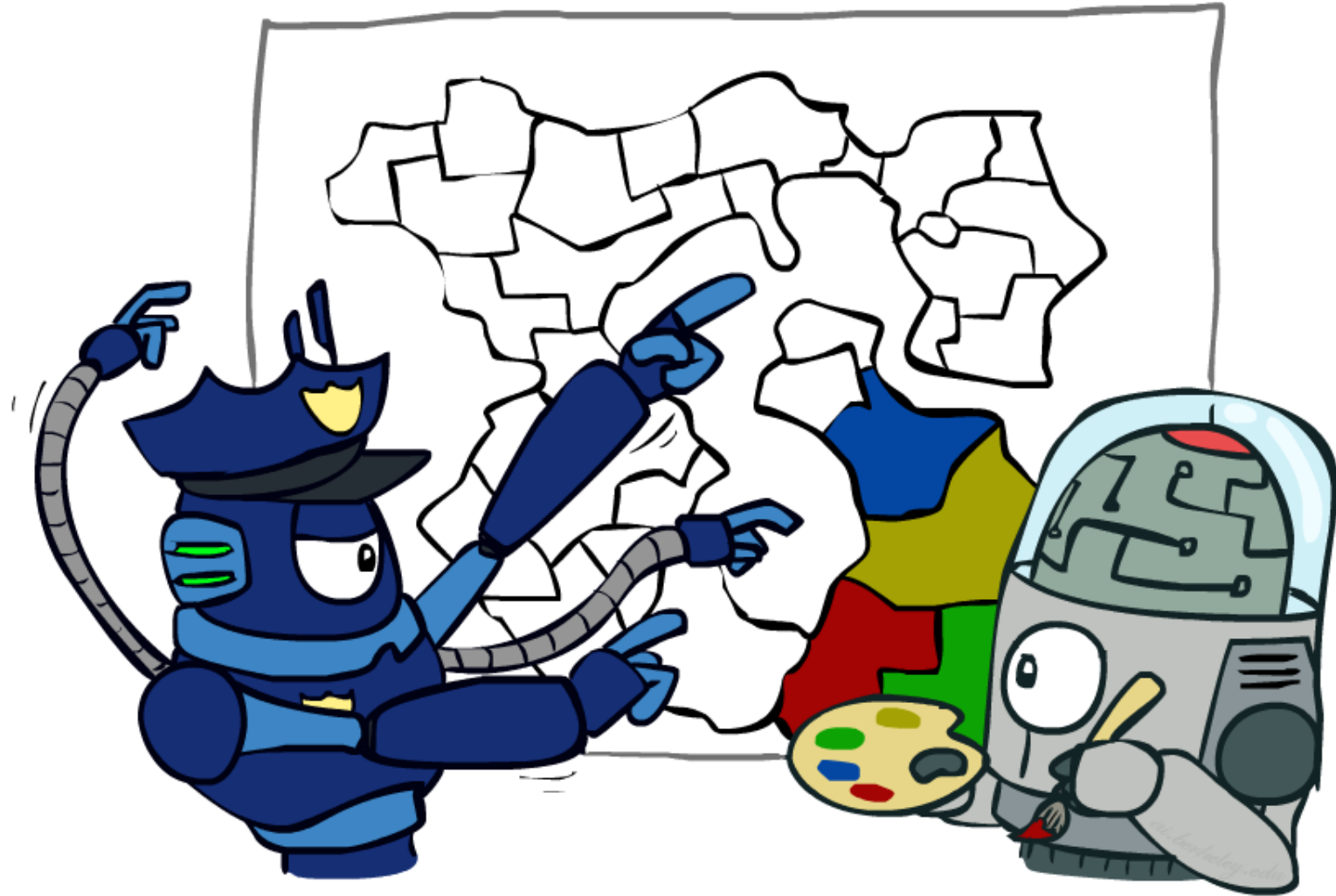


Constraint Graphs

- 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!



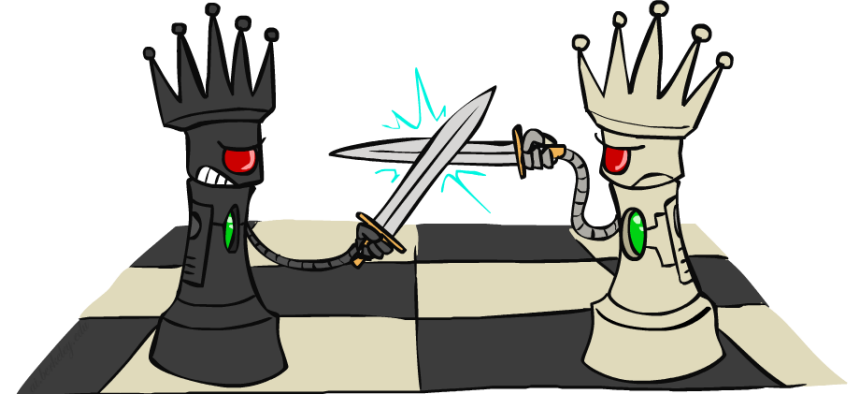
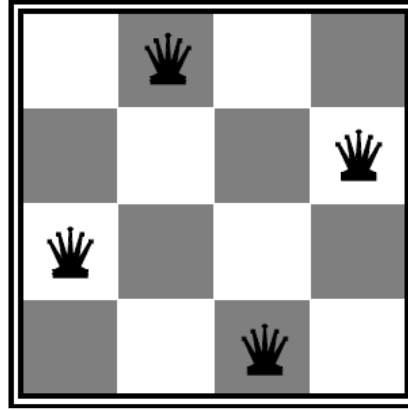
Varieties of CSPs and Constraints



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

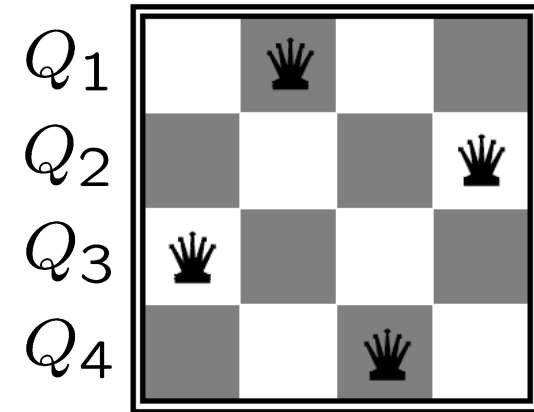
- Formulation 2:

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

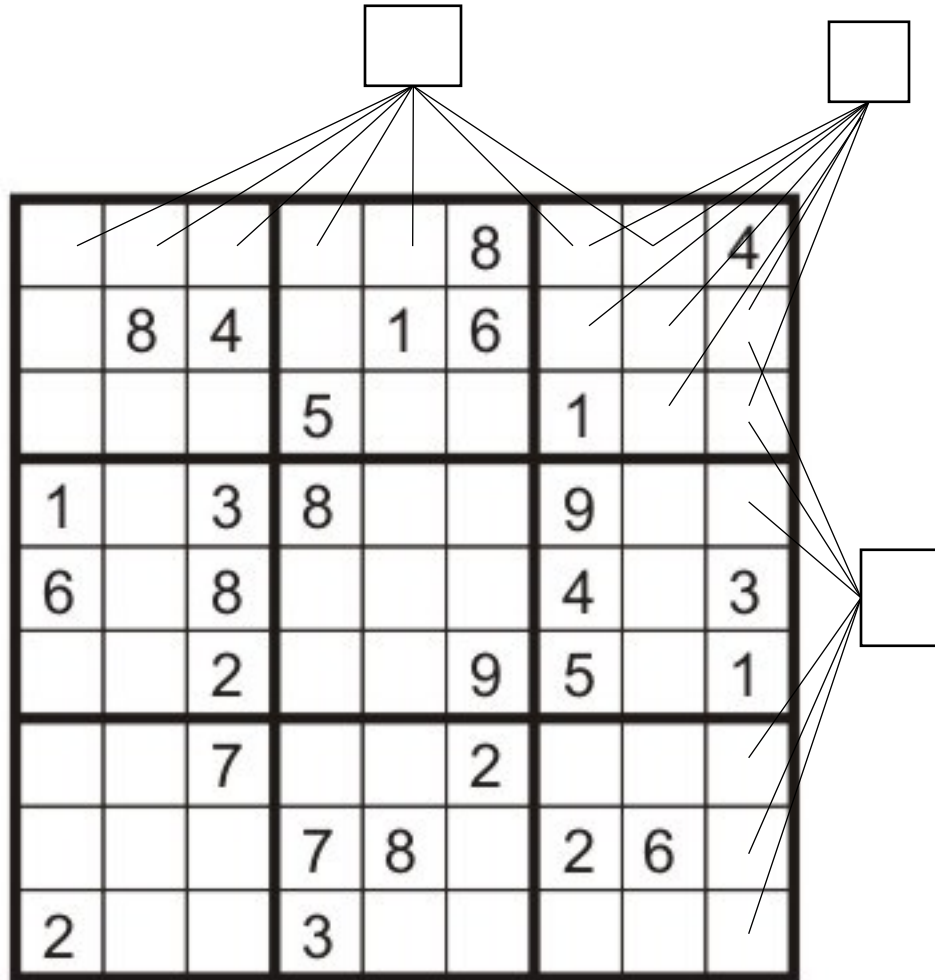
Implicit: $\forall i, j \text{ non-threatening}(Q_i, Q_j)$

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

• • •



Example: Sudoku



- Variables: Each (open) square
- Domains: $\{1,2,\dots,9\}$
- 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)

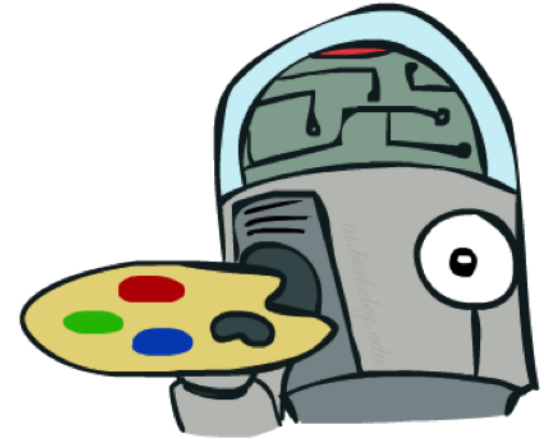
Varieties of CSPs

- 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 job
 - Linear constraints solvable, nonlinear undecidable

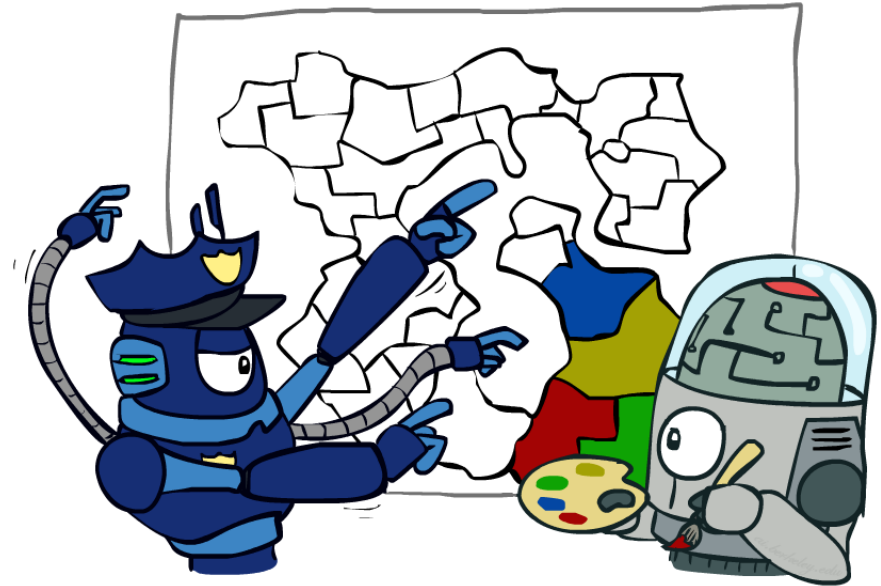
We will cover in a 07-380

- 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.:
 $SA \neq \text{green}$ Focus of today
 - 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

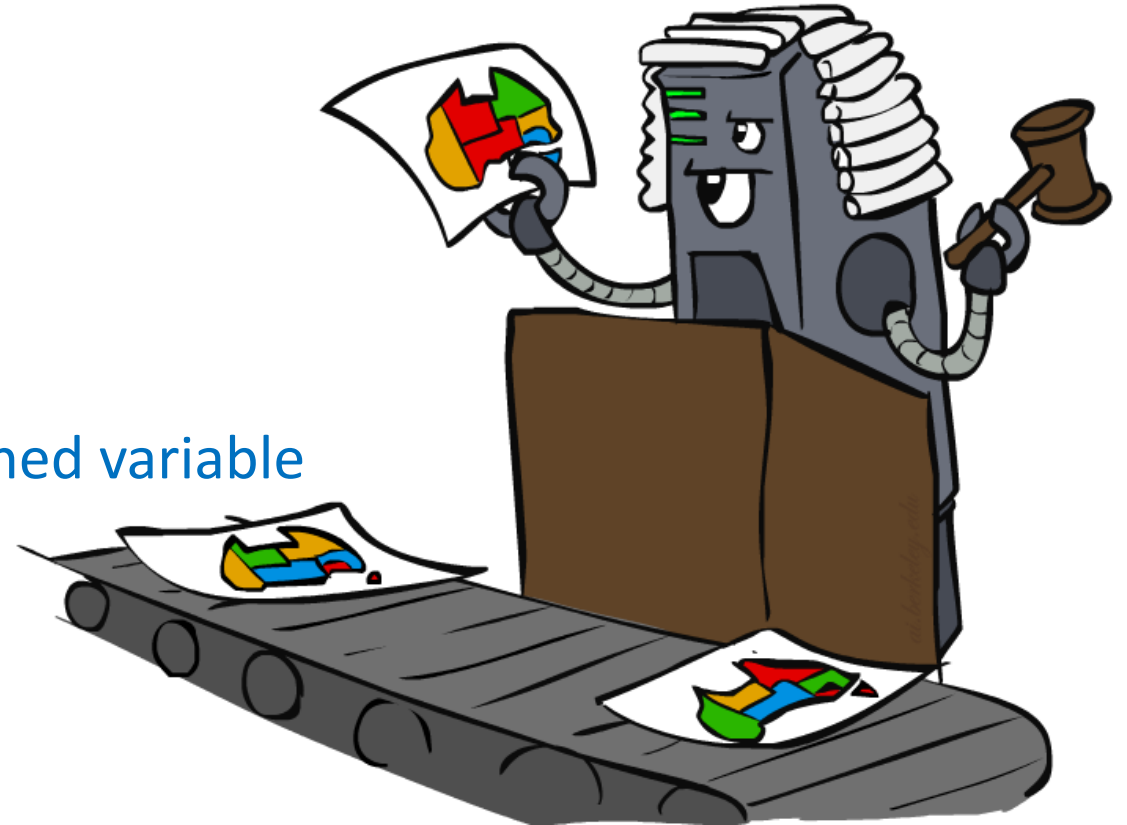


Solving CSPs



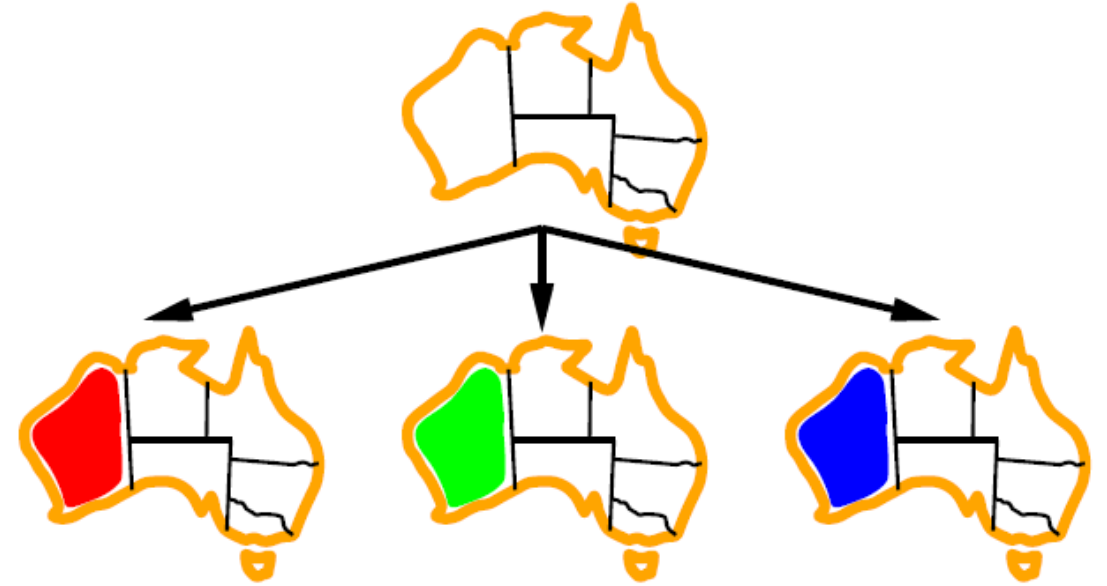
Standard Search Formulation

- 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



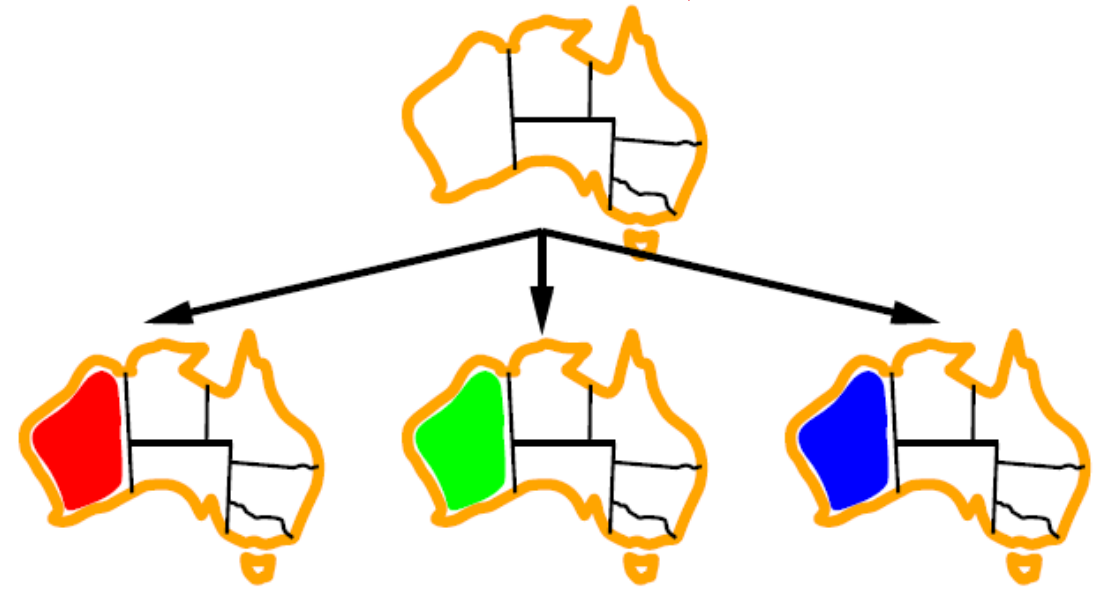
Poll 1: Search for CSPs

Should we use BFS or DFS?

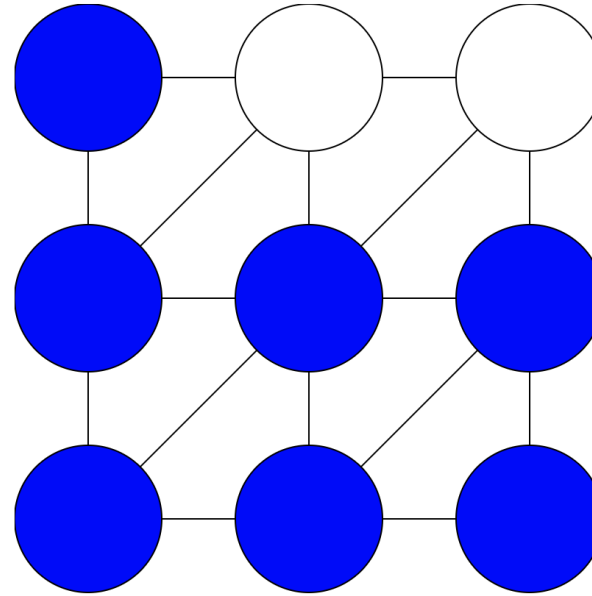
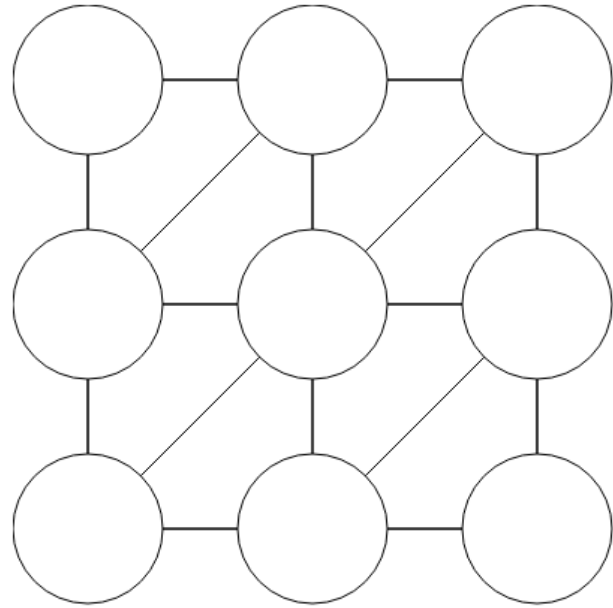


Depth First Search

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

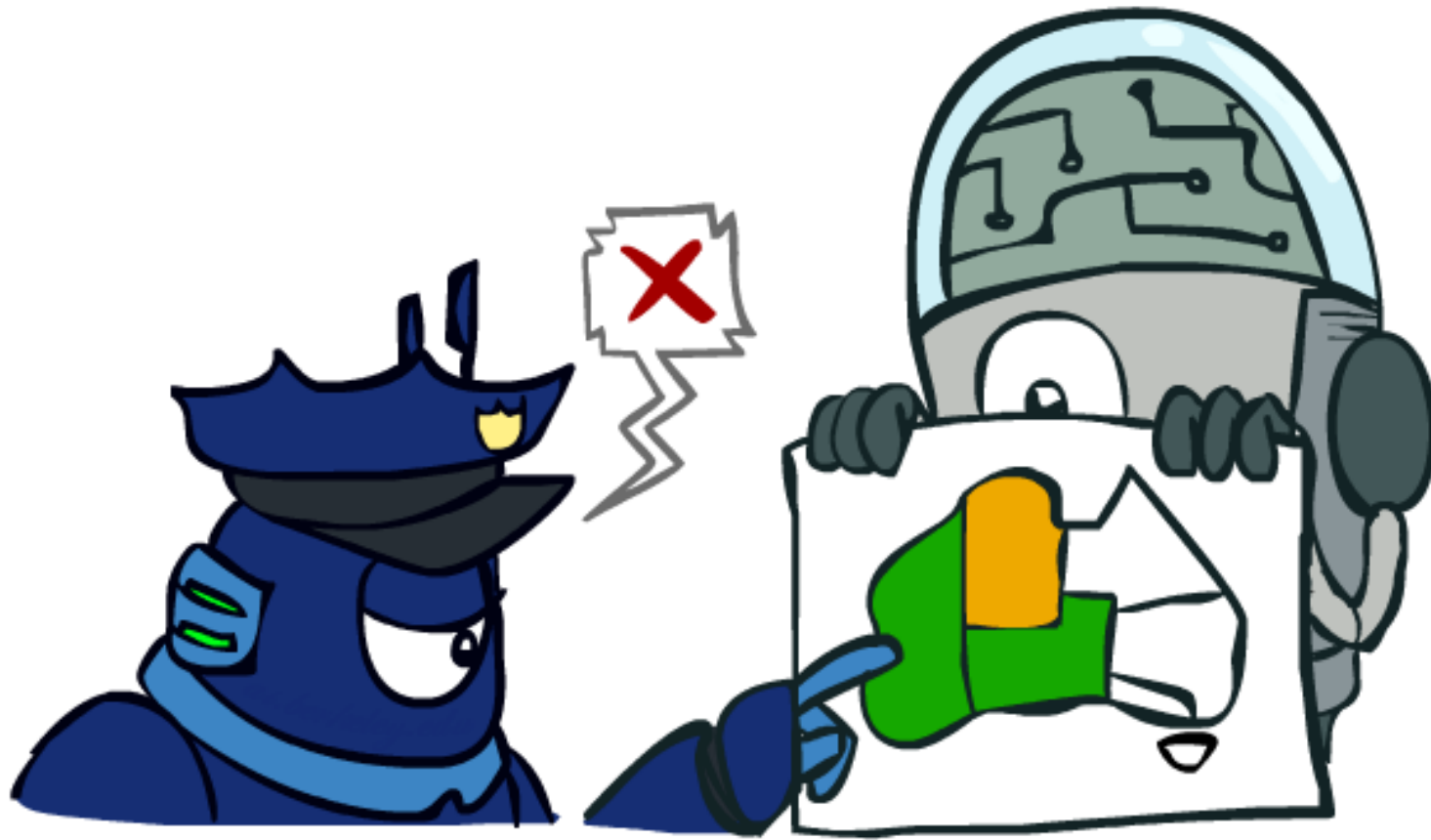


Demo – Naïve Search



15-281: Artificial Intelligence		OH	Schedule	Recitations	Exams	Assignments	Policies	Course Notes
9/5 Thu	Adversarial Search			AIMA Ch. 5.1-2, 5.5				pptx (inked) pdf (inked)
9/10 Tue	Constraint Satisfaction Problems			AIMA Ch. 6.1-3, 6.5 CSP Demo				

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

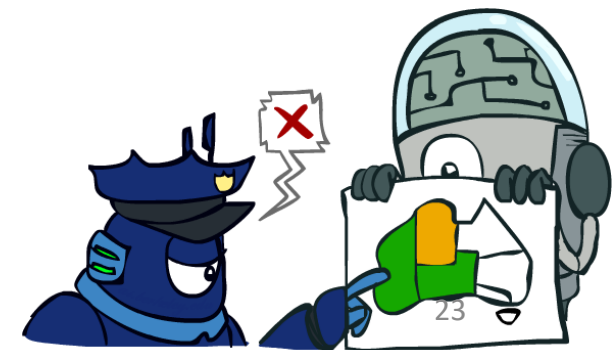
- Variable assignments are commutative
 - [WA = red then NT = green] same as [NT = green then WA = red]
- Only need to consider assign value to a single variable at each step

Idea 2: Check constraints as you go

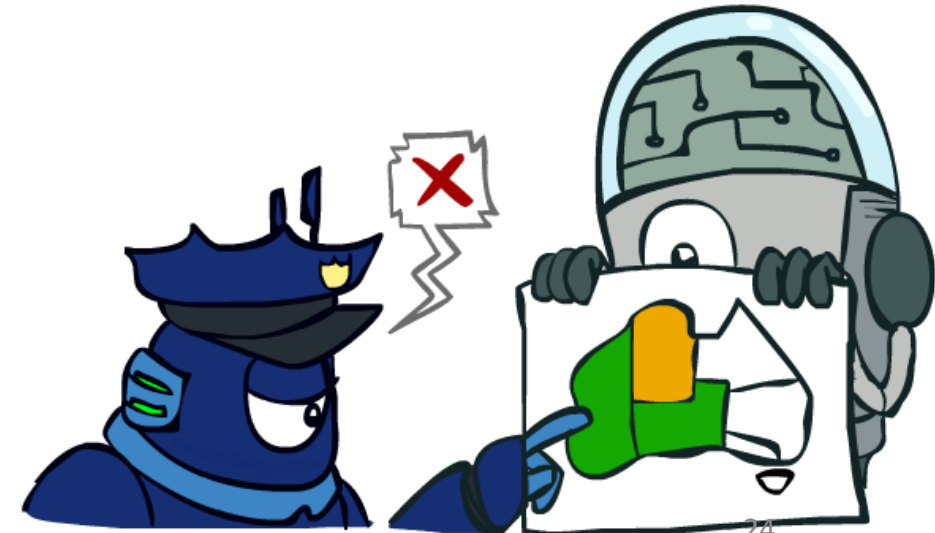
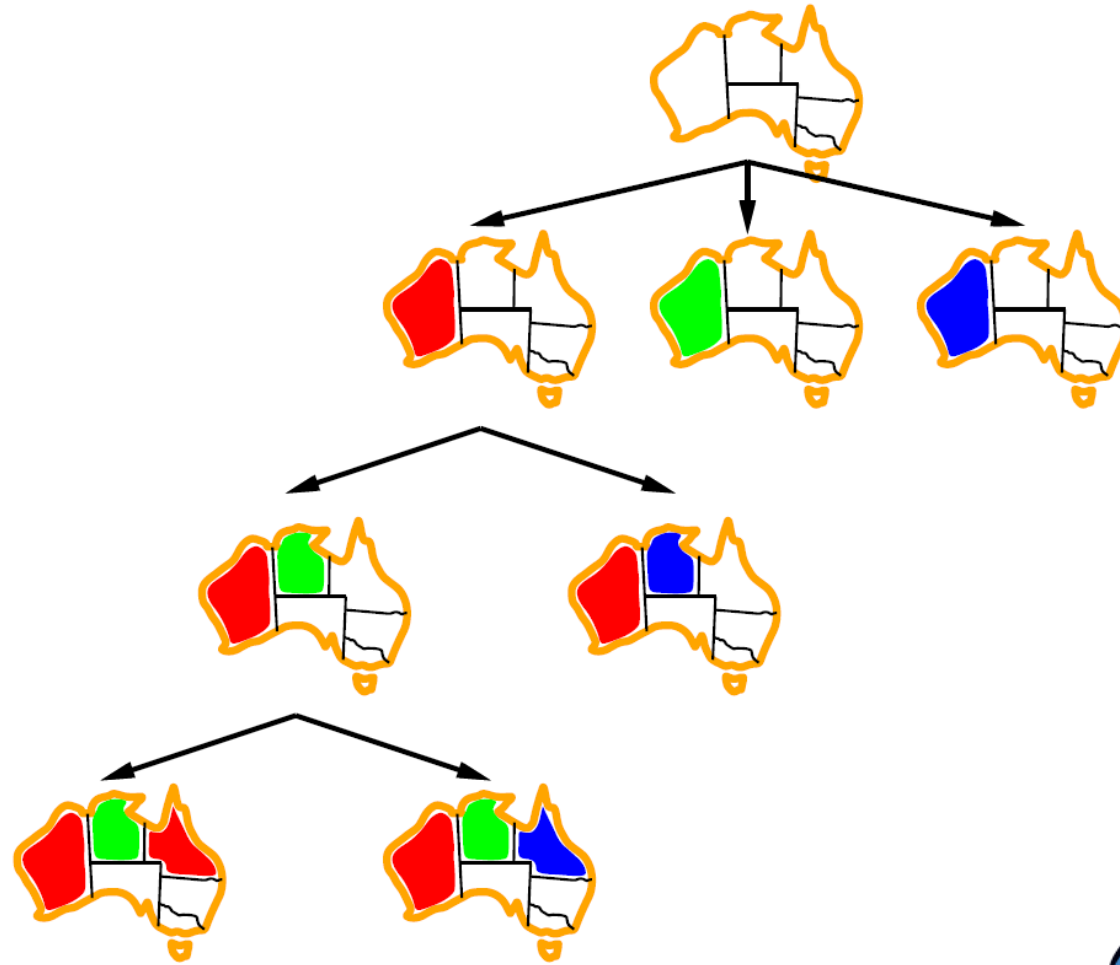
- Consider only values which do not conflict previous assignments
- May need some computation to check the constraints
- “Incremental goal test”

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

Slide/Image credit: Ketrina Yim, UCB CS 188



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
```

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 constraints 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

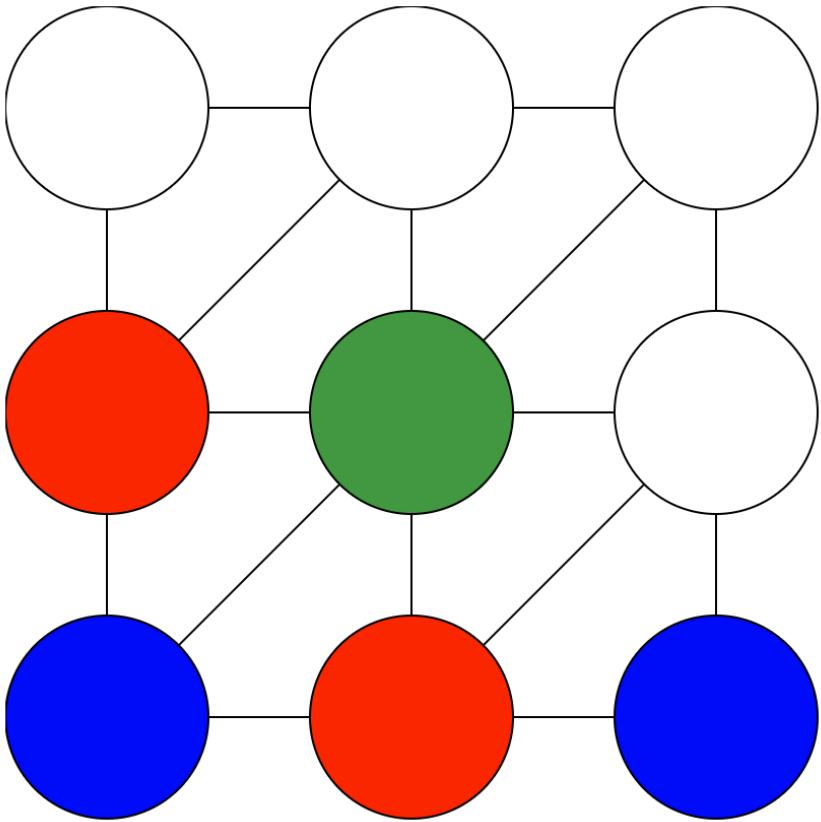
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
  return RECURSIVE-BACKTRACKING({ }, csp)

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 decision points?

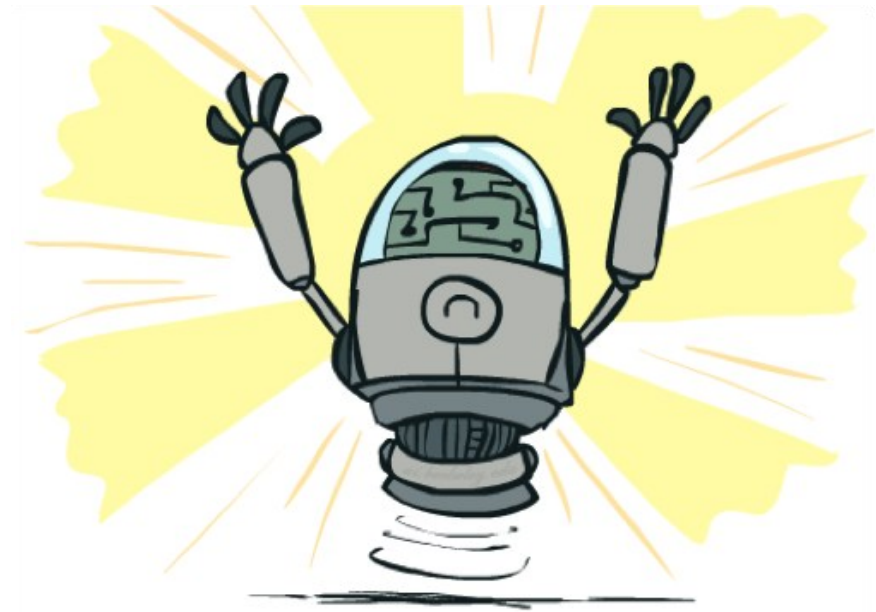
Demo – Backtracking

https://www.cs.cmu.edu/~15281/demos/csp_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

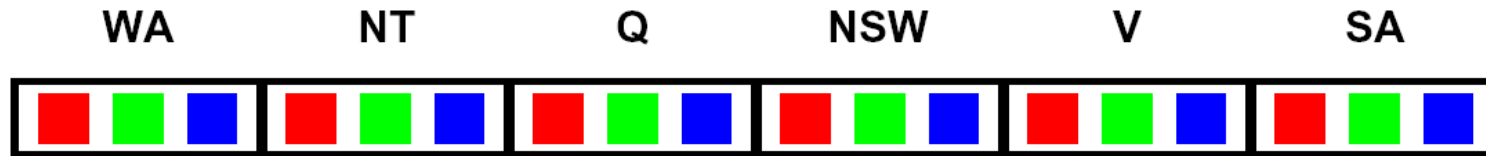
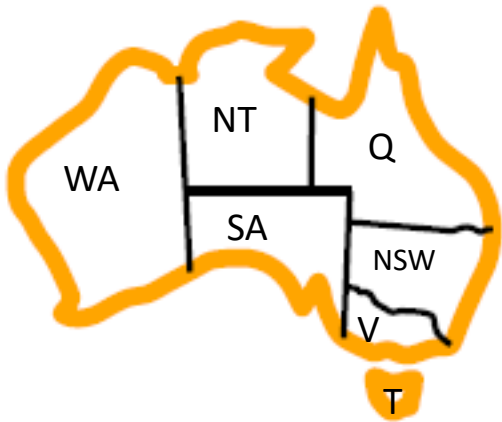
Filtering: Keep track of domains for unassigned variables and cross off bad options

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

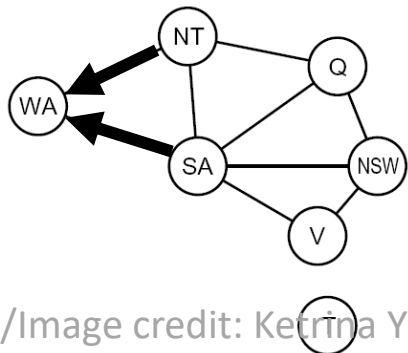
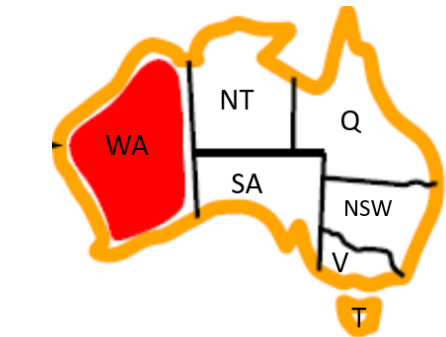
Filtering: Forward Checking

- 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

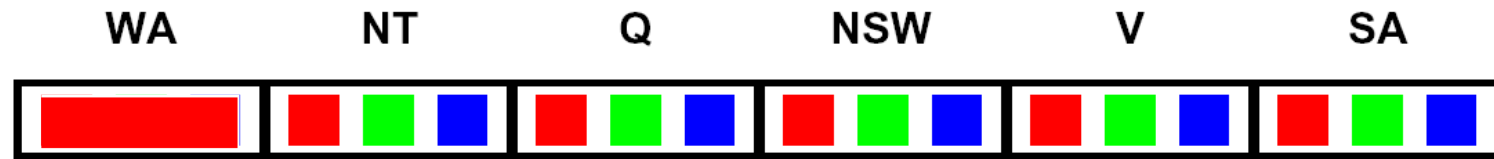


Consistency of A Single Arc

- 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



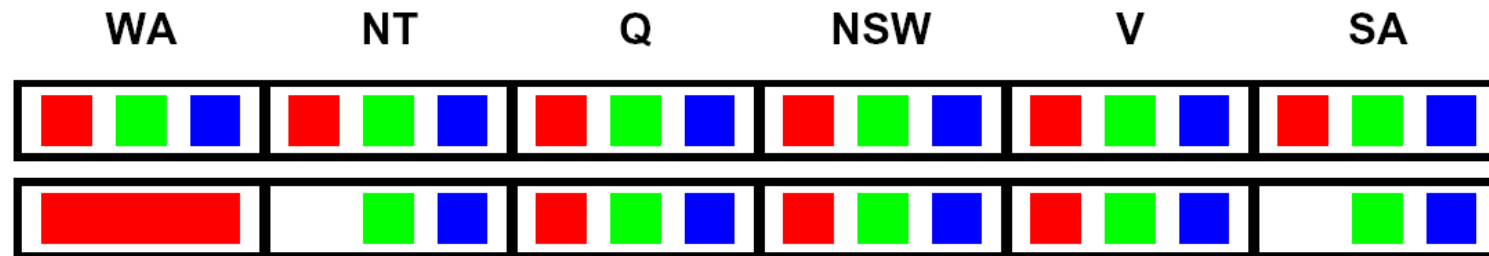
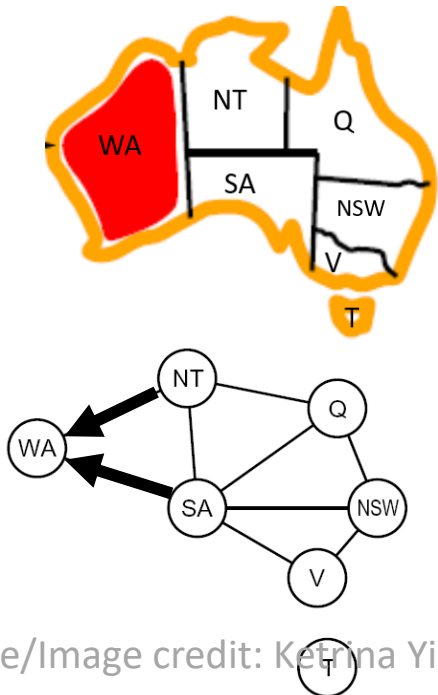
(Remove values from the tail!)



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

Filtering: Forward Checking

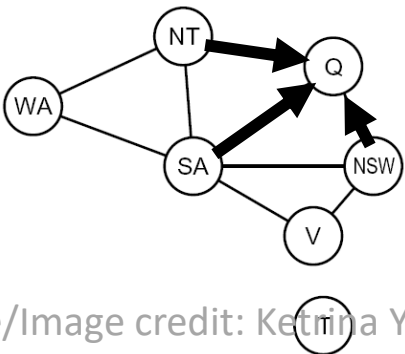
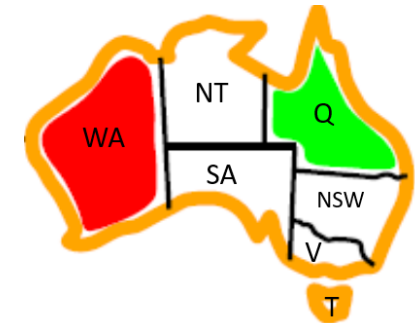
- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: A simple way for filtering
 - After a variable is assigned a value, check related constraints and cross off values of unassigned variables which violate the constraints
 - Failure detected if some 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

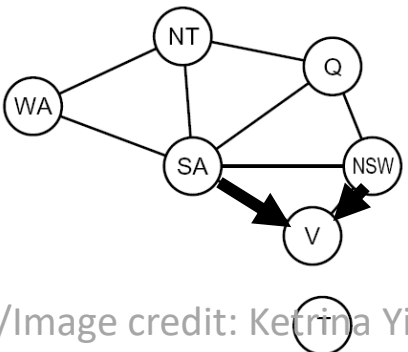
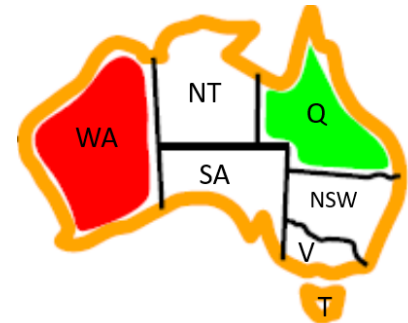
- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: A simple way for filtering
 - After a variable is assigned a value, check related constraints and cross off values of unassigned variables which violate the constraints
 - Failure detected if some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



WA	NT	Q	NSW	V	SA

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

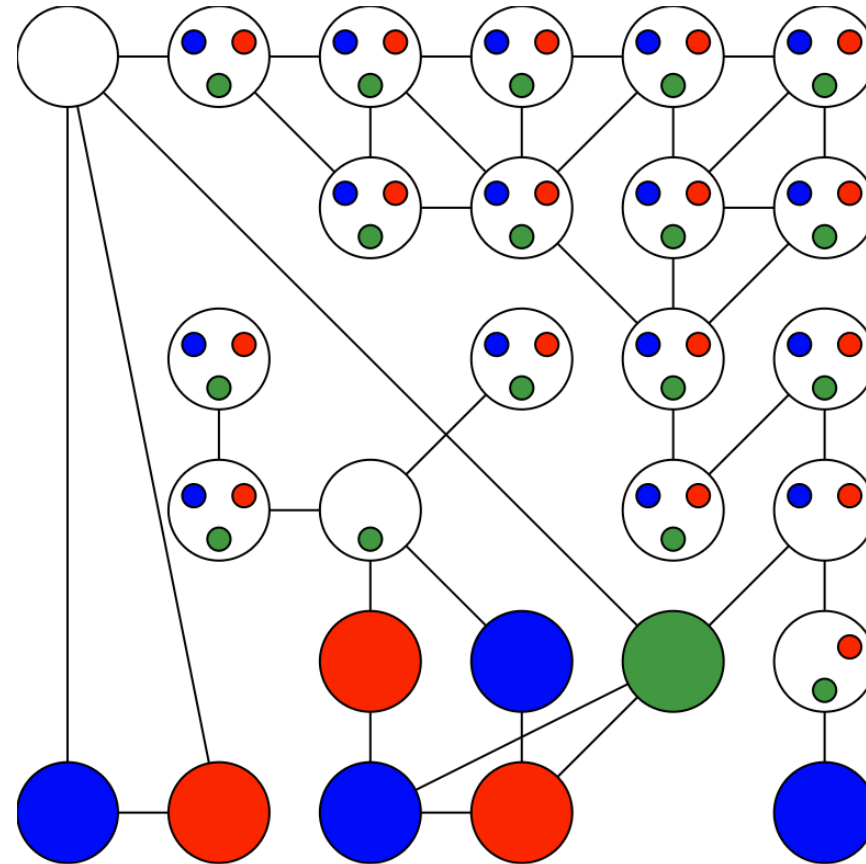
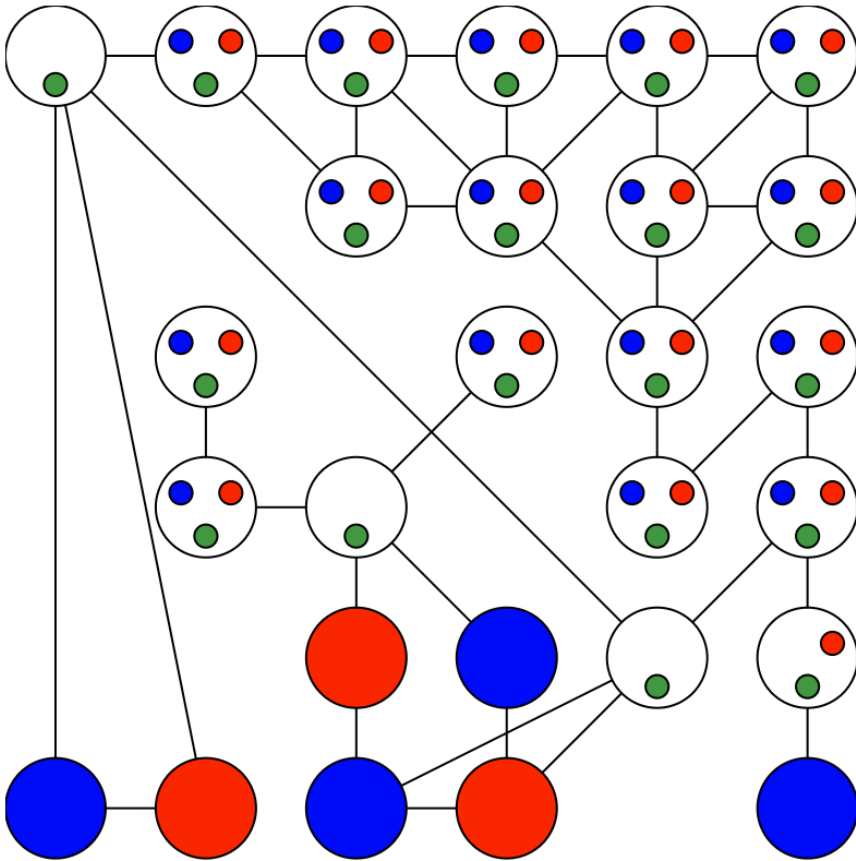


WA	NT	Q	NSW	V	SA

FAIL – variable with no possible values

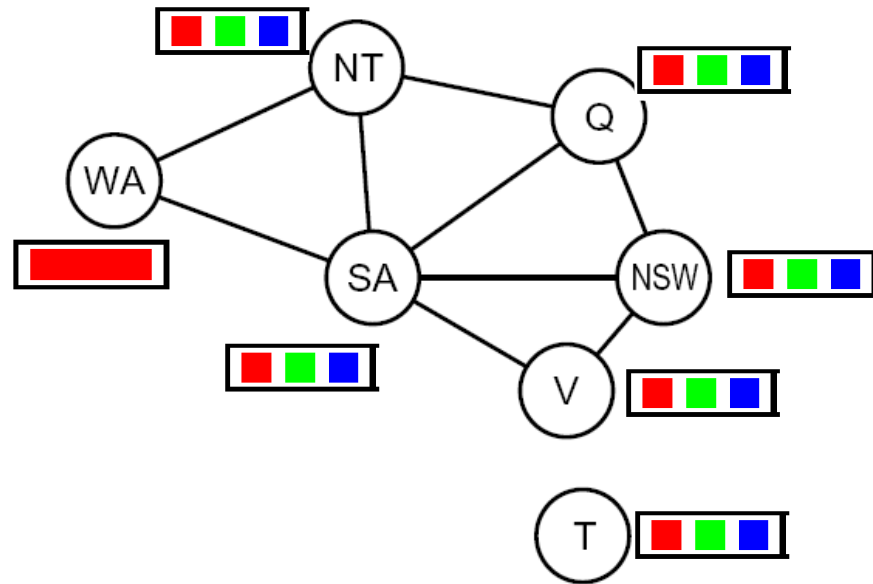
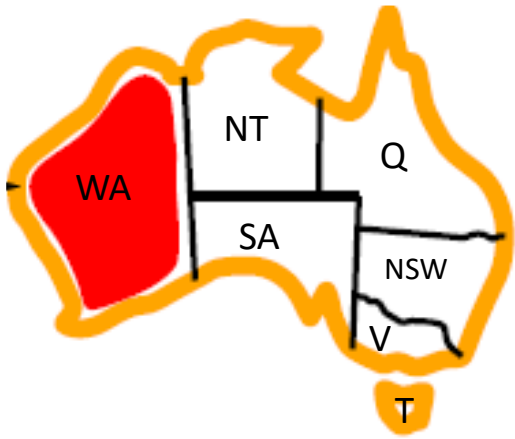
Demo – Backtracking with Forward Checking

https://www.cs.cmu.edu/~15281/demos/csp_backtracking



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

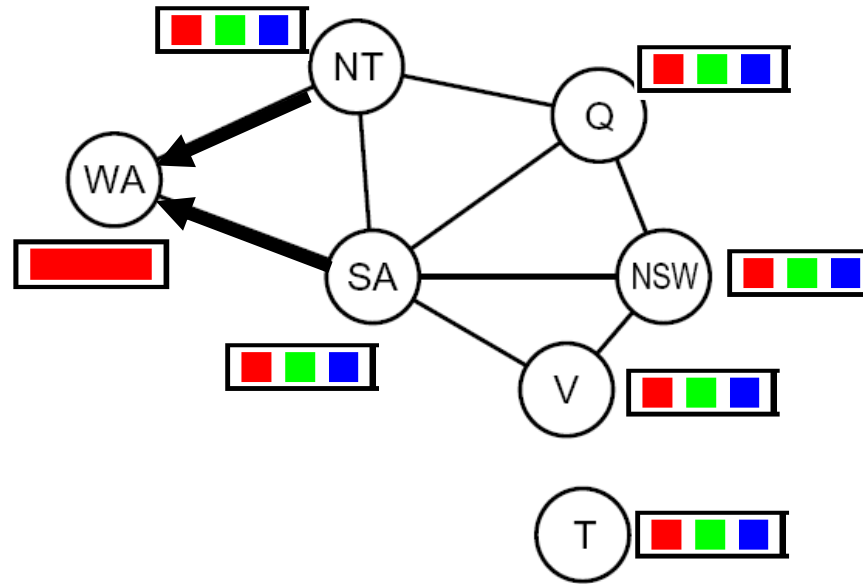
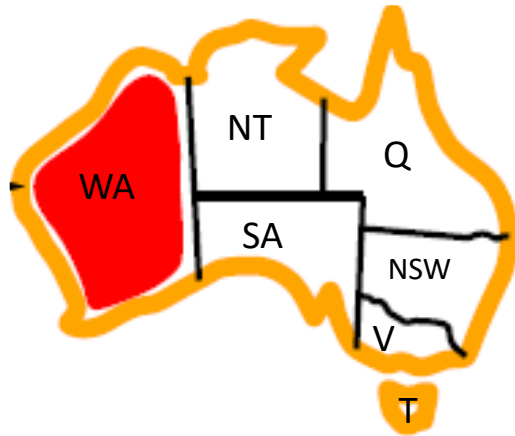
```
function AC-3(csp) returns the CSP, possibly with reduced domains
inputs: csp, a binary CSP with variables  $\{X_1, X_2, \dots, X_n\}$ 
local variables: queue, a queue of arcs, initially all the arcs in csp
while queue is not empty do
     $(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\textit{queue})$ 
    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
```

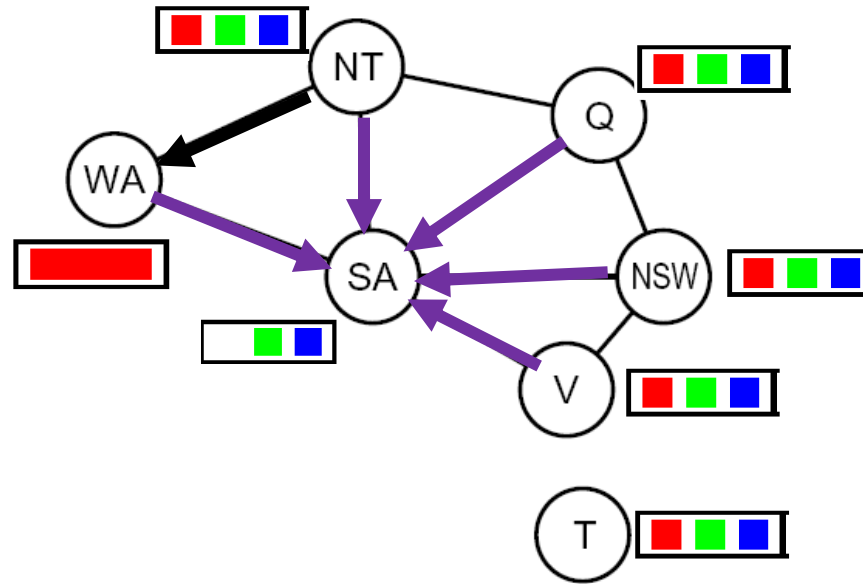
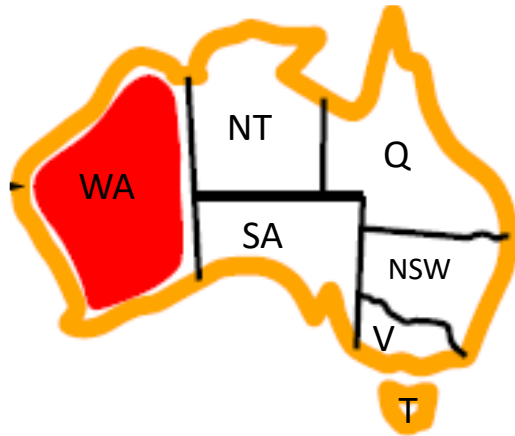
Constraint Propagation!

AC-3: Enforce Arc Consistency of Entire CSP



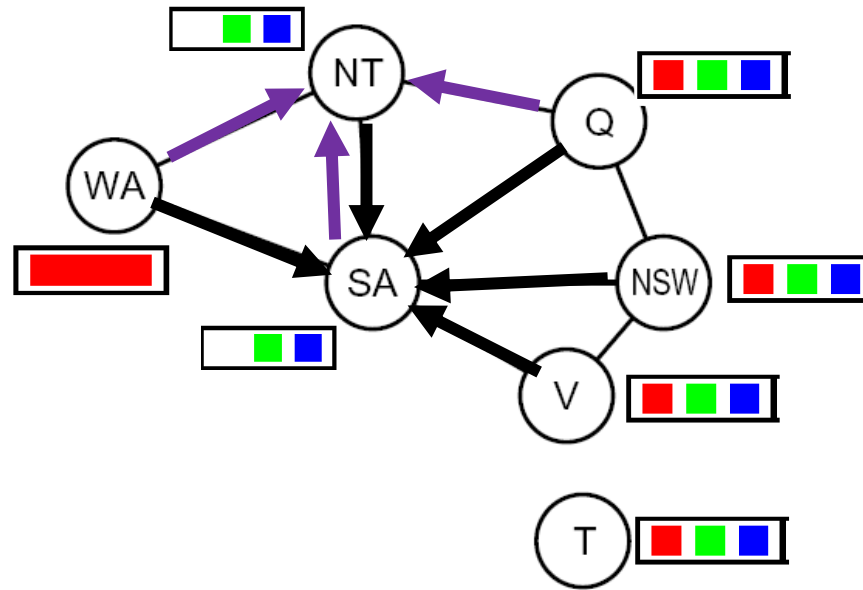
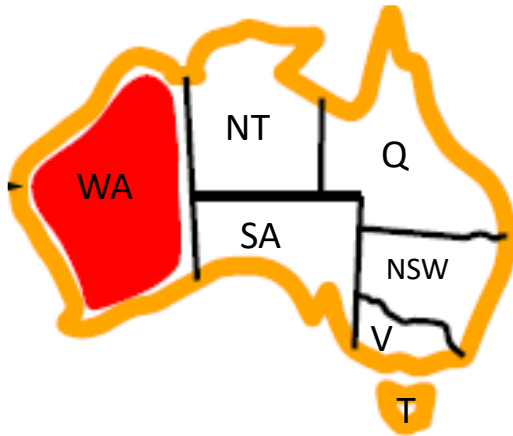
Queue:
SA->WA
NT->WA

AC-3: Enforce Arc Consistency of Entire CSP



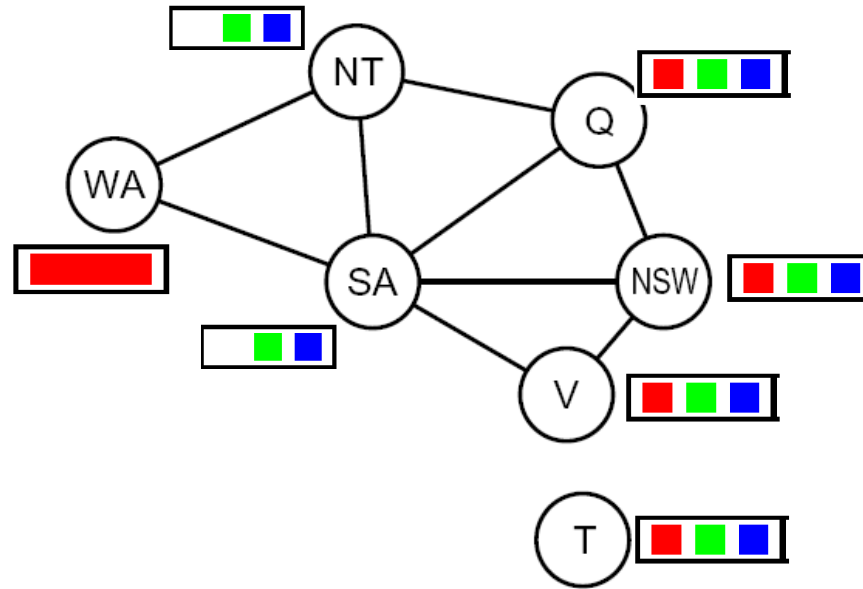
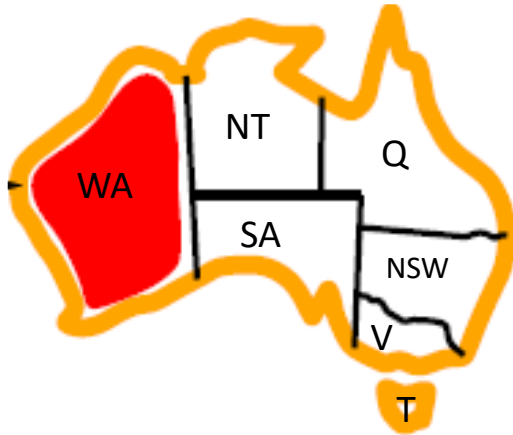
Queue:
NT->WA
WA->SA
NT->SA
Q->SA
NSW->SA
V->SA

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

AC-3: Enforce Arc Consistency of Entire CSP



Queue:

WA->SA

NT->SA

Q->SA

NSW->SA

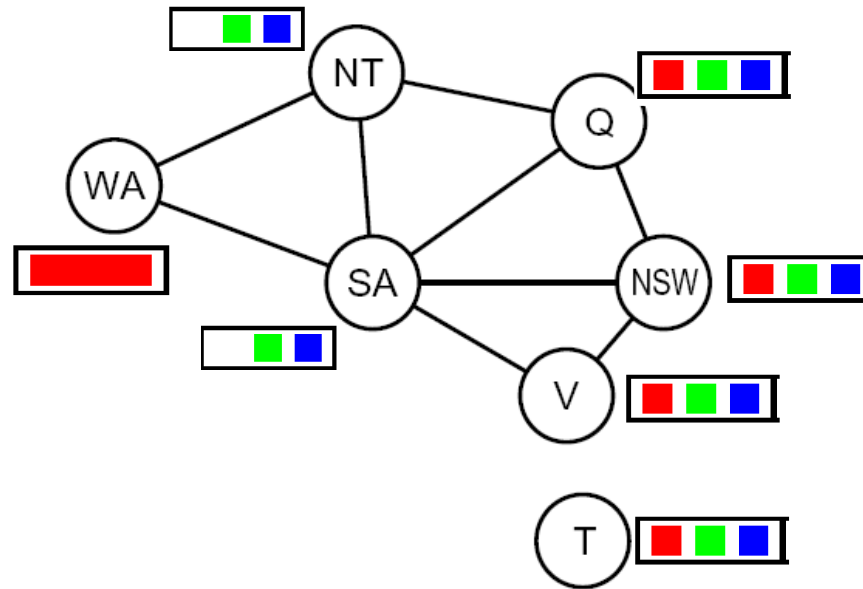
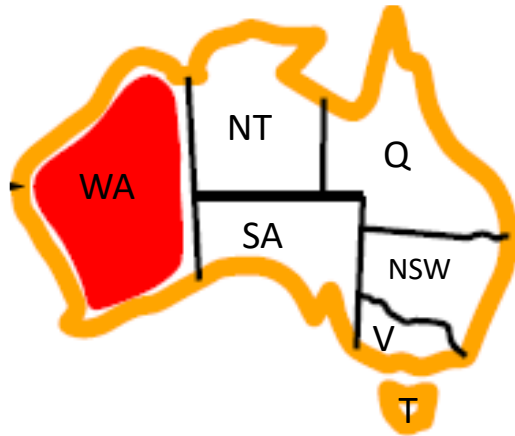
V->SA

WA->NT

SA->NT

Q->NT

AC-3: Enforce Arc Consistency of Entire CSP



Queue:

NT->SA

Q->SA

NSW->SA

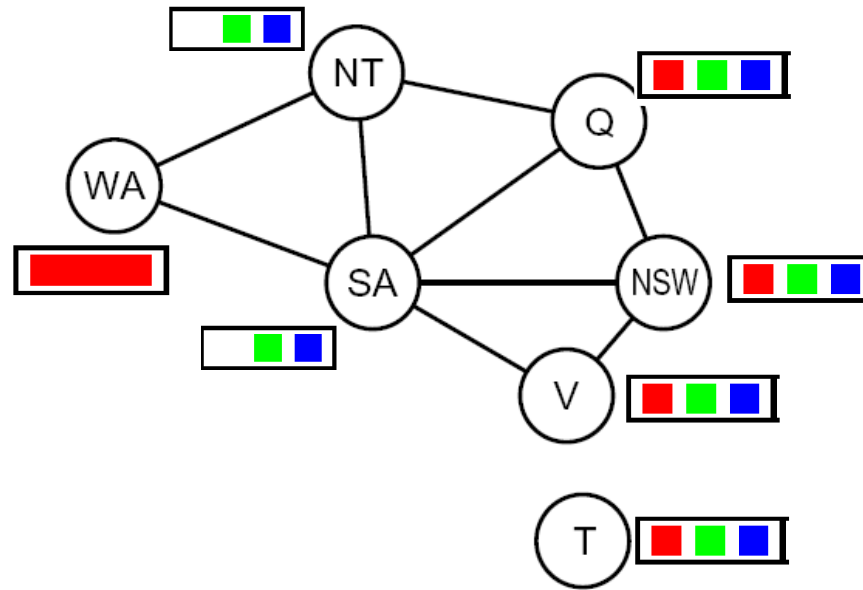
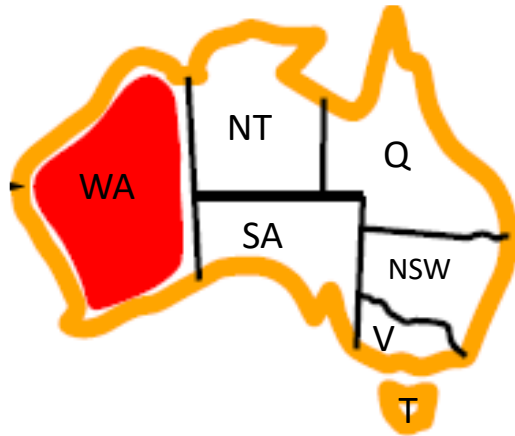
V->SA

WA->NT

SA->NT

Q->NT

AC-3: Enforce Arc Consistency of Entire CSP



Queue:

Q->SA

NSW->SA

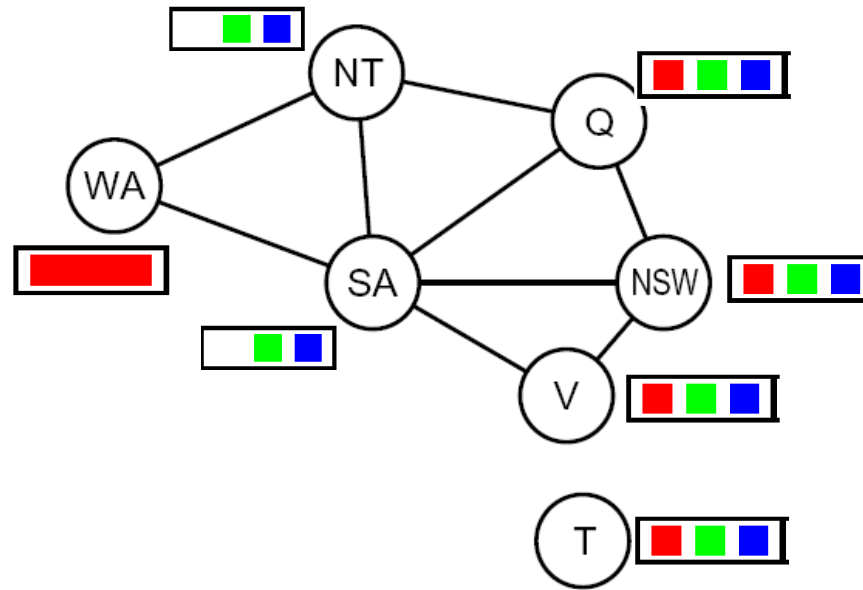
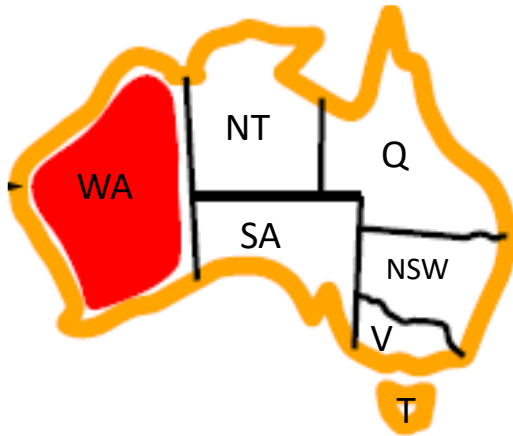
V->SA

WA->NT

SA->NT

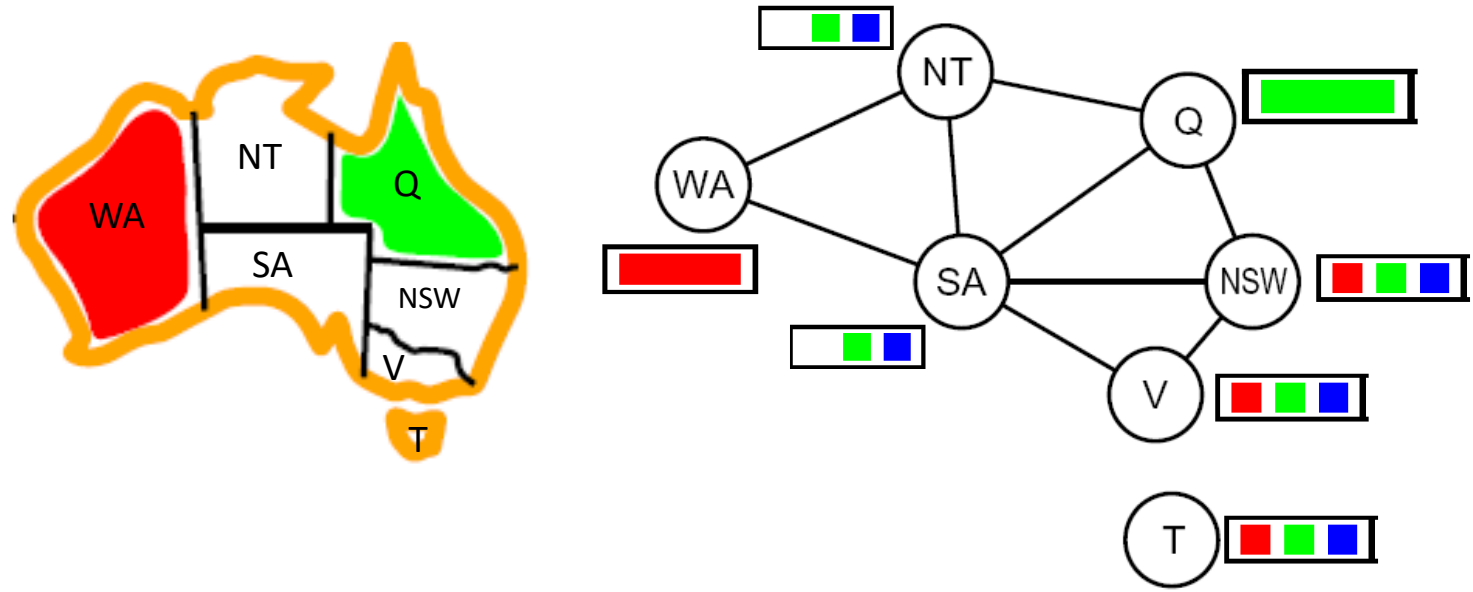
Q->NT

AC-3: Enforce Arc Consistency of Entire CSP



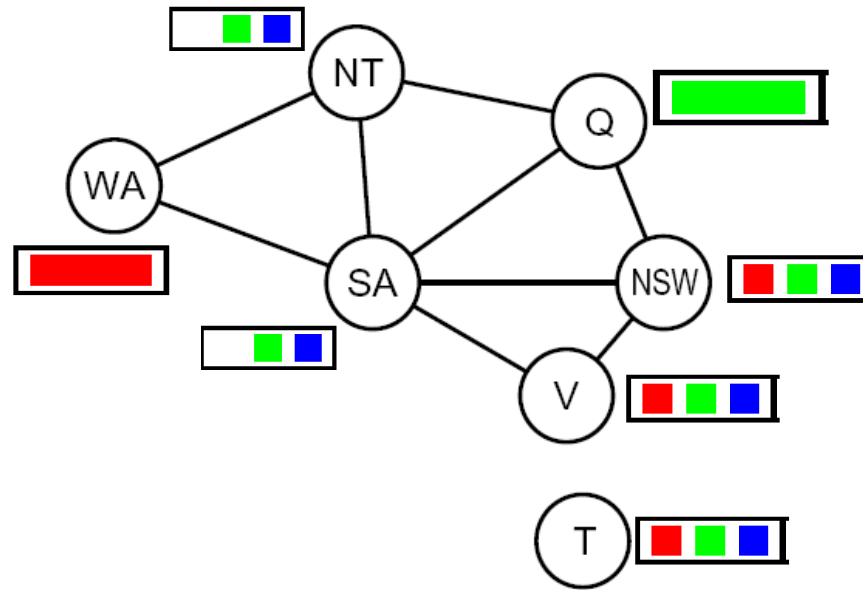
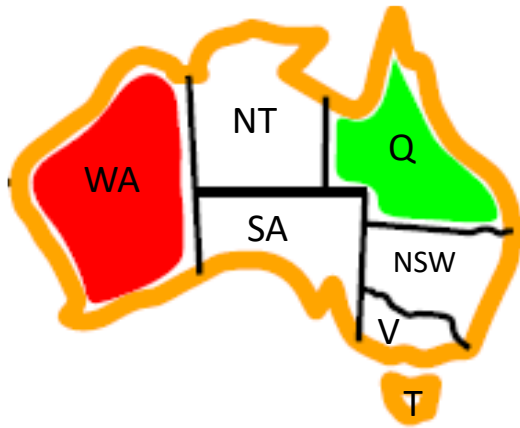
Queue:
NSW->SA
V->SA
WA->NT
SA->NT
Q->NT

AC-3: Enforce Arc Consistency of Entire CSP



Queue:

Poll 2: After assigning Q to Green, what gets added to the Queue?

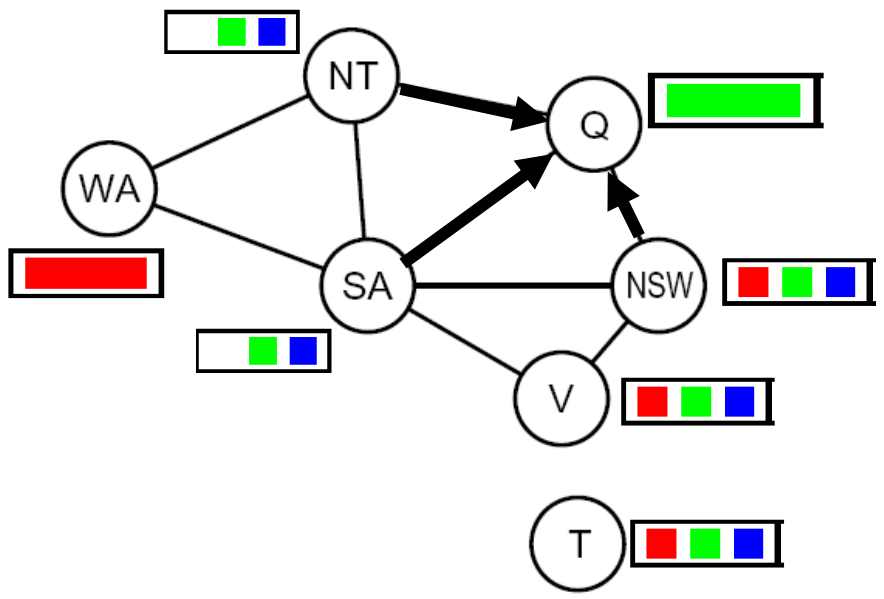
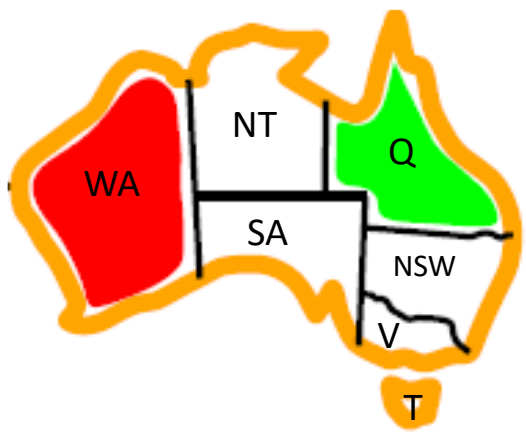


Queue:

A: NSW→Q, SA→Q, NT→Q

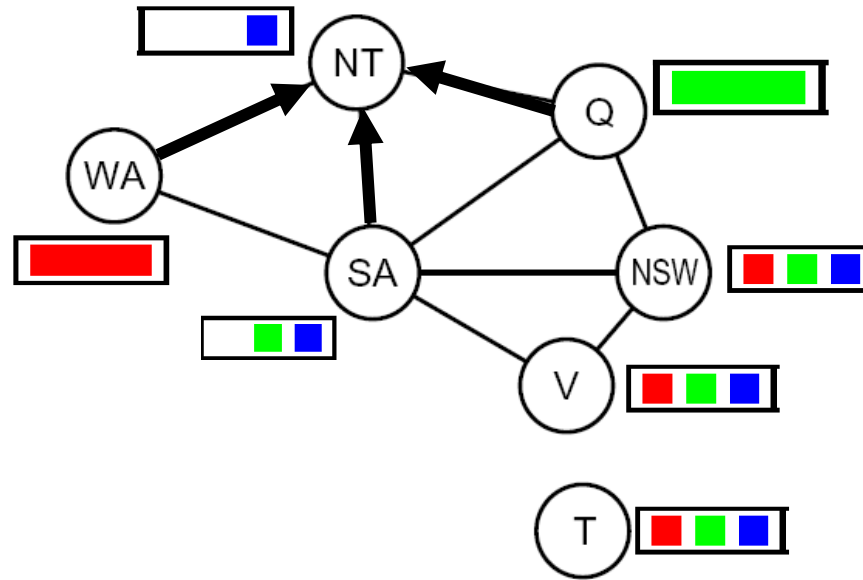
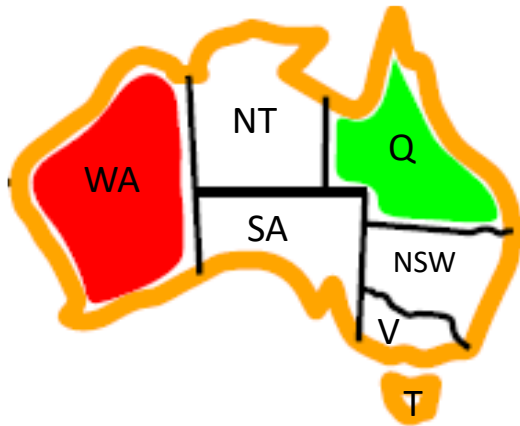
B: Q→NSW, Q→SA, Q→NT

AC-3: Enforce Arc Consistency of Entire CSP



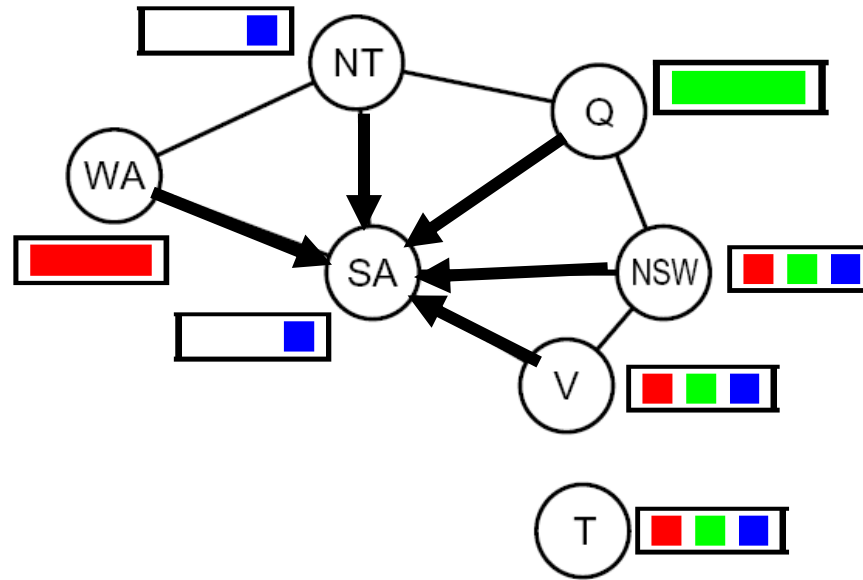
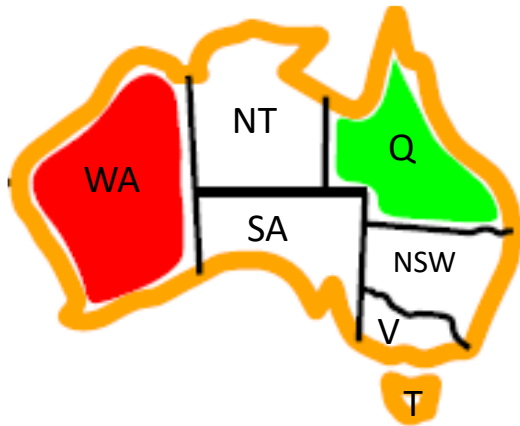
Queue:
NT->Q
SA->Q
NSW->Q

AC-3: Enforce Arc Consistency of Entire CSP



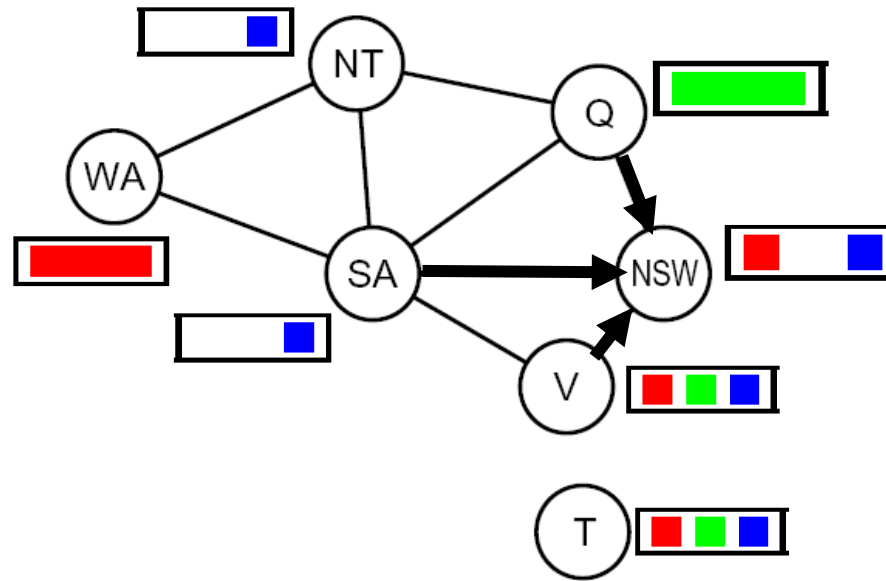
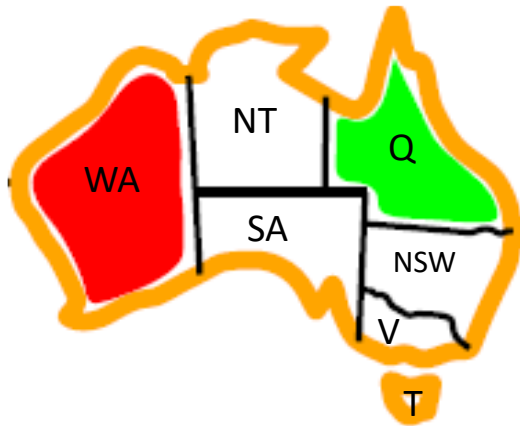
Queue:
SA->Q
NSW->Q
WA->NT
SA->NT
Q->NT

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

AC-3: Enforce Arc Consistency of Entire CSP



Queue:

WA->NT

SA->NT

Q->NT

WA->SA

NT->SA

Q->SA

NSW->SA

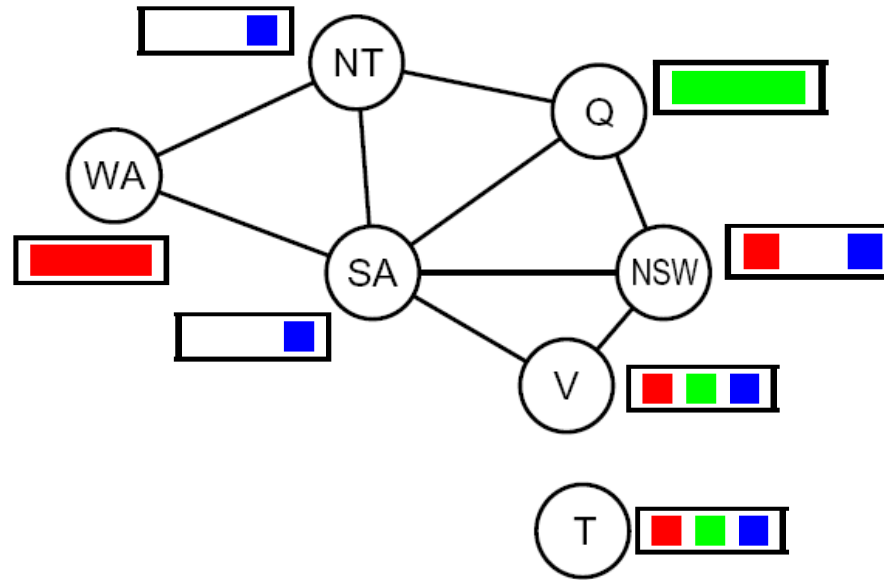
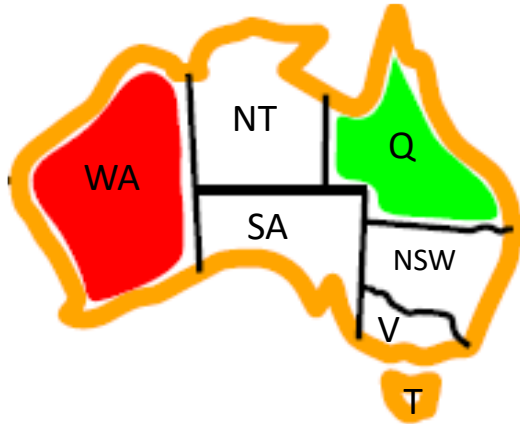
V->SA

V->NSW

Q->NSW

SA->NSW

AC-3: Enforce Arc Consistency of Entire CSP



Queue:

WA->NT

SA->NT

Q->NT

WA->SA

NT->SA

Q->SA

NSW->SA

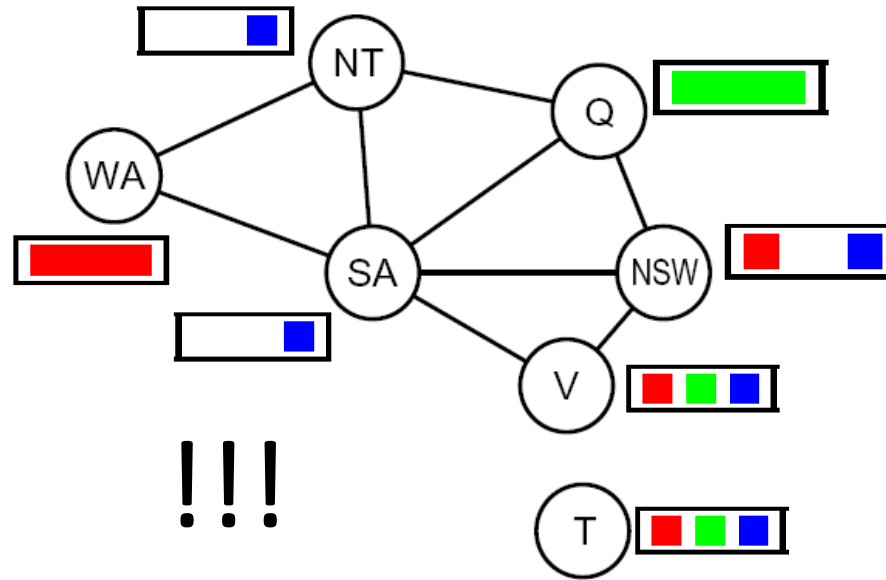
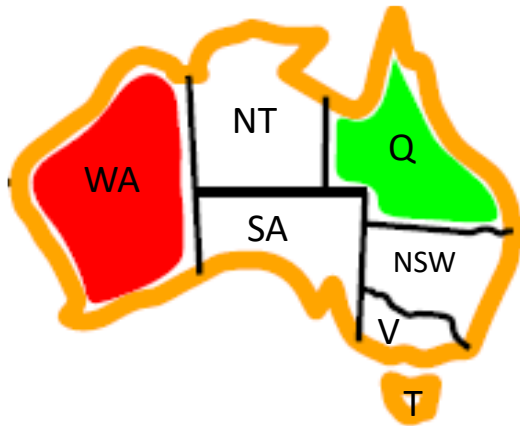
V->SA

V->NSW

Q->NSW

SA->NSW

AC-3: Enforce Arc Consistency of Entire CSP



Queue:

SA->NT

Q->NT

WA->SA

NT->SA

Q->SA

NSW->SA

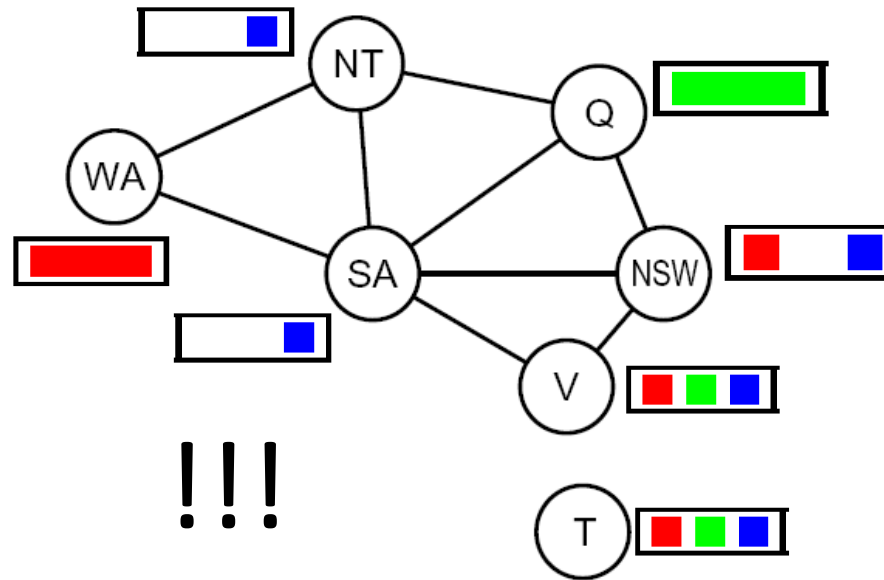
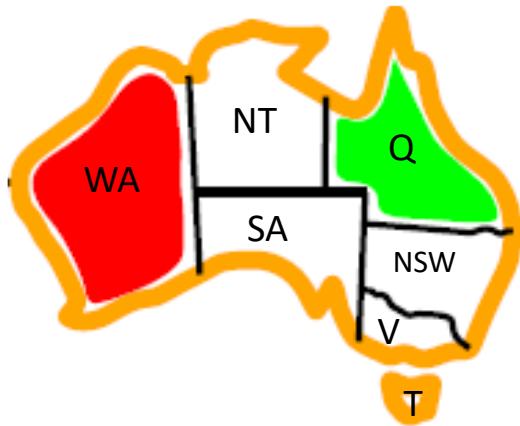
V->SA

V->NSW

Q->NSW

SA->NSW

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

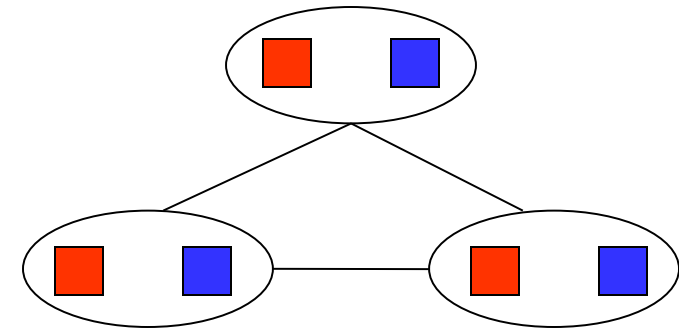
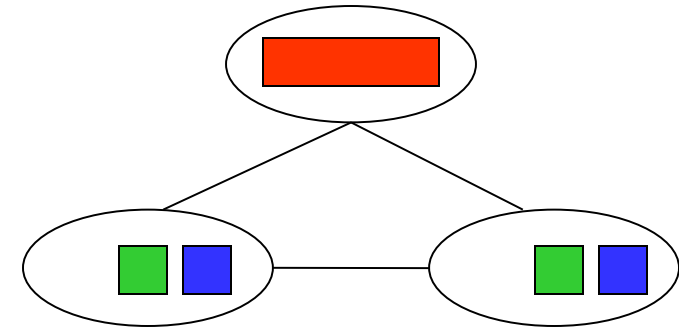
SA->NSW

- 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?

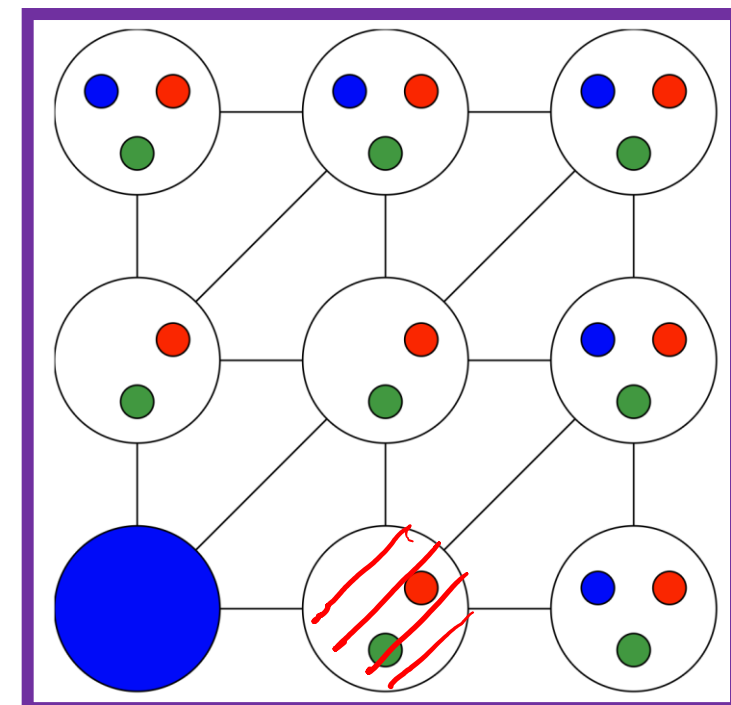
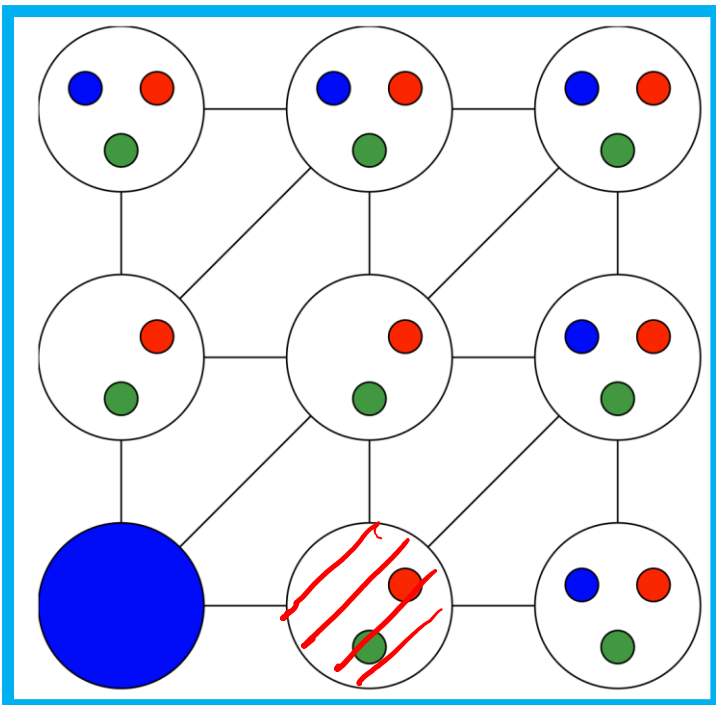
Demo – Backtracking with AC-3

Compare

- Backtracking with Forward Checking
- Backtracking with AC-3

Forward checking only check arcs connecting variables a variable that we just assigned.

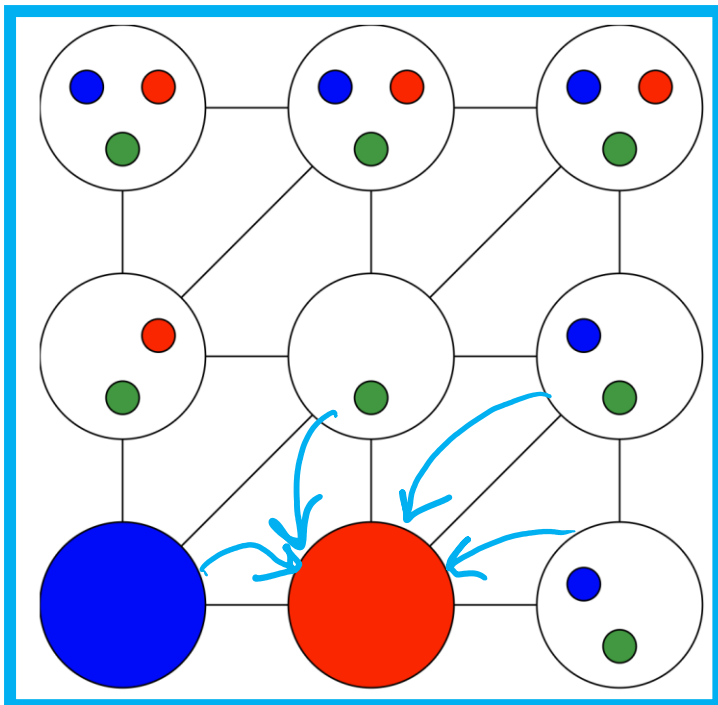
With AC-3, we make sure all arcs are consistent



Demo – Backtracking with AC-3

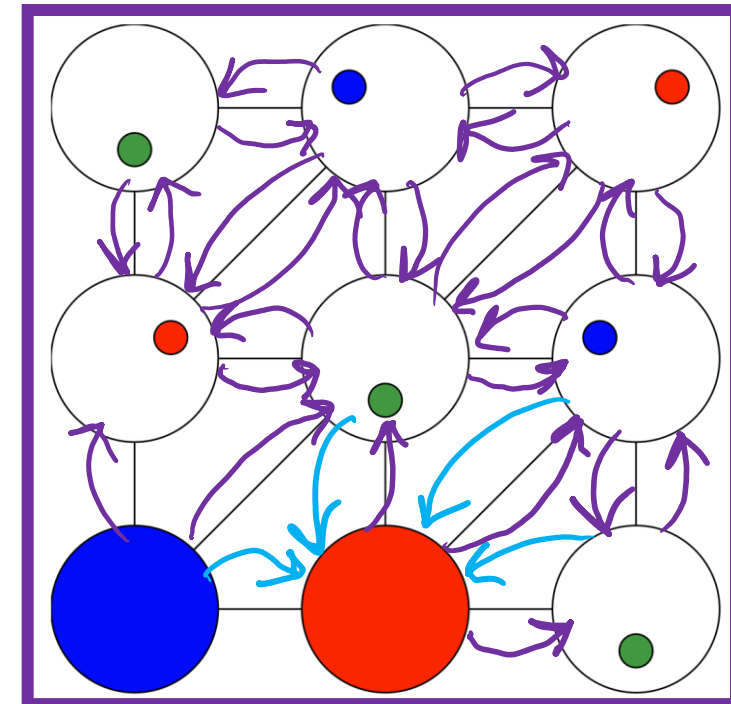
Compare

- Backtracking with Forward Checking
- Backtracking with AC-3



Forward checking only check arcs connecting variables a variable that we just assigned.

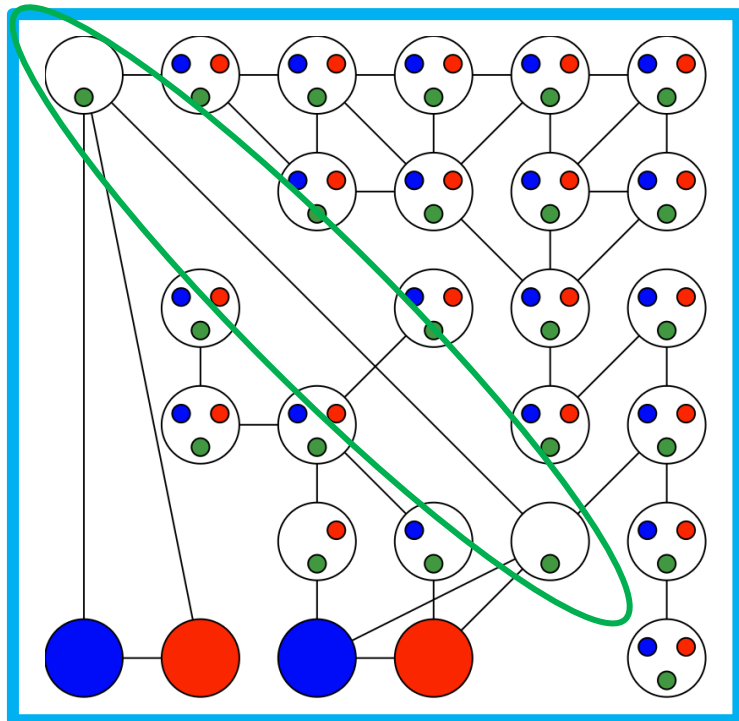
With AC-3, we make sure all arcs are consistent



Demo – Backtracking with AC-3

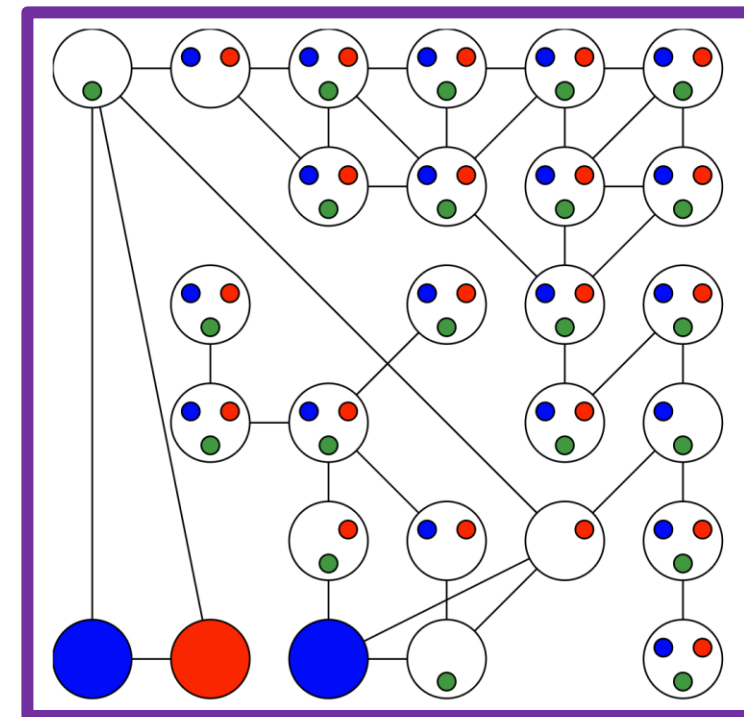
Compare

- Backtracking with Forward Checking
- Backtracking with AC-3



Forward checking only check arcs connecting variables a variable that we just assigned.

With AC-3, we can find existing problems, such as the arc between these two variables with only green left.



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  $\{X_1, X_2, \dots, X_n\}$ 
local variables: queue, a queue of arcs, initially all the arcs in csp

while queue is not empty do
     $(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\textit{queue})$ 
    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
```

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  $\{X_1, X_2, \dots, X_n\}$ 
local variables: queue, a queue of arcs, initially all the arcs in csp
while queue is not empty do
     $(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\textit{queue})$ 
    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
```

- 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  $\{X_1, X_2, \dots, X_n\}$ 
local variables: queue, a queue of arcs, initially all the arcs in csp
while queue is not empty do
     $(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\textit{queue})$ 
    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
```

- 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: $O(n^2d)$

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  $\{X_1, X_2, \dots, X_n\}$ 
local variables: queue, a queue of arcs, initially all the arcs in csp
while queue is not empty do
     $(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\textit{queue})$ 
    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
```

- 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: $O(n^2d)$
- Check arc consistency per arc: $O(d^2)$

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

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

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)
      if result ≠ failure then return result
      remove {var = value} from assignment
  return failure
```

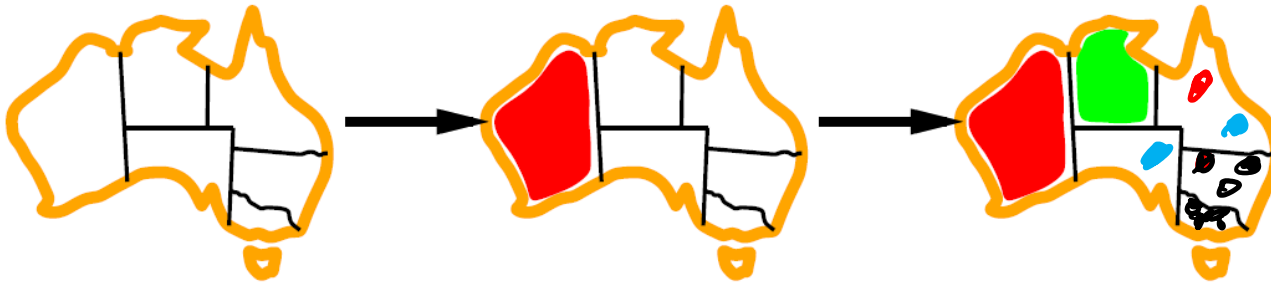
- Where do you run AC-3?

Ordering

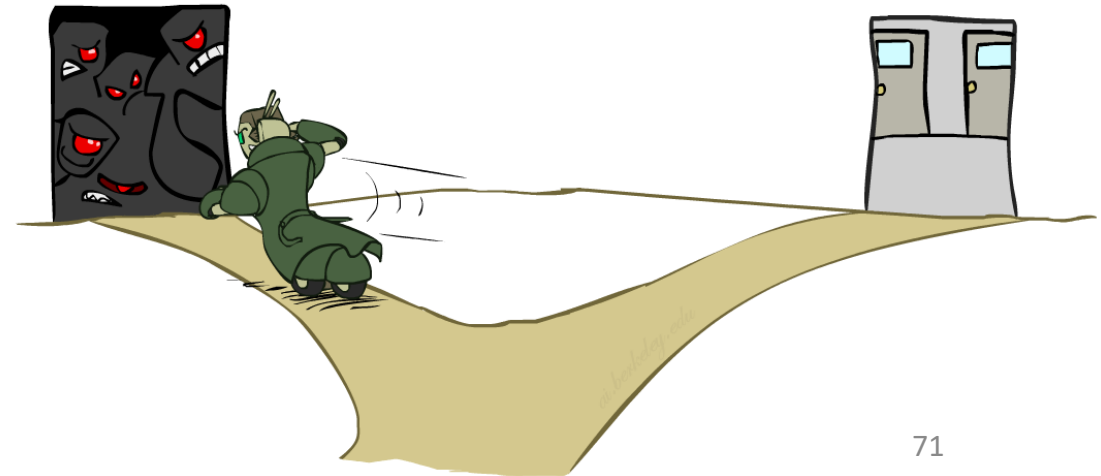


Ordering: Minimum Remaining Values

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



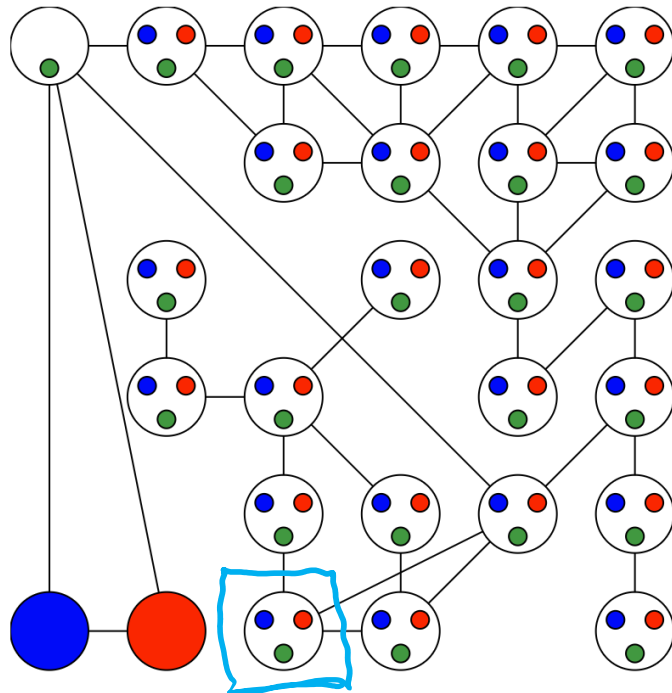
- Why min rather than max?
- Also called “most constrained variable”
- “Fail-fast” ordering



Demo – Coloring with a Complex Graph

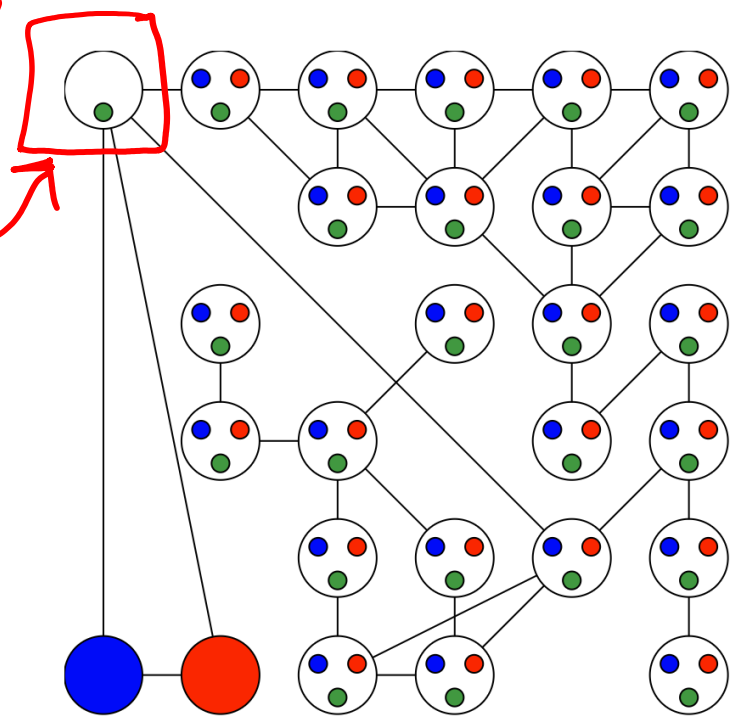
Compare

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



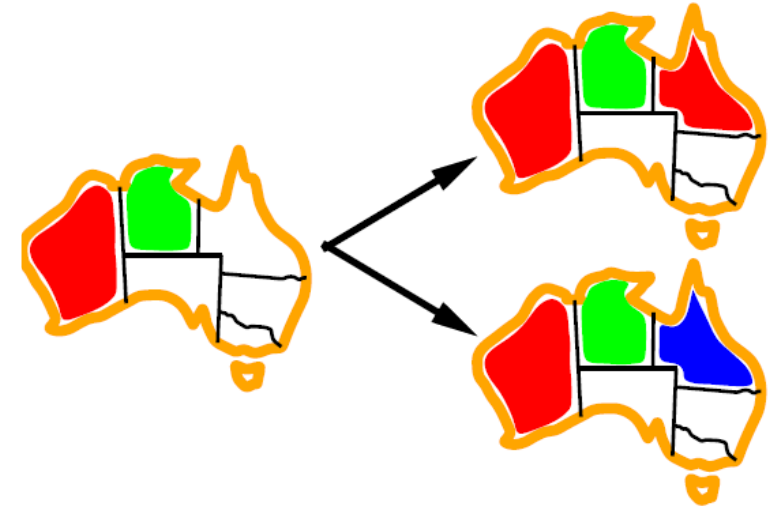
next variable
in default
ordering

next variable
with MRV



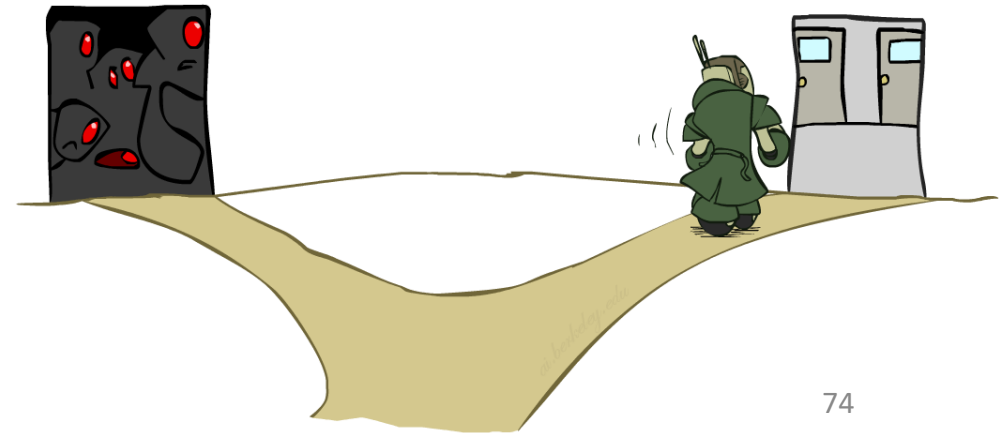
Ordering: Least Constraining Value

- 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)



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

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)

