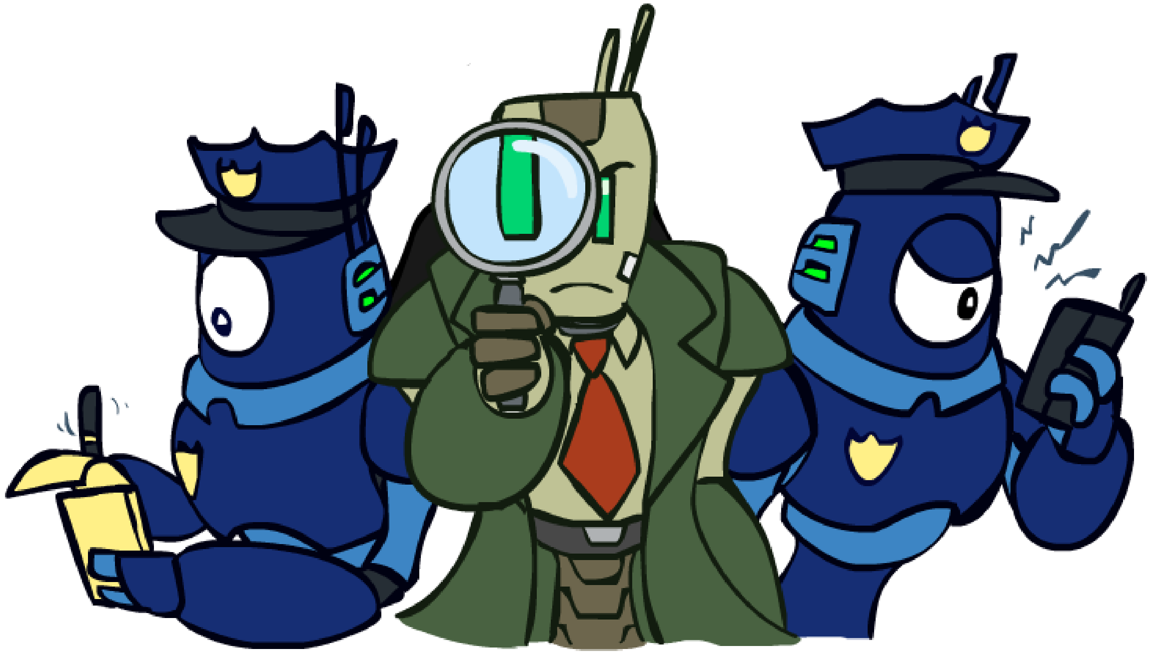
Introduction to Artificial Intelligence

Constraint Satisfaction Problems



Anish Raj

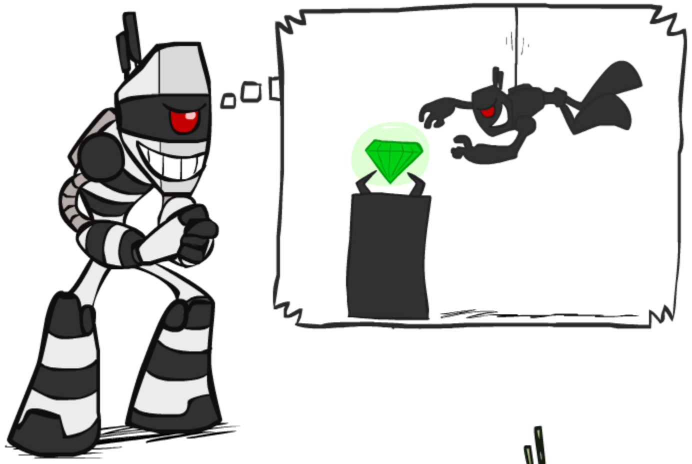
Motivation

- One of the major goals of AI is to help humans in solving complex tasks
  - Assignment Problems:
    - Who teaches what class?
  - Timetabling Problems:
    - Which class is offered when and where?
  - Transportation Scheduling
  - Job Scheduling
  - Factory resource scheduling
  - Floor Planning

What is Search For?

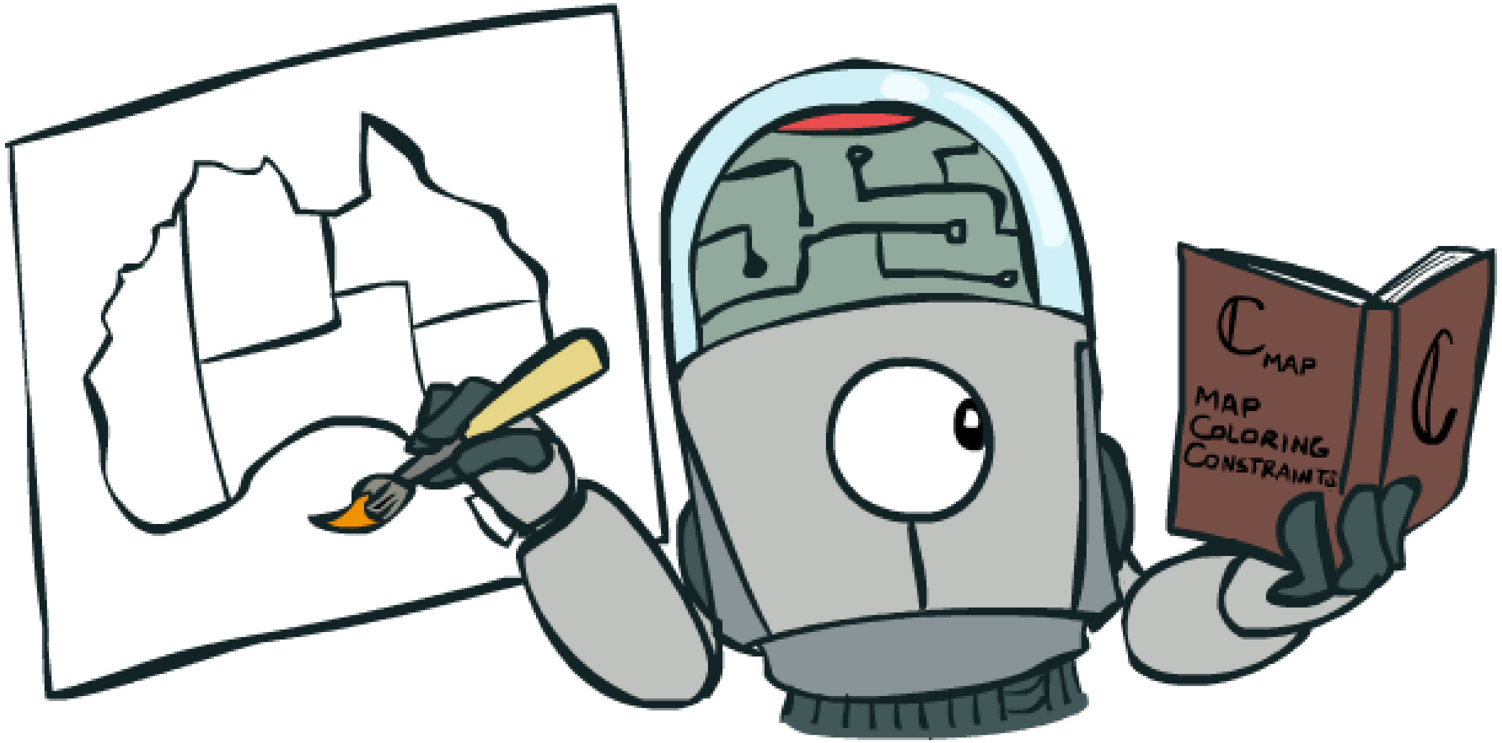
- Assumptions about the world: a single agent, deterministic actions, fully observed state, discrete state space

- 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)
  - CSPs are a specialized class of identification problems

Constraint Satisfaction Problems

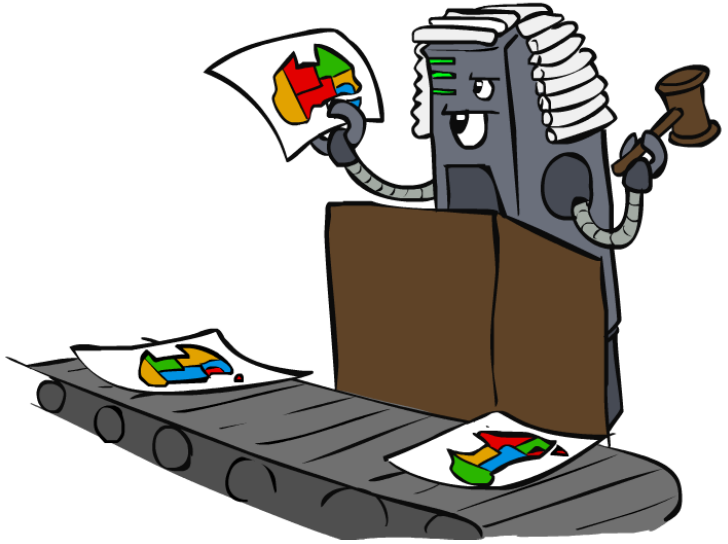


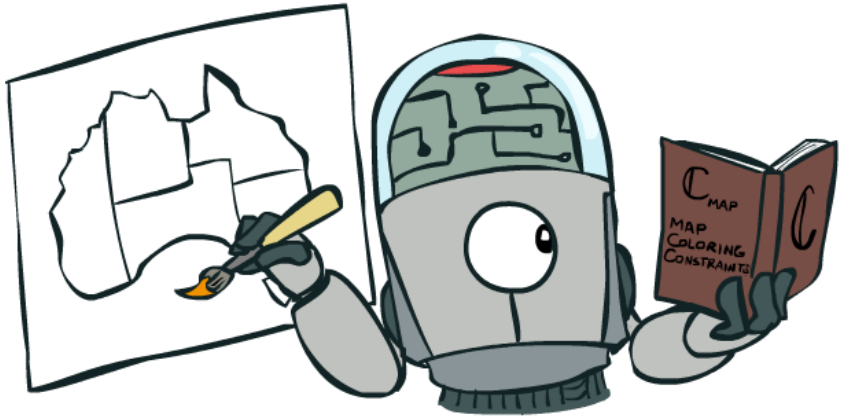
Constraint Satisfaction Problems

- 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

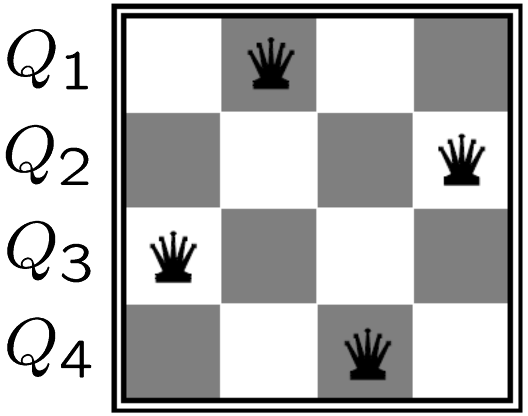
- Constraint satisfaction problems (CSPs):
  - A special subset of search problems
  - State is defined by variables X_i with values from a domain D (sometimes D depends on i)
  - Goal test is a set of constraints specifying allowable combinations of values for subsets of variables

- Allows useful general-purpose algorithms with more power than standard search algorithms



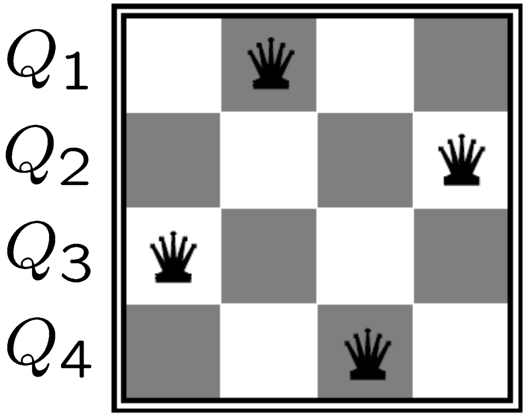


Example: N-Queens



Example: N-Queens

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



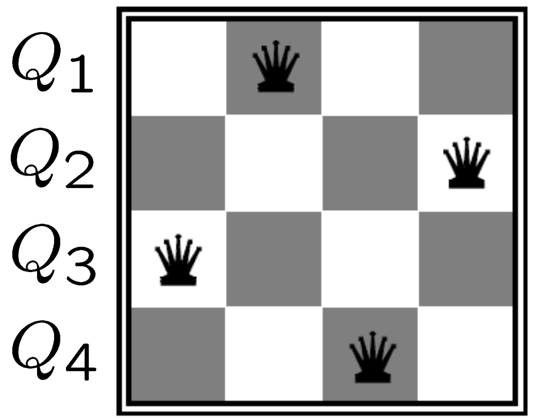
Implicit: $\forall i, j \;\; \text{non-threatening}(Q_i, Q_j)$

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

$\cdots$

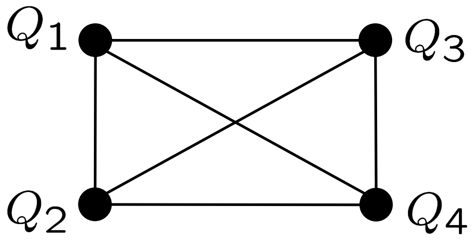
Example: N-Queens

- Formulation :
  - Variables: Q_k
  - Domains: $\{1, 2, 3, \ldots N\}$
  - Constraints: There are $C(4,2) = 6$ constraints involved



$$R_{12} = \{(1,3)(1,4)(2,4)(3,1)(4,1)(4,2)\}$$
$$R_{13} = \{(1,2)(1,4)(2,1)(2,3)(3,2)(3,4)(4,1)(4,3)\}$$
$$R_{14} = \{(1,2)(1,3)(2,1)(2,3)(2,4)(3,1)(3,2)(3,4)(4,2)(4,3)\}$$
$$R_{23} = \{(1,3)(1,4)(2,4)(3,1)(4,1)(4,2)\}$$
$$R_{24} = \{(1,2)(1,4)(2,1)(2,3)(3,2)(3,4)(4,1)(4,3)\}$$
$$R_{34} = \{(1,3)(1,4)(2,4)(3,1)(4,1)(4,2)\}$$

Constraint Graph



Example: Cryptarithmetic problems

- Find numeric substitutions that make an equation hold:

$$
\begin{array}{rcccc}
 & & T & W & O \\
+ & & T & W & O \\
\hline
= & F & O & U & R
\end{array}
$$

For example:

$O = 4$
$R = 8$
$W = 3$
$U = 6$
$T = 7$
$F = 1$

$$
\begin{array}{rcccc}
 & & 7 & 3 & 4 \\
+ & & 7 & 3 & 4 \\
\hline
= & 1 & 4 & 6 & 8
\end{array}
$$

Note: not unique – how many solutions?

Example: Cryptarithmetic

- Variables:

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

- Domains:

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

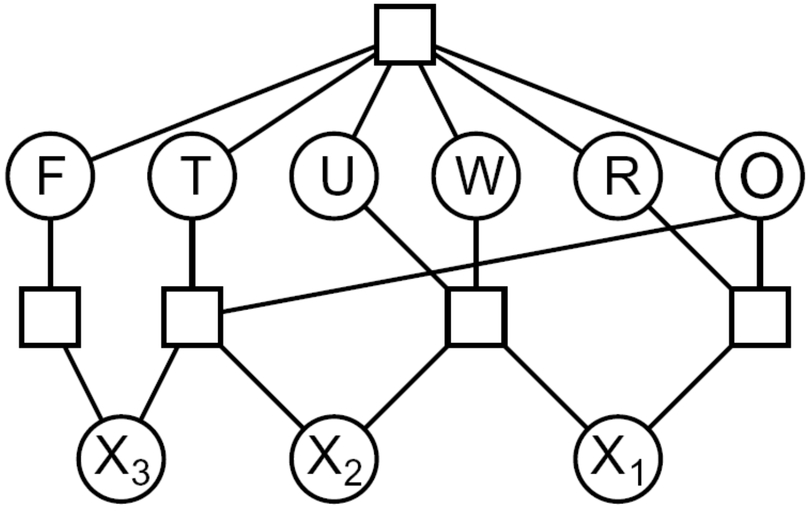
- Constraints:

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

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

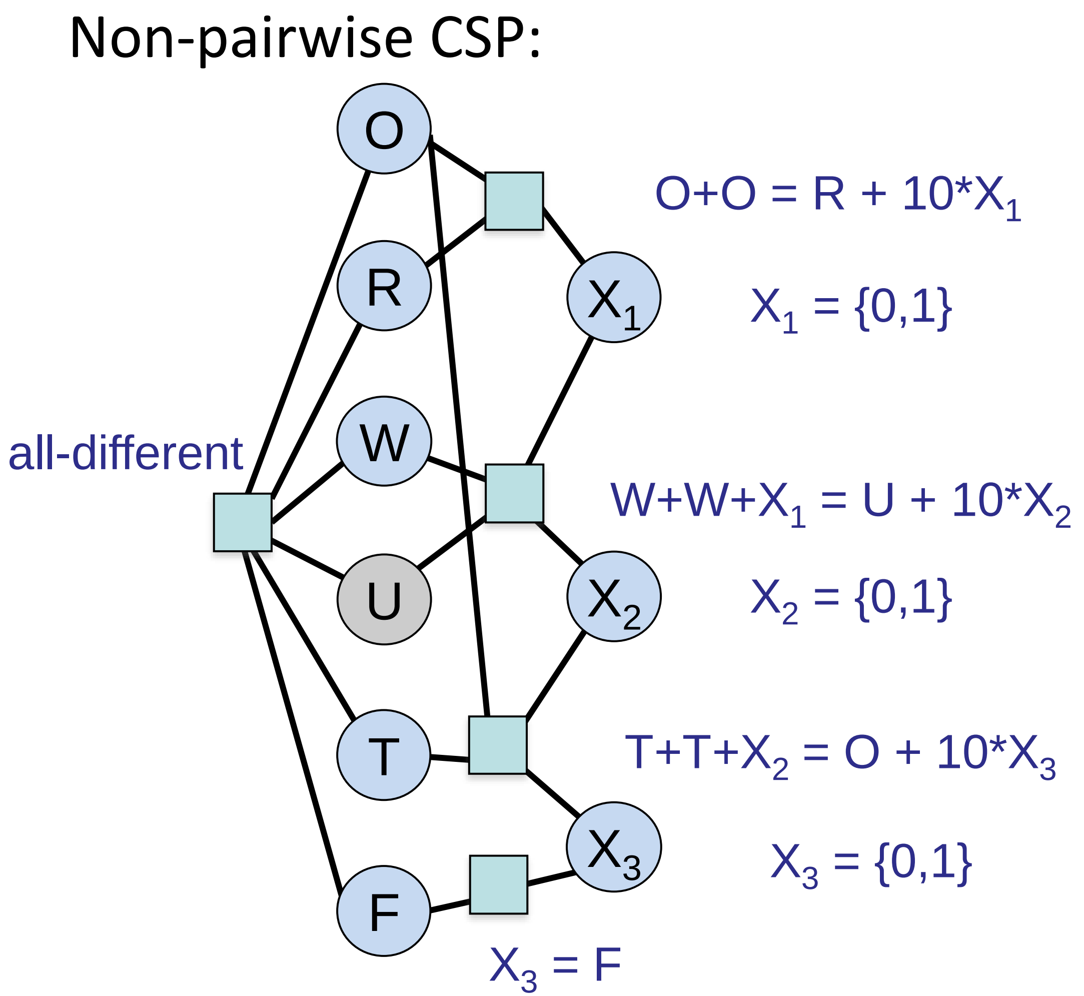
  $$\cdots$$

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



Example: Cryptarithmetic problems

- Find numeric substitutions that make an equation hold:



- Variables:

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

- Domains:

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

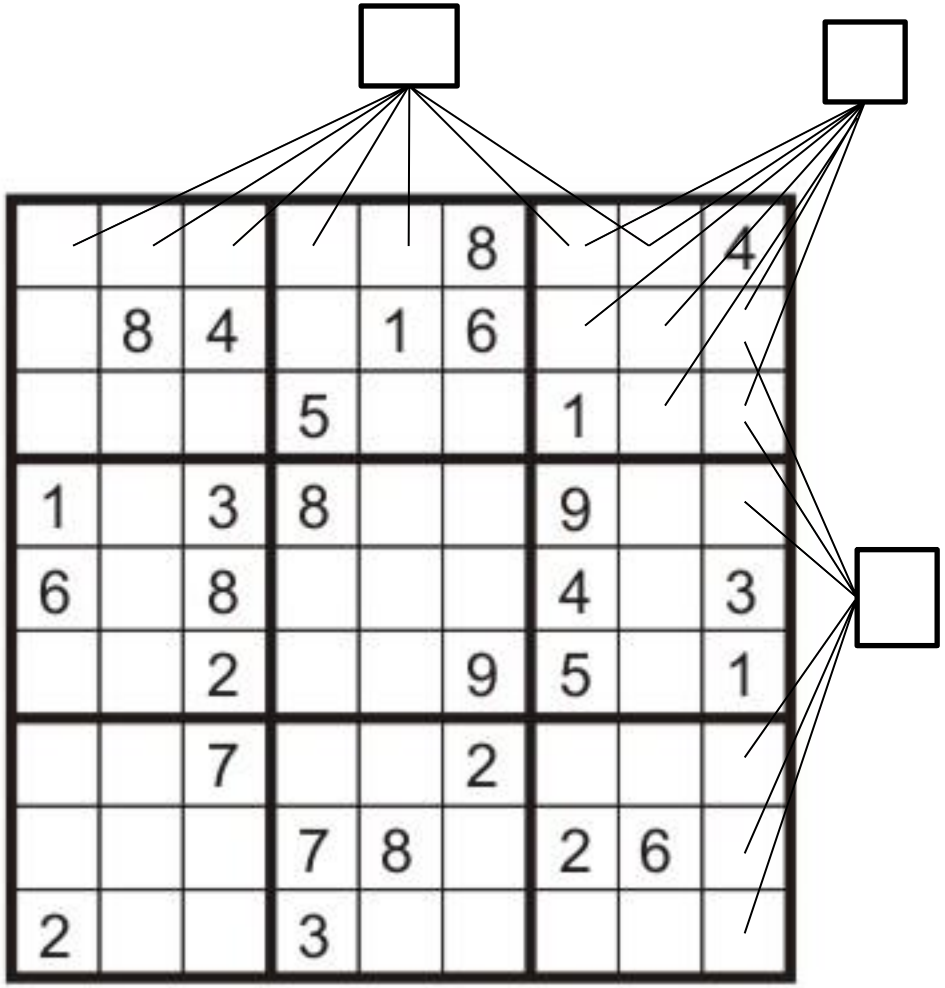
- Constraints:

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

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

$$\ldots$$

Example: Sudoku



- Variables:
  - Each (open) square
- Domains:
  - {1,2,…,9}
- Constraints:

9-way alldiff for each column

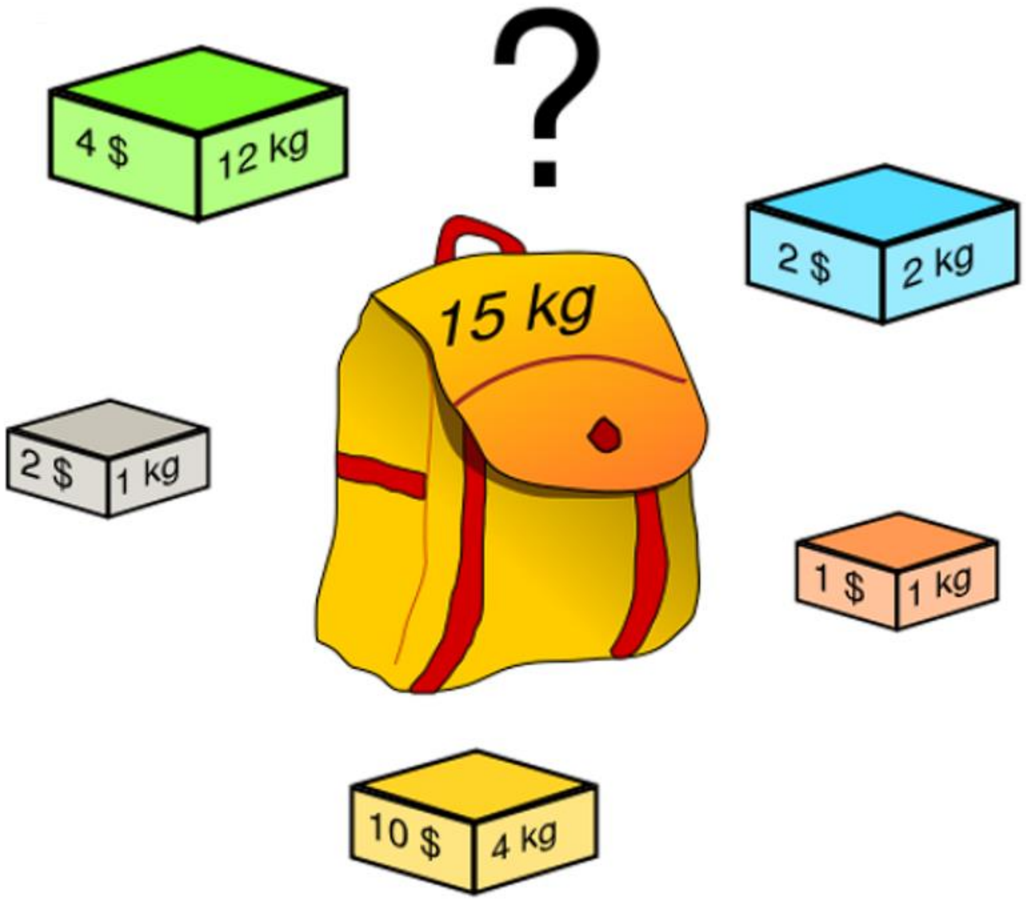
9-way alldiff for each row

9-way alldiff for each region

(or can have a bunch of pairwise inequality constraints)

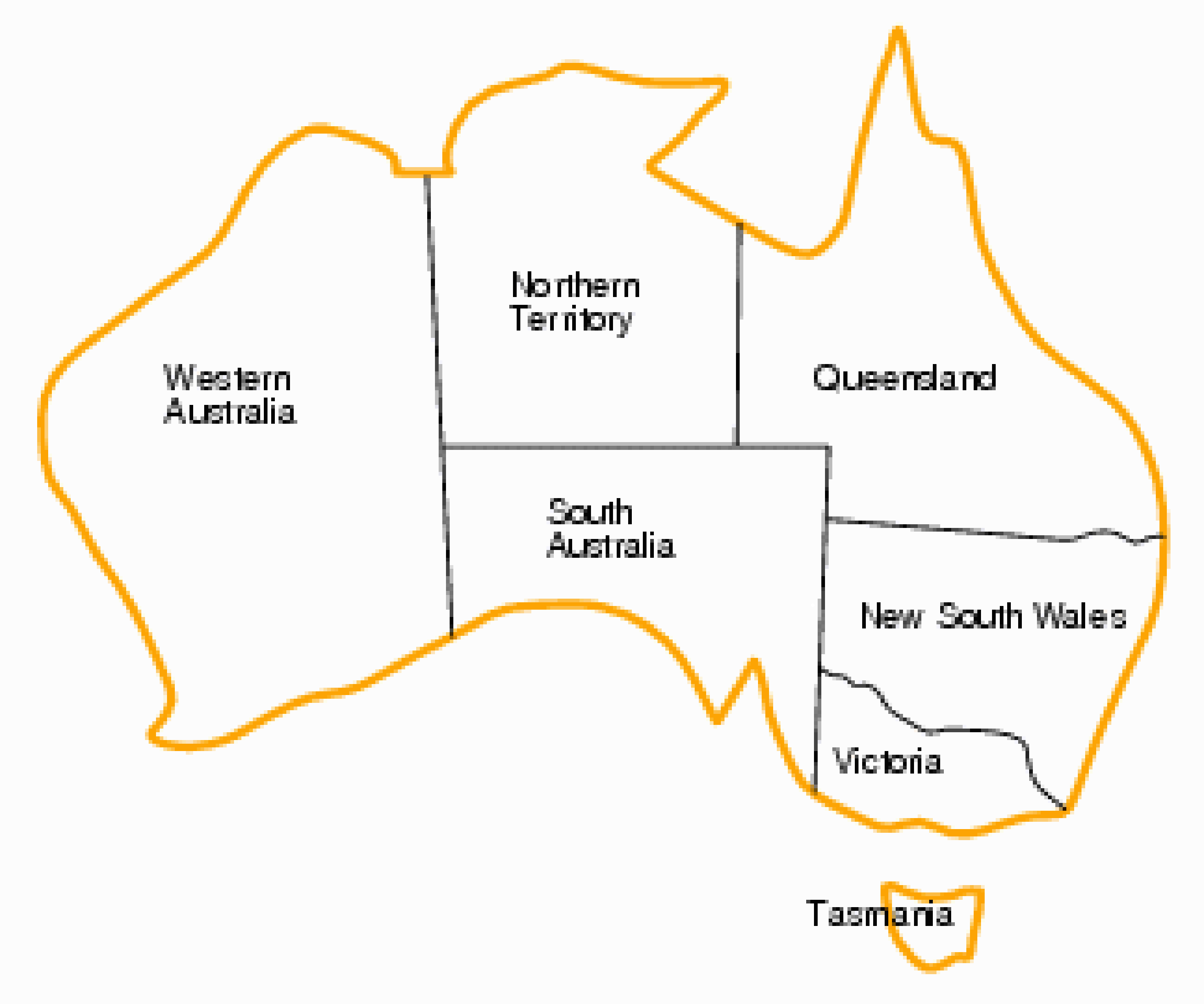
Example: Knapsack Problems

Which items should be put in the bag to maximize the profit

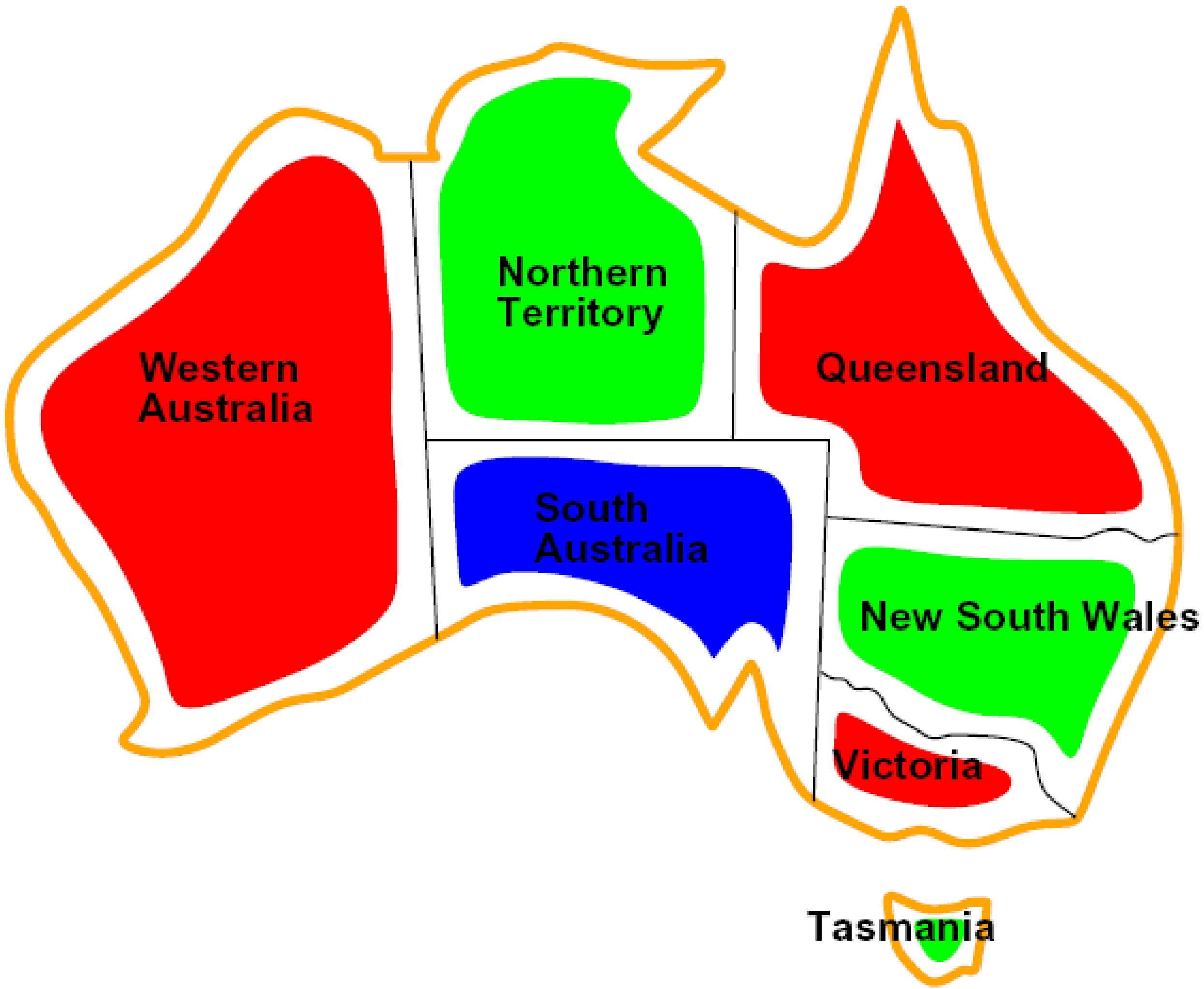


- Domain: { G, B, R, Y, P }
- Variable: { 0, …, 9 }
- Constraint:
  - 12 G + 2 B + 1 R + 4 Y + 1 P <= 15
  - Find second constraint

CSP Examples



CSP Examples



Example: Map Coloring

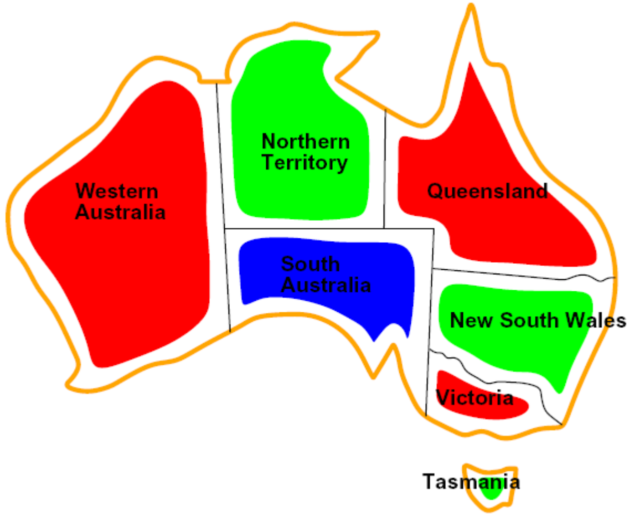
- Variables: $\mathrm{WA,\ NT,\ Q,\ NSW,\ V,\ SA,\ T}$
- Domains: $D = \{\text{red}, \text{green}, \text{blue}\}$
- Constraints: adjacent regions must have different colors

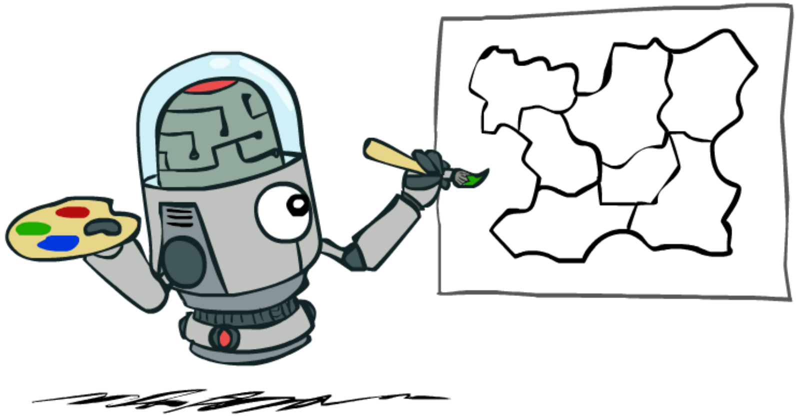
  Implicit: $\mathrm{WA} \neq \mathrm{NT}$

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

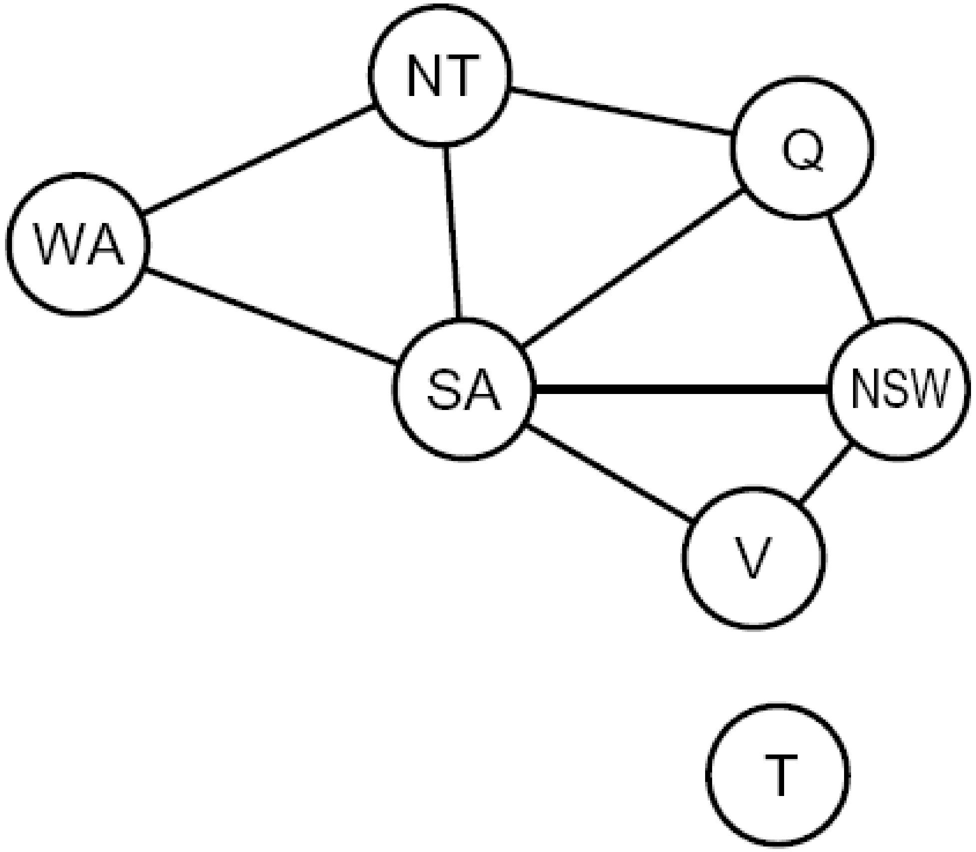
- Solutions are assignments satisfying all constraints, e.g.:

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



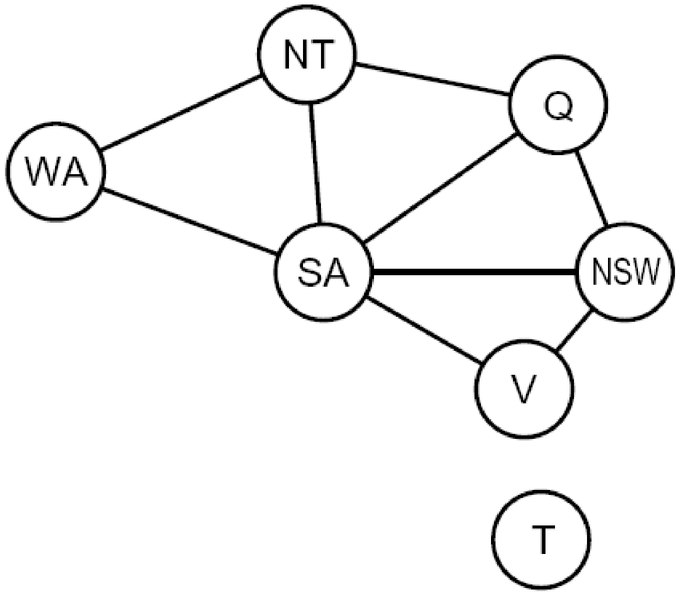


Constraint Graphs



Constraint Graphs

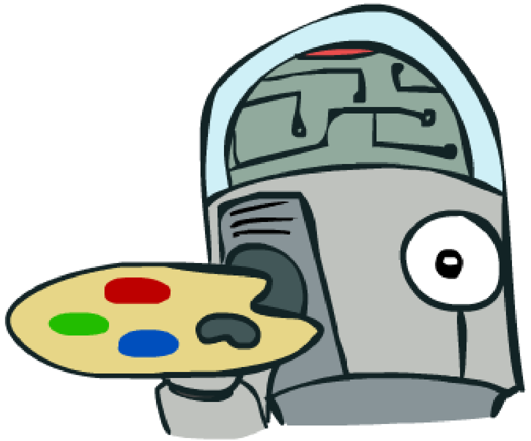
- Binary CSP: each constraint relates (at most) two variables
- Binary constraint graph: nodes are variables, arcs show constraints
- General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!



Varieties of CSPs and Constraints

Varieties of CSPs

- Discrete Variables
  - 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



- Continuous variables
  - E.g., start/end times for Hubble Telescope observations
  - Linear constraints solvable in polynomial time by LP methods (see cs170 for a bit of this theory)

Varieties of Constraints

- Varieties of Constraints
  - Unary constraints involve a single variable (equivalent to reducing domains), e.g.:

$$SA \neq green$$

  - Binary constraints involve pairs of variables, e.g.:

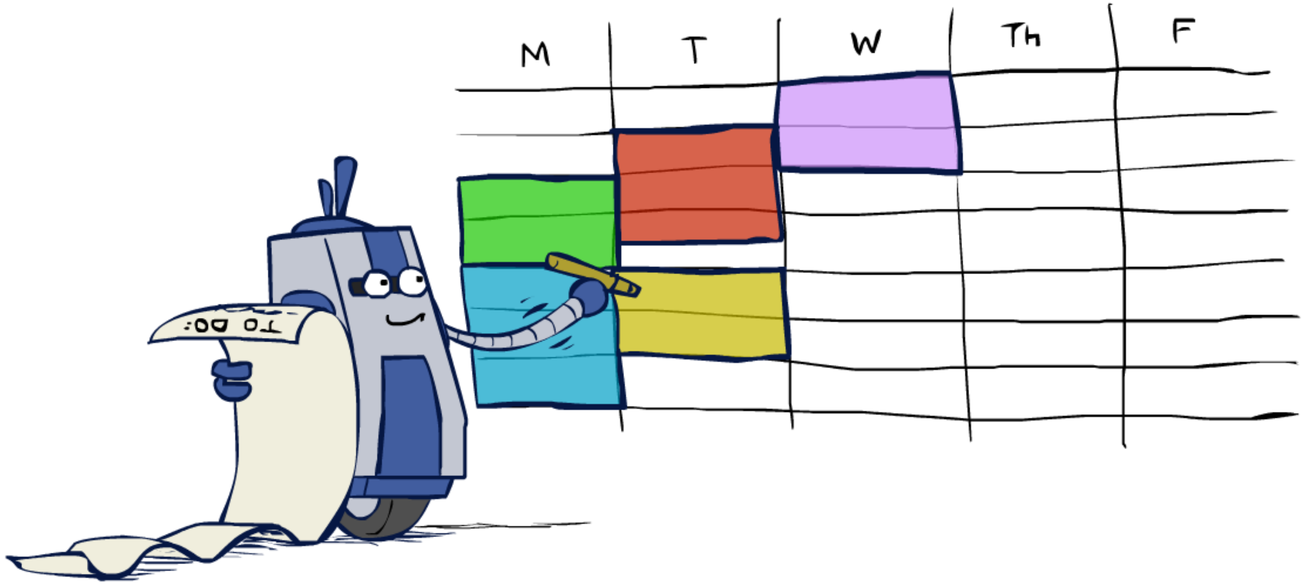
$$SA \neq WA$$

  - Higher-order constraints involve 3 or more variables:
    e.g., cryptarithmetic column constraints

- Preferences (soft constraints):
  - E.g., red is better than green
  - Often representable by a cost for each variable assignment
  - Gives constrained optimization problems
  - (We'll ignore these until we get to Bayes' nets)

Real-World CSPs

- Scheduling problems: e.g., when can we all meet?
- Timetabling problems: e.g., which class is offered when and where?
- Assignment problems: e.g., who teaches what class
- Hardware configuration
- Transportation scheduling
- Factory scheduling
- Circuit layout
- Fault diagnosis
- … lots more!



- Many real-world problems involve real-valued variables…

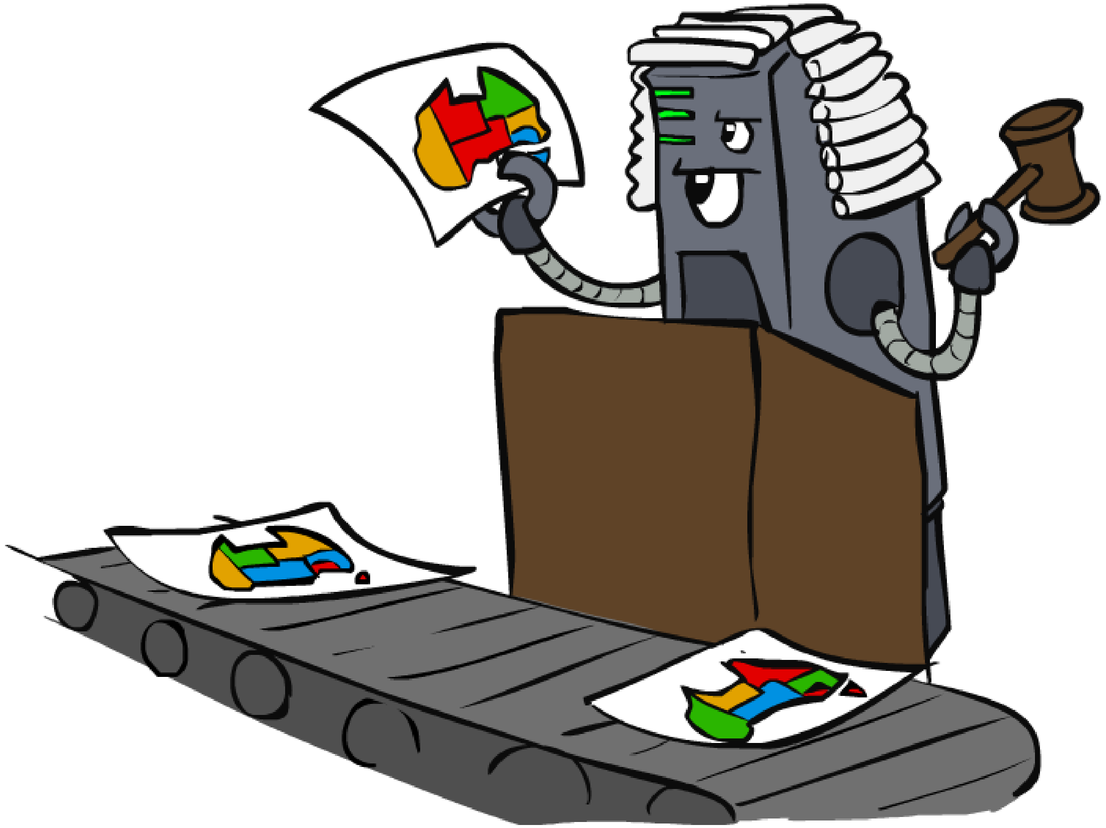
Solving CSPs

Standard Search Formulation

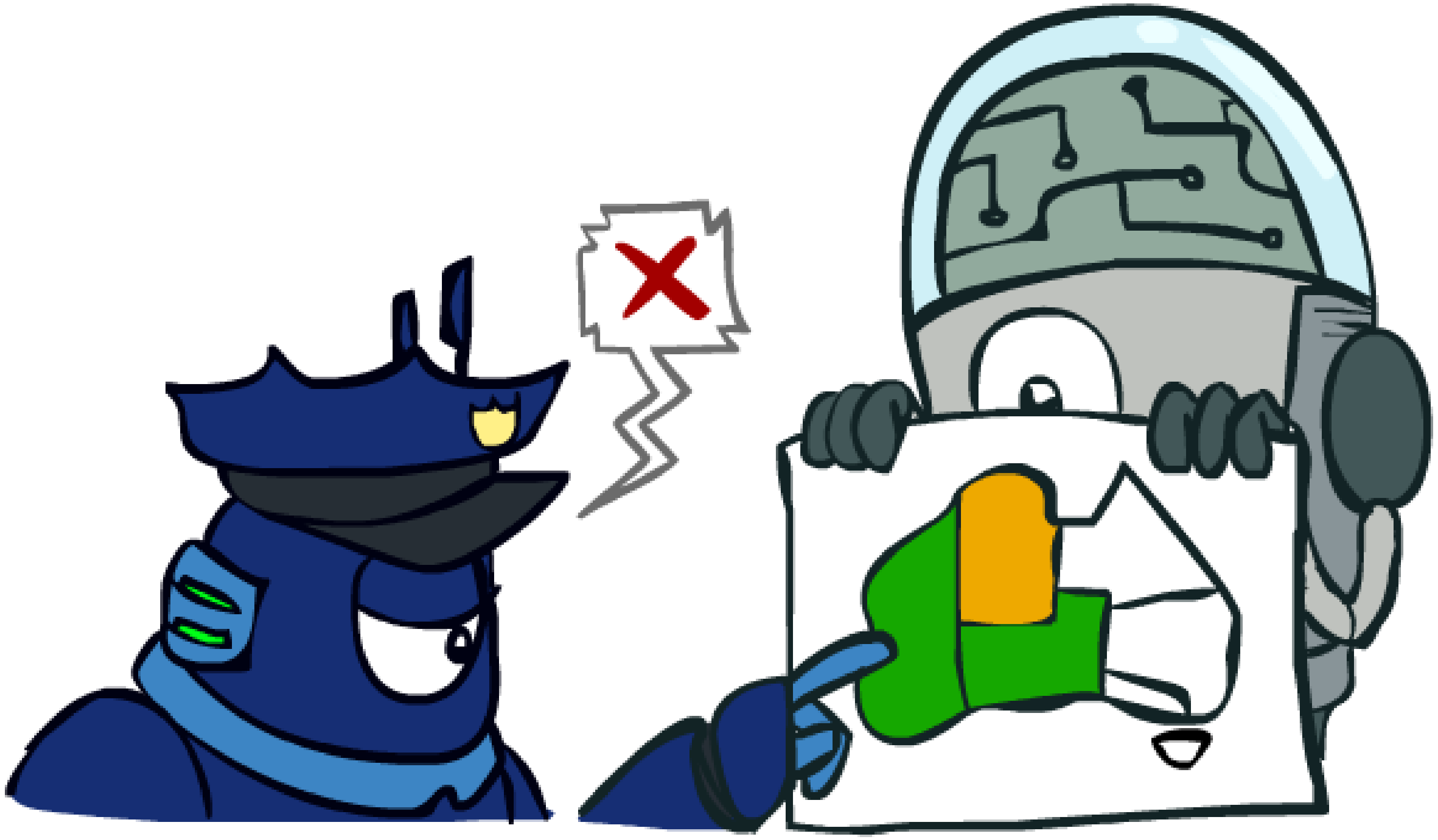
- Standard search formulation of CSPs

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

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

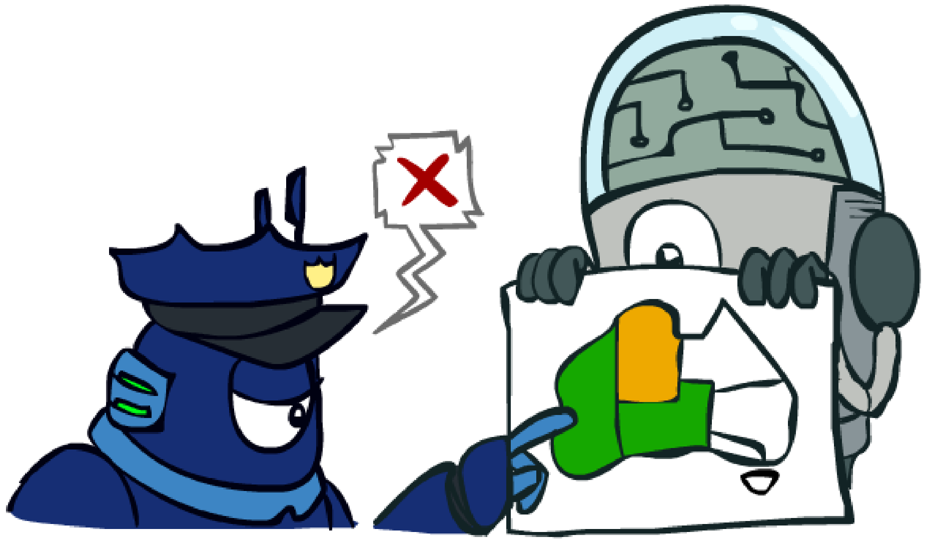


Backtracking Search

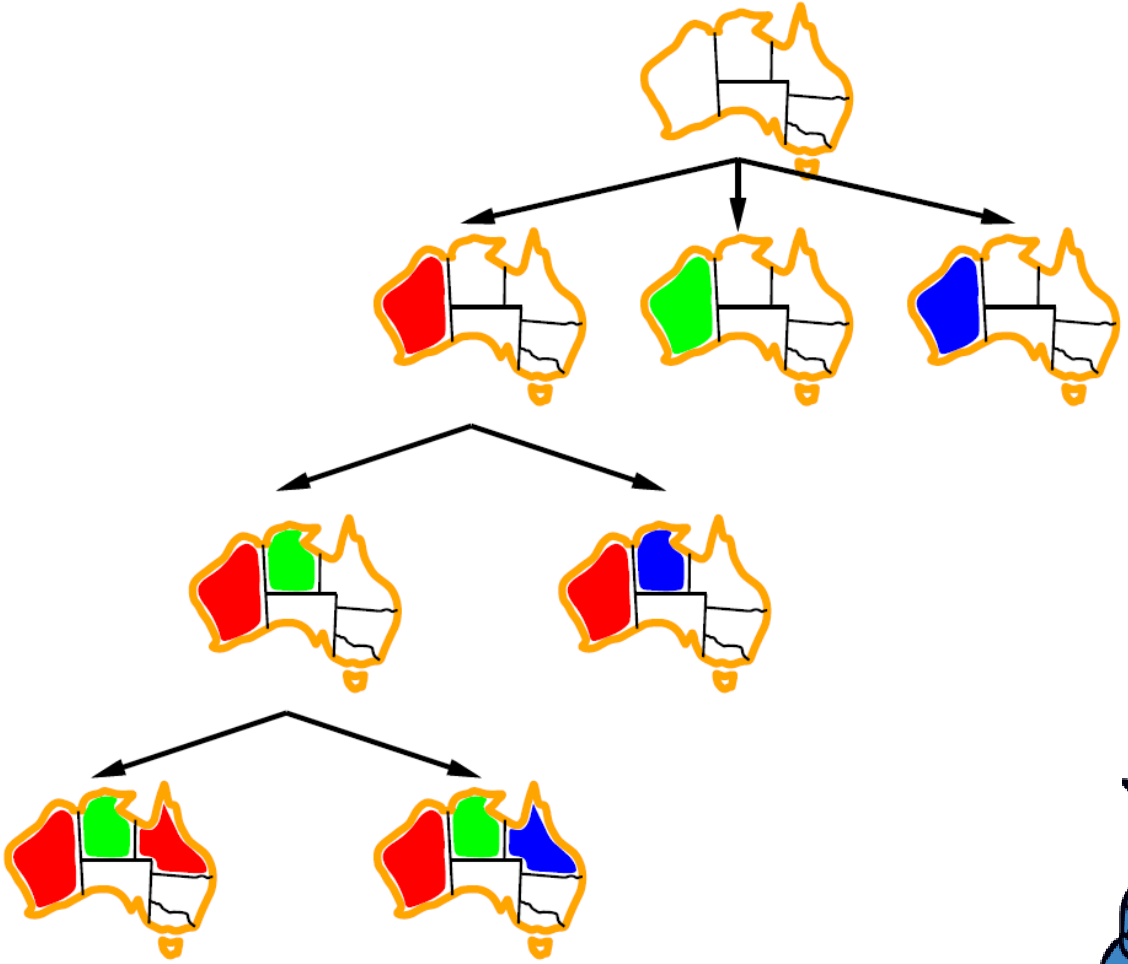


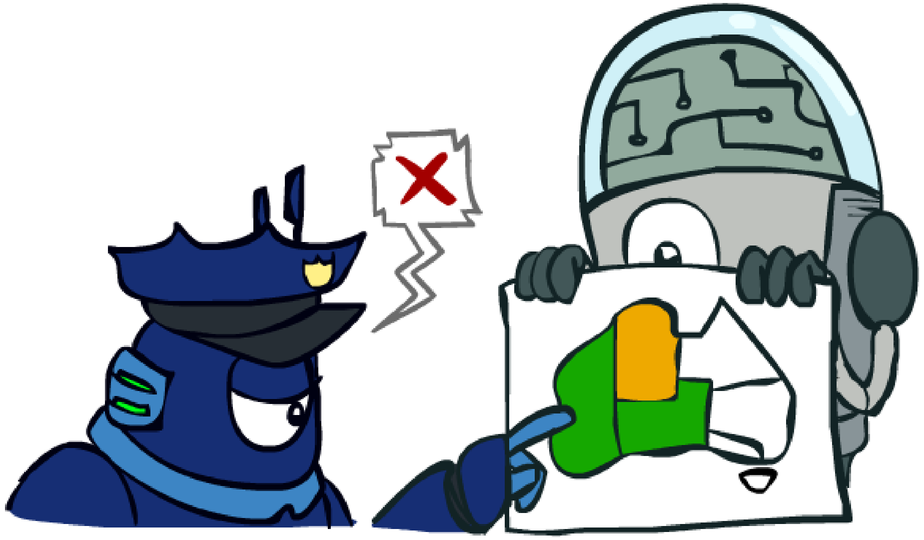
Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
- Idea 1: One variable at a time
  - Variable assignments are commutative, so fix ordering
  - I.e., [WA = red then NT = green] same as [NT = green then WA = red]
  - Only need to consider assignments to a single variable at each step
- Idea 2: Check constraints as you go
  - I.e. consider only values which do not conflict with previous assignments
  - Might have to do some computation to check the constraints
  - “Incremental goal test”
- Depth-first search with these two improvements is called *backtracking search* (not the best name)
- Can solve n-queens for $n \approx 25$



Backtracking Example





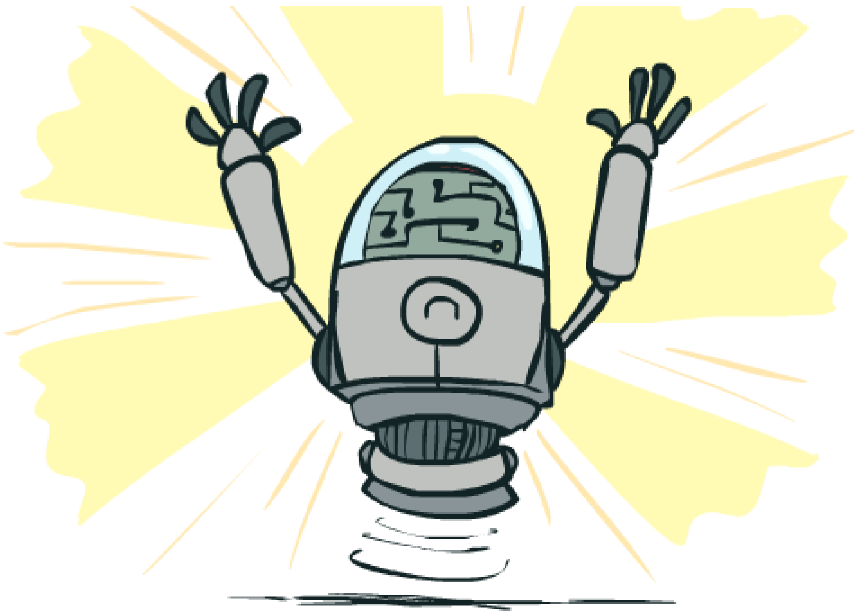
Backtracking Search

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

Improving Backtracking

- Before search:
  - **Reduce the search space**
  - Arc-consistency, path-consistency, i-consistency
  - Variable ordering (fixed)
- During search:
  - **Look-ahead schemes**:
    - Detecting failure early; reduce the search space if possible
    - Which variable should be assigned next?
    - Which value should we explore first?
  - **Look-back schemes**:
    - Backjumping
    - Constraint recording
    - Dependency-directed backtracking



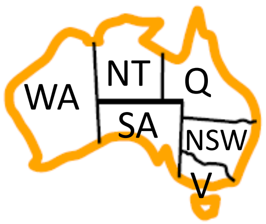
Look-Ahead

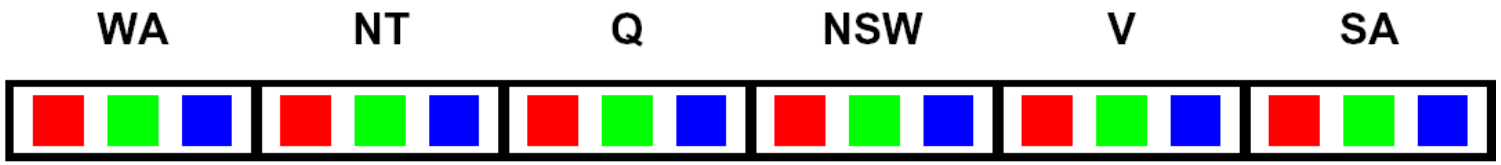
Intuition:

- Apply propagation at each node in the search tree (reduce future branching)
- Choose a variable that will detect failures early (low branching factor)
- Choose value least likely to yield a dead-end (find solution early if possible)

Filtering: Forward Checking

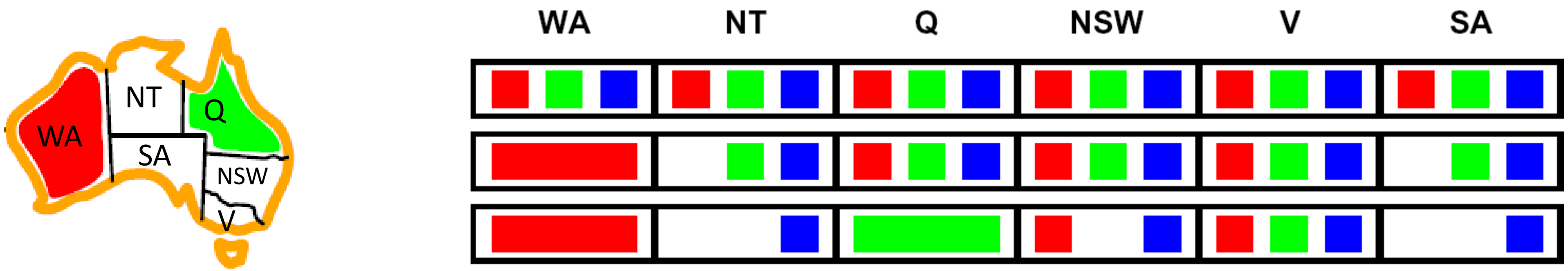
- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment





Filtering: Constraint Propagation

- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:



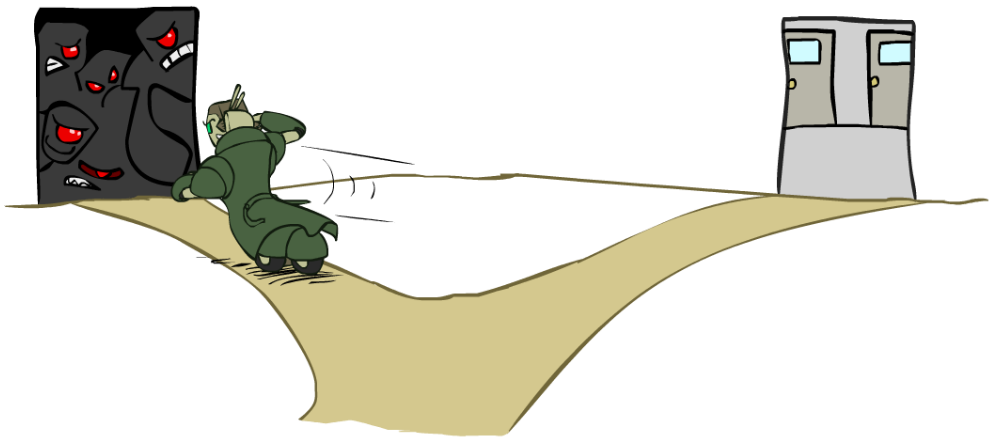
- NT and SA cannot both be blue!
- Why didn’t we detect this yet?
- *Constraint propagation:* reason from constraint to constraint

Ordering

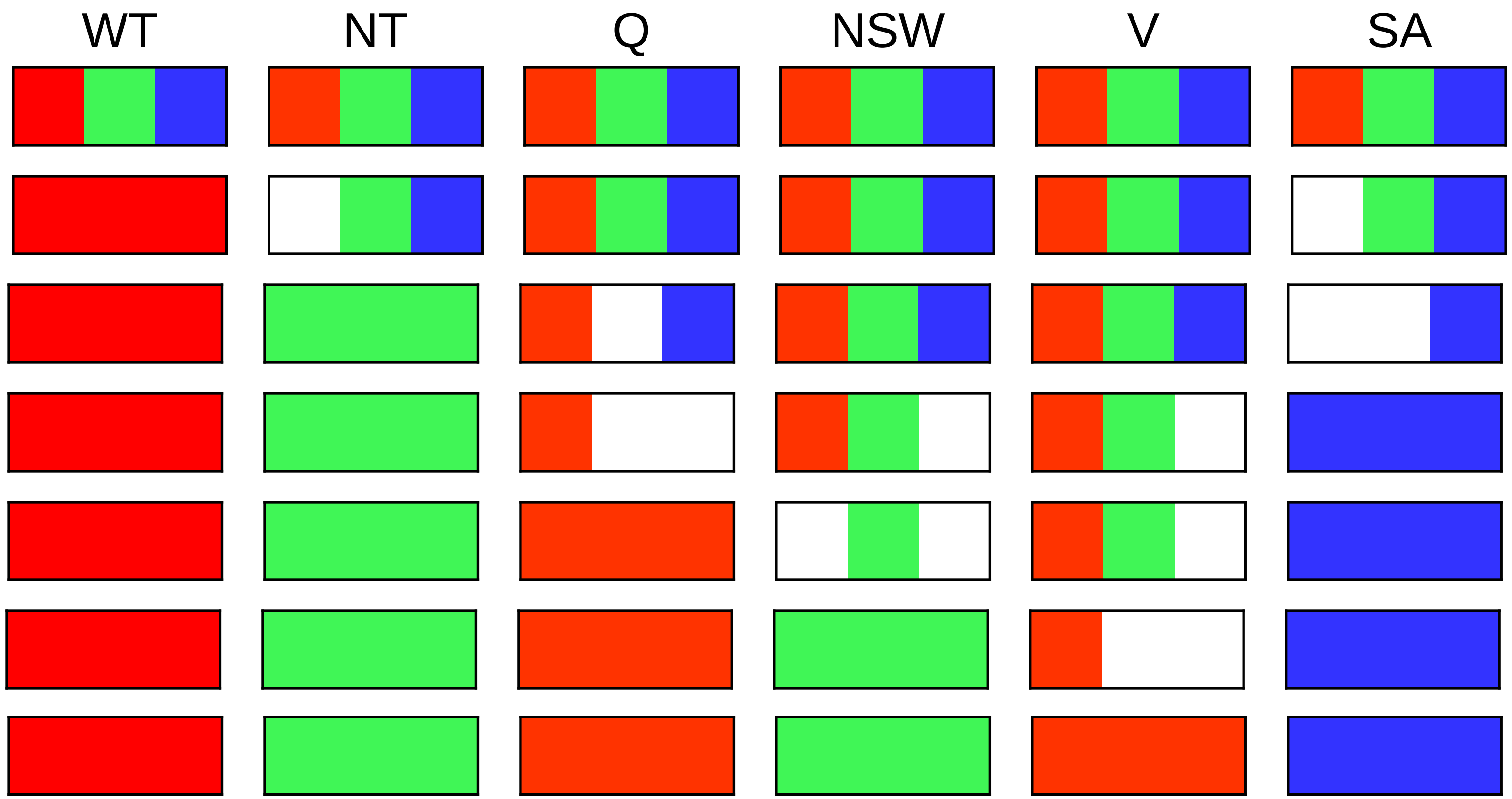
Ordering: Minimum Remaining Values

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

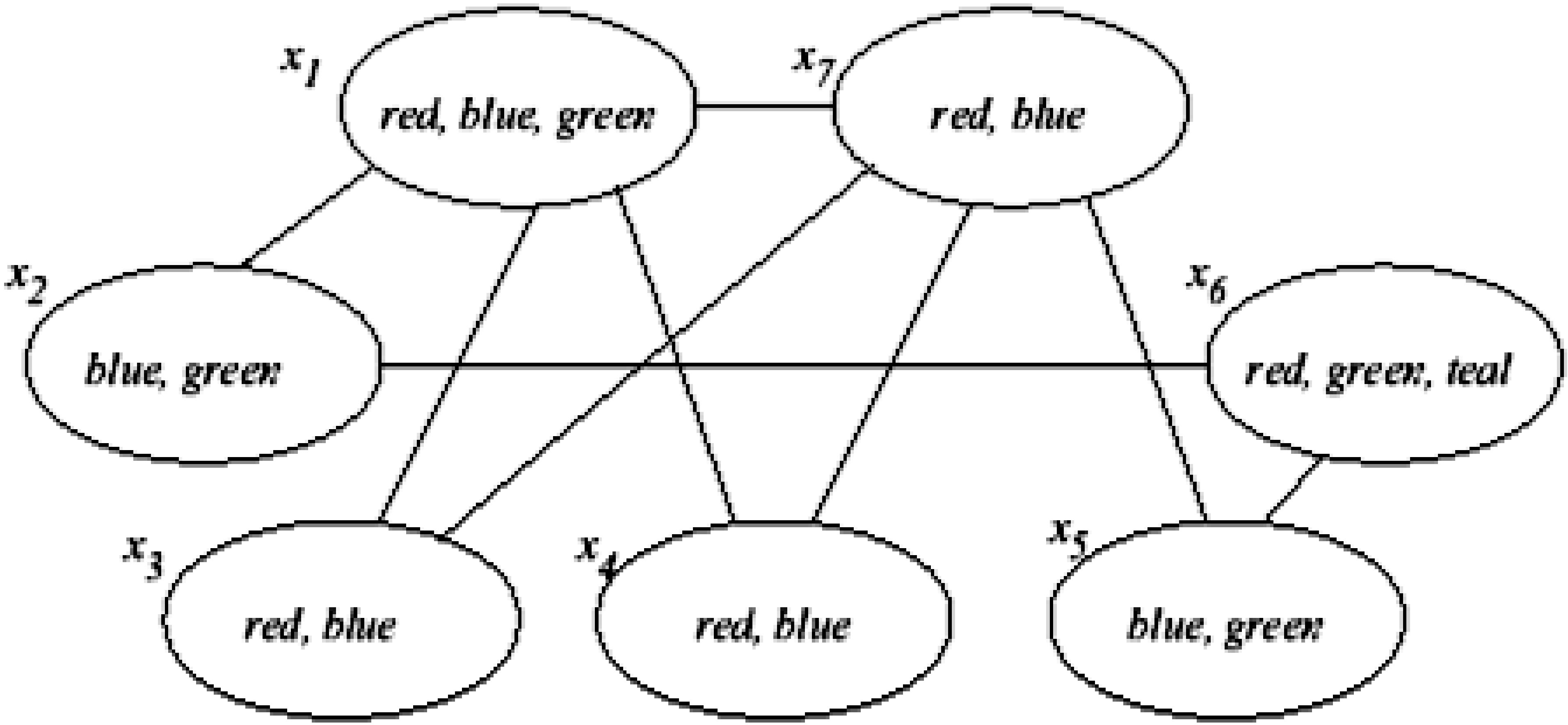
- Why min rather than max?
- Also called "most constrained variable"
- "Fail-fast" ordering



Minimum Remaining Values (MVR)



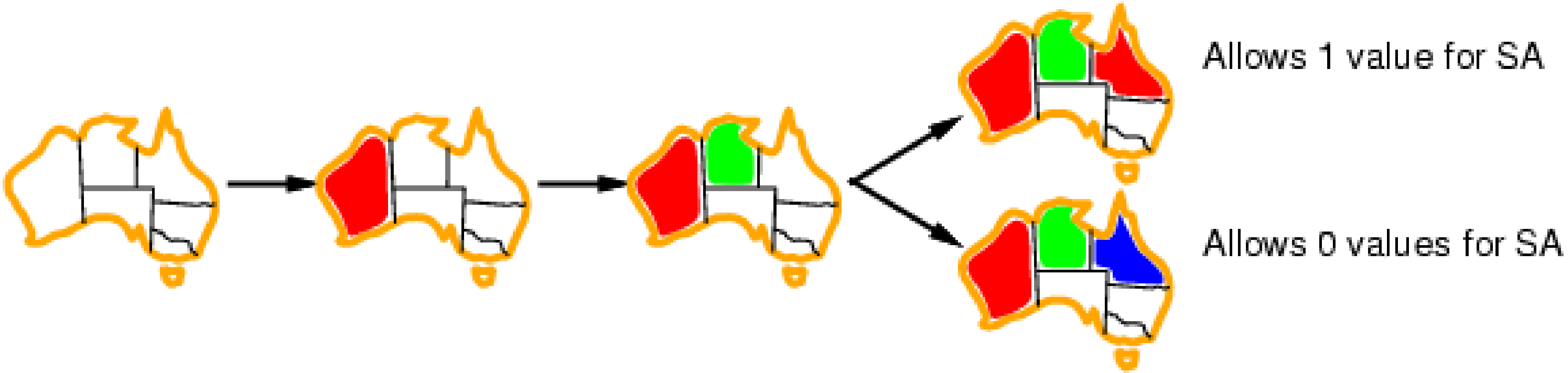
EXAMPLE



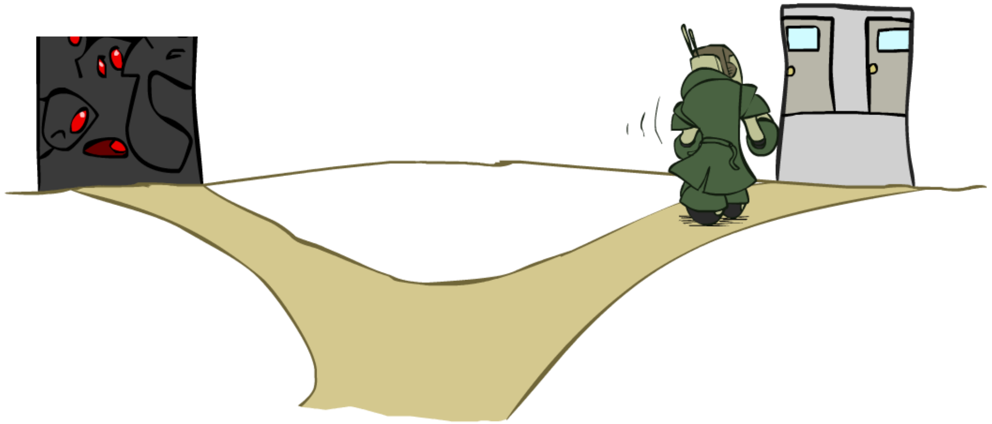
x1
red, blue, green
x7
red, blue
x2
blue, green
x6
red, green, teal
x3
red, blue
x4
red, blue
x5
blue, green

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)



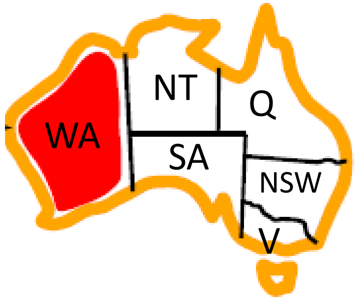
- Why least rather than most?

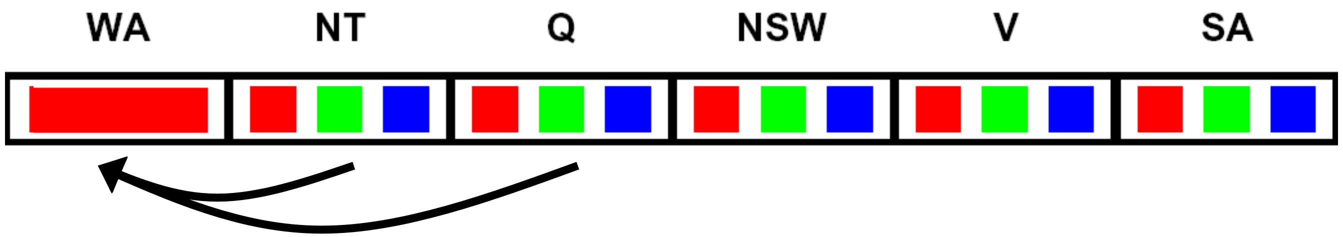


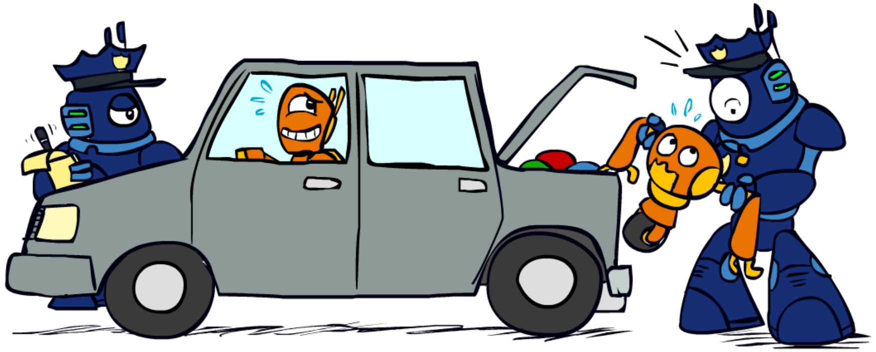
- Combining these ordering ideas makes 1000 queens feasible

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





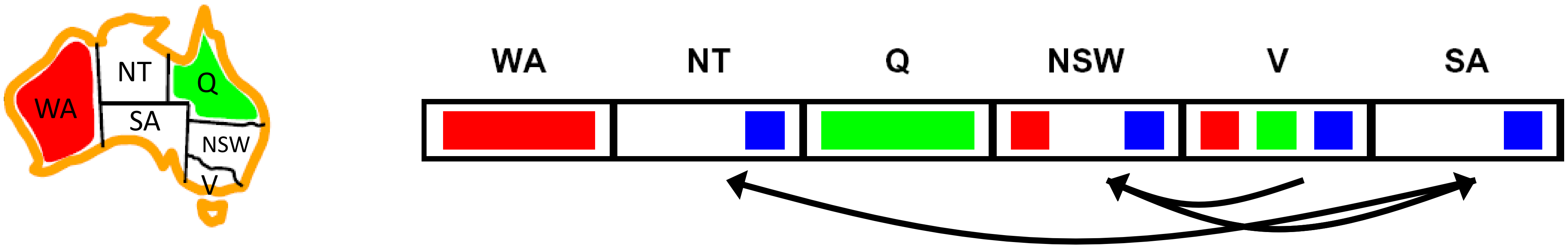


Delete from the tail!

- Forward checking: Enforcing consistency of arcs pointing to each new assignment

Arc Consistency of an Entire CSP

- A simple form of propagation makes sure all arcs are consistent:



- Important: If X loses a value, neighbors of X need to be rechecked!
- 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!

Enforcing Arc Consistency in a CSP

```
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
```

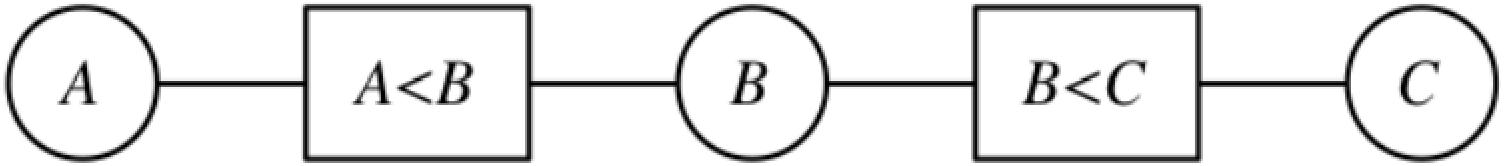
- Runtime: $O(n^2d^3)$, can be reduced to $O(n^2d^2)$
- … but detecting all possible future problems is NP-hard – why?

ARC Consistency Algorithm, AC-3

- Keep a set of arcs to be considered: pick one arc (X, Y) at the time and make it consistent (i.e., make arc X consistent to Y).
  - Start with the set of all arcs $\{(X, Y), (Y, X), (X, Z), (Z, X), \ldots.\}$.
- When an arc has been made arc consistent, does it ever need to be checked again?
  - An arc (Z, X) needs to be revisited if the domain of X is revised.

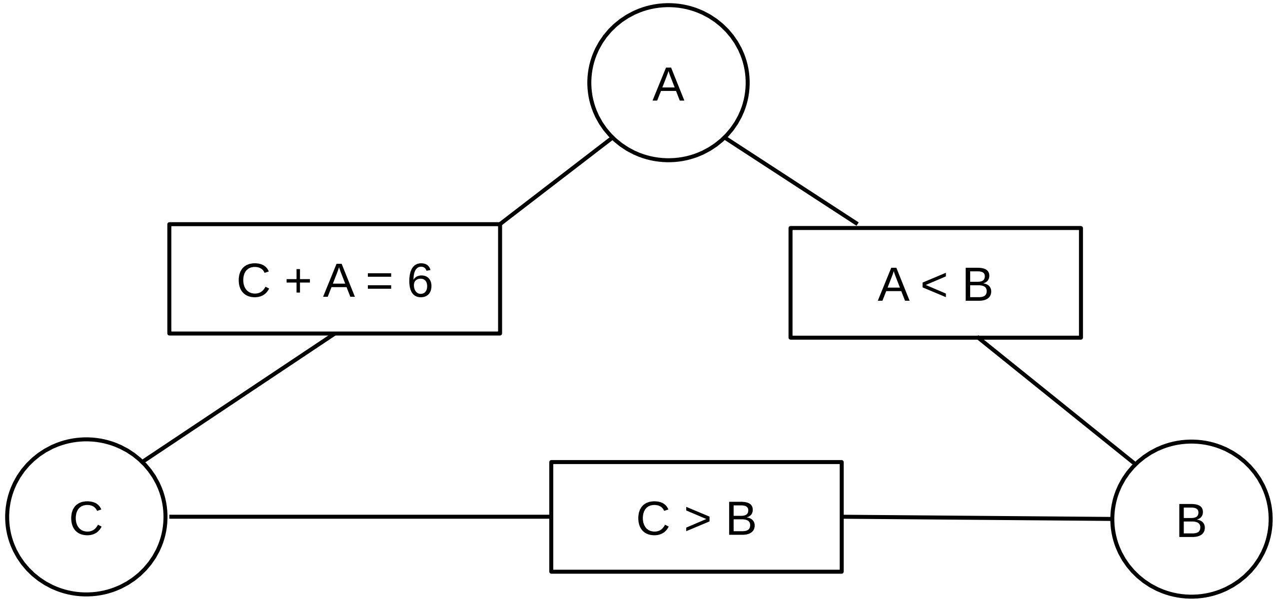
```
function AC-3(inout csp):
    initialise queue to all arcs in csp
    while queue is not empty:
        (X, Y) := RemoveOne(queue)
        if Revise(csp, X, Y):
            if D_X = Ø  then return failure
            for each Z in X.neighbors–{Y} do add (Z, X) to queue

function Revise(inout csp, X, Y):
    delete every x from D_X such that there is no value y in D_Y satisfying the constraint C_XY
    return true if D_X was revised
```

ARC-3 Example - 1



Remove	D_A	D_B	D_C	Add	Queue
	1234	1234	1234		A<B, B<C, C>B, B>A
A<B	123	1234	1234		B<C, C>B, B>A
B<C	123	123	1234	A<B	C>B, B>A, **A<B**
C>B	123	123	234		B>A, A<B
B>A	123	23	234	C>B	A<B, **C>B**
A<B	12	23	234		C>B
C>B	12	23	34		∅

ARC-3 Example



ARC-3 Example - 2

Remove	D_A	D_B	D_C	Add	Queue
	1234	1234	1234		A<B, B<C, C+A=6, A+C=6, C>B, B>A
A<B	123	1234	1234	**C+A=6**	B<C, C+A=6, A+C=6, C>B, B>A, **C+A=6**
B<C	123	123	1234	**A<B**	C+A=6, A+C=6, C>B, B>A, C+A=6, **A<B**
C+A=6	123	123	34	**C>B**	A+C=6, C>B, B>A, C+A=6, A<B, **C>B**
A+C=6	23	123	34	**B>A**	C>B, B>A, C+A=6, A<B, C>B, **B>A**
C>B	23	123	34		B>A, C+A=6, A<B, C>B, B>A
B>A	23	3	34	**C>B**	C+A=6, A<B, C>B, B>A, **C>B**
C+A=6	23	3	34		A<B, C>B, B>A, C>B
A<B	2	3	34	**C+A=6**	C>B, B>A, C>B, **C+A=6**
C>B	2	3	4	**A+C=6**	B>A, C>B, C+A=6, **A+C=6**
B>A	2	3	4		C>B, C+A=6, A+C=6

ARC-3 Example

Remove	D_A	D_B	D_C	Add	Queue
	1234	1234	1234		A<B, B<C, C+A=6, A+C=6, C>B, B>A
A<B	123	1234	1234	**C+A=6**	B<C, C+A=6, A+C=6, C>B, B>A, **C+A=6**
B<C	123	123	1234	**A<B**	C+A=6, A+C=6, C>B, B>A, C+A=6, **A<B**
C+A=6	123	123	34	**C>B**	A+C=6, C>B, B>A, C+A=6, A<B, **C>B**
A+C=6	23	123	34	**B>A**	C>B, B>A, C+A=6, A<B, C>B, **B>A**
C>B	23	123	34		B>A, C+A=6, A<B, C>B, B>A
B>A	23	3	34	**C>B**	C+A=6, A<B, C>B, B>A, **C>B**
C+A=6	23	3	34		A<B, C>B, B>A, C>B
A<B	2	3	34	**C+A=6**	C>B, B>A, C>B, **C+A=6**
C>B	2	3	4	**A+C=6**	B>A, C>B, C+A=6, **A+C=6**
B>A	2	3	4		C>B, C+A=6, A+C=6
C>B	2	3	4		C+A=6, A+C=6
C+A=6	2	3	4		A+C=6
A+C=6	2	3	4		∅

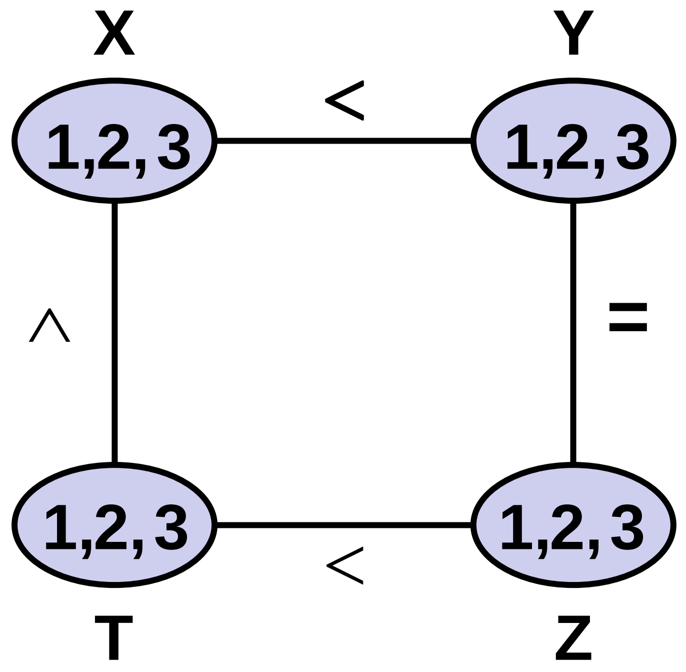
ARC Consistency Algorithm, AC-3

function AC-3(**inout** *csp*):
    initialise *queue* to all arcs in *csp*
    while *queue* is not empty:
        (X, Y) := RemoveOne(*queue*)
        if Revise(*csp*, X, Y):
            if $D_X = \emptyset$ **then return** failure
            for each Z **in** X.neighbors–{Y} **do** add (Z, X) to *queue*

function Revise(**inout** *csp*, X, Y):
    delete every x from D_X such that there is no value y in D_Y satisfying the constraint C_{XY}
    return true **if** D_X was revised

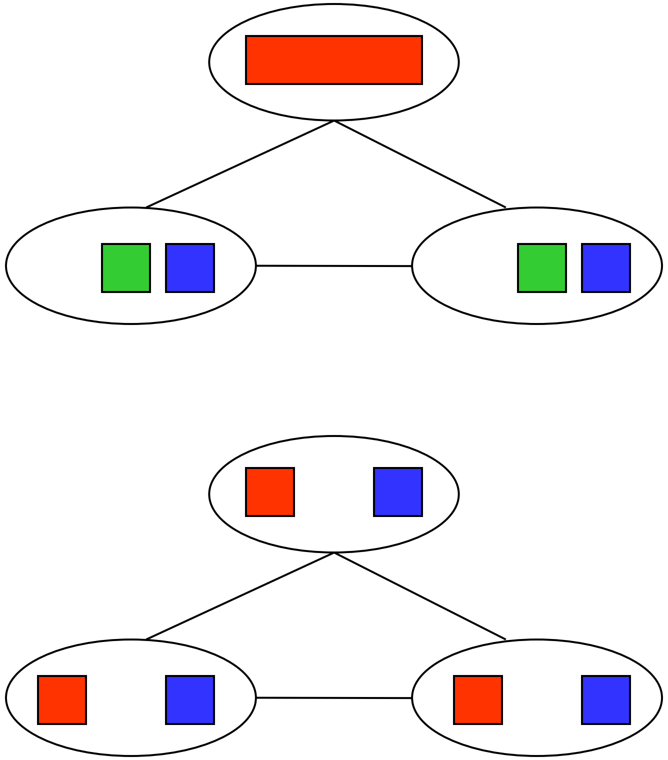
ARC-3 Example - 3

$1 \leq X, Y, Z, T \leq 3$
$X < Y$
$Y = Z$
$T < Z$
$X \leq T$



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 still runs inside a backtracking search!



What went wrong here?

Summary

- CSPs
  - special kind of problem: states defined by values of a fixed set of variables, goal test defined by constraints on variable values
- Backtracking = depth-first search with one variable assigned per node
- Forward checking prevents assignments that guarantee later failure
- Heuristics
  - Variable ordering and value selection heuristics help significantly
- Variable ordering (selection) heuristics
  - Choose variable with Minimum Remaining Values (MRV)
  - Degree Heuristic – break ties after applying MRV
- Value ordering (selection) heuristic
  - Choose Least Constraining Value
- Constraint propagation (e.g. ARC consistency) does additional work to constraint values and detect inconsistencies