

Artificial Intelligence

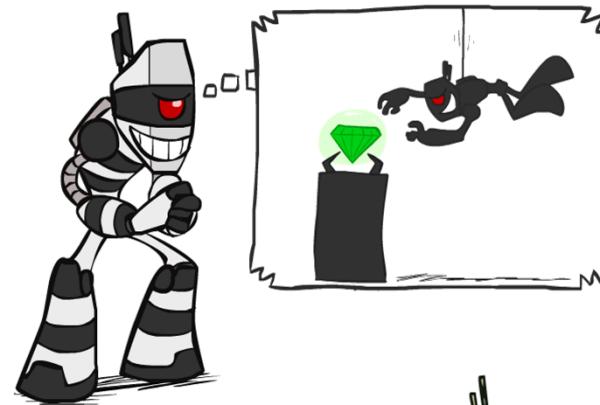
Constraint Satisfaction Problems



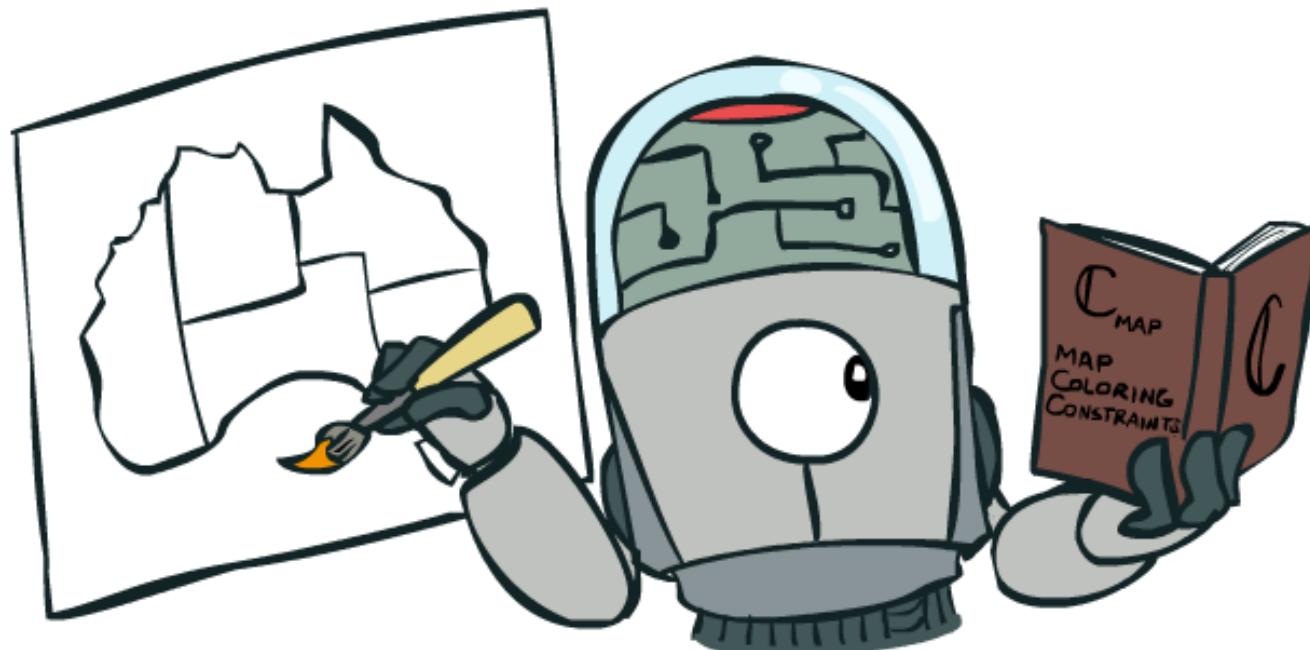
Fatemeh Mansoori
University of Isfahan

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

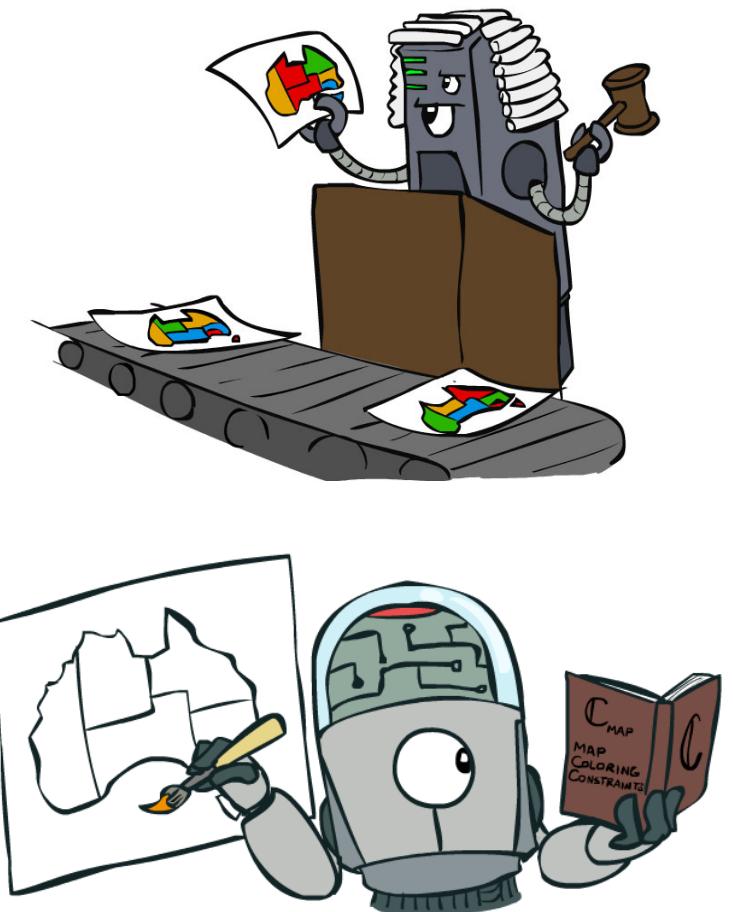


Constraint Satisfaction Problems

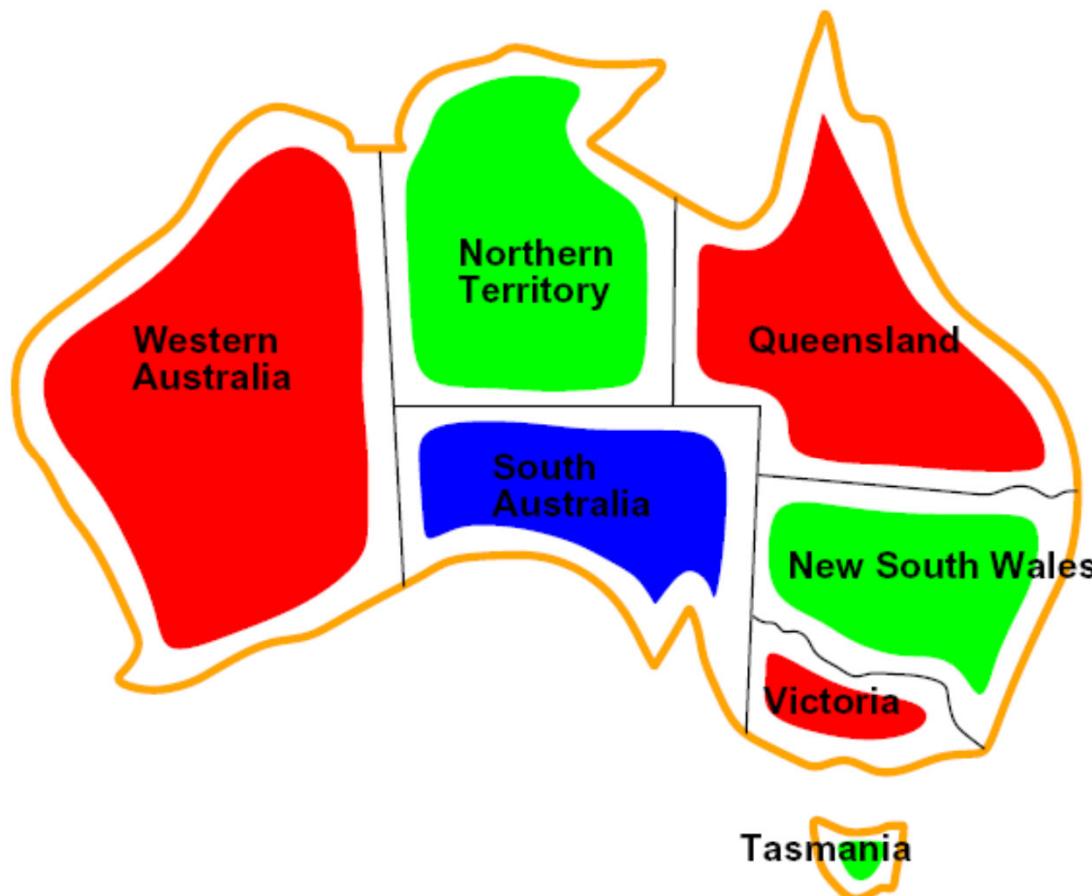


Constraint Satisfaction Problems

- 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
- Simple example of a *formal representation language*
- Allows useful general-purpose algorithms with more power than standard search algorithms



CSP Examples



Example: Map Coloring

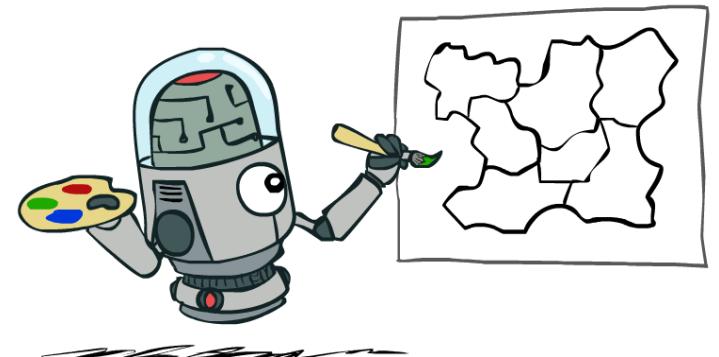
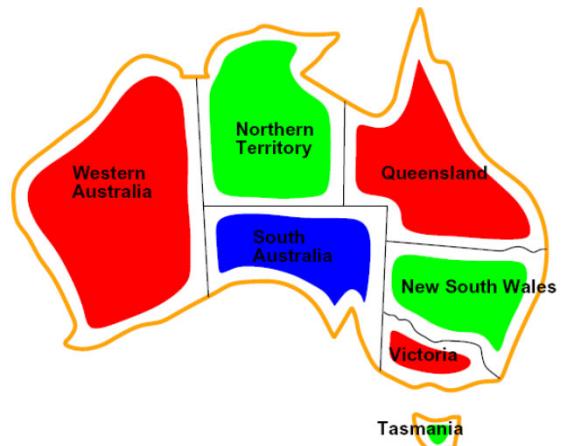
- Variables: WA, NT, Q, NSW, V, SA, T
- Domains: $D = \{\text{red, green, blue}\}$
- Constraints: adjacent regions must have different colors

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

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

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

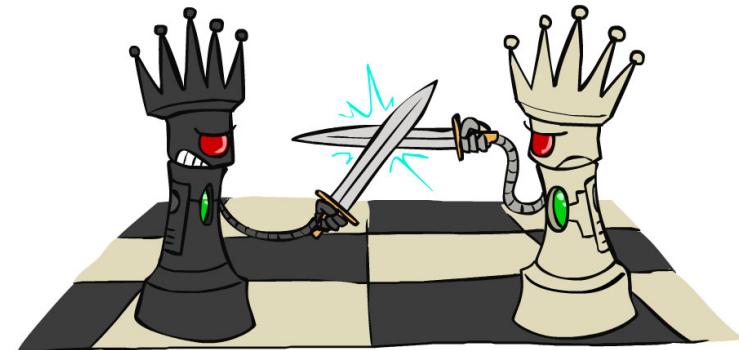
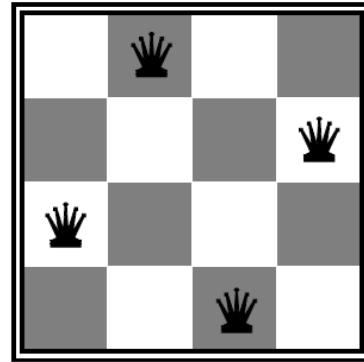
$\{\text{WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green}\}$



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

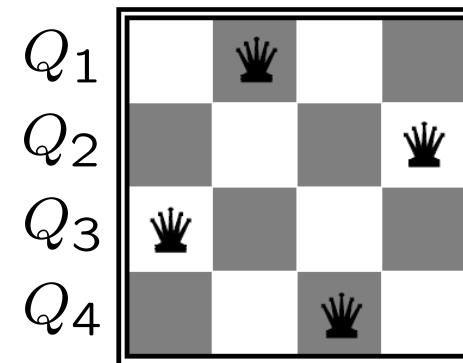
- Formulation 2:

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

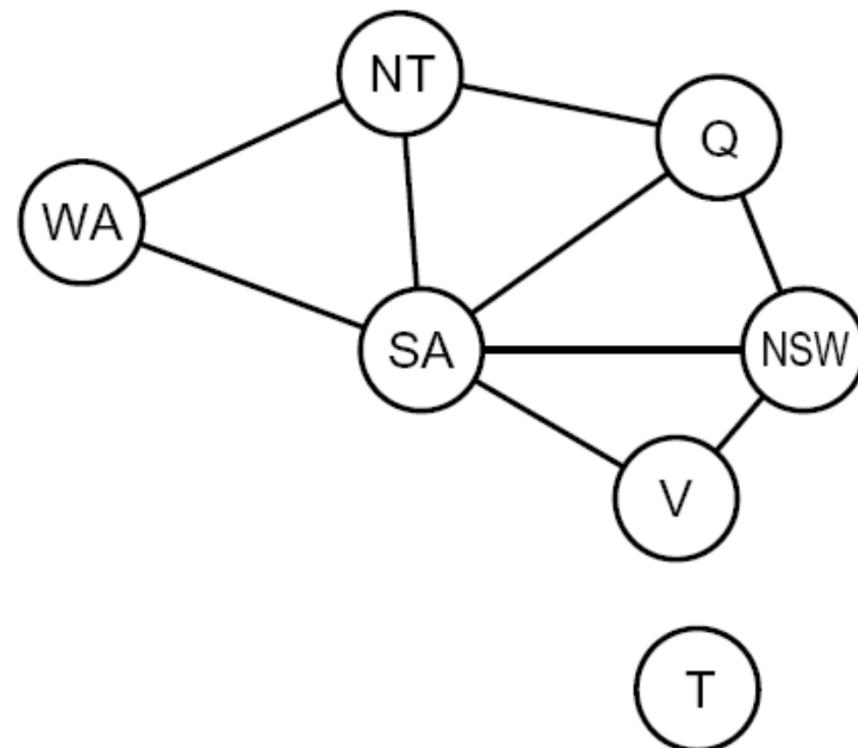
Implicit: $\forall i, j \text{ non-threatening}(Q_i, Q_j)$

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

...

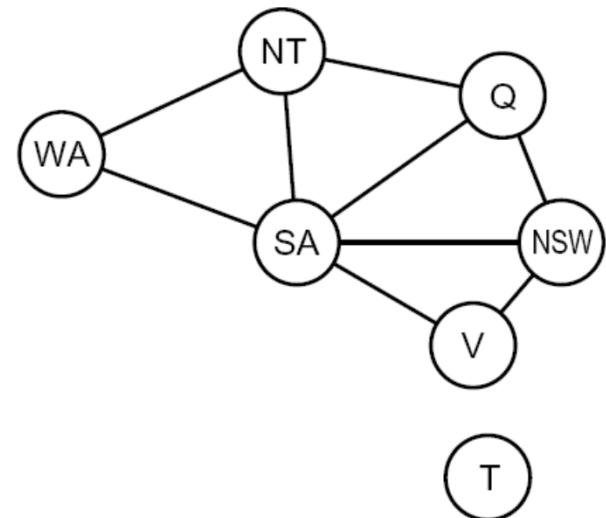


Constraint Graphs



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!



[Demo: CSP applet (made available by aispace.org) -- n-queens]

Example: Cryptarithmetic

- Variables:

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

- Domains:

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

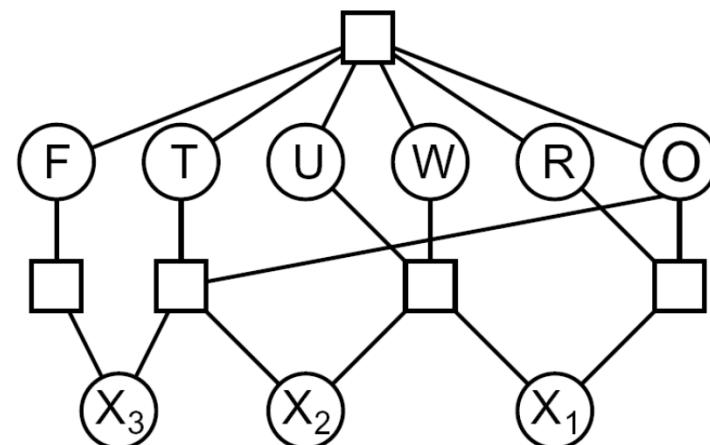
- Constraints:

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

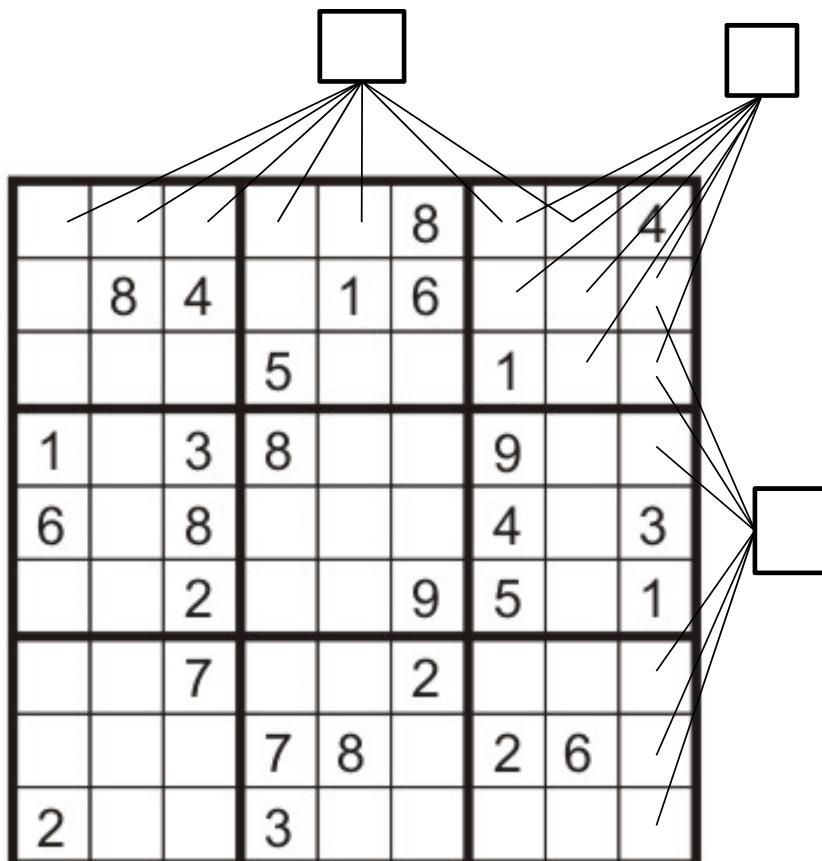
$$O + O = R + 10 \cdot X_1$$

...

$$\begin{array}{r} \text{T} \ \text{W} \ \text{O} \\ + \ \text{T} \ \text{W} \ \text{O} \\ \hline \text{F} \ \text{O} \ \text{U} \ \text{R} \end{array}$$

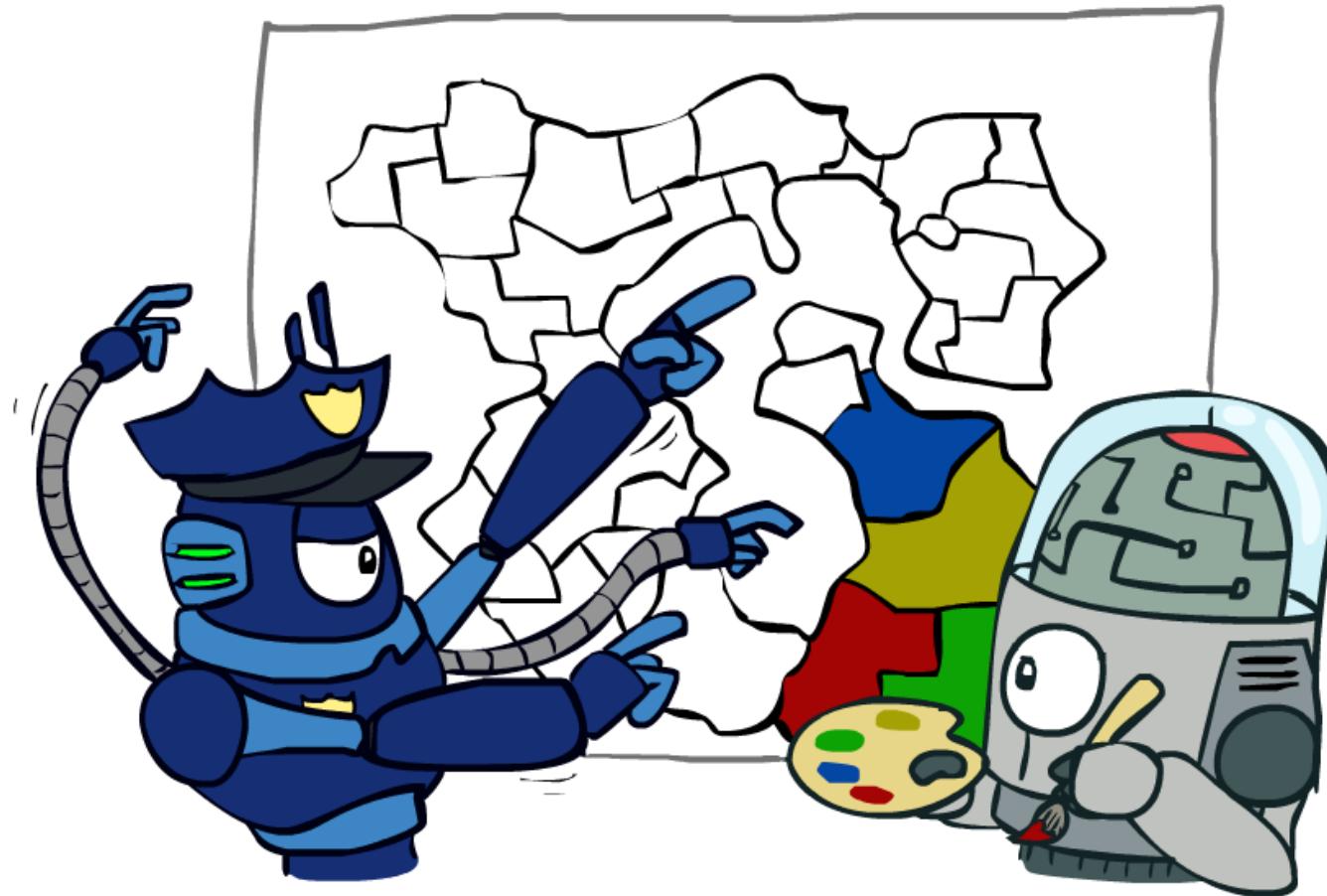


Example: Sudoku



- 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)

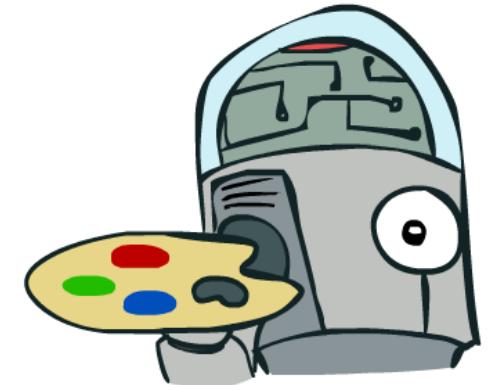
Varieties of CSPs and Constraints



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.:

$$SA \neq \text{green}$$

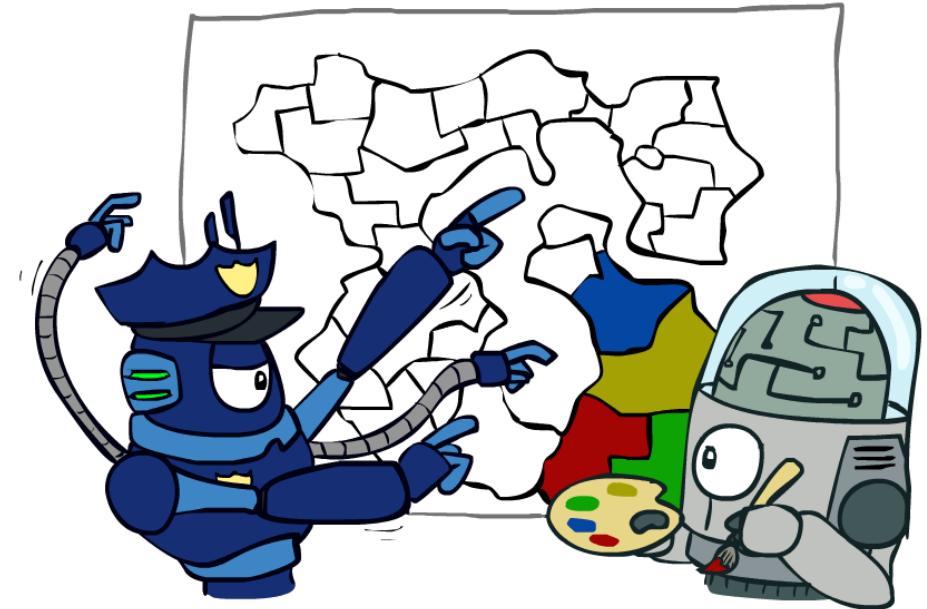
- Binary constraints involve pairs of variables, e.g.:

$$SA \neq WA$$

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

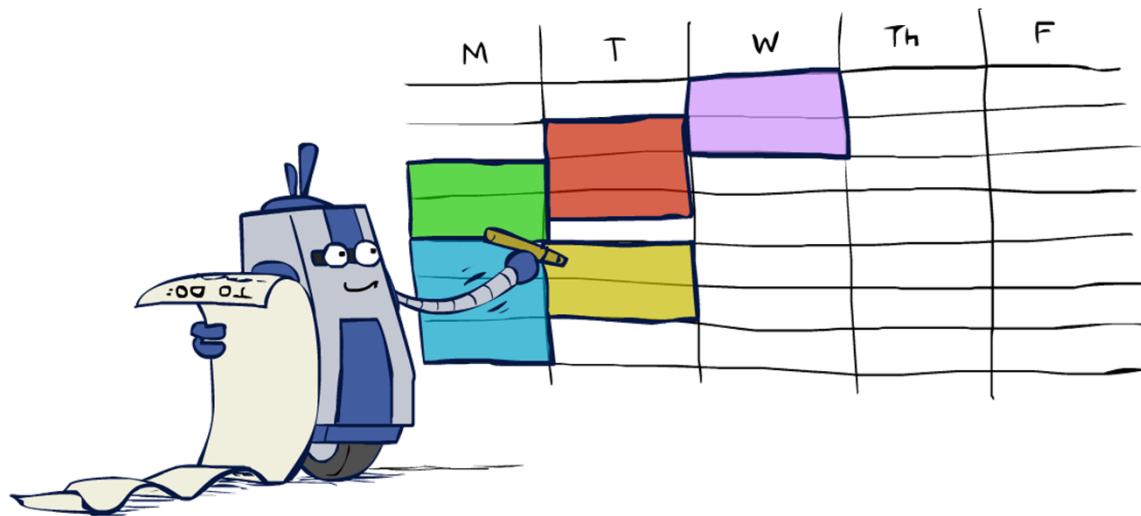
- **Preferences (soft constraints):**

- 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!



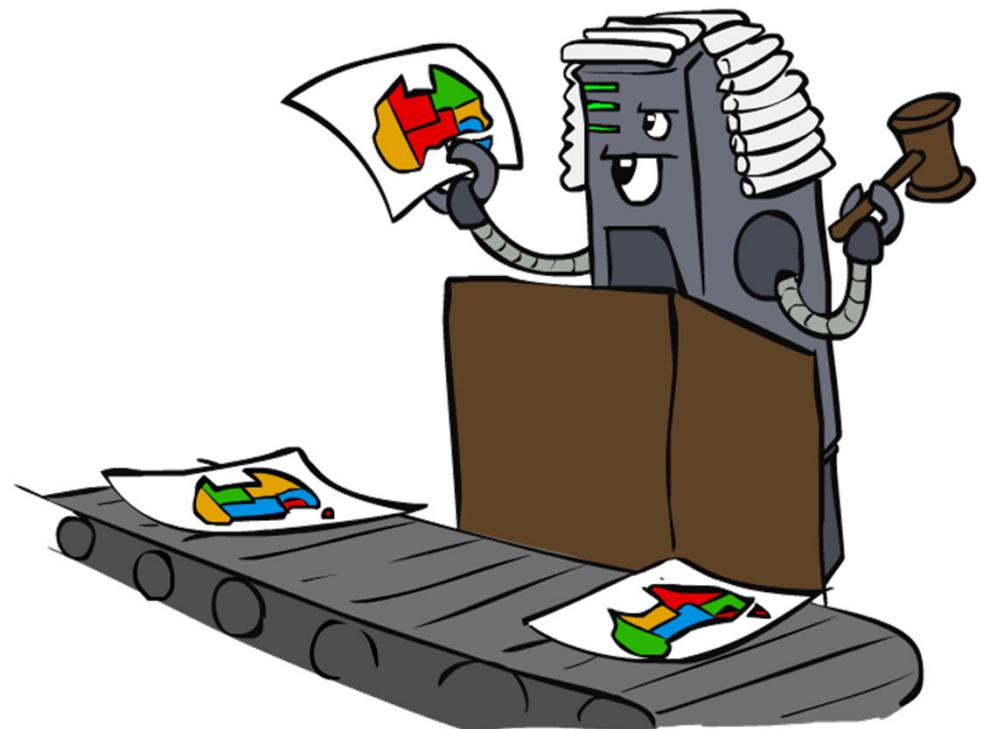
- Many real-world problems involve real-valued variables...

Solving CSPs



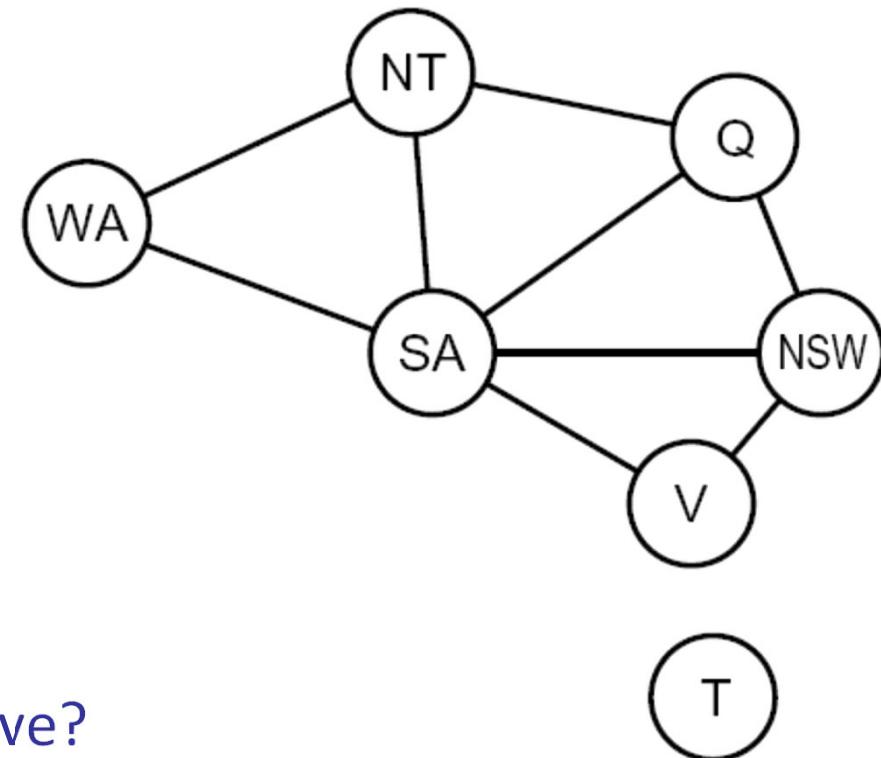
Standard Search Formulation

- 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



Search Methods

- What would BFS do?

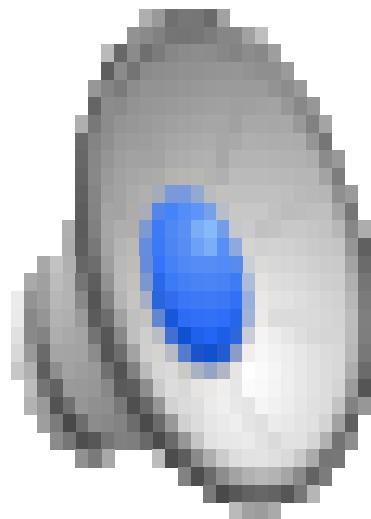


- What would DFS do?

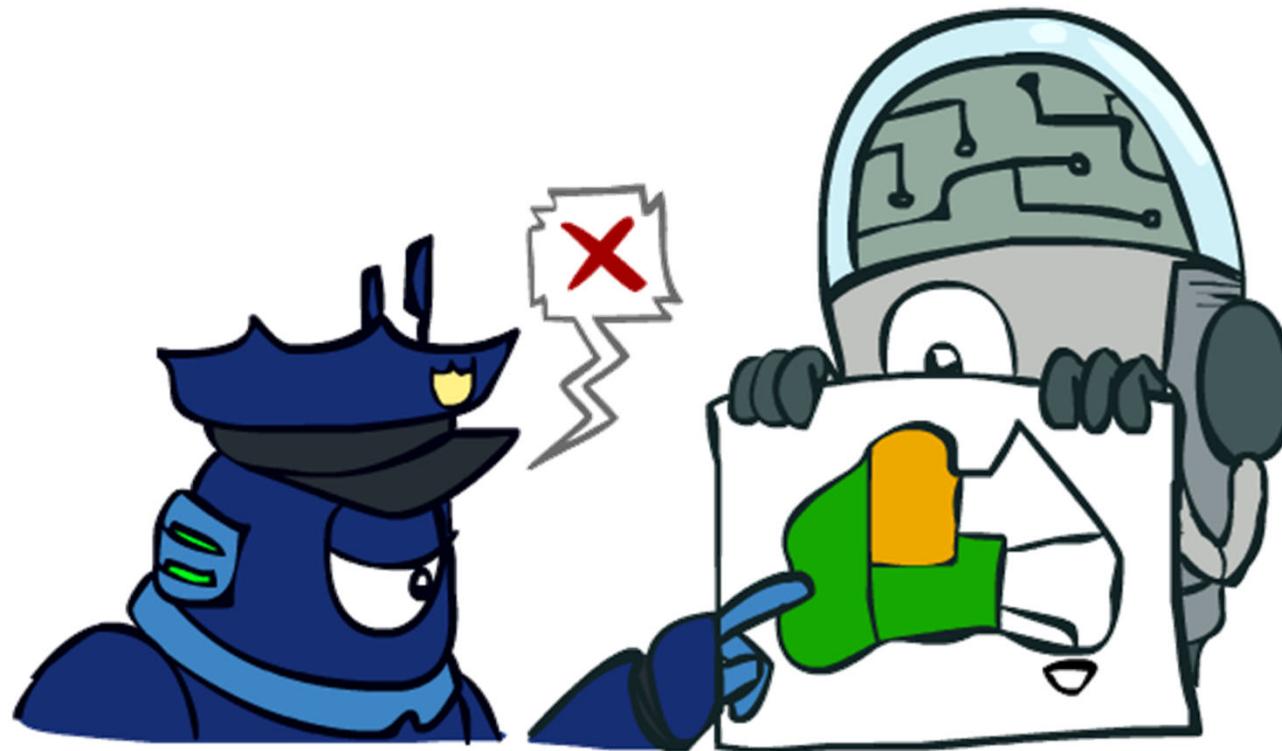
- What problems does naïve search have?

[Demo: coloring -- dfs]

Video of Demo Coloring -- DFS

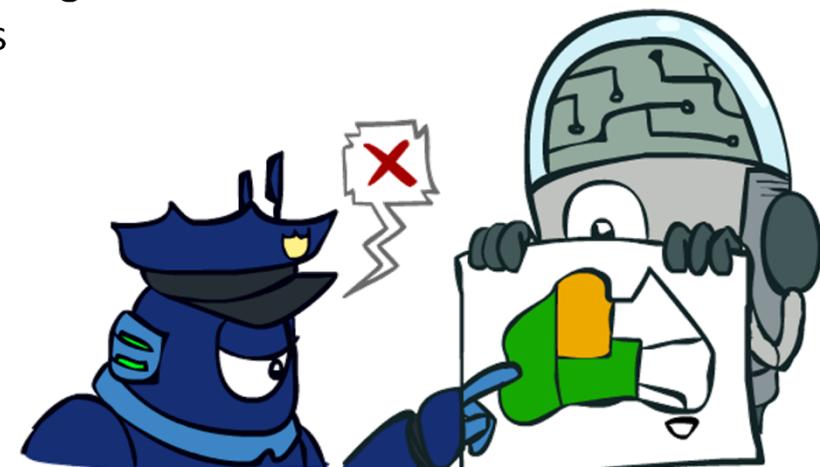


Backtracking Search

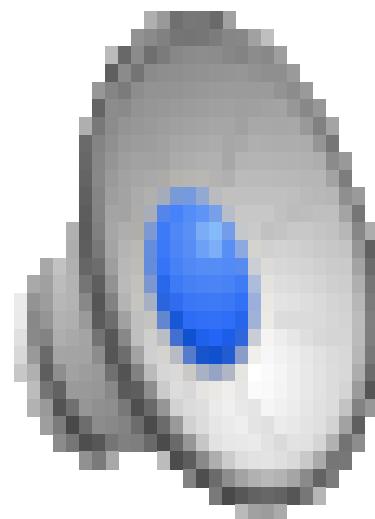


Backtracking Search

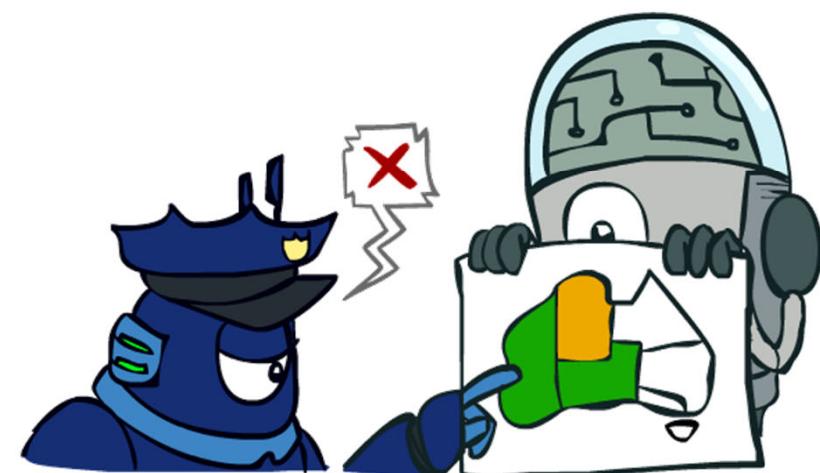
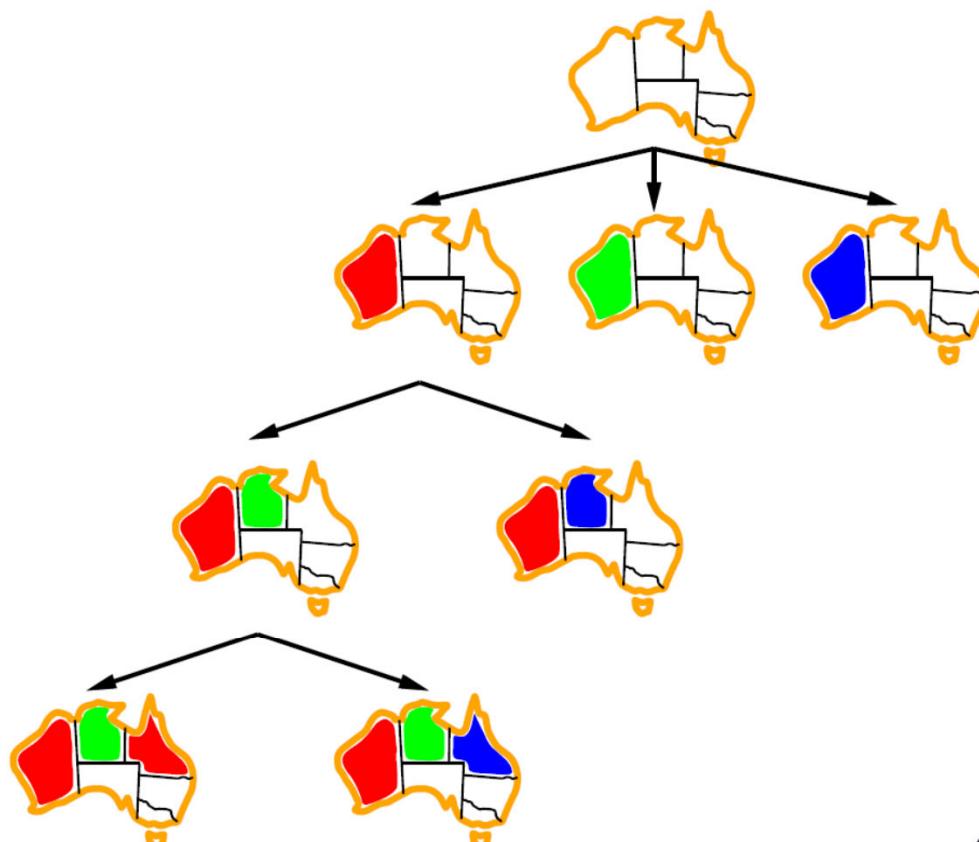
- Backtracking search is the basic uninformed algorithm for solving CSPs
- Idea 1: One variable at a time
 - Variable assignments are commutative, so fix ordering
 - I.e., [WA = red then NT = green] same as [NT = green then WA = red]
 - Only need to consider assignments to a single variable at each step
- 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 $n \approx 25$



Video of Demo Coloring – Backtracking



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  $\leftarrow$  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  $\leftarrow$  RECURSIVE-BACKTRACKING(assignment, csp)
            if result  $\neq$  failure then return result
            remove {var = value} from assignment
    return failure
```

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

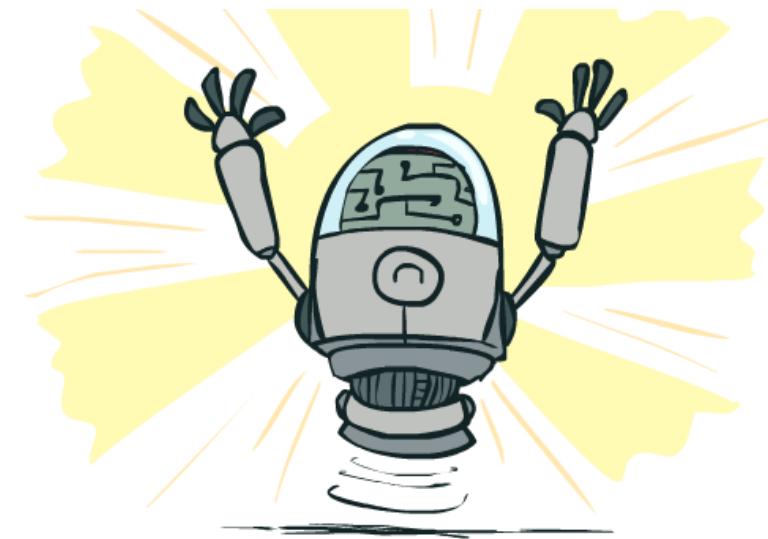
[Demo: coloring -- backtracking]

Improving Backtracking

Problem	Backtracking	BT+MRV	Forward Checking	FC+MRV	Min-Conflicts
USA	(> 1,000K)	(> 1,000K)		2K	60
n-Queens	(> 40,000K)	13,500K	(> 40,000K)	817K	4K

Improving Backtracking

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

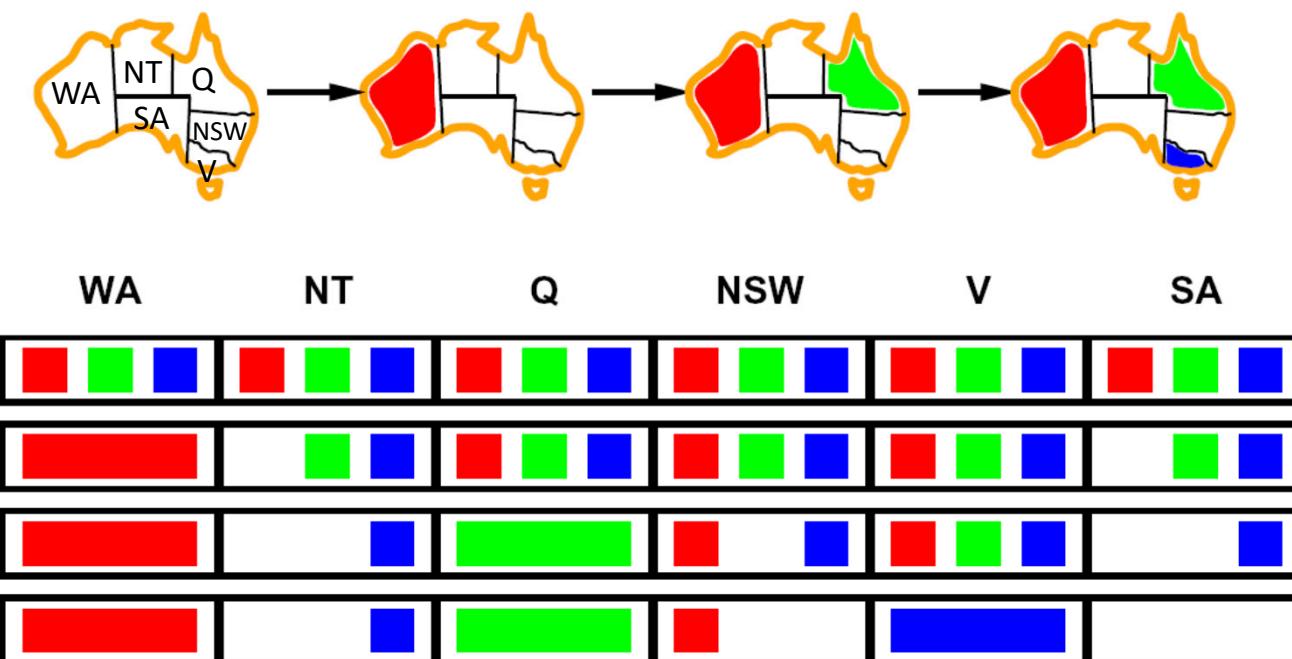


Filtering

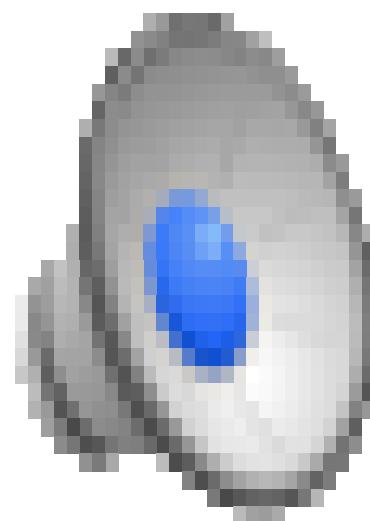


Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking:
 - Cross off values that violate a constraint when added to the existing assignment
 - When selecting a value for a variable, infer new domain reductions on neighboring unassigned variables
 - Terminate search when a variable has no legal value

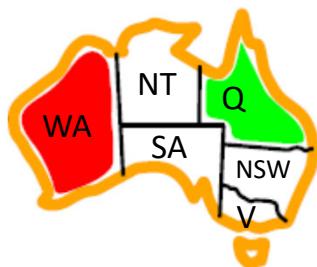


Video of Demo Coloring – Backtracking with Forward Checking



Filtering: Constraint Propagation

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

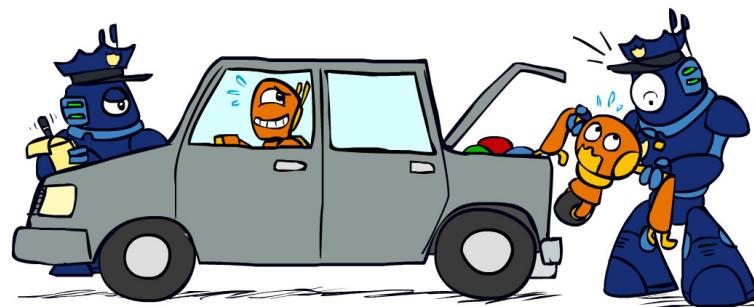


WA	NT	Q	NSW	V	SA
■ Red	■ Green	■ Blue	■ Red	■ Green	■ Blue
■ Red	■ Green	■ Blue	■ Red	■ Green	■ Blue
■ Red	■ Blue	■ Green	■ Red	■ Blue	■ Blue

- 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 $X \rightarrow Y$ is **consistent** iff for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint

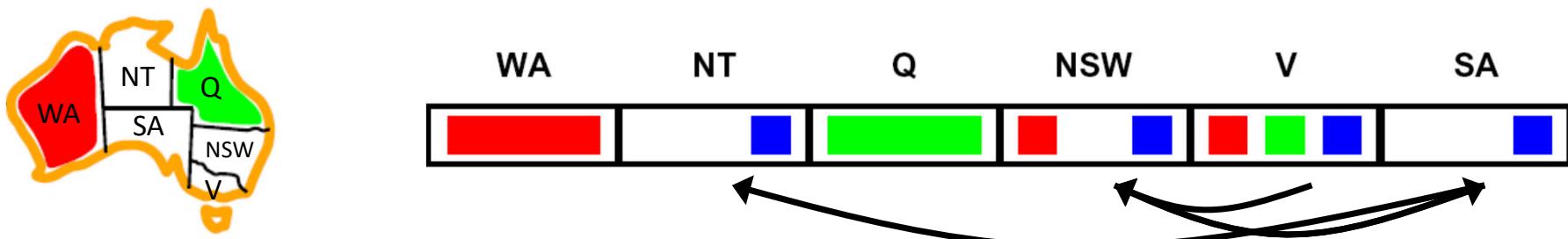


Delete from the tail!

- 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  $\{X_1, X_2, \dots, X_n\}$ 
    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



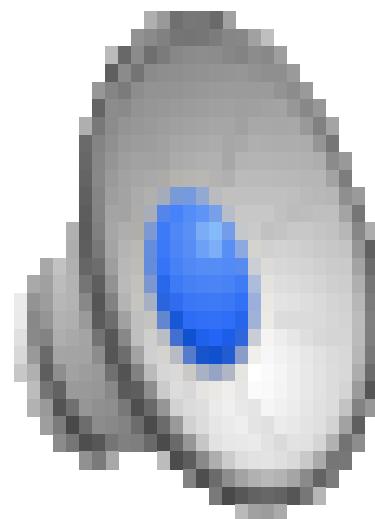
---


function REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) returns true iff succeeds
    removed  $\leftarrow \text{false}$ 
    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  $\leftarrow \text{true}$ 
    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?

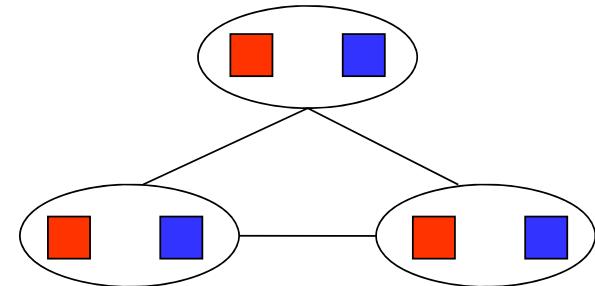
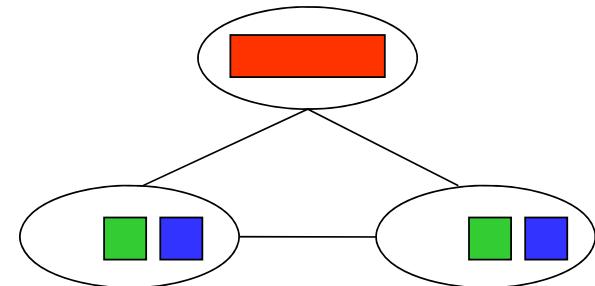
[Demo: CSP applet (made available by aispace.org) -- n-queens]

Video of Demo Arc Consistency – CSP Applet – n Queens



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!



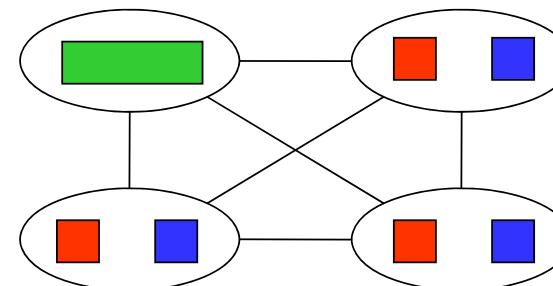
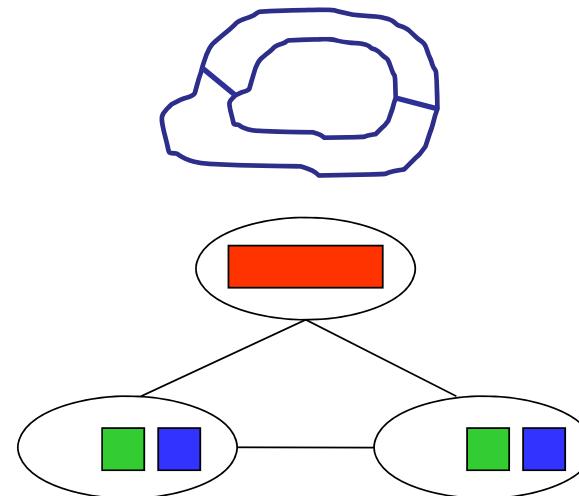
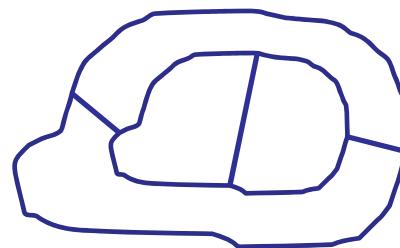
What went wrong here?

[Demo: coloring -- forward checking]
[Demo: coloring -- arc consistency]

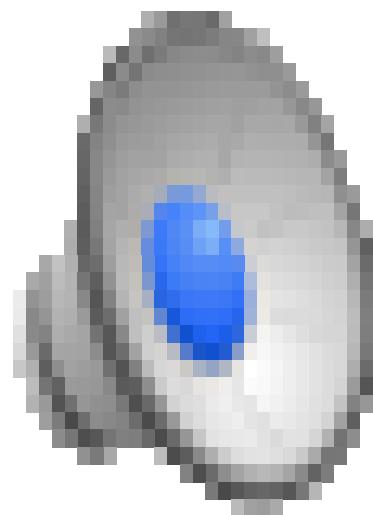
Limitations of Arc Consistency

- After running arc consistency:

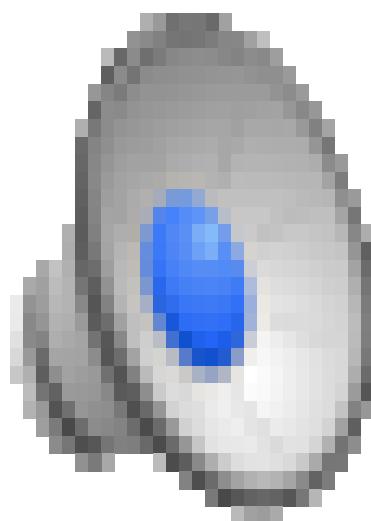
- Can have one solution left
- Can have multiple solutions left
- Can have no solutions left (and not know it)



Video of Demo Coloring – Backtracking with Forward Checking – Complex Graph



Video of Demo Coloring – Backtracking with Arc Consistency – Complex Graph



Ordering

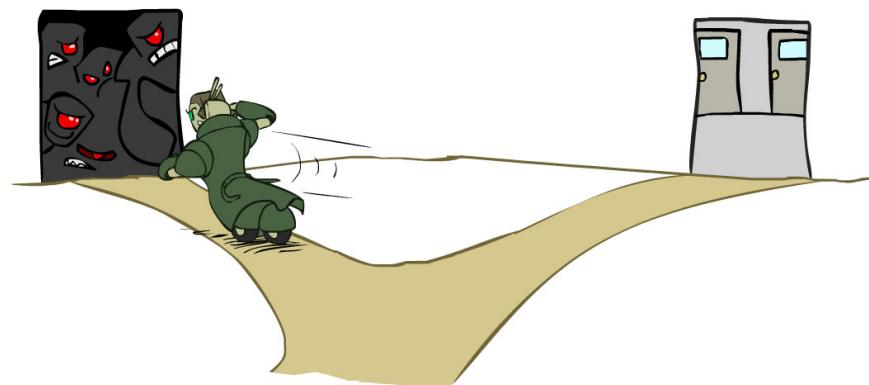


Ordering: Minimum Remaining Values

- 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



Degree Heuristic

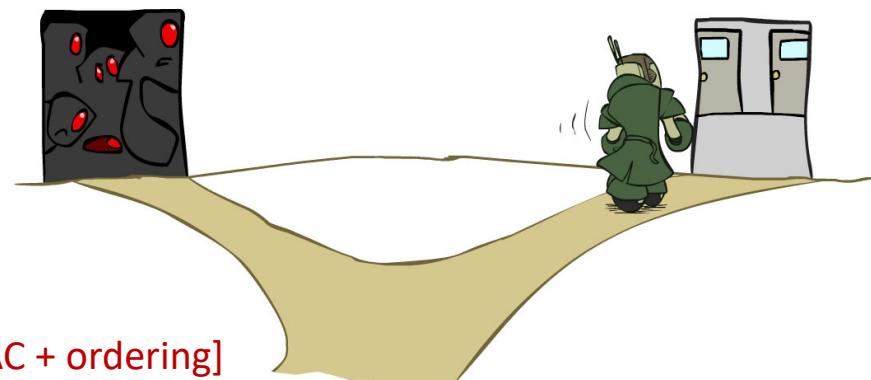
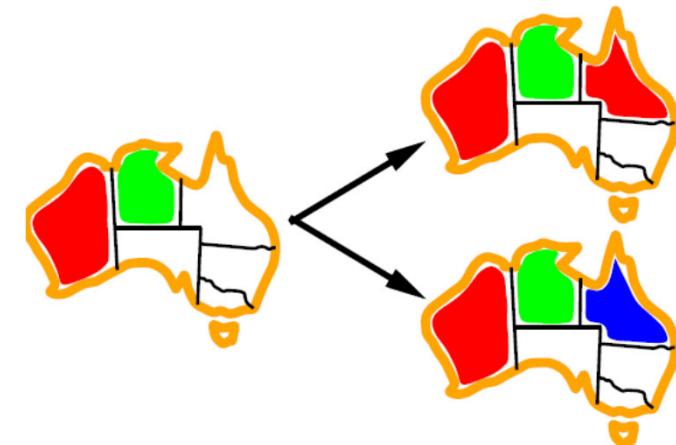
- Tie-breaker among MRV variables
- Degree heuristic:
 - Choose the variable participating in the most constraints on remaining variables



- Why most rather than fewest constraints?

Ordering: Least Constraining Value

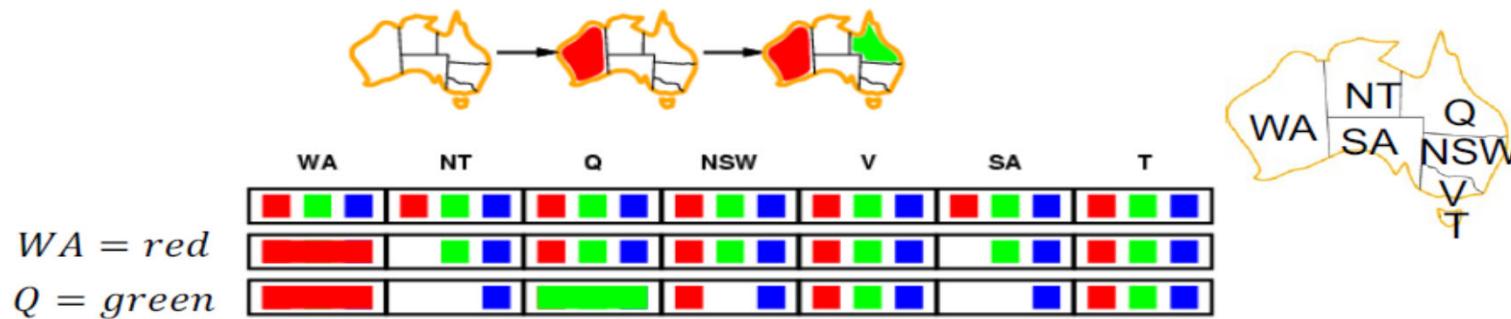
- **Value Ordering: Least Constraining Value**
 - Given a choice of variable, choose the *least constraining value*
 - I.e., the one that rules out the fewest values in the remaining variables
 - Note that it may take some computation to determine this! (E.g., rerunning filtering)
- Why least rather than most?
- Combining these ordering ideas makes 1000 queens feasible



[Demo: coloring – backtracking + AC + ordering]

Constraint propagation

- FC makes the current variable arc-consistent but does not make all the other variables arc-consistent

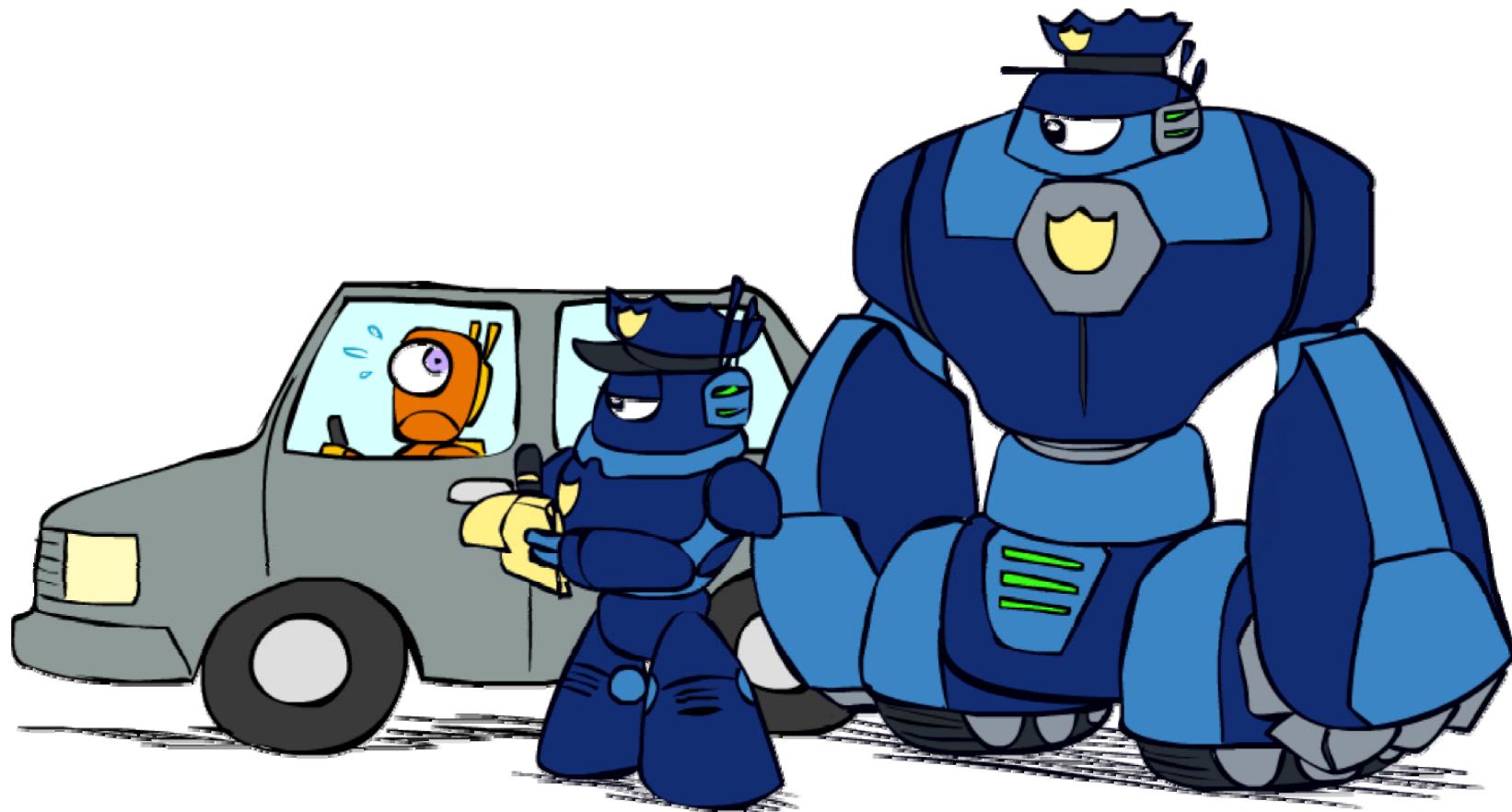


- NT and SA cannot both be blue!
 - FC does not look far enough ahead to find this inconsistency
- **Maintaining Arc Consistency (MAC) - Constraint propagation**
 - Forward checking + recursively propagating constraints when changing domains
 - similar to AC-3 but only arcs related to the current variable are put in the queue at start

Arc consistency: map coloring example

- For general map coloring problem all pairs of variables are arc-consistent if $|D_i| \geq 2(i = 1, \dots, n)$
- In this case, arc consistency as preprocessing can do nothing.
 - Fails to make enough inference
- need stronger notion of consistency to detect failure at start.
- 3-consistency (path consistency): for any consistent assignment to each set of two variables, a consistent value can be assigned to any other variable.

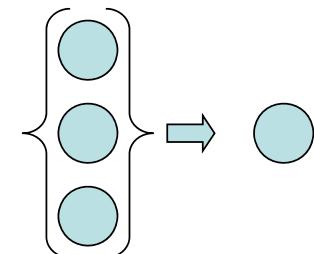
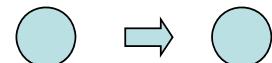
K-Consistency



K-Consistency

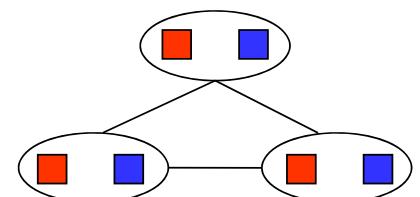
- Increasing degrees of consistency

- 1-Consistency (Node Consistency): Each single node's domain has a value which meets that node's unary constraints
- 2-Consistency (Arc Consistency): For each pair of nodes, any consistent assignment to one can be extended to the other
- K-Consistency: For each k nodes, any consistent assignment to k-1 can be extended to the kth node.



- Higher k more expensive to compute

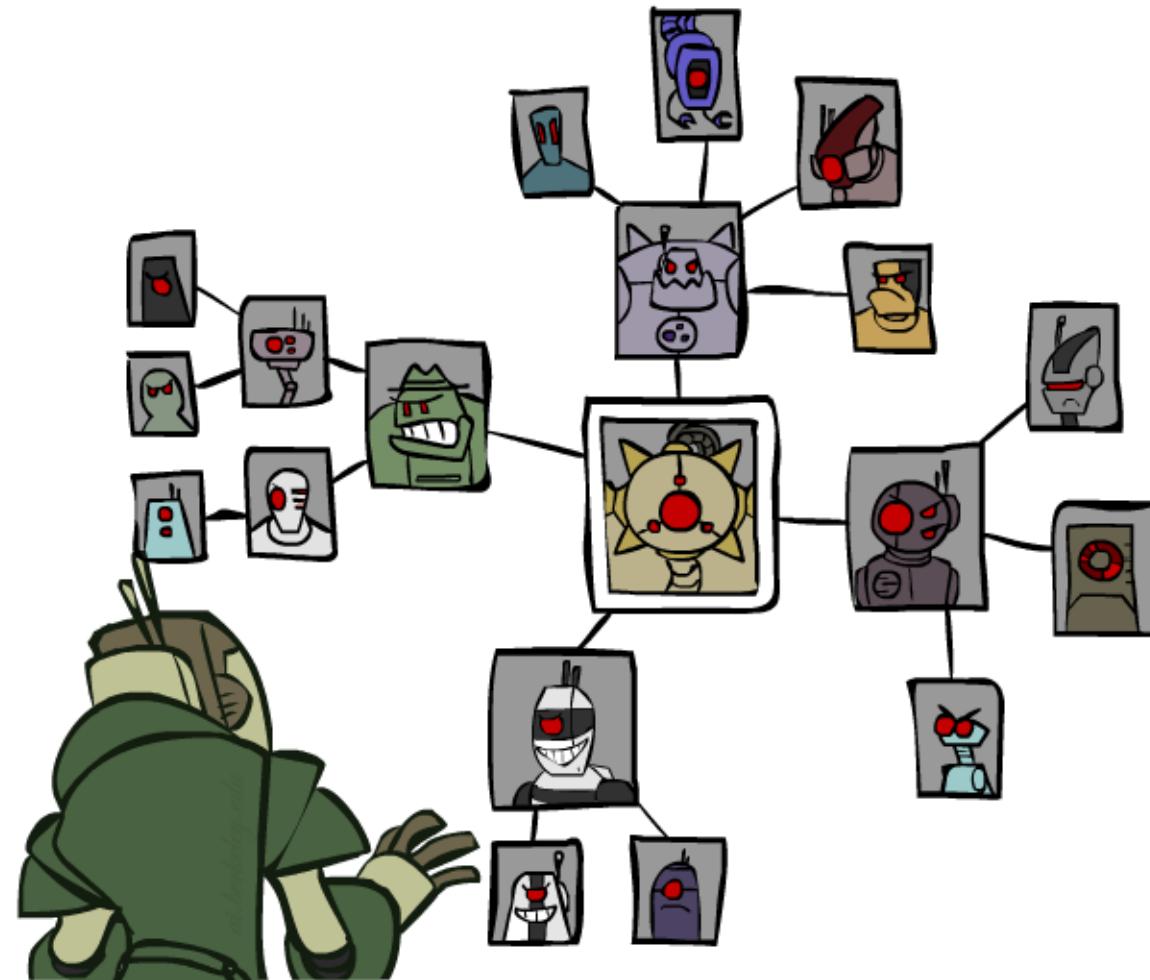
- (You need to know the k=2 case: arc consistency)



Strong K-Consistency

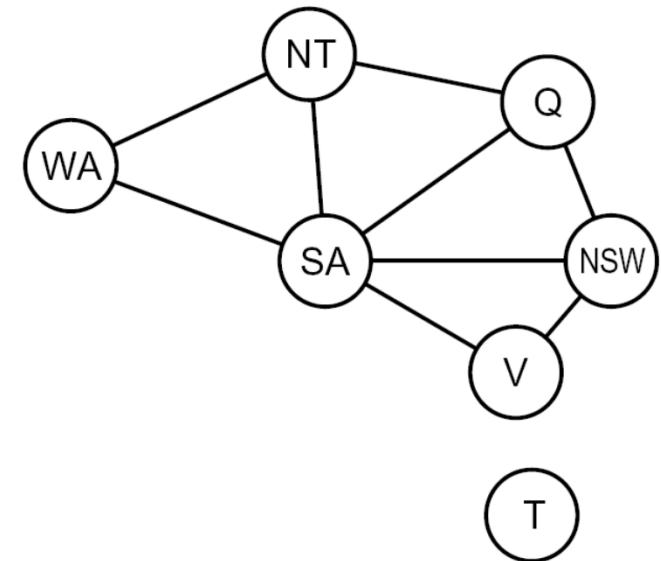
- Strong k-consistency: also k-1, k-2, ... 1 consistent
- Claim: strong n-consistency means we can solve without backtracking!
- Why?
 - Choose any assignment to any variable
 - Choose a new variable
 - By 2-consistency, there is a choice consistent with the first
 - Choose a new variable
 - By 3-consistency, there is a choice consistent with the first 2
 - ...
- Lots of middle ground between arc consistency and n-consistency! (e.g. k=3, called path consistency)

Structure

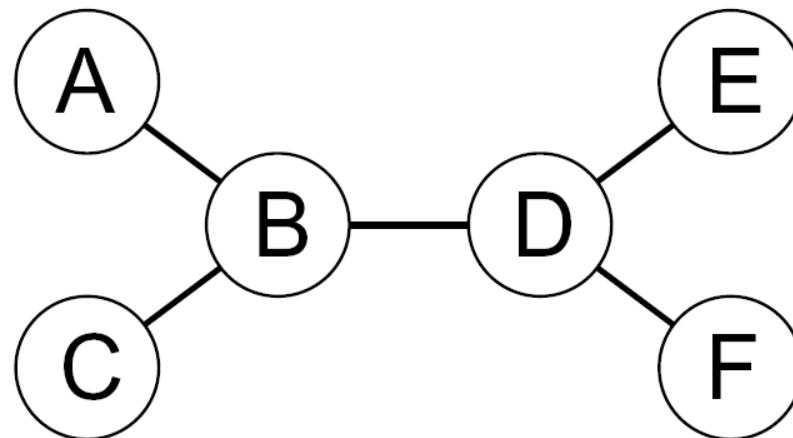


Problem Structure

- General purpose CSP algorithms use graph structure to speed up search
 - Extreme case: independent subproblems
 - Example: Tasmania and mainland do not interact
 - Independent subproblems are identifiable as connected components of constraint graph
- Suppose a graph of n variables can be broken into subproblems of only c variables:
 - Worst-case solution cost is $O((n/c)(d^c))$, linear in n
 - E.g., $n = 80$, $d = 2$, $c = 20$
 - $2^{80} = 4$
 - $(4)(2^{20}) = 0.4$ seconds at 10 million nodes/sec



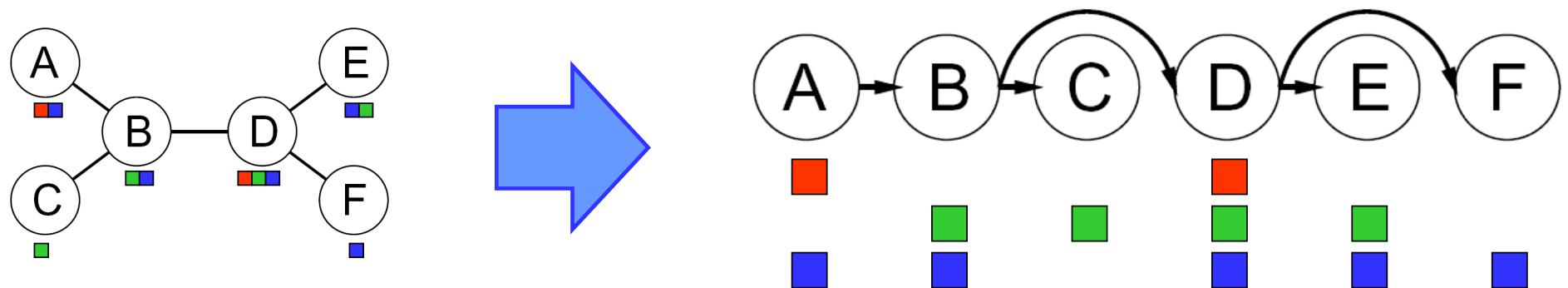
Tree-Structured CSPs



- Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n d^2)$ time
 - Compare to general CSPs, where worst-case time is $O(d^n)$
- This property also applies to probabilistic reasoning (later): an example of the relation between syntactic restrictions and the complexity of reasoning

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($\text{Parent}(X_i), X_i$)
- Assign forward: For $i = 1 : n$, assign X_i consistently with $\text{Parent}(X_i)$
- Runtime: $O(n d^2)$ (why?)



Tree-Structured CSPs- Algorithm

```
function TREE_CSP_SOLVER(csp) returns a solution or failure
```

input: *csp*, a CSP with components X, D, C

$n \leftarrow$ number of variables in X

assignment \leftarrow an empty assignment

root \leftarrow any variable in X

$X \leftarrow \text{TOPOLOGICAL}(X, \text{root})$

for $j = n$ **down to** 2 **do**

MAKE_ARC_CONSISTENT(*PARENT*(X_j), X_j)

if it cannot be made consistent **then return** failure

for $i = 1$ **to** n **do**

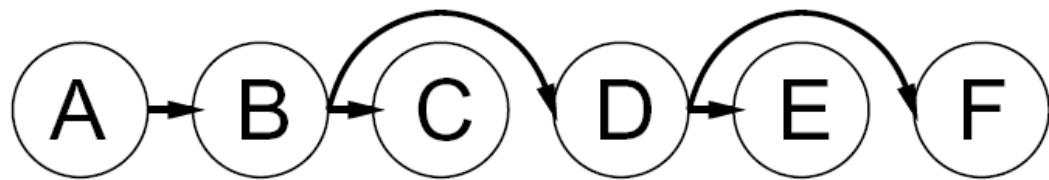
assignment[X_i] \leftarrow anyconsistent value from D_i

if there is no consistent value **then return** failure

return *assignment*

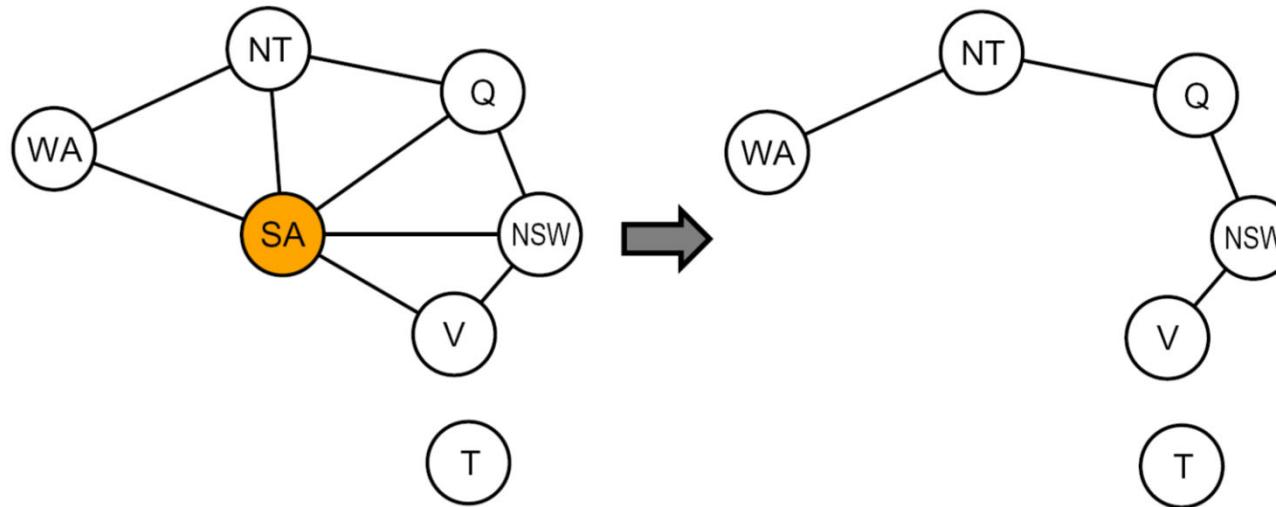
Tree-Structured CSPs

- Claim 1: After backward pass, all root-to-leaf arcs are consistent
- Proof: Each $X \rightarrow Y$ was made consistent at one point and Y 's domain could not have been reduced thereafter (because Y 's children were processed before Y)



- Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack
- Proof: Induction on position
- Why doesn't this algorithm work with cycles in the constraint graph?
- Note: we'll see this basic idea again with Bayes' nets

Nearly Tree-Structured CSPs



- Conditioning: instantiate a variable, prune its neighbors' domains
- Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree
- Cutset size c gives runtime $O((d^c) (n-c) d^2)$, very fast for small c

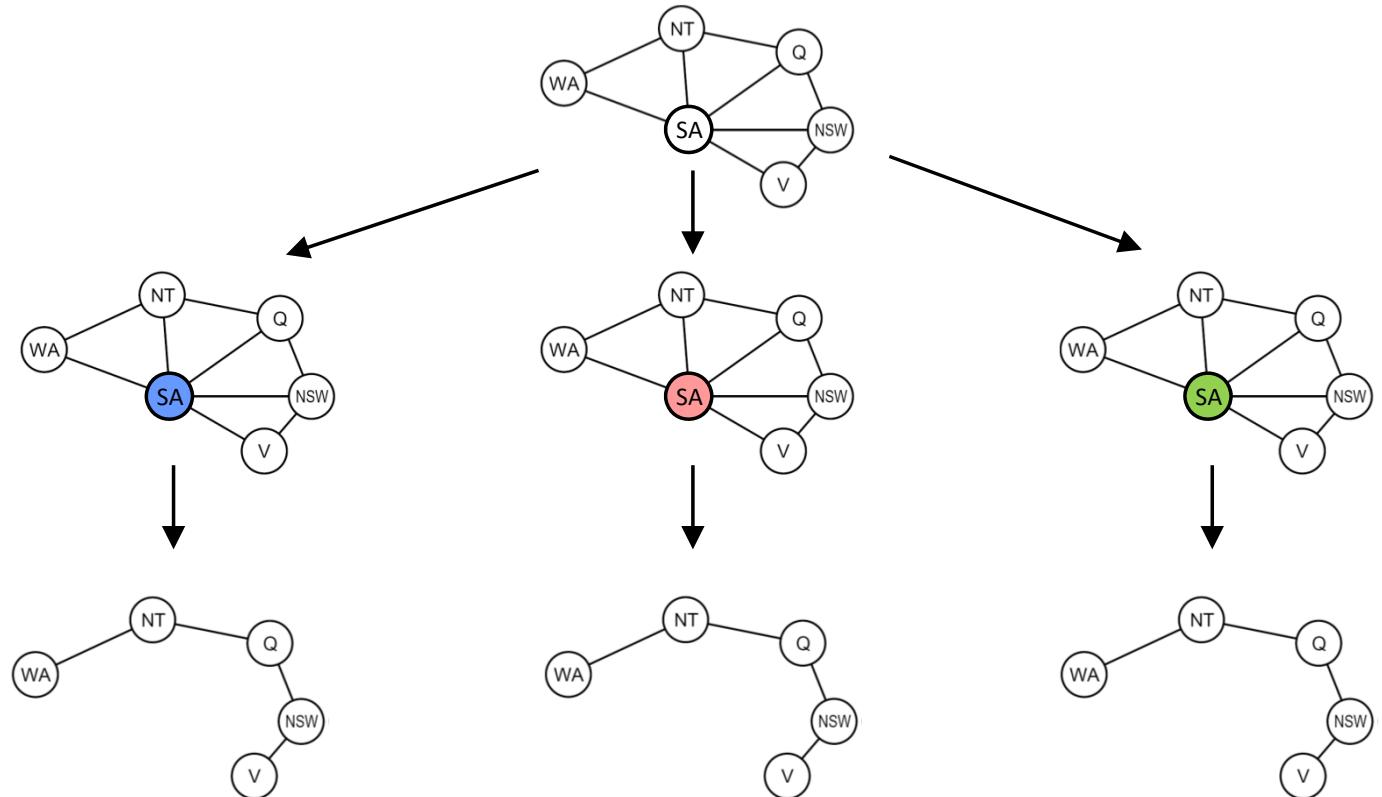
Cutset Conditioning

Choose a cutset

Instantiate the cutset
(all possible ways)

Compute residual CSP
for each assignment

Solve the residual CSPs
(tree structured)

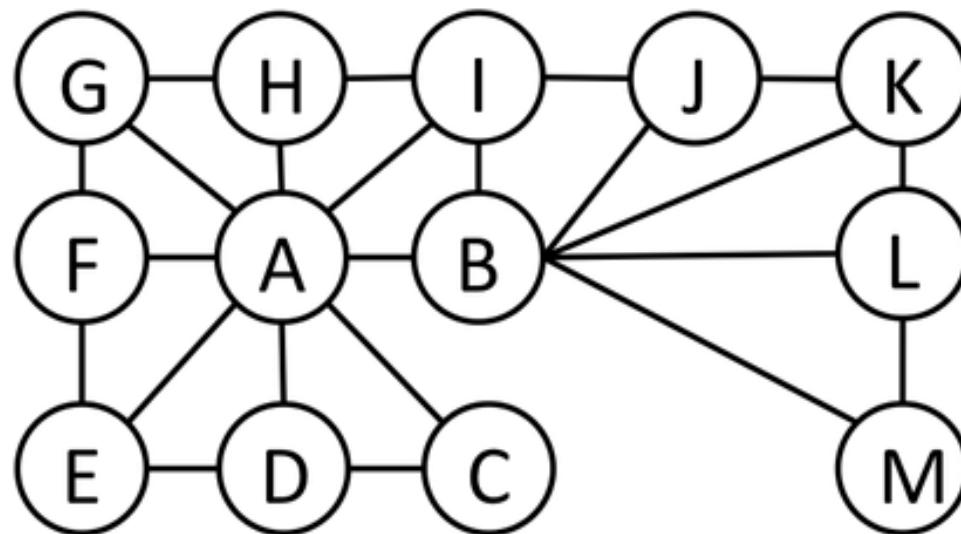


Cut set conditioning

- 1) find a subset S such that the remaining graph become a tree
- 2) for each possible consistent assignment to S
 - Remove inconsistent values from domains of remaining variables
 - Solve the remaining CSP which has a tree structure

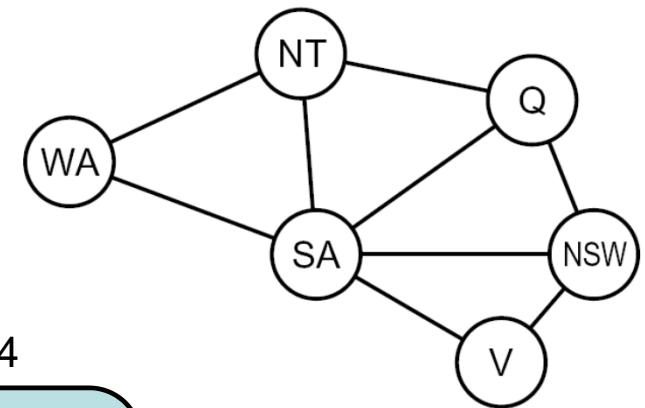
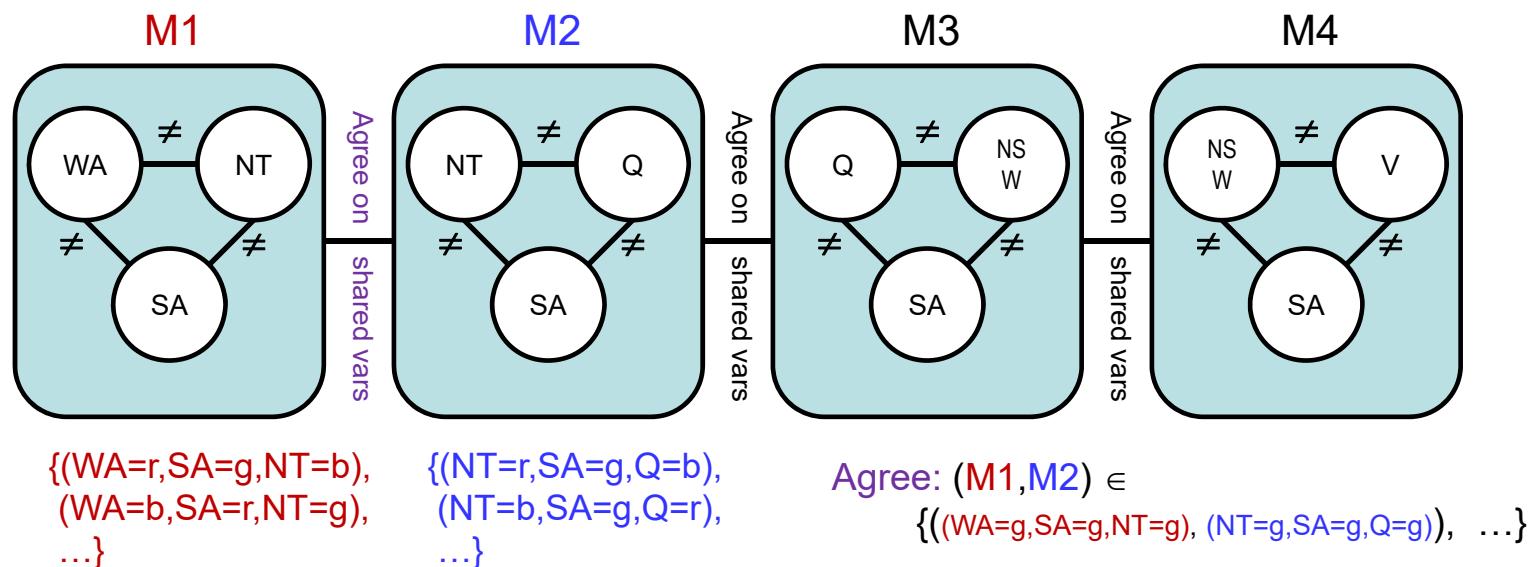
Cutset Quiz

- Find the smallest cutset for the graph below.



Tree Decomposition*

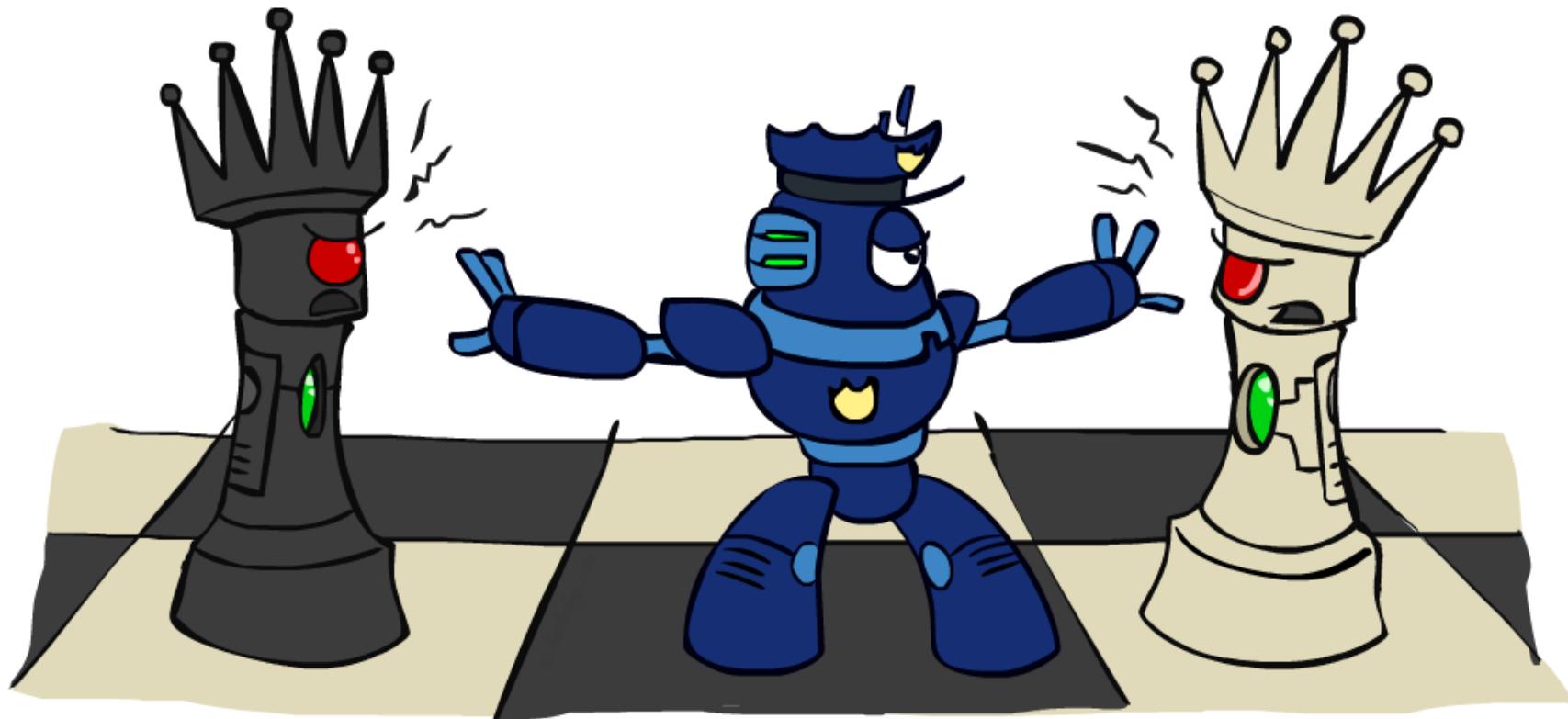
- Idea: create a tree-structured graph of mega-variables
- Each mega-variable encodes part of the original CSP
- Subproblems overlap to ensure consistent solutions



Tree Decomposition

- **Conditions :**
 - Every variable in the original problem appears in at least one of the tree nodes.
 - If two variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the tree nodes.
 - If a variable appears in two nodes in the tree, it must appear in every node along the path connecting those nodes
- The first two conditions ensure that all the variables and constraints are represented in the tree decomposition
- The third condition ensure that any variable from the original problem have the same value wherever it appears

Iterative Improvement

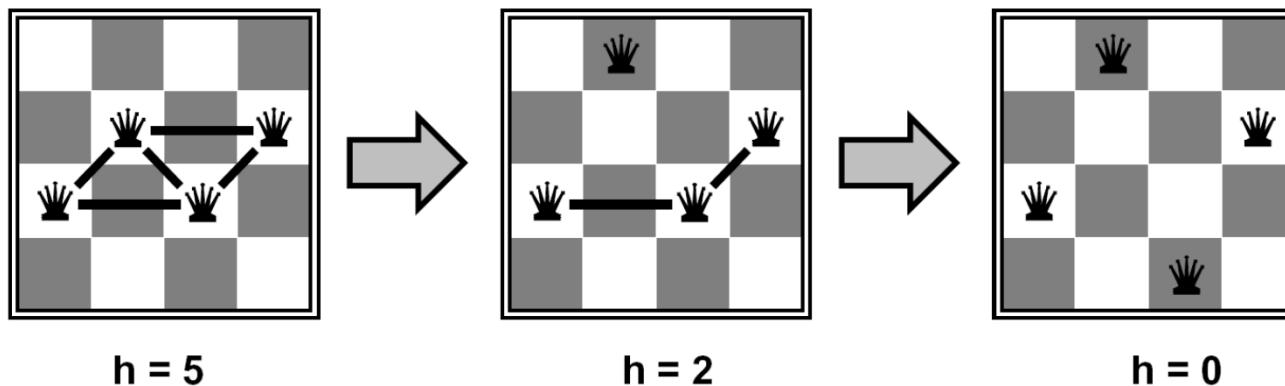


Iterative Algorithms for CSPs

- Local search methods typically work with “complete” states, i.e., all variables assigned
- To apply to CSPs:
 - Take an assignment with unsatisfied constraints
 - Operators *reassign* variable values
 - No fringe! Live on the edge.
- Algorithm: While not solved,
 - Variable selection: randomly select any conflicted variable
 - Value selection: min-conflicts heuristic:
 - Choose a value that violates the fewest constraints
 - I.e., hill climb with $h(n) = \text{total number of violated constraints}$



Example: 4-Queens



- States: 4 queens in 4 columns ($4^4 = 256$ states)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation: $c(n) = \text{number of attacks}$

[Demo: n-queens – iterative improvement (L5D1)]
[Demo: coloring – iterative improvement]

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
- Iterative min-conflicts is often effective in practice

