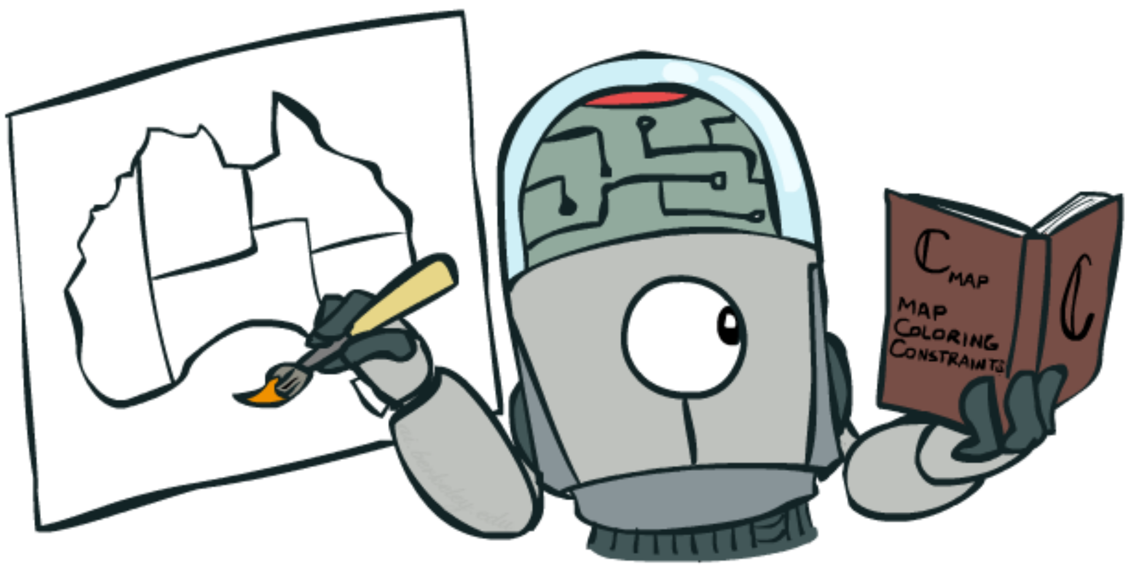
Constraint Satisfaction Problems

CE417: Introduction to Artificial Intelligence
Sharif University of Technology
Fall 2022

Soleymani

“Artificial Intelligence: A Modern Approach”, 3rd Edition, Chapter 6
Most slides have been adapted from Klein and Abdeel, CS188, UC Berkeley.

Constraint Satisfaction Problems



Outline

- Constraint Satisfaction Problems (CSP)
  - Representation for wide variety of problems
  - CSP solvers can be faster than general state-space searchers
- Backtracking search for CSPs
- Inference in CSPs
- Problem Structure
- Local search for CSPs

What is CSPs?

- In CSPs, the problem is to search for a set of values for the variables (features) so that the assigned values satisfy constraints.

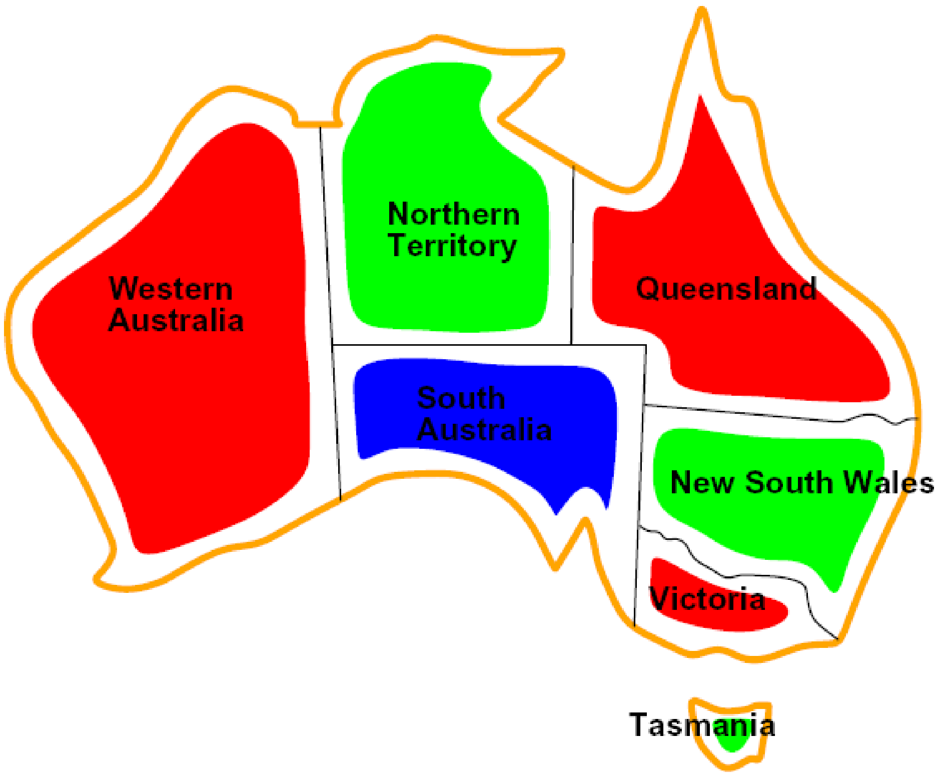
- CSPs yield a natural representation for a **wide variety of problems**
  - CSP search algorithms use <u>general-purpose heuristics</u> based on the structure of states

What is CSPs?

- Components of a CSP
  - X is a set of variables $\{X_1, X_2, \ldots, X_n\}$
  - D is the set of domains $\{D_1, D_2, \ldots, D_n\}$ where D_i is the domain of X_i
  - C is a set of constraints $\{C_1, C_2, \ldots, C_m\}$
    - Each constraint limits the values that variables can take (e.g., $X_1 \neq X_2$)

- Solving a CSP
  - State: An assignment of values to some or all of the variables
  - Solution (goal): A <u>complete</u> and <u>consistent</u> assignment
    - Consistent: An assignment that does not violate any constraint
    - Complete: All variables are assigned.

CSP Examples

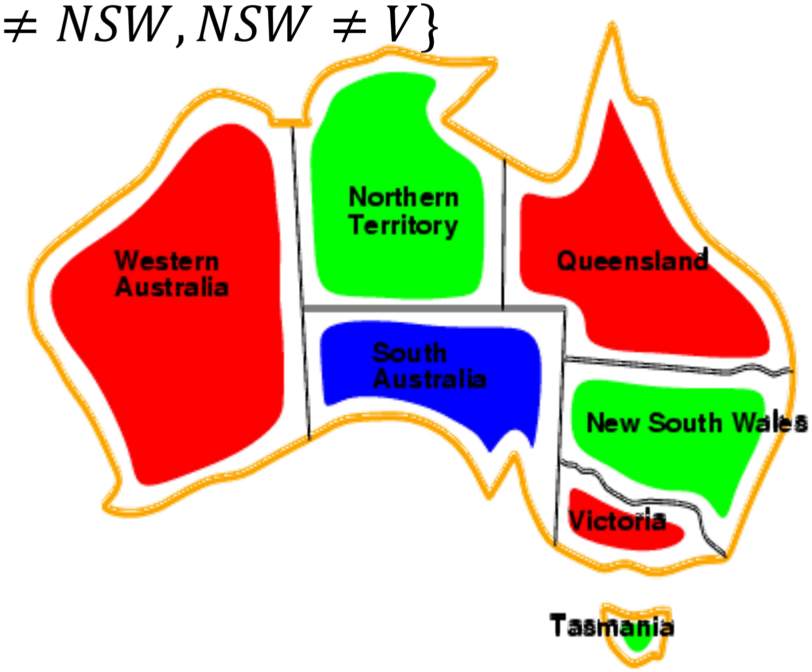


CSP: Map coloring example

- Coloring regions with tree colors such that no neighboring regions have the same color
  - Variables corresponding to regions: $X = \{WA, NT, Q, NSW, V, SA, T\}$
  - The domain of all variables is $\{red, green, blue\}$
  - Constraints: adjacent regions must have different colors

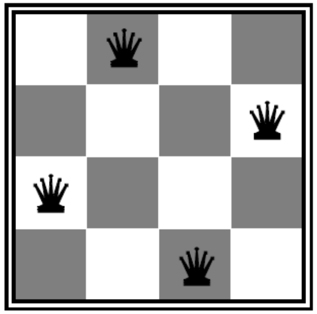
$$C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, S \neq V, WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}$$

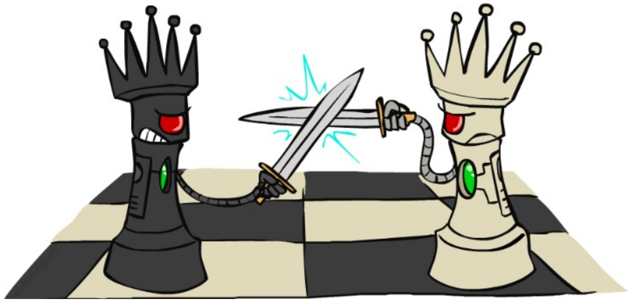
- A solution:
$\{WA = red, NT = green, Q = red, NSW = green, V = red, T = green\}$



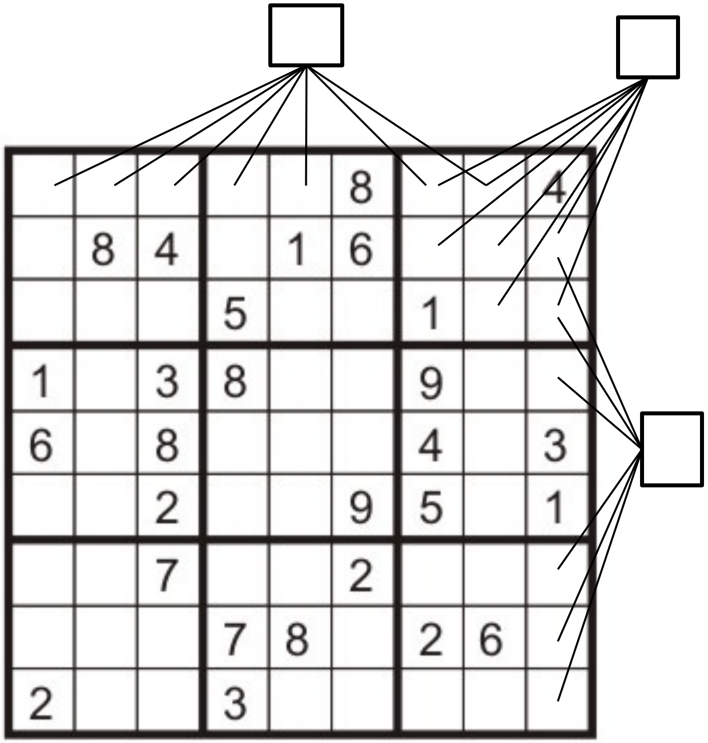
Example: N-Queens

- Variables: $\{Q_1, Q_2, \dots, Q_N\}$
- Domains: $\{1,2,\dots,N\}$
- Constraints:
  - Implicit: $\forall i, j \neq i \;\; non\_threatening(Q_i, Q_j)$
  - Explicit: $(Q_i, Q_j) \in \{(1,3), (1,4), \dots, (8,6)\}$





Example: Sudoku



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

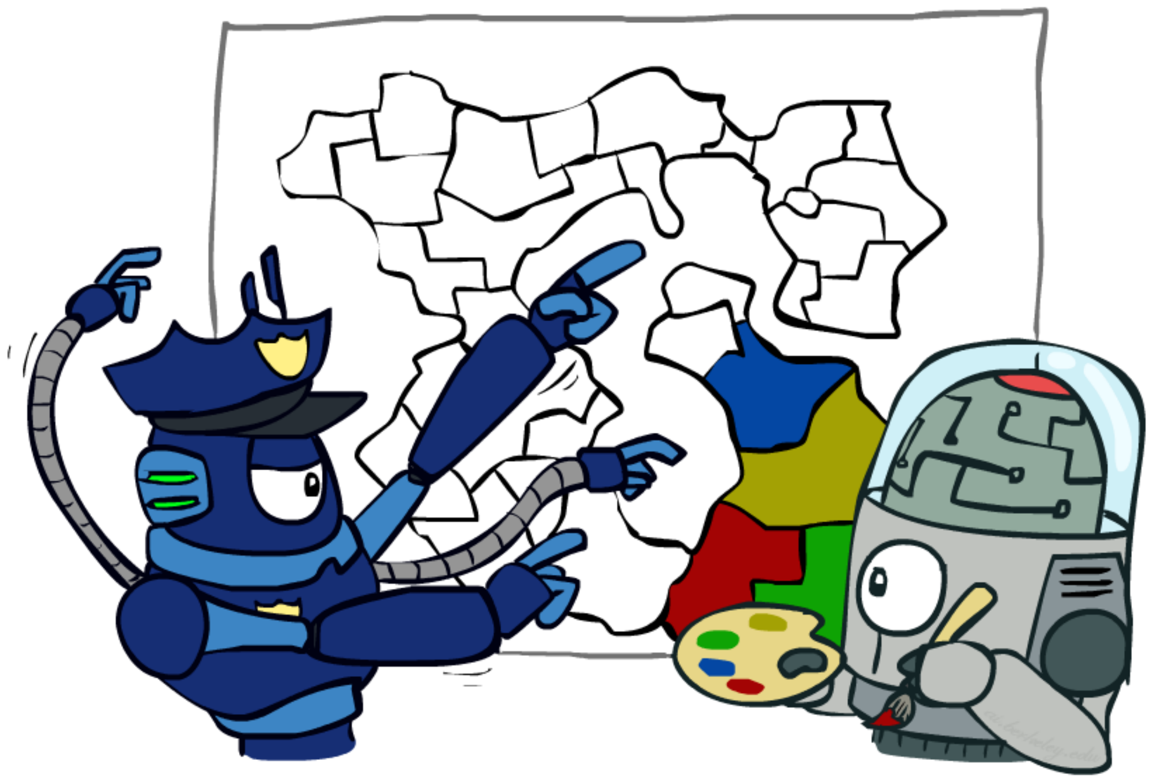
  9-way alldiff for each column

  9-way alldiff for each row

  9-way alldiff for each region

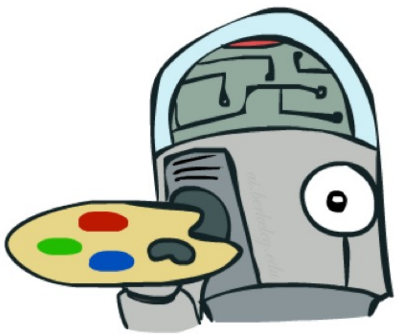
  (or can have a bunch of pairwise inequality constraints)

Varieties of CSPs 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

Varieties of constraints

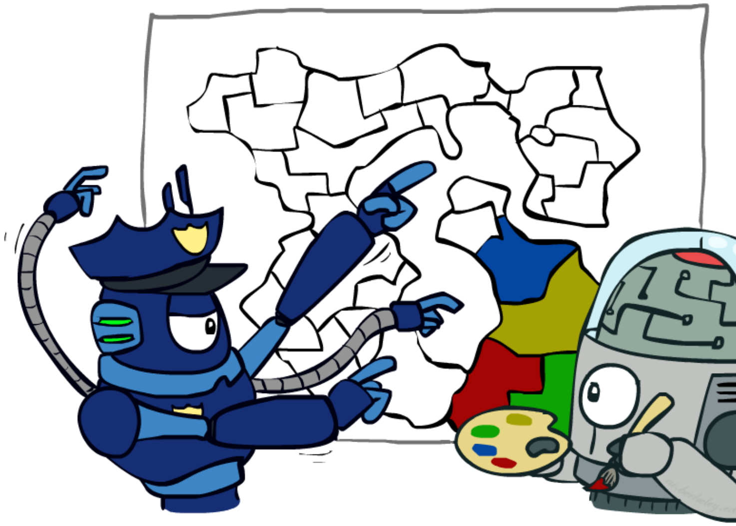
- Varieties of Constraints
  - Unary constraints involve a single variable (equivalent to reducing domains), e.g.:

$$SA \neq green$$

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

$$SA \neq WA$$

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

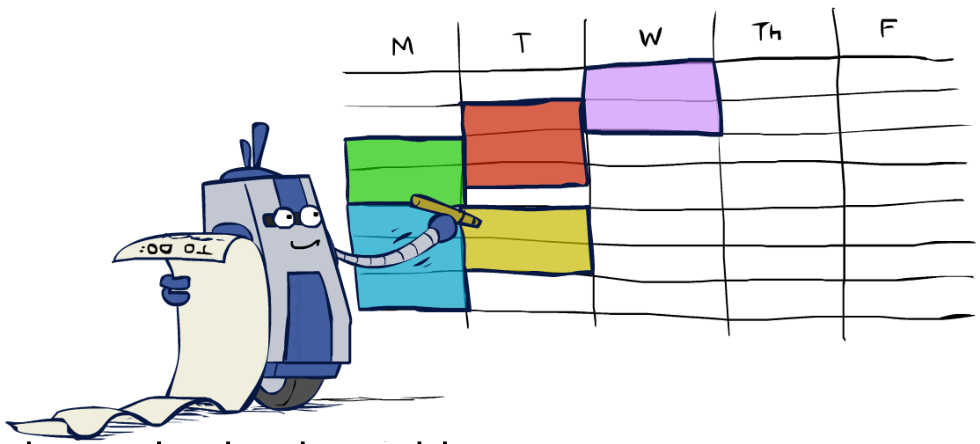


- Preferences (soft constraints):
  - E.g., red is better than green
  - Often representable by a cost for each variable assignment
  - Gives constrained optimization problems

  (We'll ignore these until we get to Bayes' nets)

Real-world CSPs

- 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!



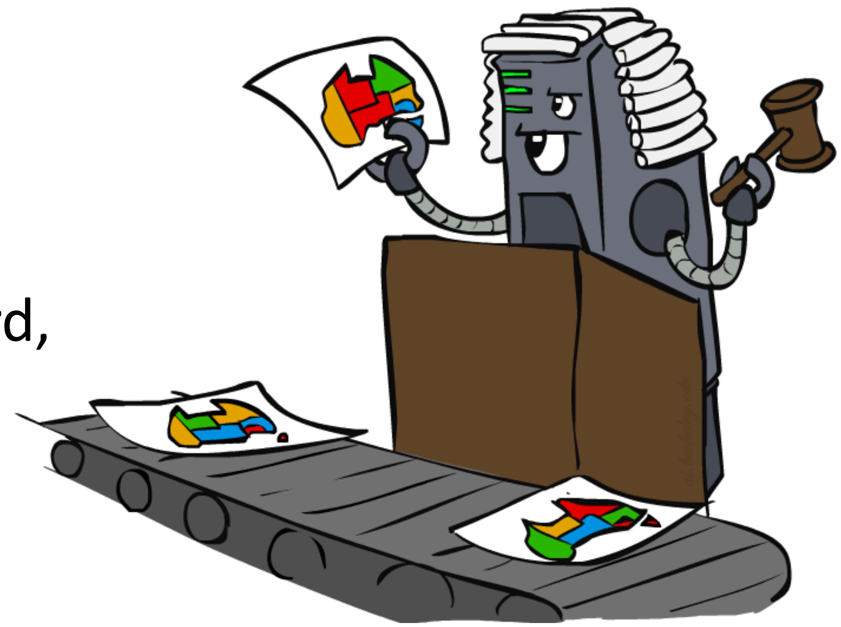
- Many real-world problems involve real-valued variables…

Solving CSPs

Solving CSPs as a systematic search problem

- **Initial State:** No assignment { }
- **Actions or successor function:** assign a value to an unassigned variable that does not conflict with current assignment
- **Goal test:** Consistent & complete assignment
- **Path cost:** not important

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



Properties of CSPs as a systematic search problem

- Generic problem formulation: same formulation for all CSPs

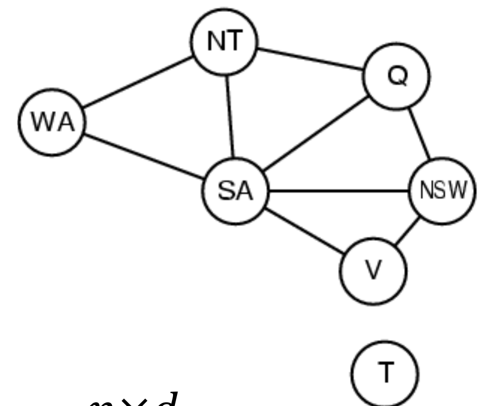
- Every solution appears at depth n with n variables

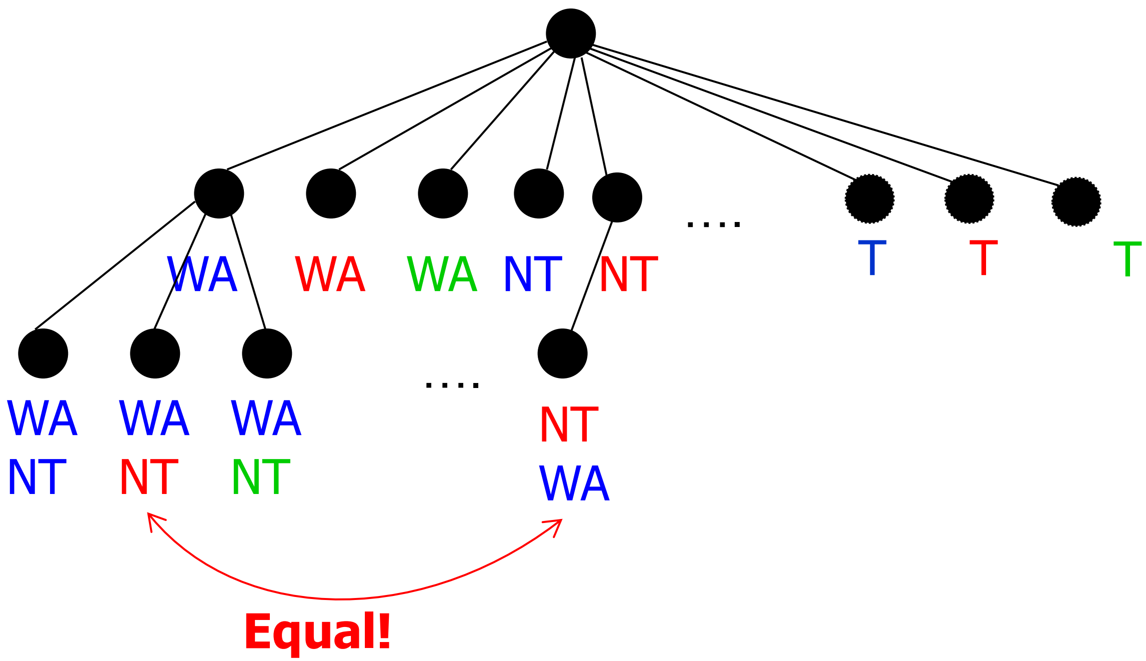
- Which search algorithm is proper?
  - Depth-limited search

- Branching factor is nd at the top level, $b = (n - l)d$ at depth l, hence there are $n!\,d^n$ leaves.
  - However, there are only d^n complete assignments.

Assignment community

- When assigning values to variables, we reach the same partial assignment regardless of the order of variables





$n \times d$

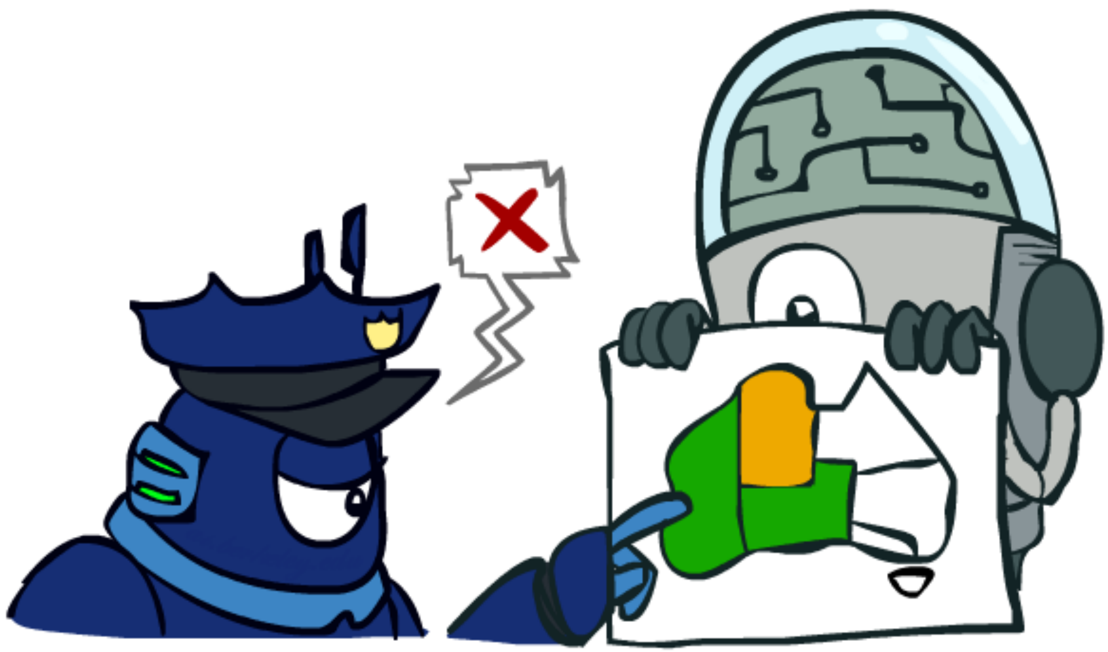
$(n \times d) \times ((n-1) \times d)$

…

$n! \times d^n$

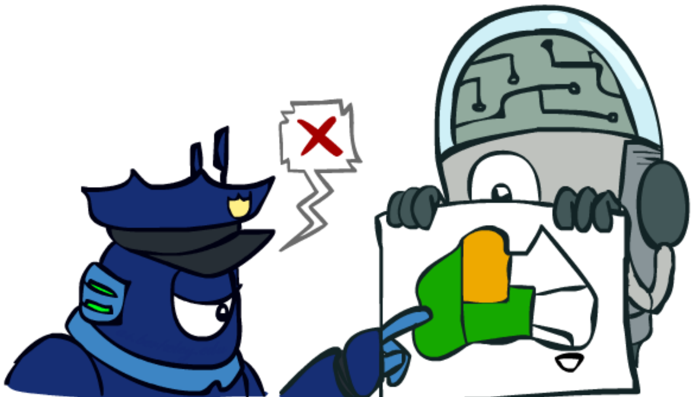
There are $n! \times d^n$ leaves in the tree but only d^n distinct states!

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 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 search

- Depth-first search for CSPs with single-variable assignments is called backtracking search
  - assigns one variable at each level (eventually they all have to be assigned.)

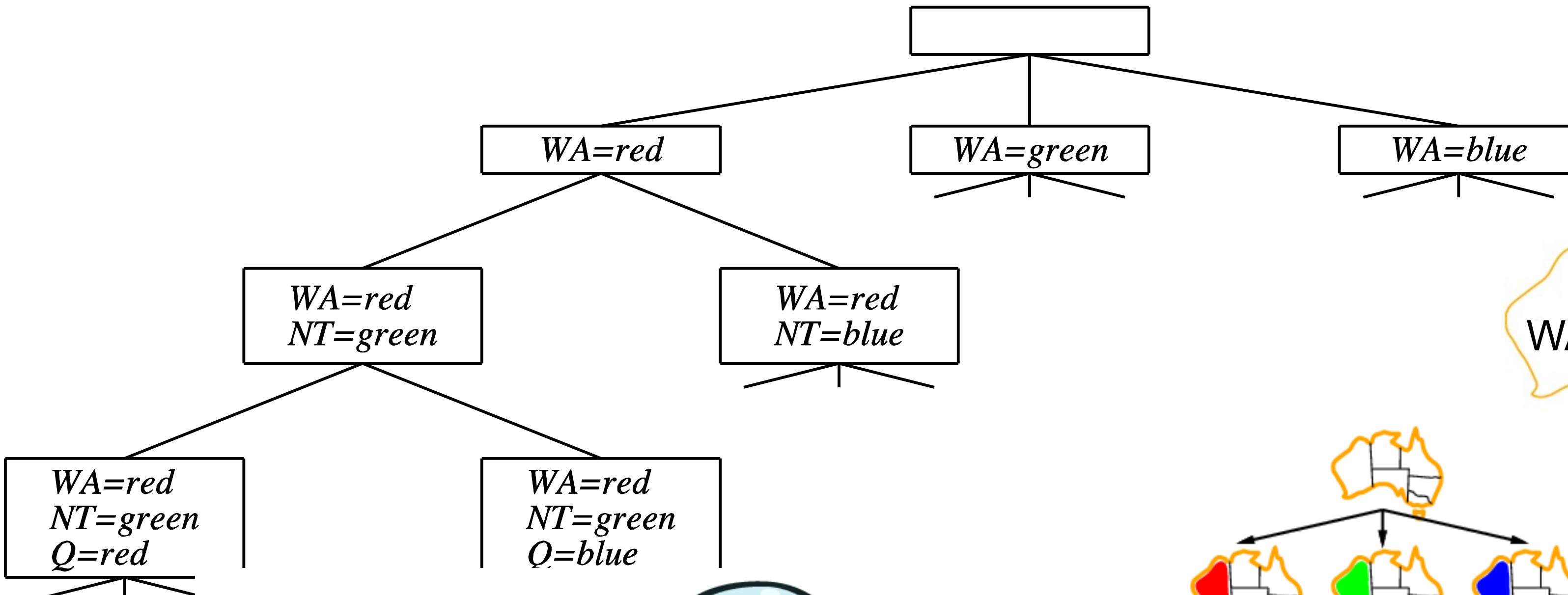
- Naïve backtracking is not generally efficient for solving CSPs.
  - More heuristics will be introduced later to speedup it.

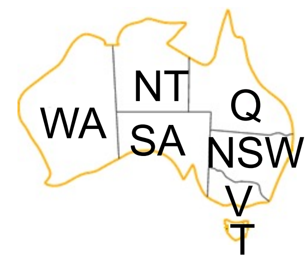
Backtracking search

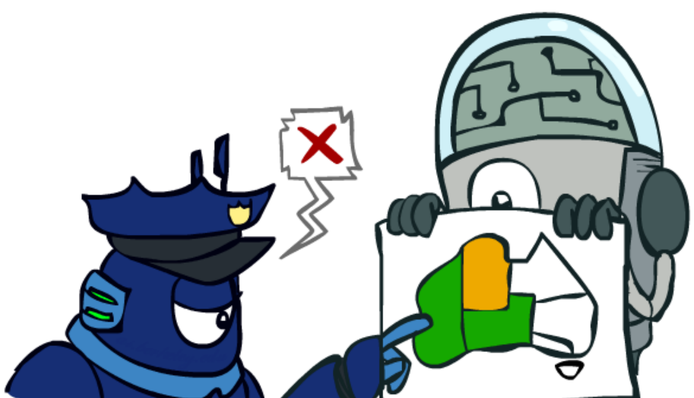
- Nodes are partial assignments
- Incremental completion
  - Each partial candidate is the parent of all candidates that differ from it by a single extension step.
- Traverses the search tree in depth first order.
- At each node *c*
  - If it cannot be completed to a valid solution, the whole sub-tree rooted at *c* is skipped (not promising branches are *pruned*).
  - Otherwise, the algorithm (1) checks whether *c* itself is a valid solution, returns it; and (2) recursively enumerates all sub-trees of *c*.

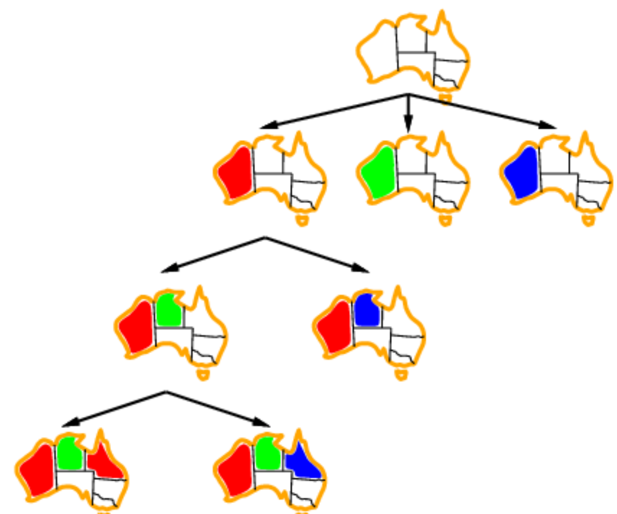
Search tree

- Variable assignments in the order: $WA, NT, Q, \ldots$









General backtracking search

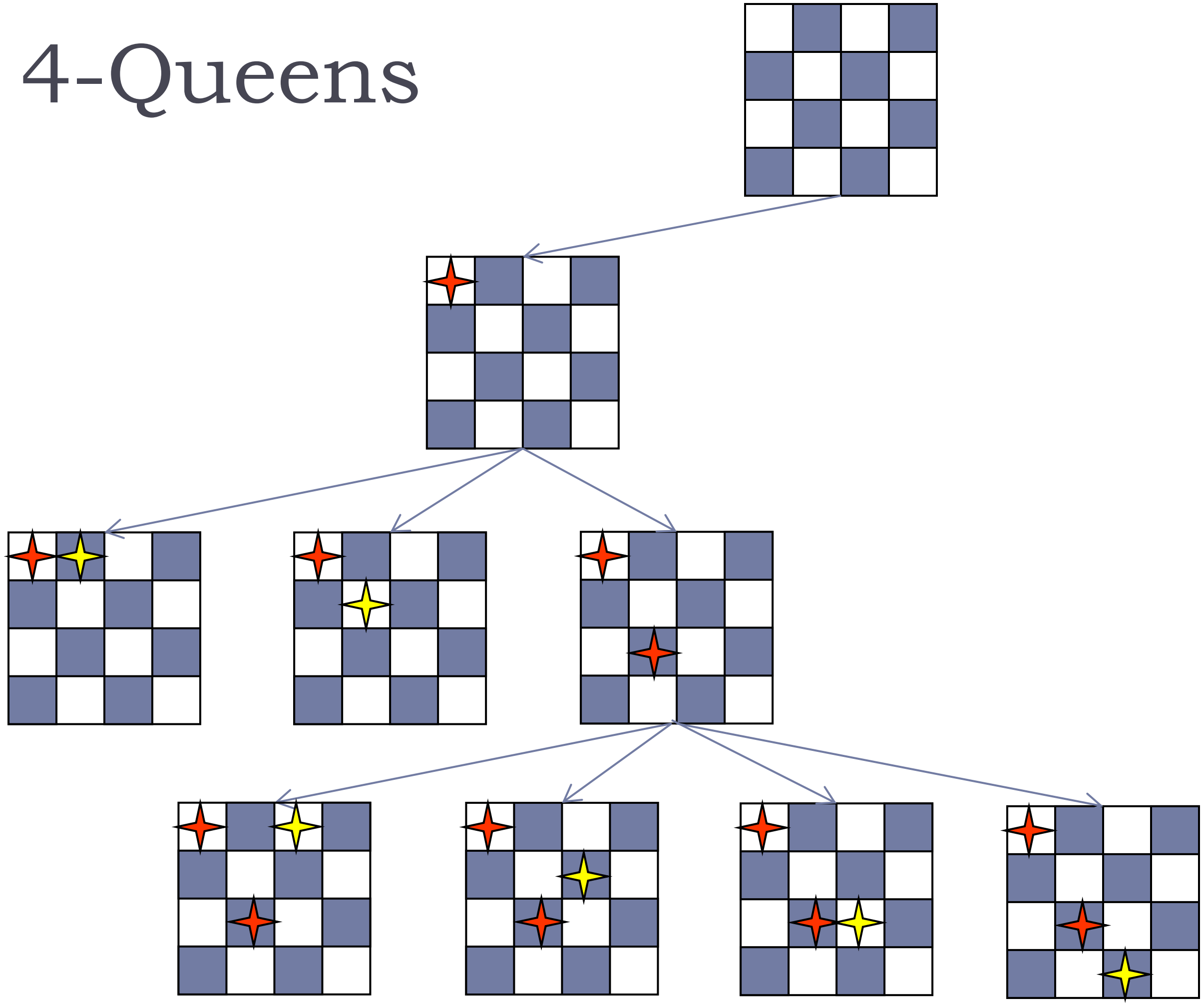
function $BACKTRACK(v)$ **returns** a solution, or failure
    if there is a solution at v **then return** solution
    for each child u of v **do**
        if $Promising(u)$ **then**
            $result \leftarrow BACKTRACK(u)$
            if $result \neq$ failure **return** $result$
    return failure

⇓

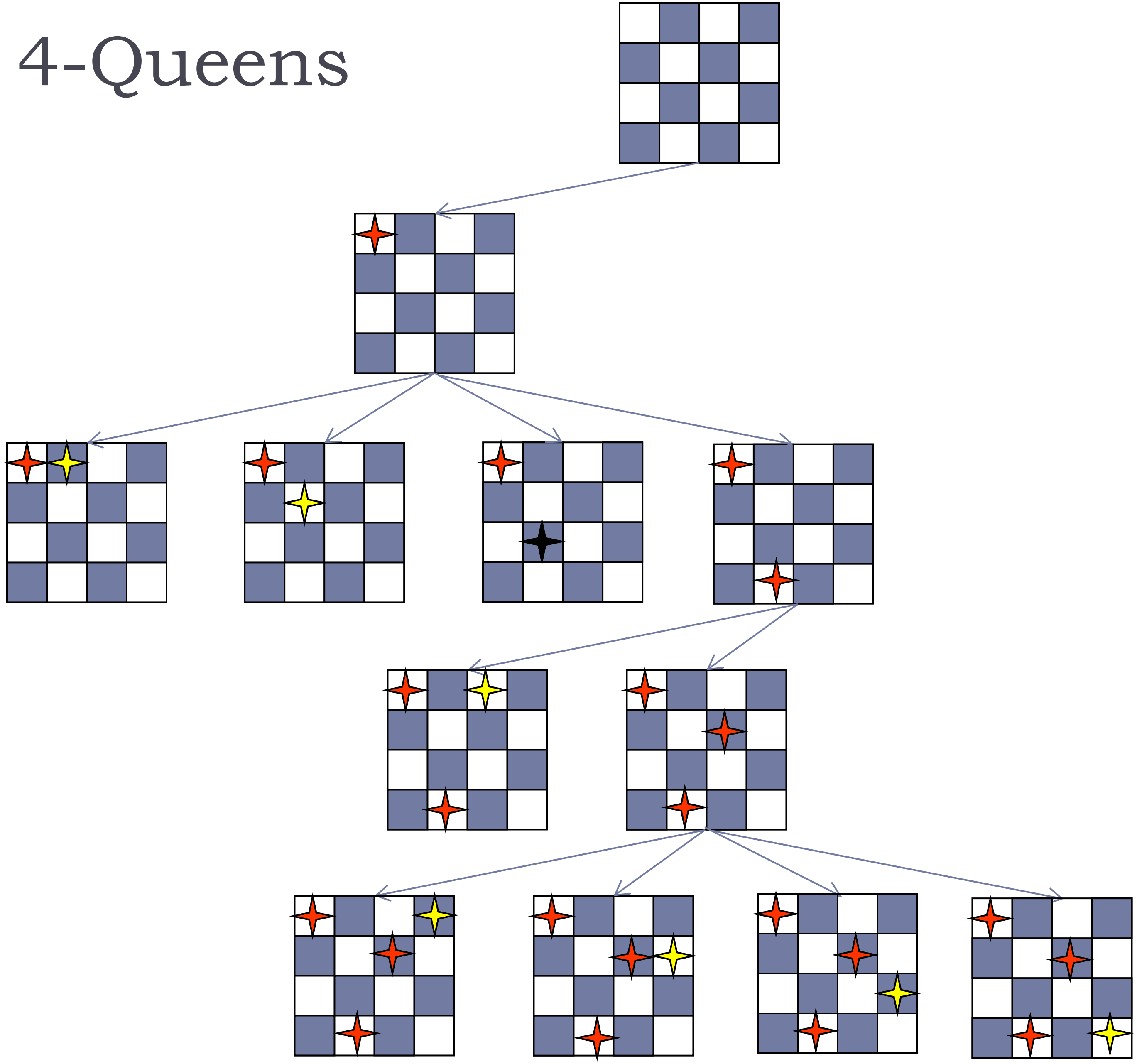
function $BACKTRACK(assignment, csp)$ **returns** an assignment, or failure
    If $assignment$ is complete **then return** $assignment$
    $var \leftarrow$ select an unassigned variable
    for each val **in** $Domain(var)$ **do**
        if $Consistent(assignment \cup \{var \leftarrow value\}, csp)$ **then**
            $result \leftarrow BACKTRACK(assignment \cup \{var \leftarrow value\}, csp)$
            if $result \neq$ failure **return** $result$
    return failure

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

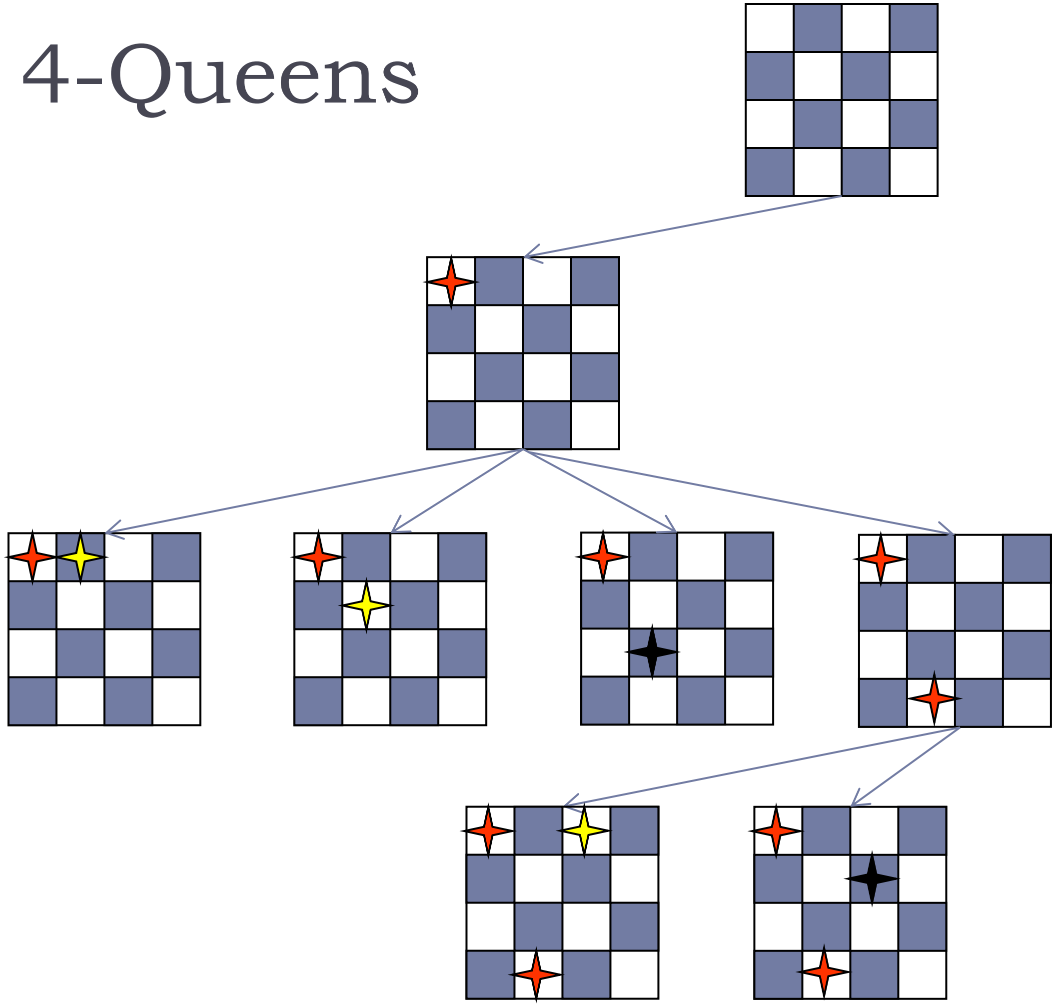
4-Queens



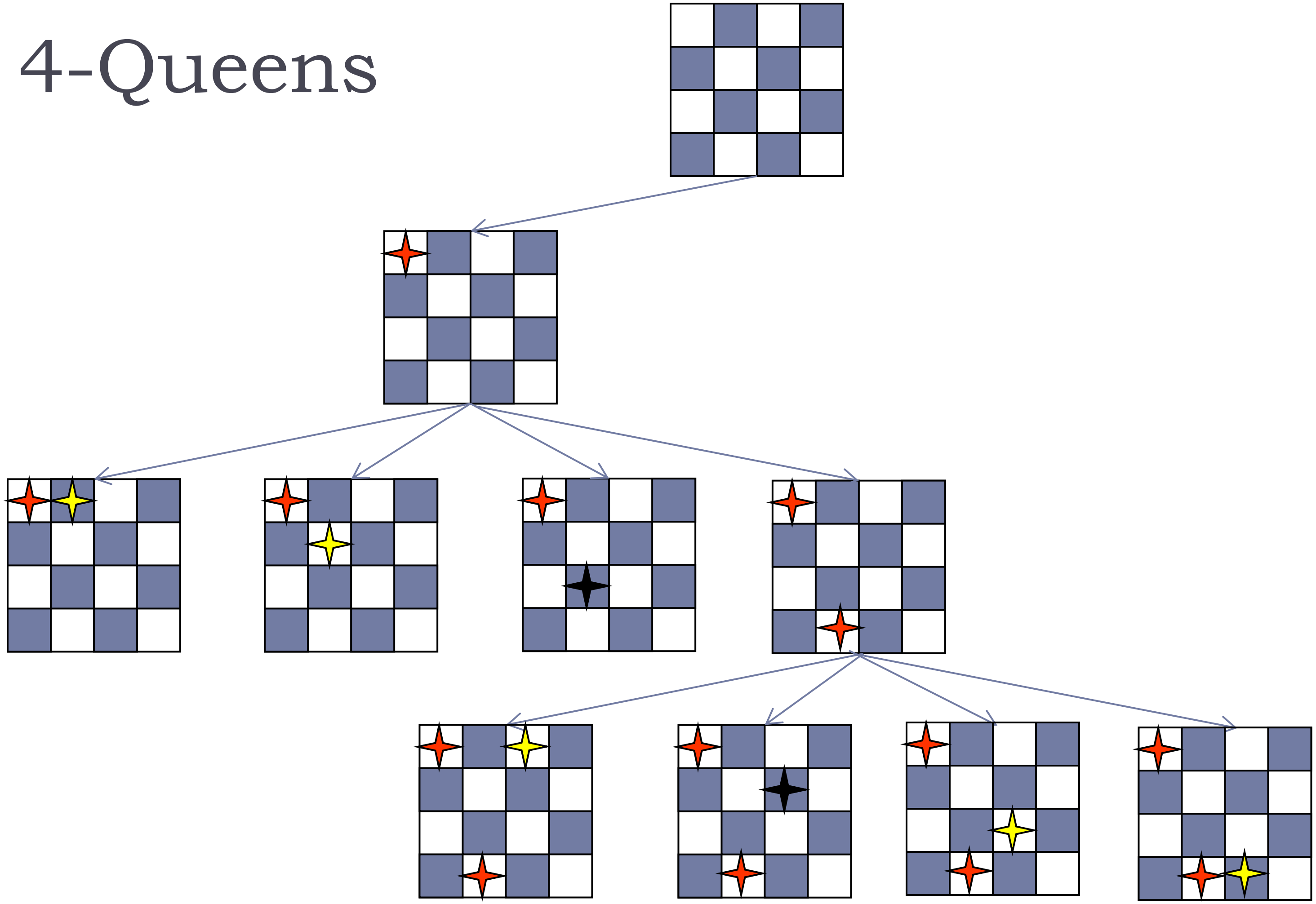
4-Queens



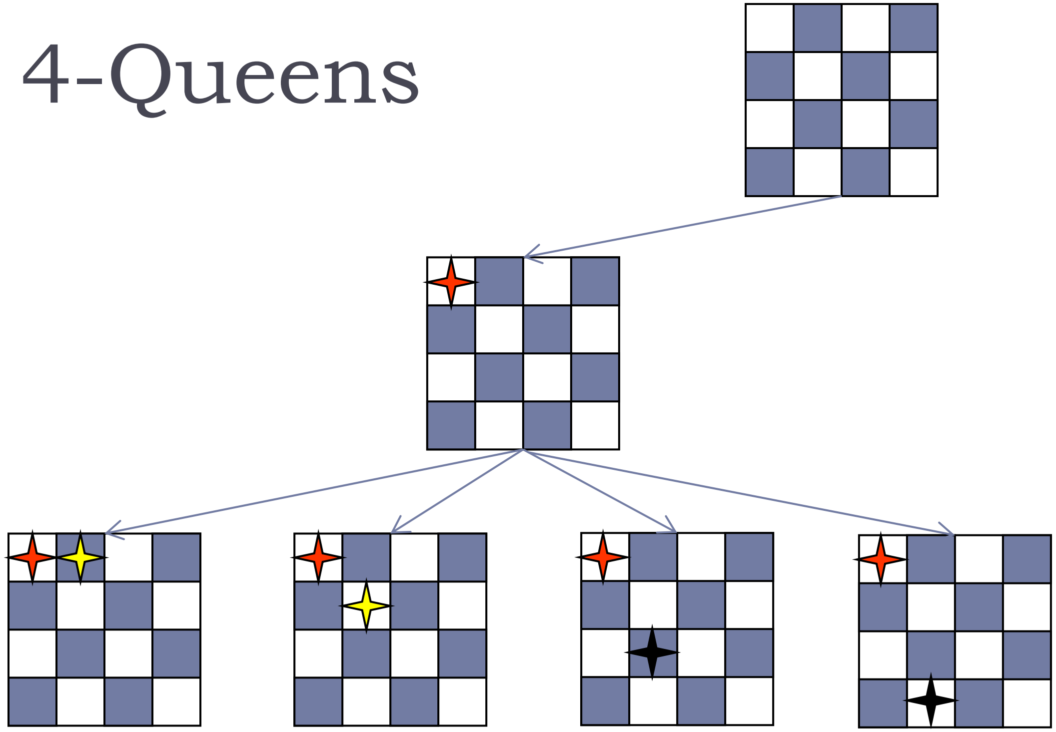
4-Queens



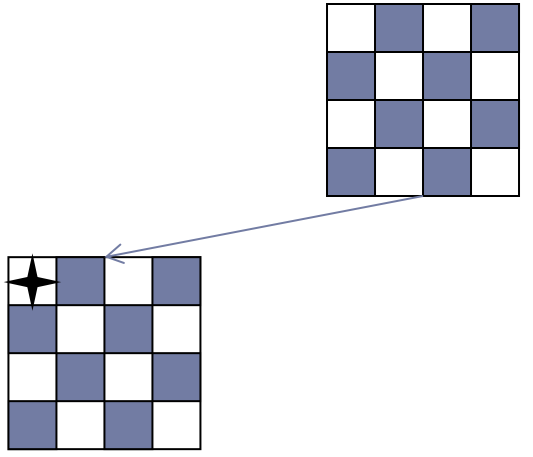
4-Queens



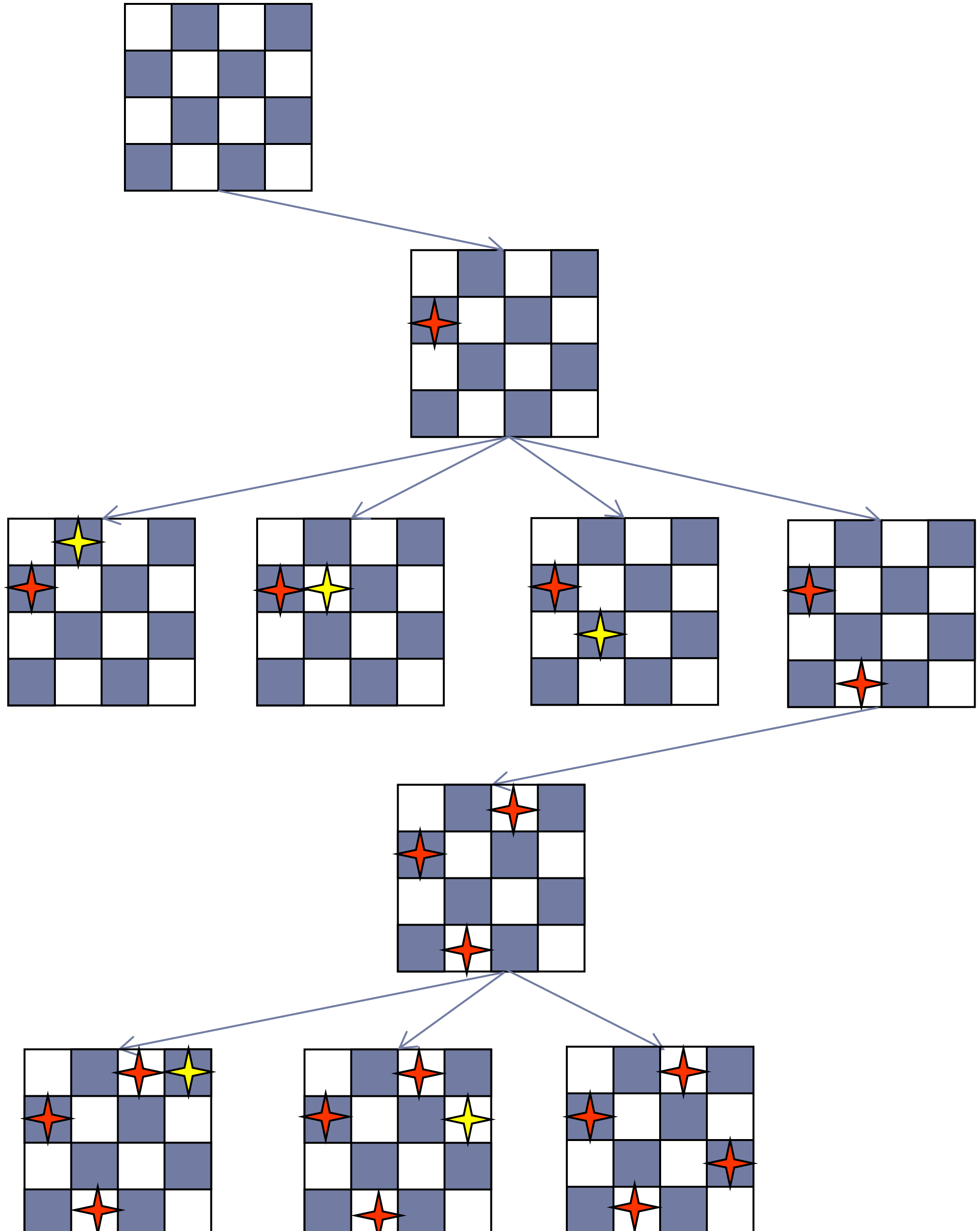
4-Queens



4-Queens



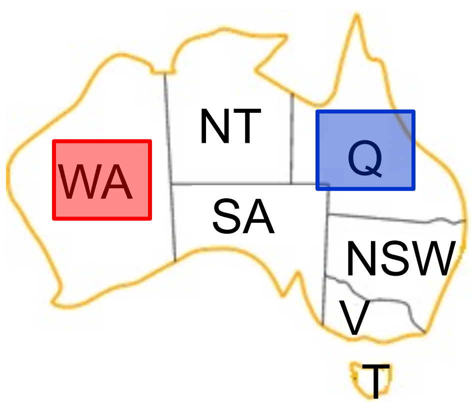
4-Queens

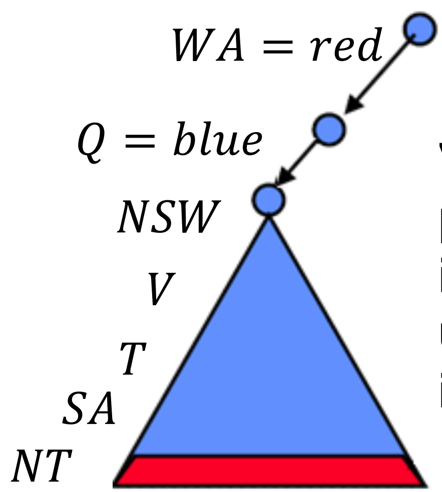


Solution

Naïve backtracking (late failure)

- Naïve backtracking is not generally efficient for solving CSPs.
- Map coloring with three colors
  - $\{WA = red, Q = blue\}$ can not be completed.
  - However, the backtracking search does not detect this before selecting but NT and SA variables

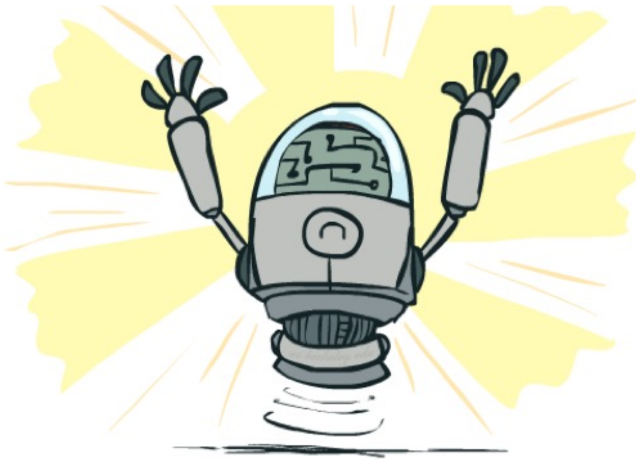


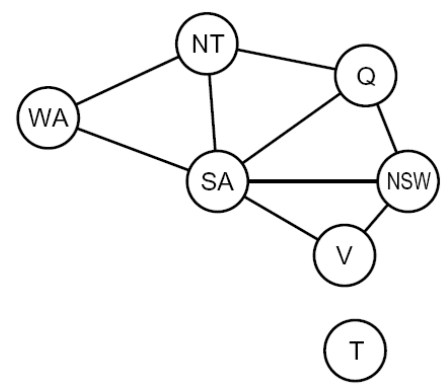


Variable NT has no possible value. But it is not detected until tying to assign it a value.

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?



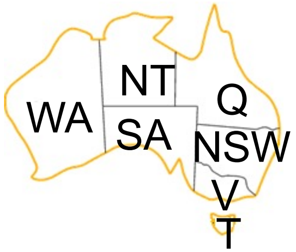


Filtering

- Filtering: Keep track of domains for unassigned variables and cross off bad options
  - Filtering by inference (looking ahead) in solving CSPs

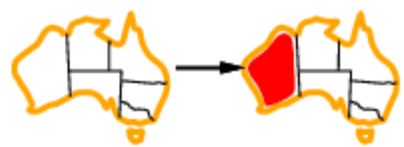
Forward Checking (FC)

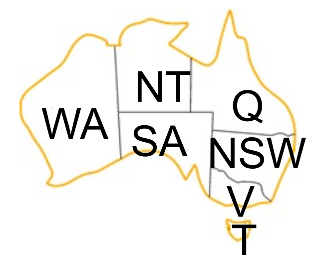
- When selecting a value for a variable, infer <u>new domain reductions on neighboring unassigned variables</u>.
  - Terminate search when a variable has no legal value

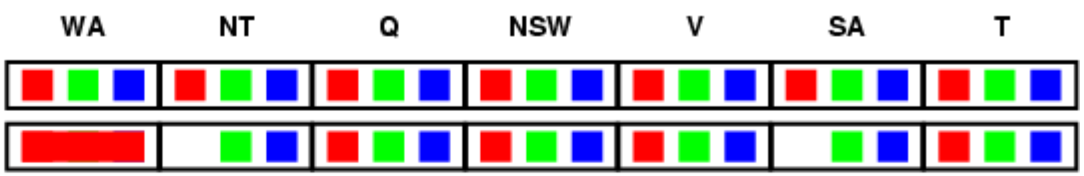


Forward Checking (FC)

- When selecting a value for a variable, infer new domain reductions on neighboring unassigned variables.
  - Terminate search when a variable has no legal value



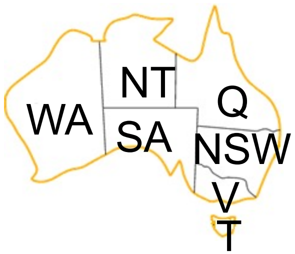


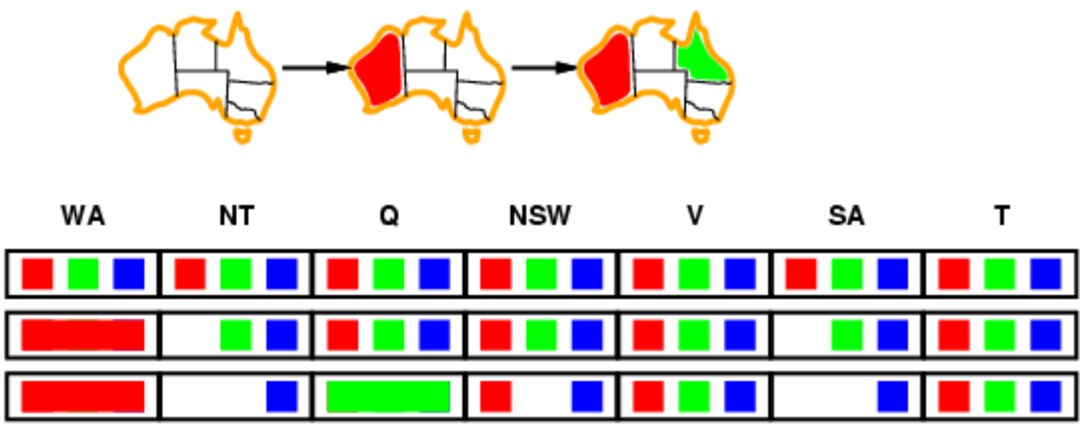


$WA = red$

Forward Checking (FC)

- When selecting a value for a variable, infer new domain reductions on neighboring unassigned variables.
  - Terminate search when a variable has no legal value



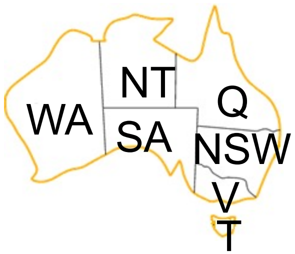


$WA = red$

$Q = green$

Forward Checking (FC)

- When selecting a value for a variable, infer new domain reductions on neighboring unassigned variables.
  - Terminate search when a variable has no legal value



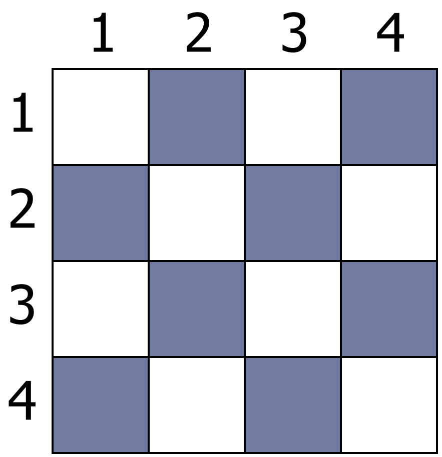
$WA = red$

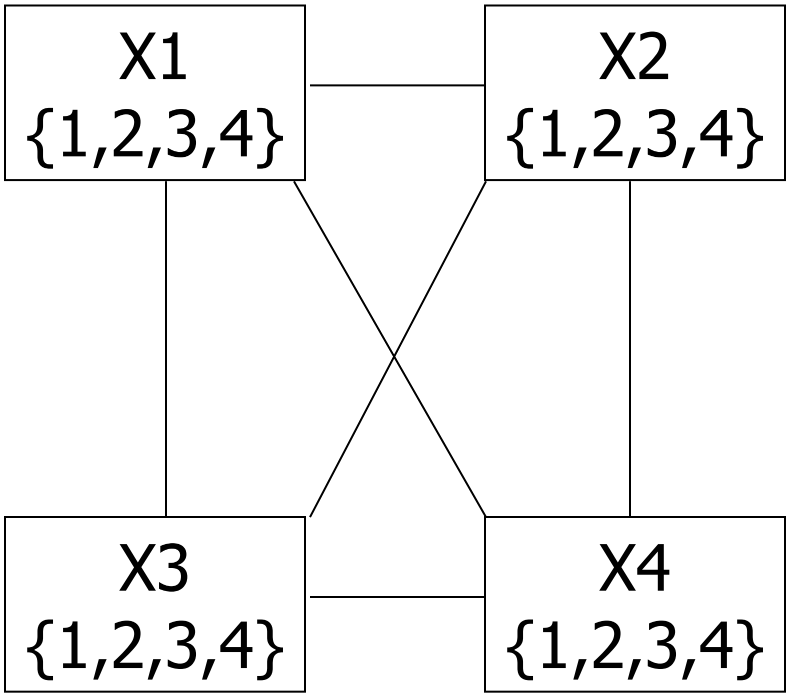
$Q = green$

$V = blue$

$\Longrightarrow \{WA = red, Q = green, V = blue\}$ is an inconsistent partial assignment

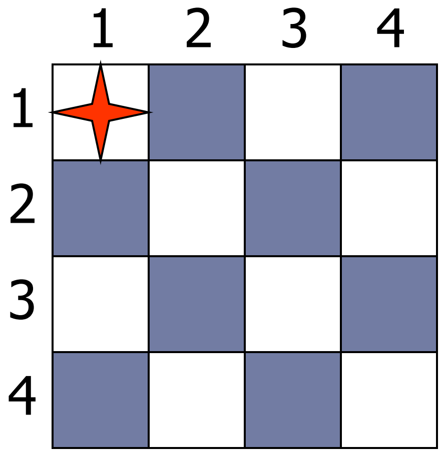
Example: 4-Queens

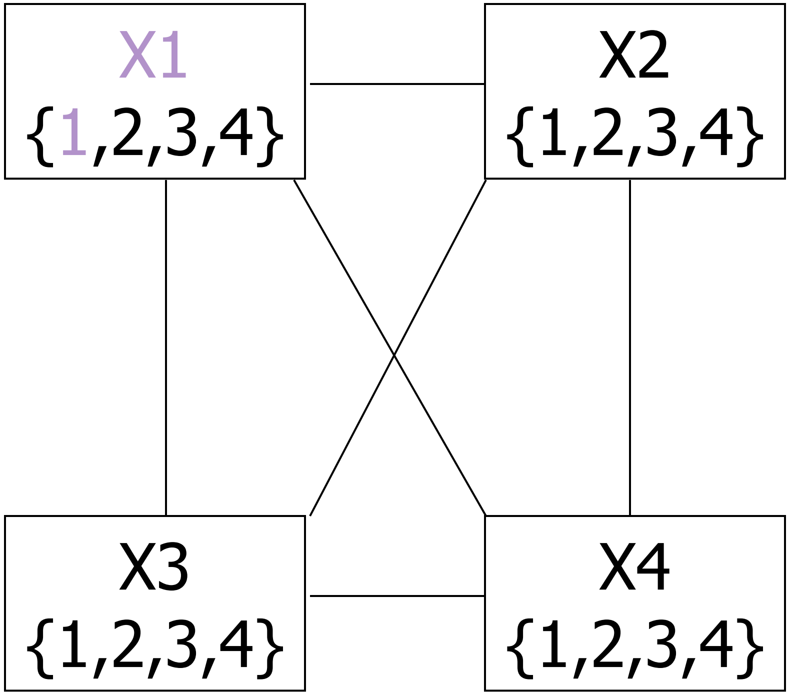




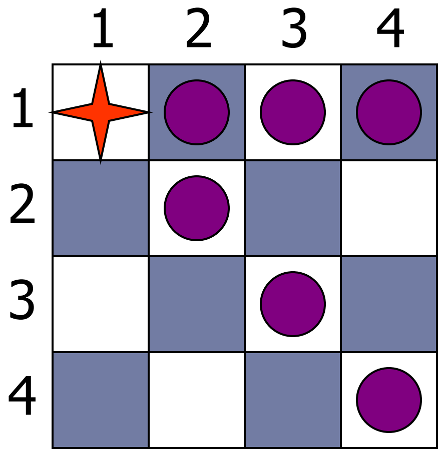
4-Qeens slides have been adopted from Dorr's slides on CMSC-421 course

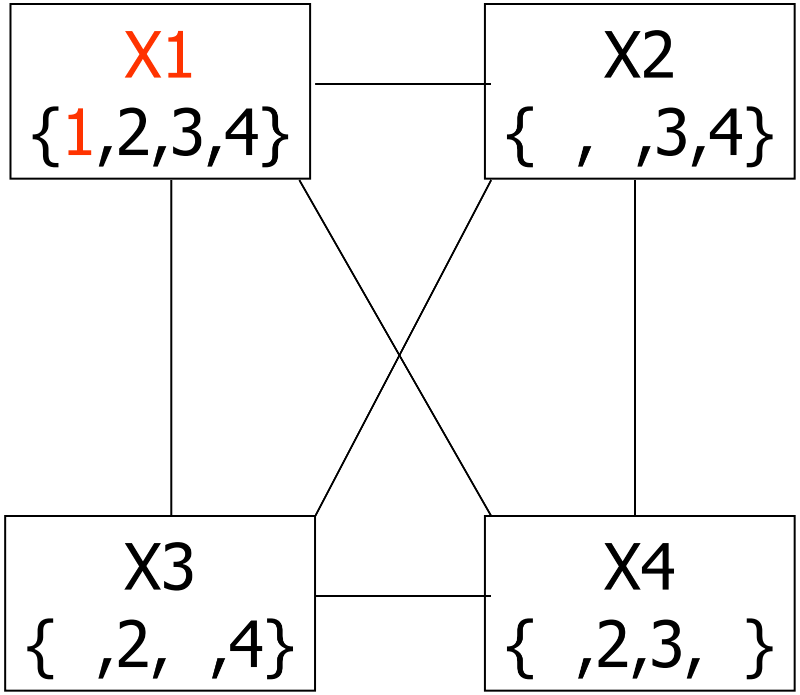
Example: 4-Queens



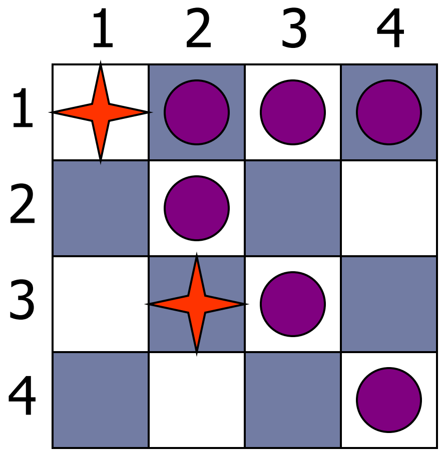


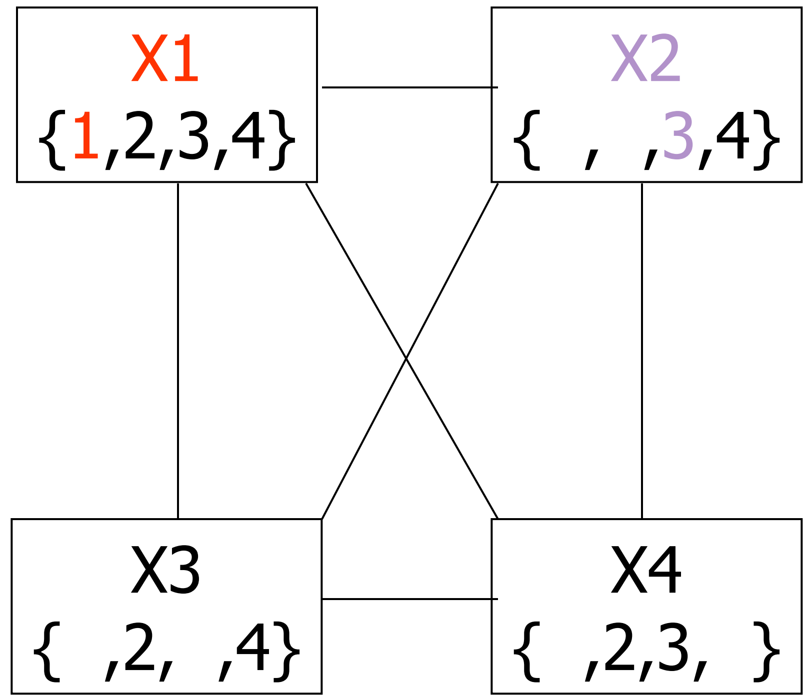
Example: 4-Queens



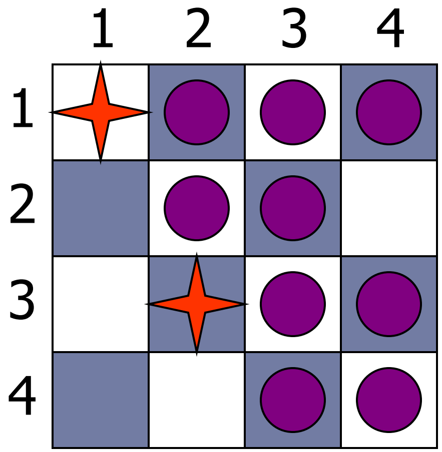


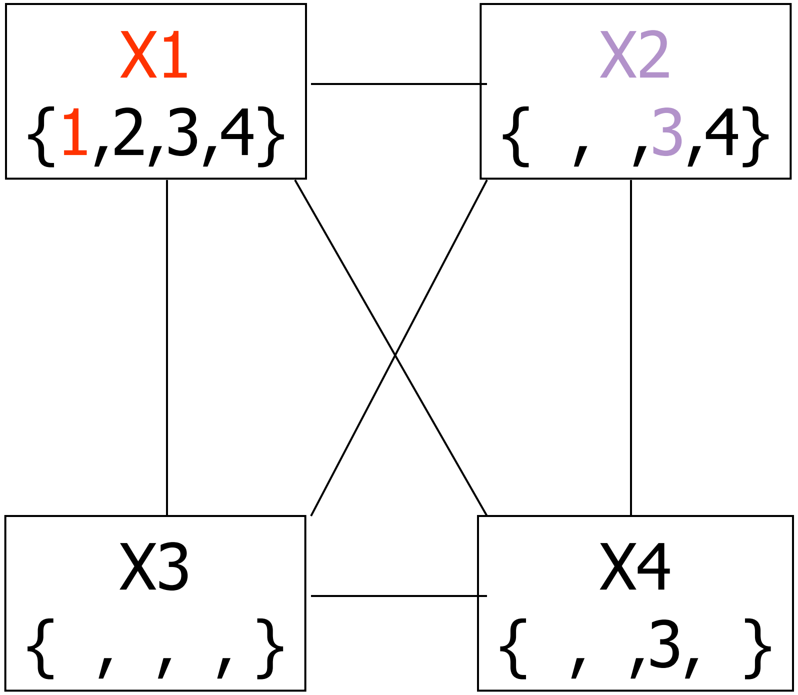
Example: 4-Queens



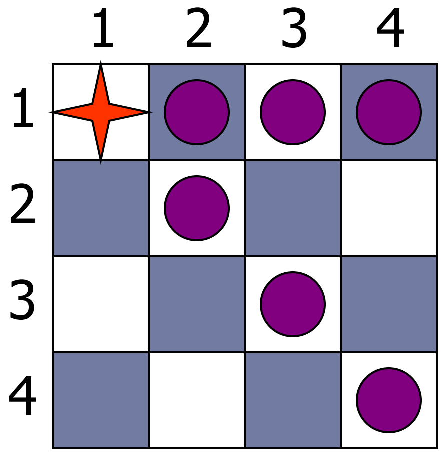


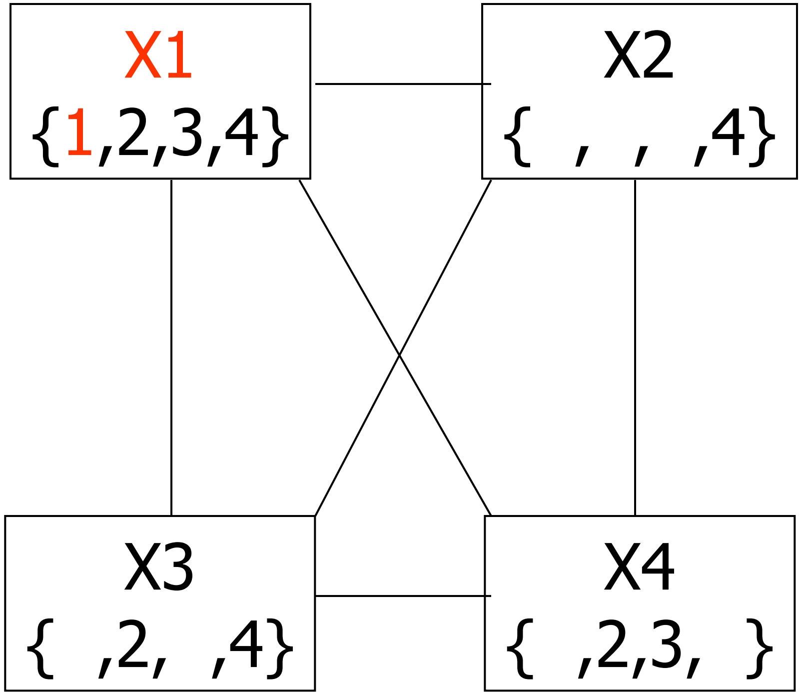
Example: 4-Queens



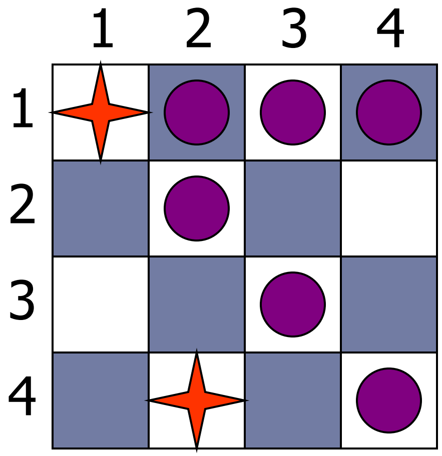


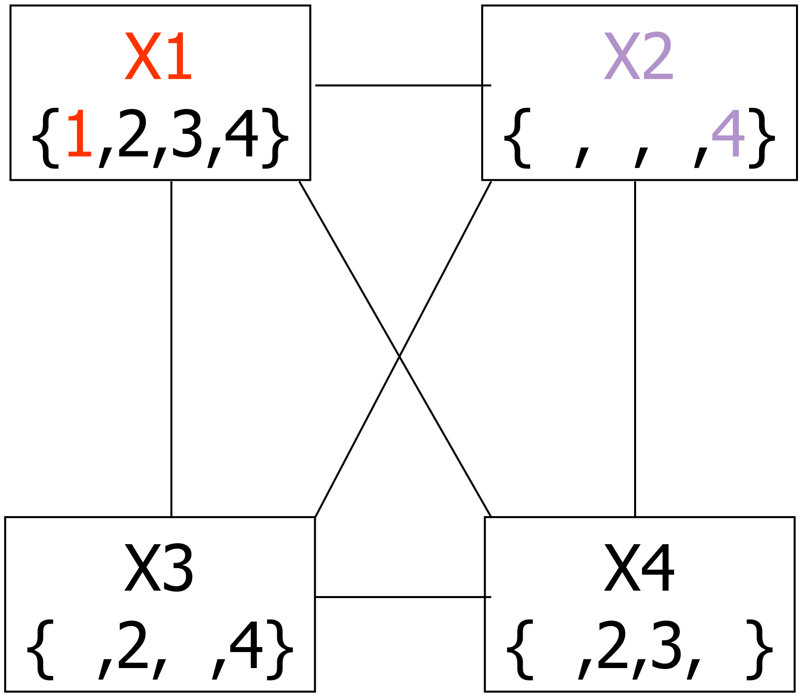
Example: 4-Queens



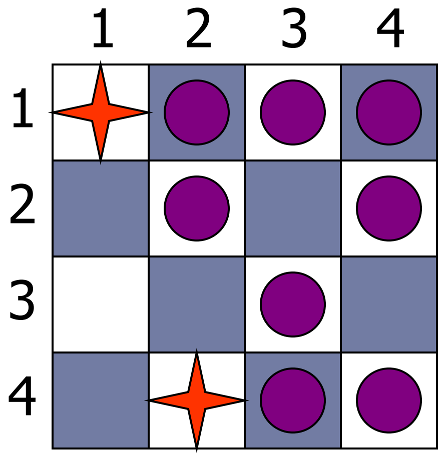


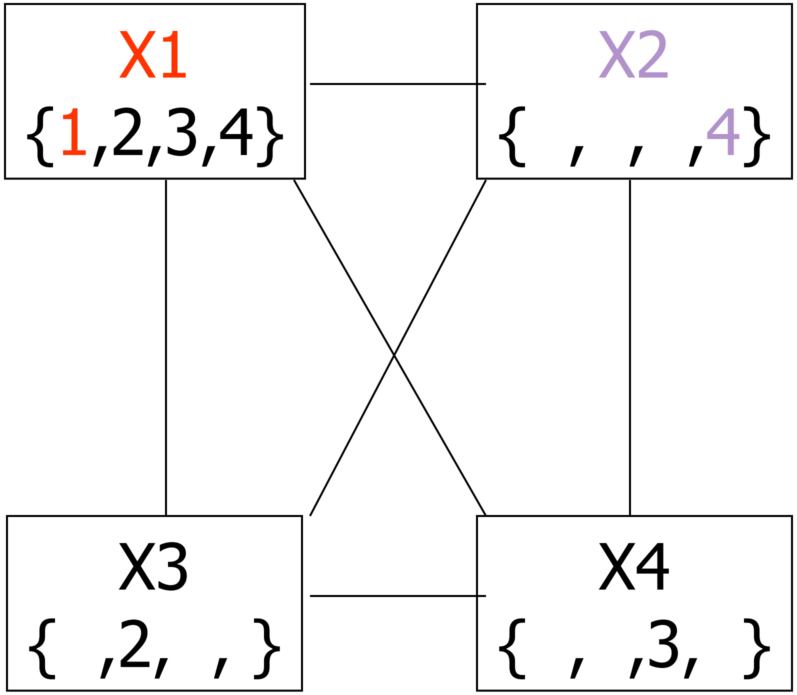
Example: 4-Queens



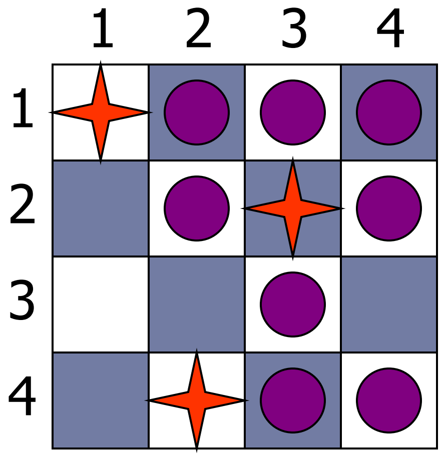


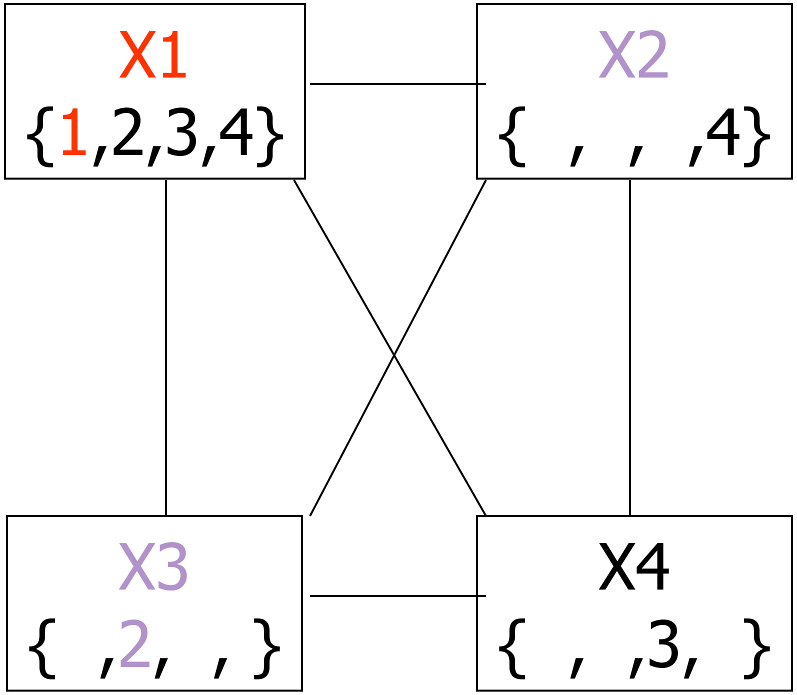
Example: 4-Queens



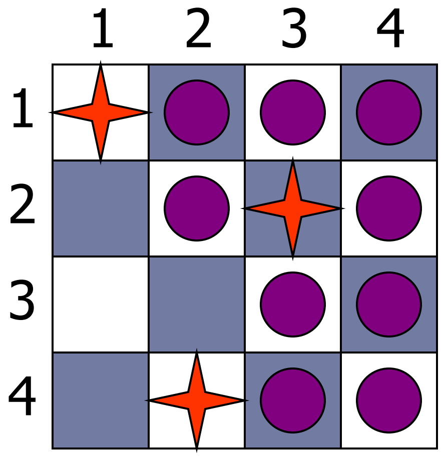


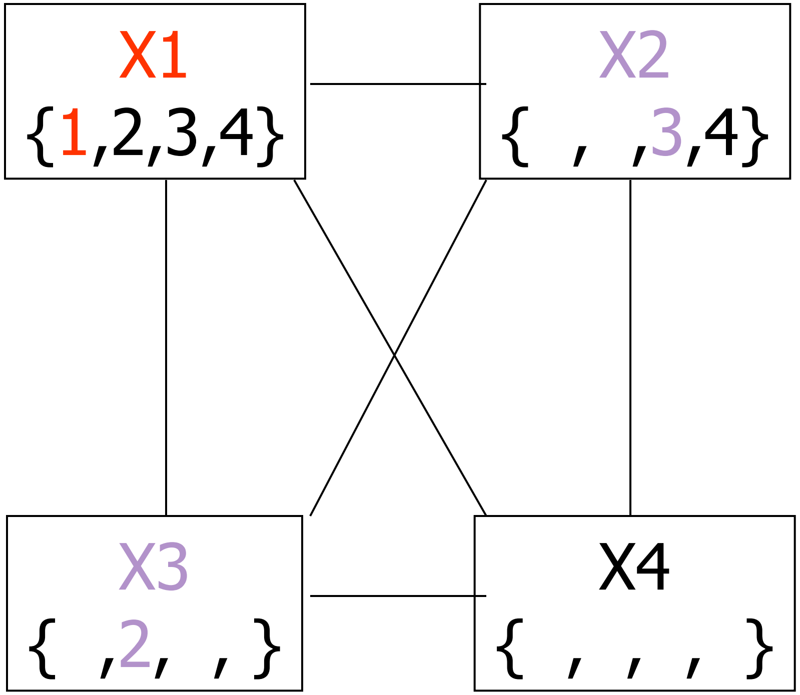
Example: 4-Queens





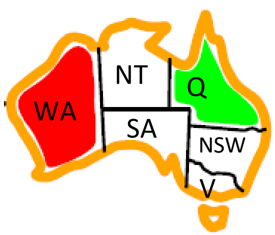
Example: 4-Queens

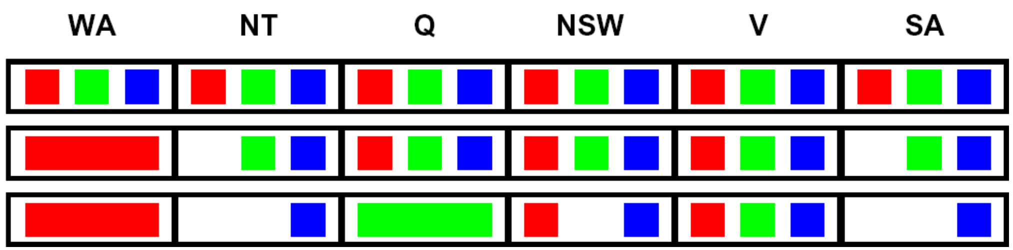




Filtering: shortcoming

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

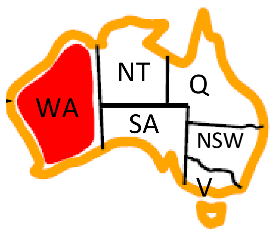


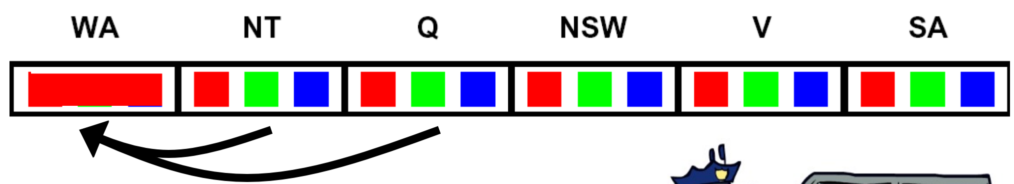


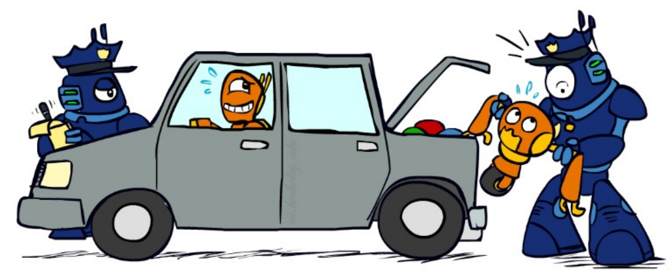
- NT and SA cannot both be blue!
- Why didn’t we detect this yet?
- *Constraint propagation:* reason from constraint to constraint

Consistency of a single arc

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







Delete from the tail!

- NT -> WA
  - If NT = blue: we could assign WA = red
  - If NT = green: we could assign WA = red
  - If NT = red: there is no remaining assignment to WA that we can use
  - Deleting NT = red from the tail makes this arc consistent

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

Arc consistency

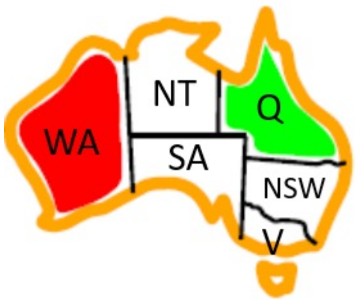
- X_i is arc-consistent with respect to X_j
  if for every value in D_i there is a consistent value in D_j

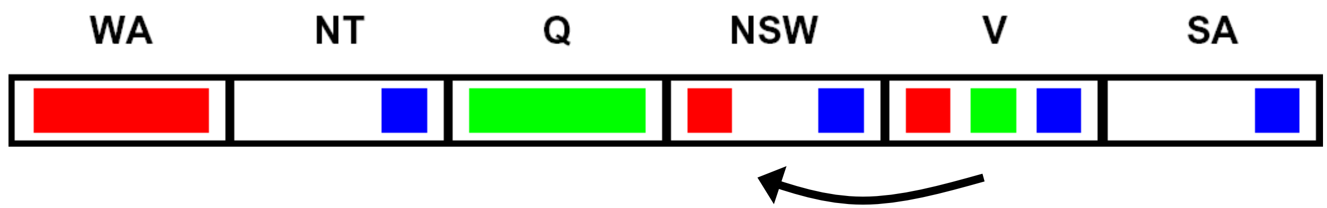
- Example
  - Variables: $X = \{X_1, X_2\}$
  - Domain: $\{0,1,2, \ldots, 9\}$
  - Constraint: $X_1 = {X_2}^2$

  - Is X_1 is arc-consistent w.r.t. X_2?
    - No, to be arc-consistent $Domain(X_1) = \{0,1,4,9\}$
  - Is X_2 is arc-consistent w.r.t. X_1?
    - No, to be arc-consistent $Domain(X_2) = \{0,1,2,3\}$

Arc consistency of an entire CSP (1/6)

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

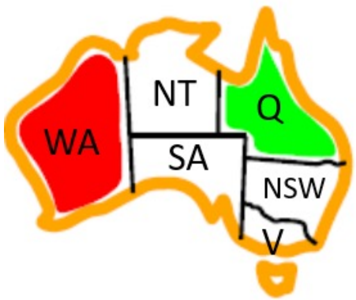


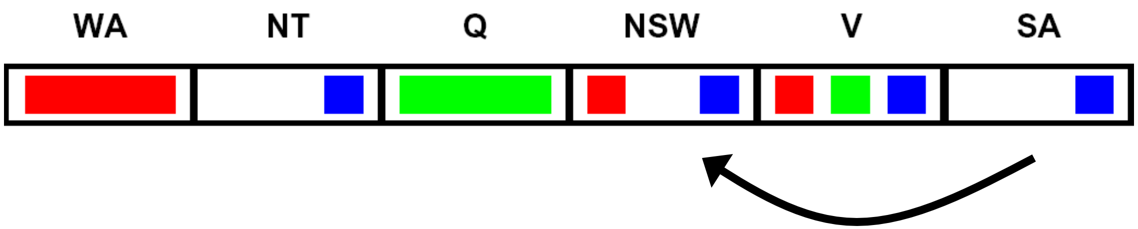


- Arc V to NSW is consistent: for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint

Arc consistency of an entire CSP (2/6)

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

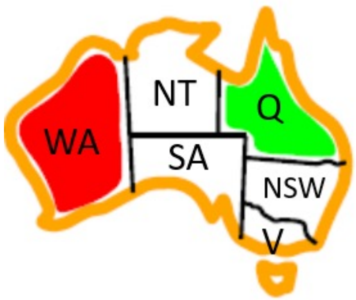


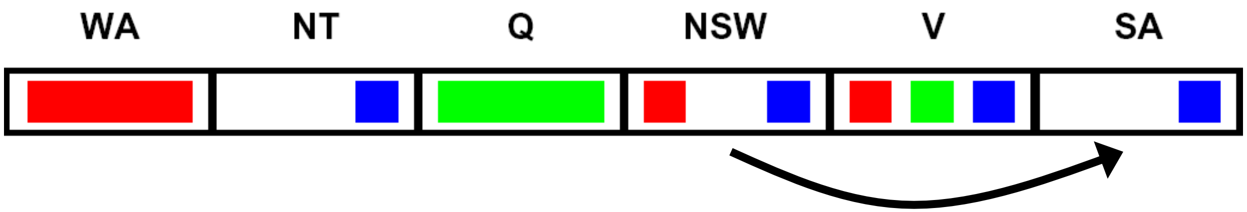


- Arc SA to NSW is consistent: for every x in the tail there is some y in the head which could be assigned without violating a constraint

Arc consistency of an entire CSP (3/6)

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

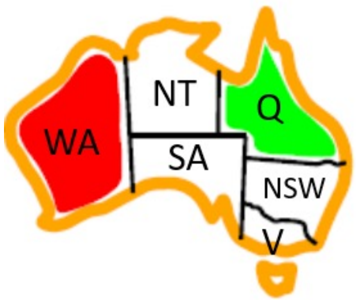


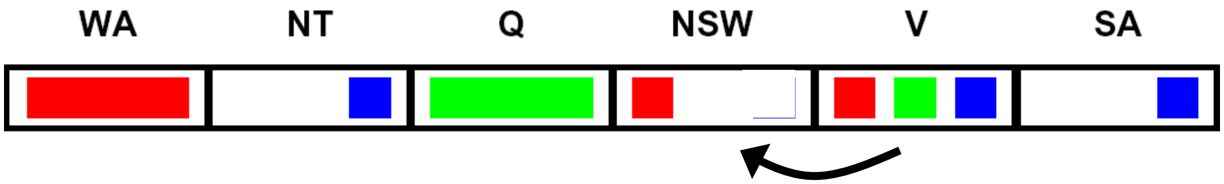


- Arc NSW to SA is not consistent: if we assign NSW = blue, there is no valid assignment left for SA
- To make this arc consistent, we delete NSW = blue from the tail

Arc consistency of an entire CSP (4/6)

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

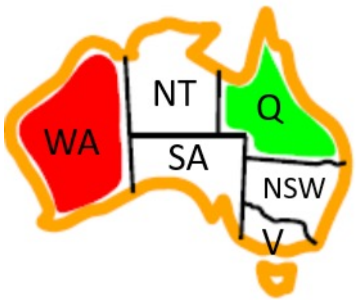


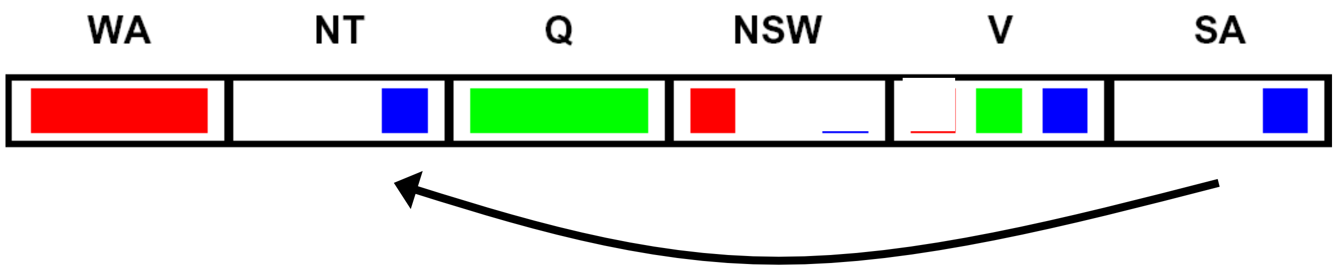


- Remember that arc V to NSW was consistent, when NSW had red and blue in its domain
- After removing blue from NSW, this arc might not be consistent anymore! We need to recheck this arc.
- Important: If X loses a value, neighbors of X need to be rechecked!

Arc consistency of an entire CSP (5/6)

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

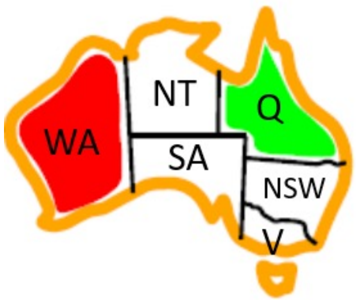


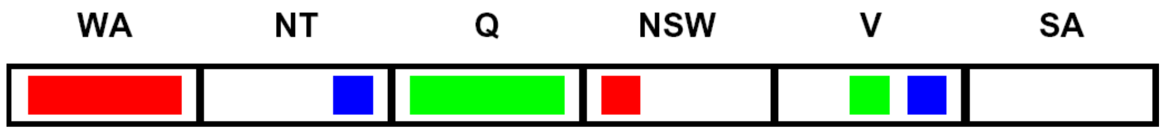


- Arc SA to NT is inconsistent. We make it consistent by deleting from the tail (SA = blue).

Arc consistency of an entire CSP (6/6)

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





- SA has an empty domain, so we detect failure. There is no way to solve this CSP with WA = red and Q = green, so we backtrack.
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment

Arc consistency algorithm: AC-3

For each arc (X_i, X_j) in the queue

Remove it from queue

Makes X_i arc-consistent with respect to X_j

1) If D_i remains unchanged then continue
2) If $|D_i| = 0$ then return false
3) For each neighbor X_k of X_i except to X_j do

    add (X_k, X_i) to queue

If domain of X_i loses a value, neighbors of X_i must be rechecked

- Removing a value from a domain may cause further inconsistency, so we have to repeat the procedure until everything is consistent.
- When queue is empty, resulted CSP is equivalent to the original CSP.
  - Same solution (usually reduced domains speed up the search)

Arc consistency algorithm: AC-3

function $AC\_3(csp)$ **returns** false if an inconsistency is found and true otherwise
    inputs: csp, a binary CSP with components X, D, C
    local variables: $queue$, a queue of arcs, initially all the arcs in csp

while $queue$ is not empty **do**
    $(X_i, X_j) \leftarrow REMOVE\_FIRST(queue)$
    if $REVISE(csp, X_i, X_j)$ **then**
        If size of $D_i = 0$ **then return** $false$
        for each X_k **in** $X_i.NEIGHBORS - \{X_j\}$ **do**
            add (X_k, X_i) to $queue$

function $REVISE(csp, X_i, X_j)$ **returns** true iff we revise the domain of X_i
    $revised \leftarrow false$
    for each x **in** D_i **do**
        if no value y in D_j allows (x, y) to satisfy the constraint between X_i and X_j **then**
            delete x from D_i
            $revised \leftarrow true$
return $revised$

Makes X_i arc-consistent with respect to X_j

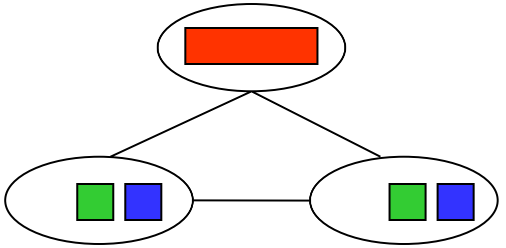
AC-3: time complexity

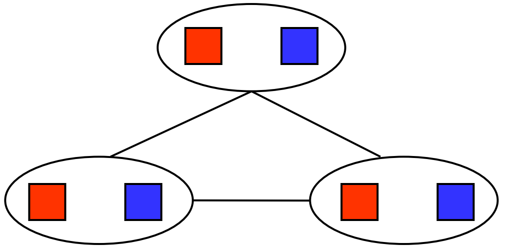
- Time complexity (n variables, c binary constraints, d domain size): $O(cd^3)$
  - Each arc (X_k, X_i) is inserted in the queue at most d times.
    - At most all values in domain X_i can be deleted.
  - Checking consistency of an arc: $O(d^2)$

- Detecting all possible future problems is NP-hard – why?

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)

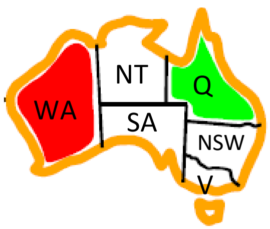




What went wrong here?

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?

Inference during the search process

- It can be more powerful than inference in the preprocessing stage.

- Interleaving search and inference

Arc consistency: map coloring example

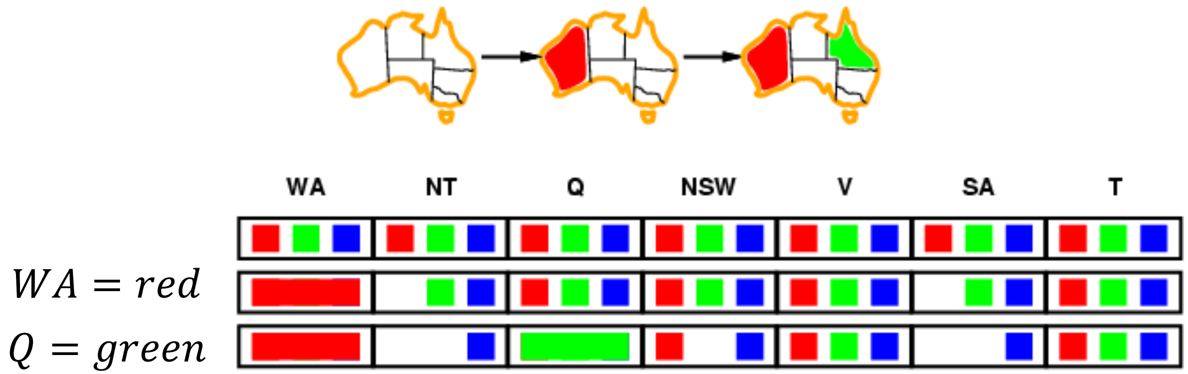
- For general map coloring problem all pairs of variables are arc-consistent if $|D_i| \geq 2(i = 1, \ldots, n)$

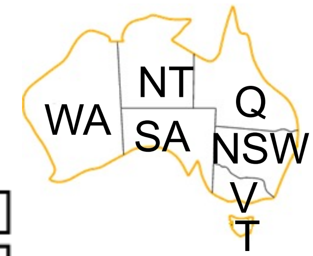
- In this case, arc consistency as preprocessing can do nothing.
  - Fails to make enough inference

- We may 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.
  - Both of the possible assignments to set $\{WA, SA\}$ are inconsistent with NT.

Constraint propagation

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



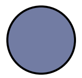


- NT and SA cannot both be blue!
  - FC does not look far enough ahead to find this inconsistency

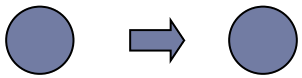
- Maintaining Arc Consistency (MAC) - Constraint propagation
  - Forward checking + recursively propagating constraints when changing domains
  - similar to AC-3 but only arcs related to the current variable are put in the queue at start

Local consistency

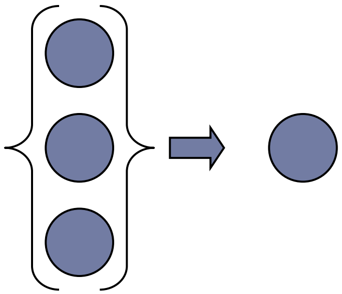
- Node consistency (1-consistency)
  - Each single node’s domain has a value which meets that node’s unary constraints



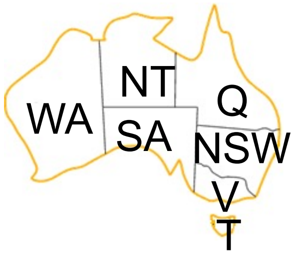
- Arc consistency (2-consistency)
  - For each pair of nodes, any consistent assignment to one can be extended to the other



- k-consistency
  - For each k nodes, any consistent assignment to (k-1) nodes can be extended to the kth node.



k-consistency



- Arc consistency does not detect all inconsistencies
- A CSP is k-consistent if for any set of $k-1$ variables and for any consistent assignment to those variables, a consistent value can always be assigned to any kth variable.
  - E.g. 1-consistency = node-consistency
  - E.g. 2-consistency = arc-consistency
  - E.g. 3-consistency = path-consistency
- Higher k more expensive to compute

Which level of consistency?

- **Trade off** between the required time to establish k-consistency and amount of the eliminated search space.
  - If establishing consistency is slow, this can slow the search down to the point where no propagation is better.

- Establishing k-consistency need exponential time and space in k (in the worst case)

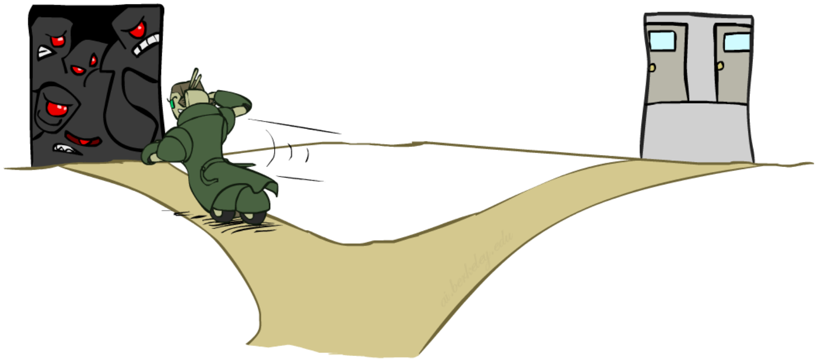
- Commonly computing 2-consistency and less commonly 3-consistency

Ordering

Ordering: Minimum Remaining Values (MRV)

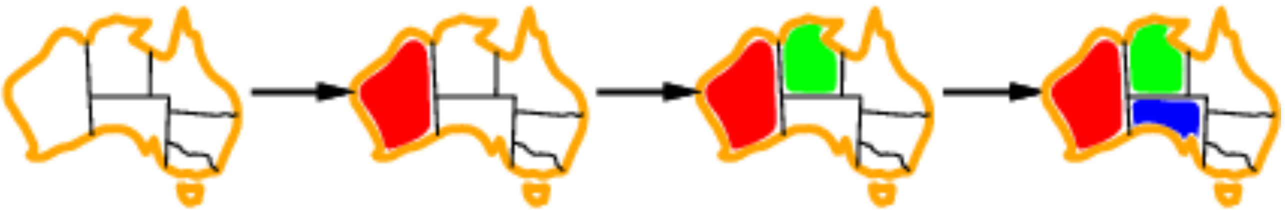
- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal left values in its domain

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



Ordering: Minimum Remaining Values (MRV)

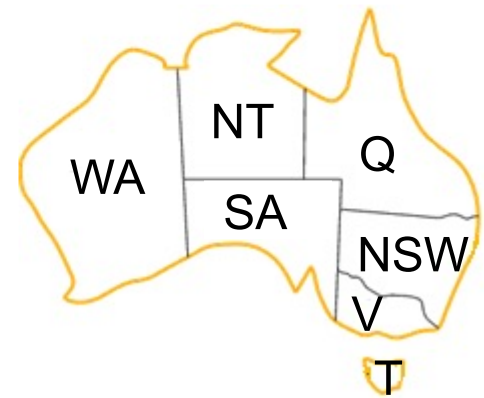
- Chooses the variable with the fewest legal values
  - Fail first
- Also known as Most Constrained Variable (MCS)
- Most likely to cause a failure soon and so pruning the search tree

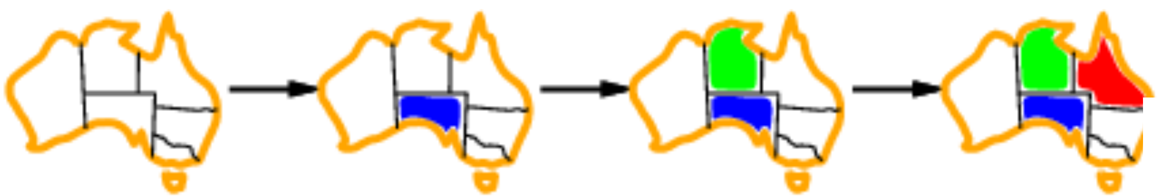


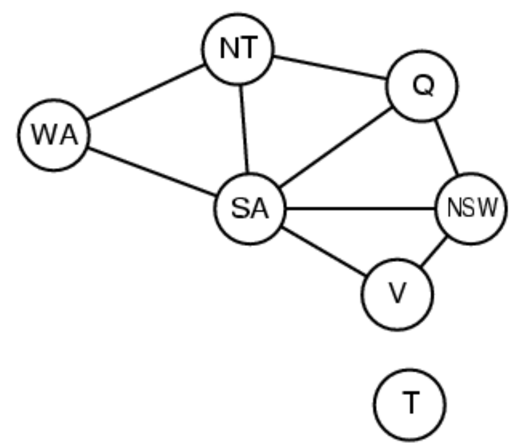
Degree heuristic

- Tie-breaker among MRV variables

- Degree heuristic: choose the variable with the most constraints on remaining variables
  - To choose one who interferes the others most!
  - reduction in branching factor

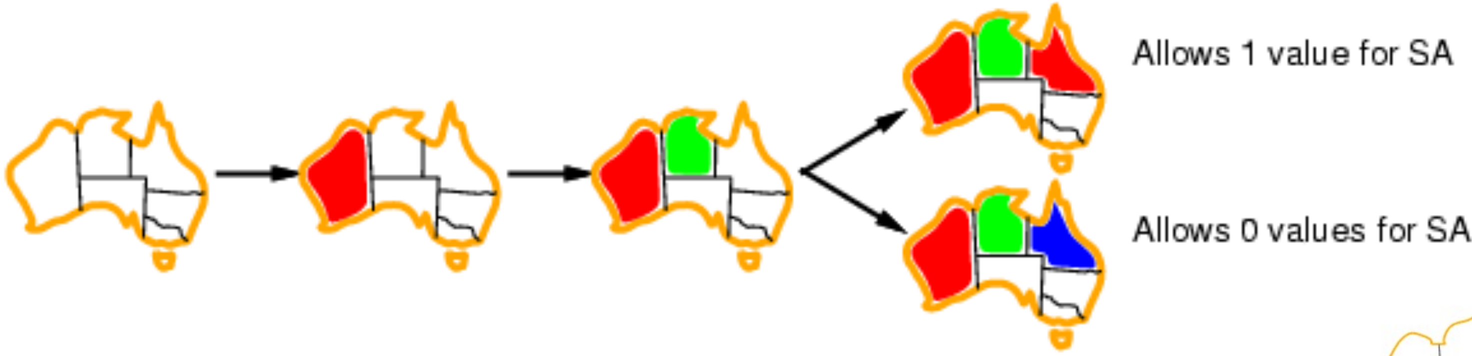


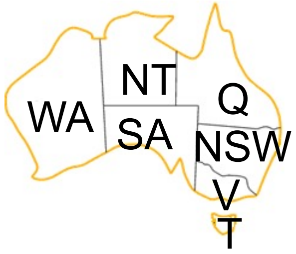




Ordering: Least Constraining Value (LCV)

- Given a variable, choose the least constraining value:
  - one that rules out the fewest values in the remaining variables
  - leaving maximum flexibility for subsequent variable assignments
    - Fail last (the most likely values first)

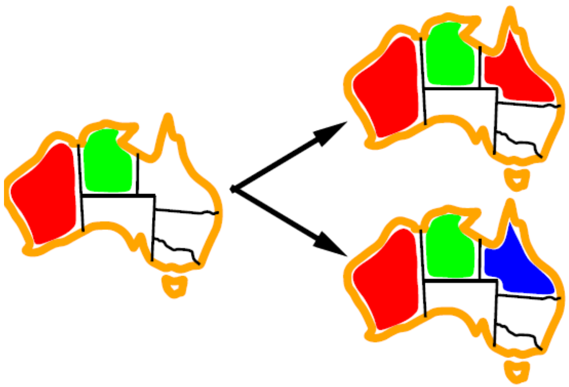


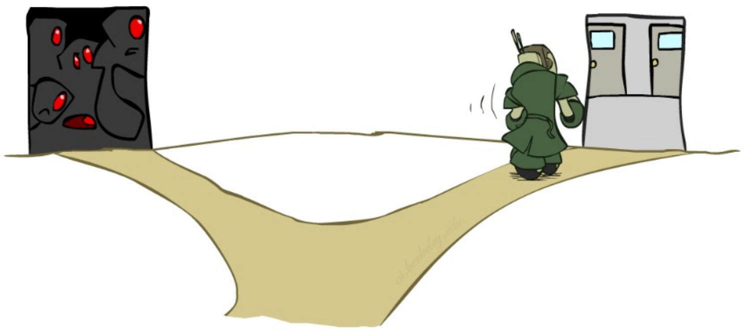


- Assumption: we only need one solution

Ordering: Least Constraining Value (LCV)

- Value Ordering: Least Constraining Value
  - 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





Solving CSP efficiently

- Which variable should be assigned next?
  - $SELECT\_UNASSIGNED\_VARIABLE$
- In what order should values of the selected variable be tried?
  - $ORDER\_DOMAIN\_VALUES$
- What inferences should be performed at each step in the search?
  - $INFERENCE$

CSP backtracking search

```
function BACKTRACKIN_SEARCH(csp) returns a solution, or failure
    return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
    if assignment is complete then return assignment
    var ← SELECT_UNASSIGNED_VARIABLE(csp, assignment)
    for each value in ORDER_DOMAIN_VALUES(var, assignment, csp) do
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences ← INFERENCE(csp, var, value)
            if inferences ≠ failure then
                add inferences to assignment
                result ← BACKTRACK(assignemnt, csp)
                if result ≠ failure then return result
            remove {var = value} and inferences from assignment
return failure
```

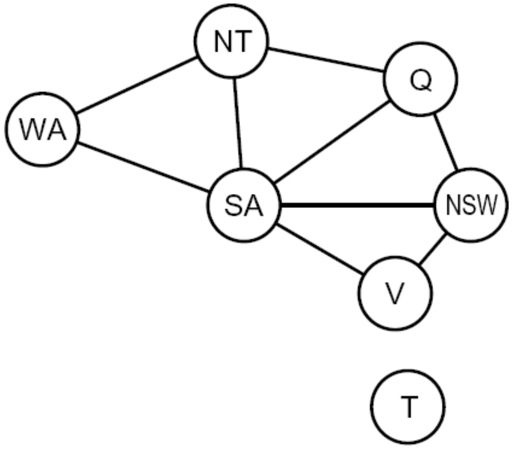
- Backtracking = DFS + variable-ordering + fail-on-violation
- What are the choice points?

CSPs solver phases: summary

- Combination of combinatorial search and heuristics to reach reasonable complexity:
  - Search
    - Select a new variable assignment from several possibilities of assigning values to unassigned variables
    - Base of the search process is a backtracking algorithm
  - Inference in CSPs (constraint propagation)
    - "looking ahead" in the search at unassigned variables to eliminate some possible part of the future search space.
      - Using the constraints to reduce legal values for variables
    - Key idea is local consistency

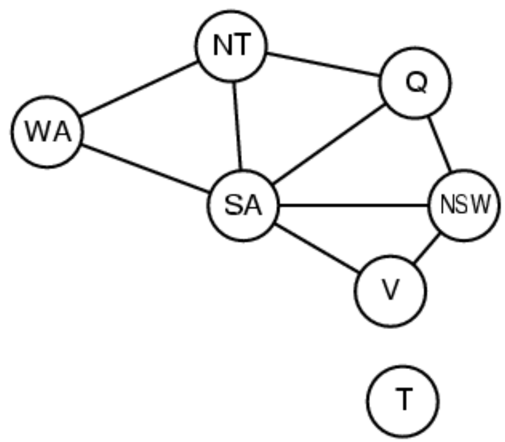
Constraint graph

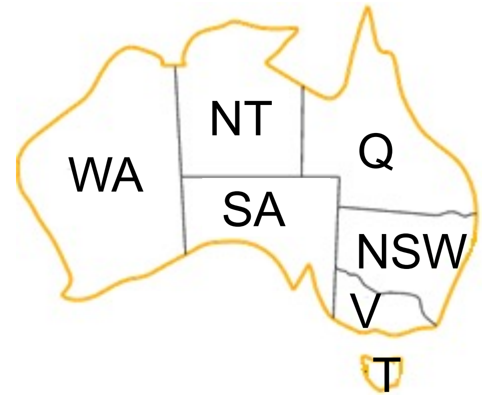
- Binary CSP: each constraint relates (at most) two variables
- Binary constraint graph: nodes are variables, arcs show constraints
- General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!



Constraint graph

- Nodes are variables, arcs are constraints



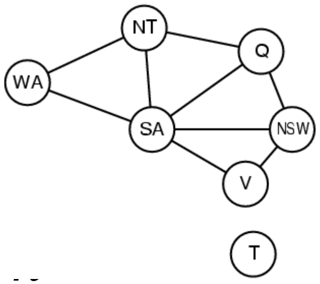


- Enforcing local consistency in each part of the graph can cause inconsistent values to be eliminated

Graph structure

- Extreme case: independent subproblems
  - Example: Tasmania and mainland do not interact

- Connected components as independent sub-problems
  - The color of T is independent of those of other region



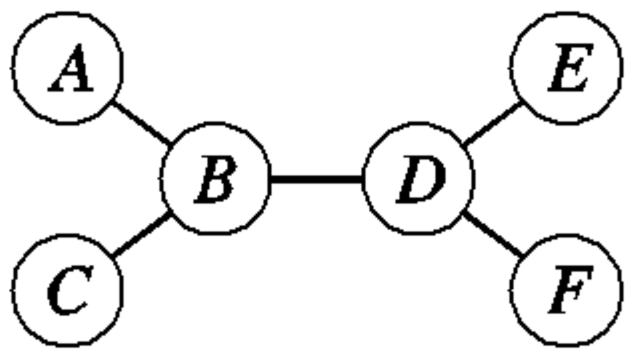
- Suppose each sub-problem has h variables out of n
  - Worst-case solution cost is $O((n/h)(d^h))$ that is linear in ..

- Example: $n = 80,\ d = 2,\ h = 20$ (processing: 10^6 nodes/sec)
  - 2^{80} = 4 billion years
  - $(4)(2^{20})$ = 0.4 seconds

Tree structured CSPs

- Any two variables are connected by only one path

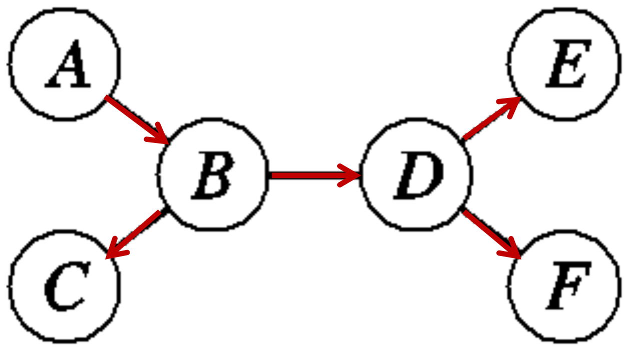
- Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\ d^2)$ time
  - Compare to general CSPs, where worst-case time is $O(d^n)$



- This property also applies to probabilistic reasoning (later): an example of the relation between syntactic restrictions and the complexity of reasoning

Tree structured CSPs: topological ordering

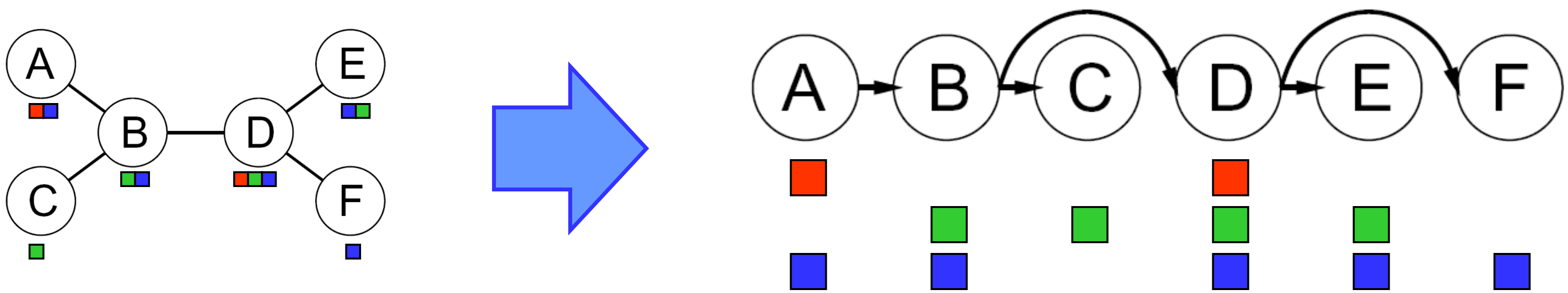
- Construct a rooted tree (picking any variable to be root, …)



- Order variables from root to leaves such that every node's parent precedes it in the ordering (<u>topological ordering</u>)

Tree structured CSPs

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

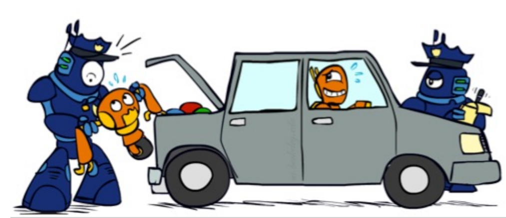


Remove backward:

For i=n:2, apply ArcConsistent(Parent(X_i),X_i)

Assign forward:

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



Tree structured CSP Solver

$X \leftarrow$ Topological Sort
for $i = n$ downto 2 do
    Make-Arc-Consistent(Parent(X_i), X_i) → remove all values from domain of Parent(X_i) which may violate arc-consistency.
for $i = 1$ to n do
    $X_i \leftarrow$ any consistent value (with its parent) in D_i

- After running loop1, any arc from a parent to its child is arc-consistent.

- ⇒ if the constraint graph has no loops, the CSP can be solved in $O(nd^2)$ time.

function $TREE\_CSP\_SOLVER(csp)$ **returns** a solution or failure

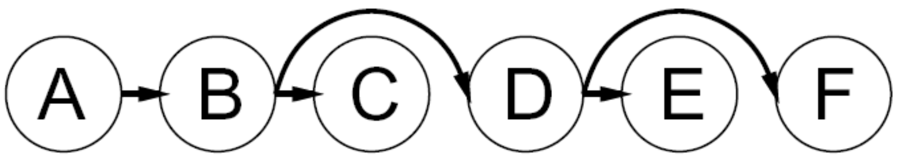
input: csp, a CSP with components X, D, C

$n \leftarrow$ number of variables in X
$assignment \leftarrow$ an empty assignment
$root \leftarrow$ any variable in X
$X \leftarrow TOPOLOGICAL(X, root)$
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→Y was made consistent at one point and Y's domain could not have been reduced thereafter (since Y's children were processed before Y)



- **Claim 2**: If root-to-leaf arcs are consistent, forward assignment will not backtrack

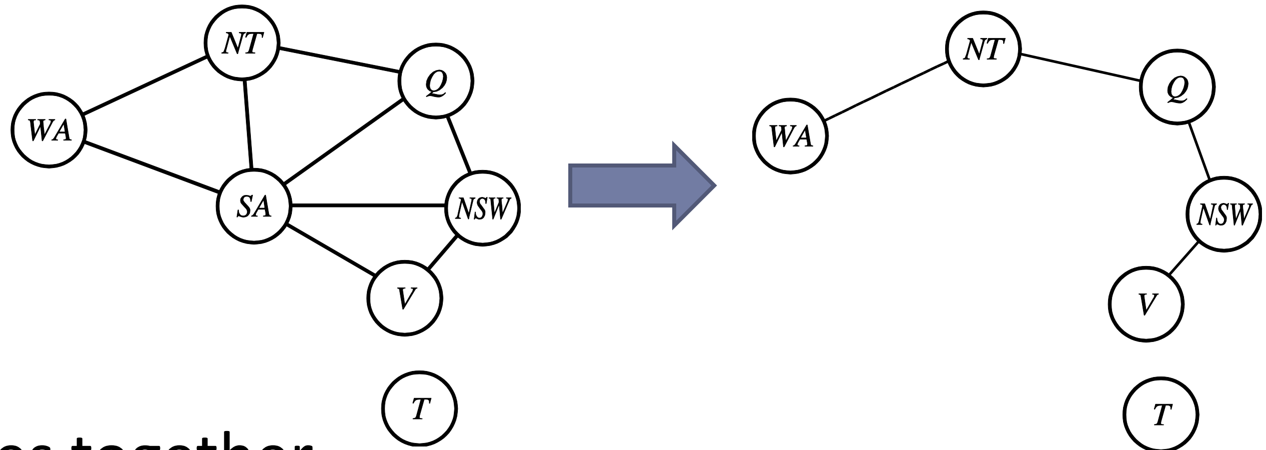
  Proof: Induction on position

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

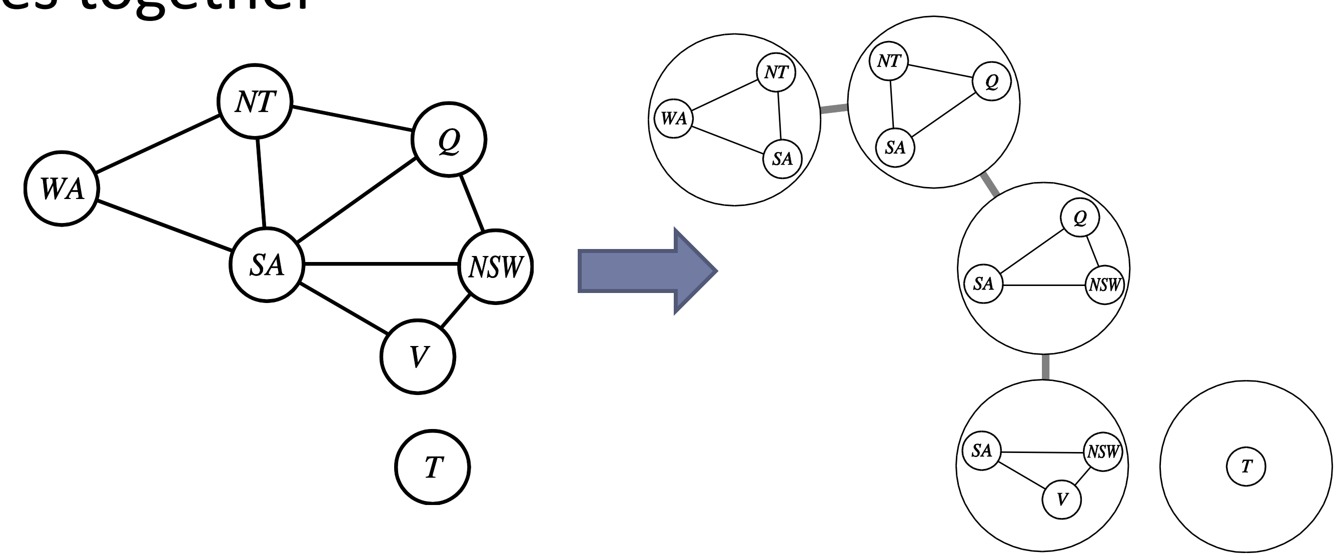
- Note: we'll see this basic idea again with Bayes' nets

Reduction of general graphs into trees

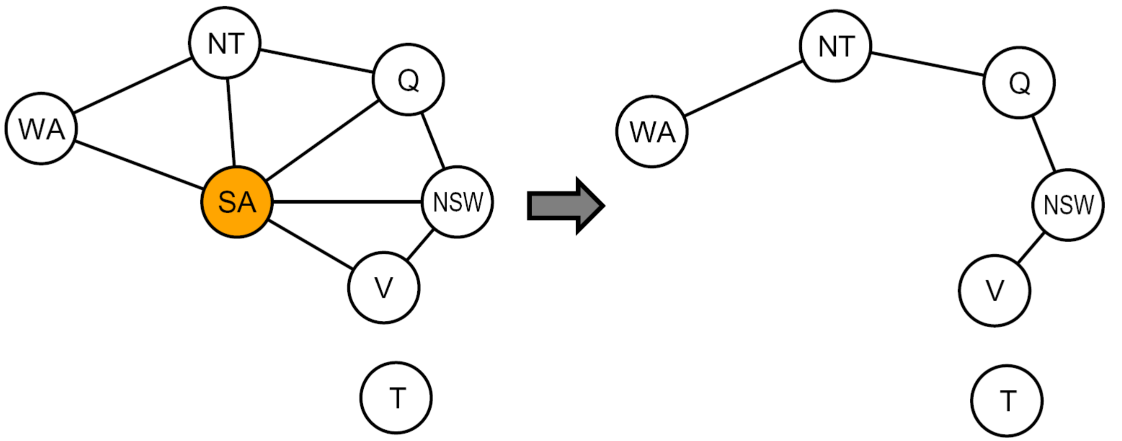
- Removing nodes



- Collapsing nodes together



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

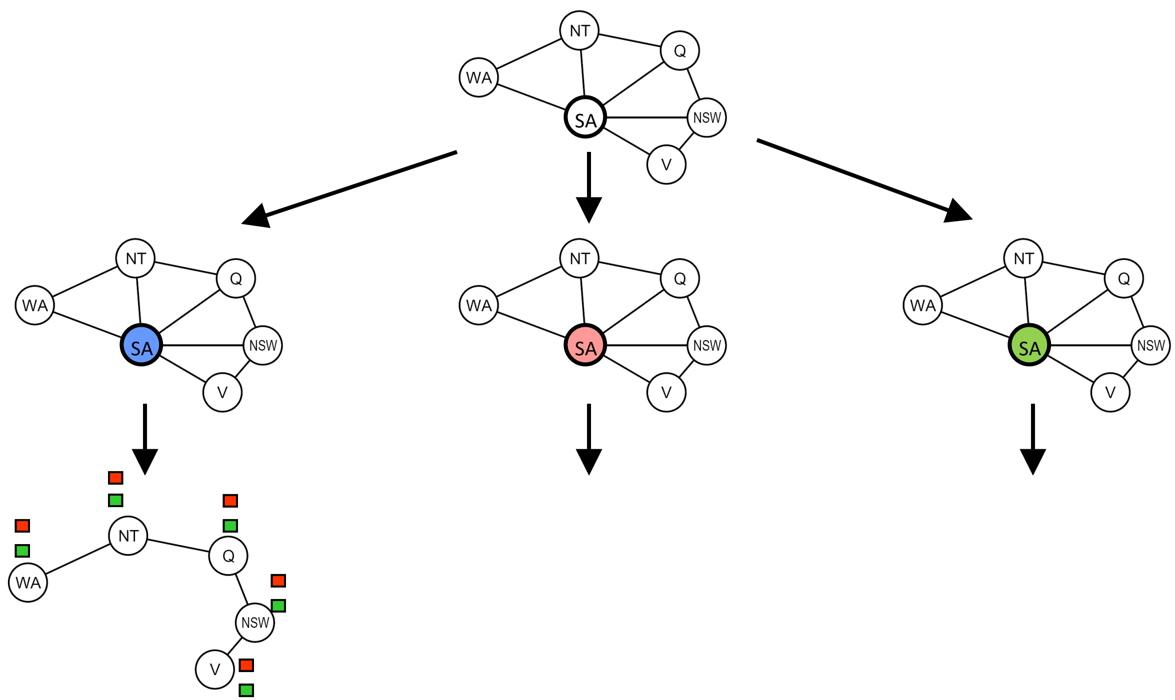
Cut-set conditioning

Choose a cutset

Instantiate the cutset (all possible ways)

Compute residual CSP for each assignment

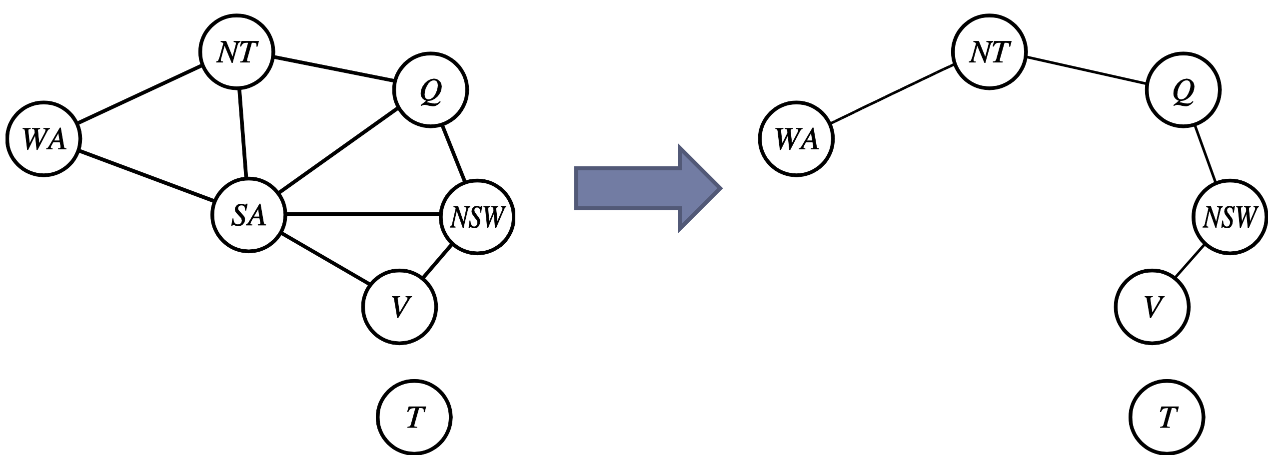
Solve the residual CSPs (tree structured)



Cut-set conditioning

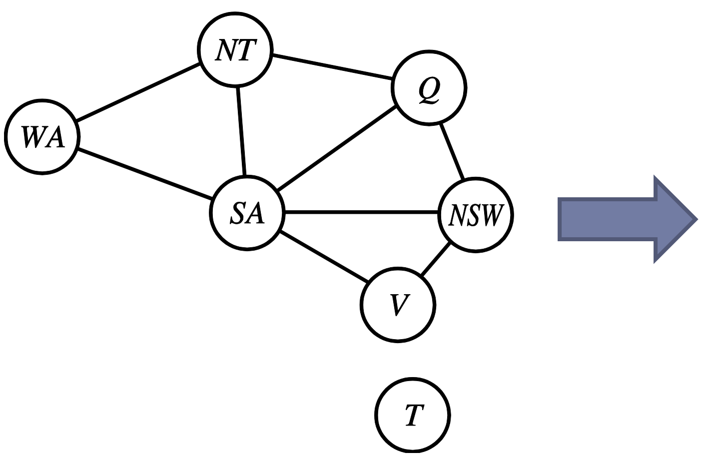
1) Find a <u>subset S</u> such that the remaining graph becomes a tree
2) For each possible consistent assignment to S
   a) remove inconsistent values from domains of remaining variables
   b) solve the remaining CSP which has a tree structure

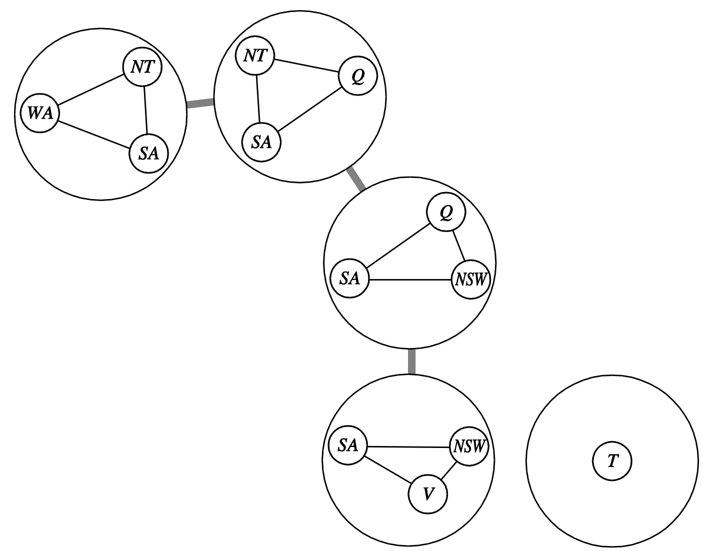
- Cutset size c gives runtime $O((d^c)\ (n-c)\ d^2\ )$
  - very fast for small c
  - c can be as large as $n-2$



Tree decomposition

- Create a tree-structured graph of overlapping sub-problems (each sub-problem as a mega-variable)
- Solve each sub-problem (enforcing local constraints)
- Solve the tree-structured CSP over mega-variables



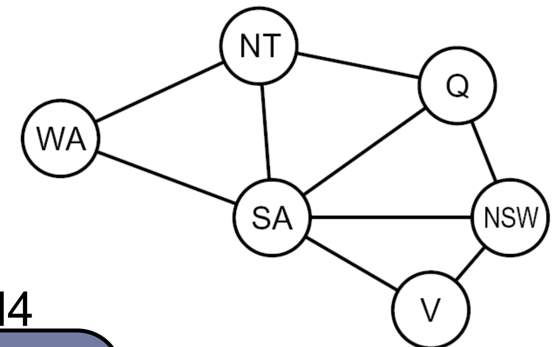


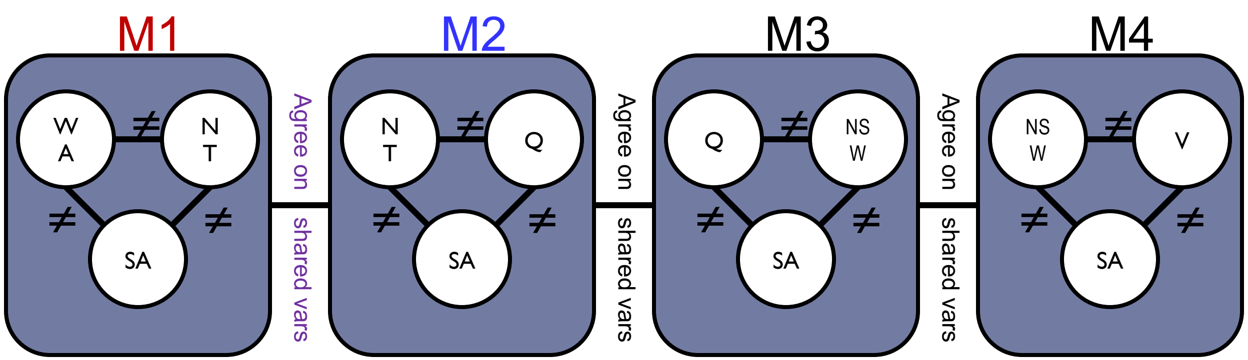
Tree decomposition

- Include all variables
- Each constraint must be in at least one sub problem.
- If a variable is in two sub-probs, it must be in all sub-probs along the path.

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





{(WA=r,SA=g,NT=b),
(WA=b,SA=r,NT=g),
…}

{(NT=r,SA=g,Q=b),
(NT=b,SA=g,Q=r),
…}

Agree: (M1,M2) ∈
{((WA=g,SA=g,NT=g), (NT=g,SA=g,Q=g)), …}

Solving CSPs by local search algorithms

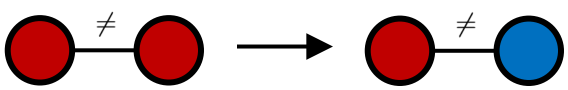
- In the CSP formulation as a search problem, path is irrelevant, so we can use complete-state formulation

- State: an assignment of values to variables
- Successors(s): all states resulted from s by choosing a new value for a variable
- Cost function h(s): Number of violated constraints
- Global minimum: $h(s) = 0$

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



- 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) = -total number of violated constraints

function $MIN\_CONFLICTS(csp, \max\_steps)$ **returns** a solution or failure

inputs: csp, a constraint satisfaction problem

$\mathrm{m}ax\_steps$, the number of steps allowed before giving up

$current \leftarrow$ an initial complete assignment for csp

for $i = 1$ to $\max\_steps$ do

if $current$ is a solution for csp **then return** current

$var \leftarrow$ a randomly chosen conflicted variable from $csp.VARIABLES$

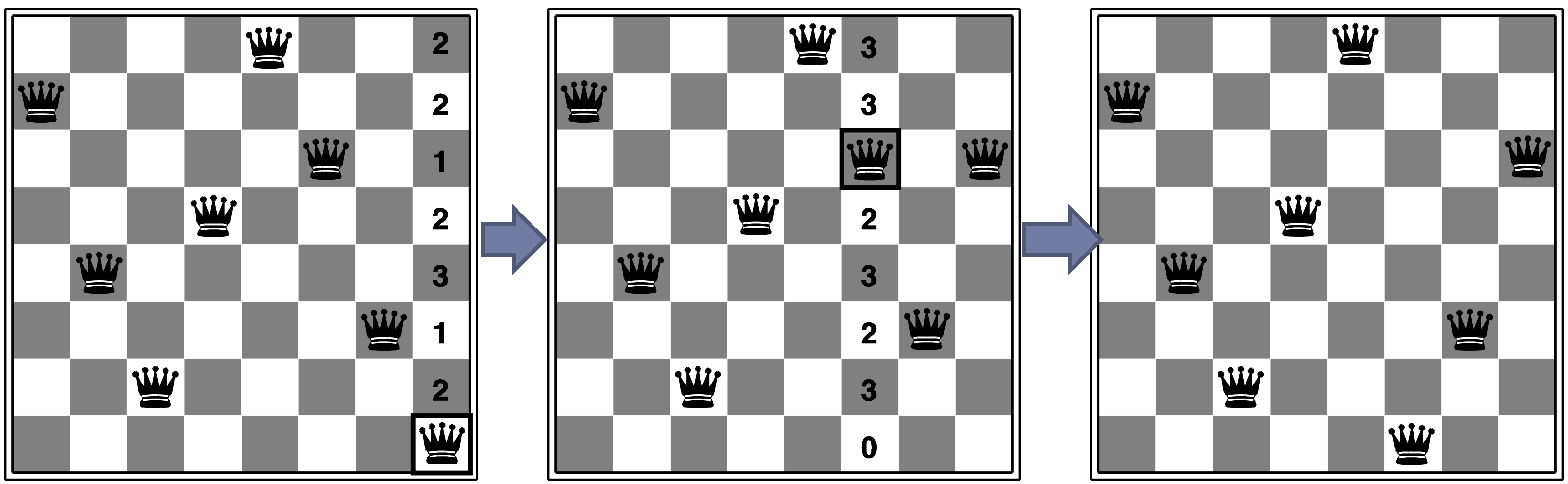
$value \leftarrow$ the value v for var that minimizes $CONFLICTS(var, v, current, csp)$

set $var = value$ in $current$

return $failure$

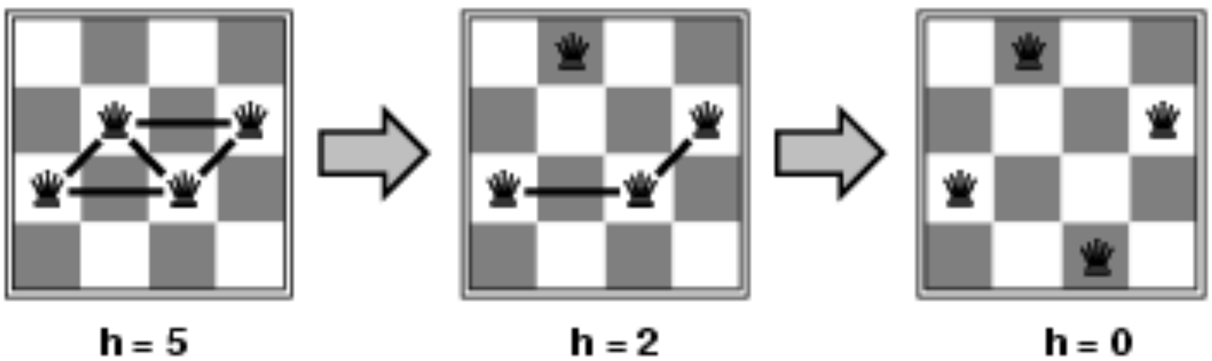
if current state is consistent then
return it
else
choose a random variable v, and change assignment of v
to a value that causes minimum conflict.

8-Queens example



$Q_8 = 3$ $Q_6 = 8$

4-Queens example



Local search for CSPs

- Variable selection: randomly select any conflicted variable
- Value selection by min-conflicts heuristic
  - choose value that violates the fewest constraints
    - i.e., hill-climbing

- Given random initial state, it can solve *n*-queens in almost constant time for arbitrary *n* with high probability
  - $n = 1000000$ in an average of 50 steps

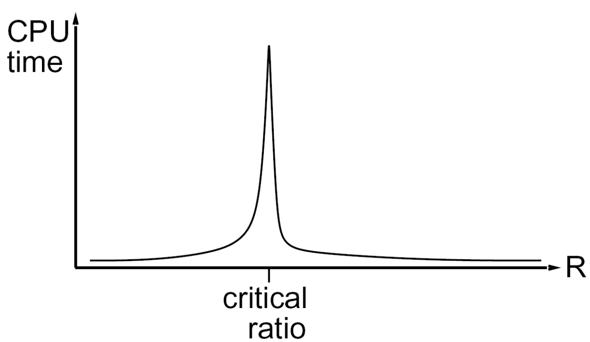
- N-queens is easy for local search methods (while quite tricky for backtracking)
  - Solutions are very densely distributed in the space and any initial assignment is guaranteed to have a solution nearby.

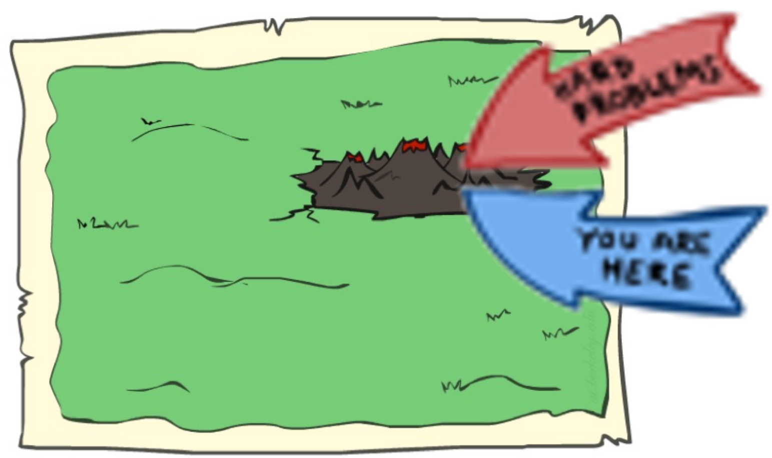
Performance of Min-Conflicts

- Given random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (e.g., n = 10,000,000)!

- The same appears to be true for any randomly-generated CSP *except* in a narrow range of the ratio

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$

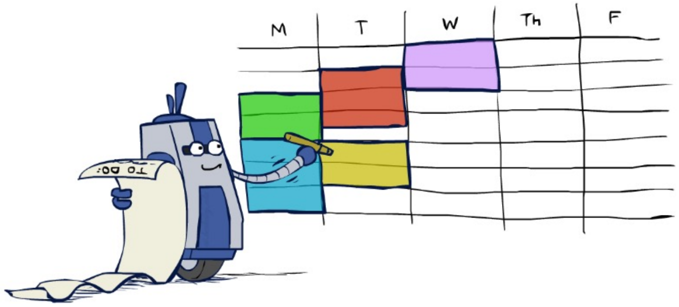




Summary

- CSP benefits
  - Standard representation of many problems
  - Generic heuristics (no domain specific expertise)

- CSPs solvers (based on systematic search)
  - Basic solution: backtracking search
  - Speed-ups:
    - Ordering
    - Filtering
    - Structure



- Graph structure may be useful in solving CSPs efficiently.

- Local search methods for CSPs: Iterative min-conflicts is usually effective in solving CSPs.