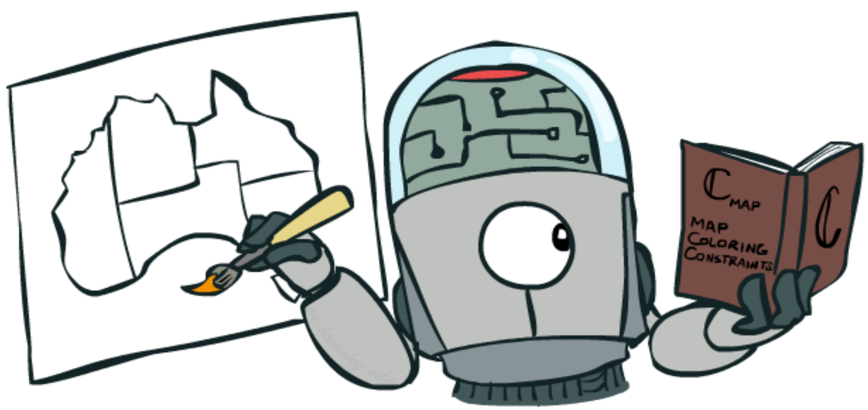
Constraint Satisfaction Problems (CSPs)

Chris Amato
Northeastern University

Some images and slides are used from: Rob Platt, CS188 UC Berkeley, AIMA

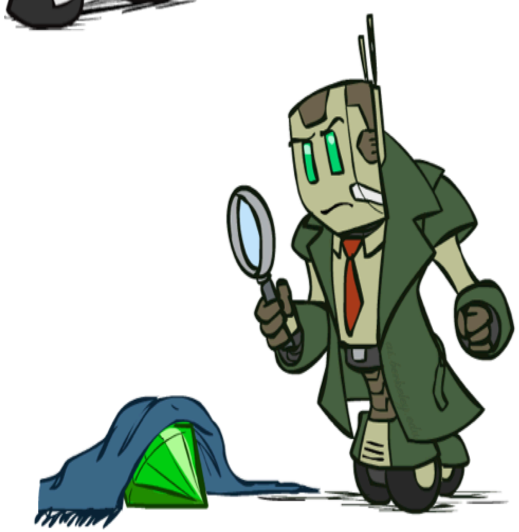


What is search for?

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

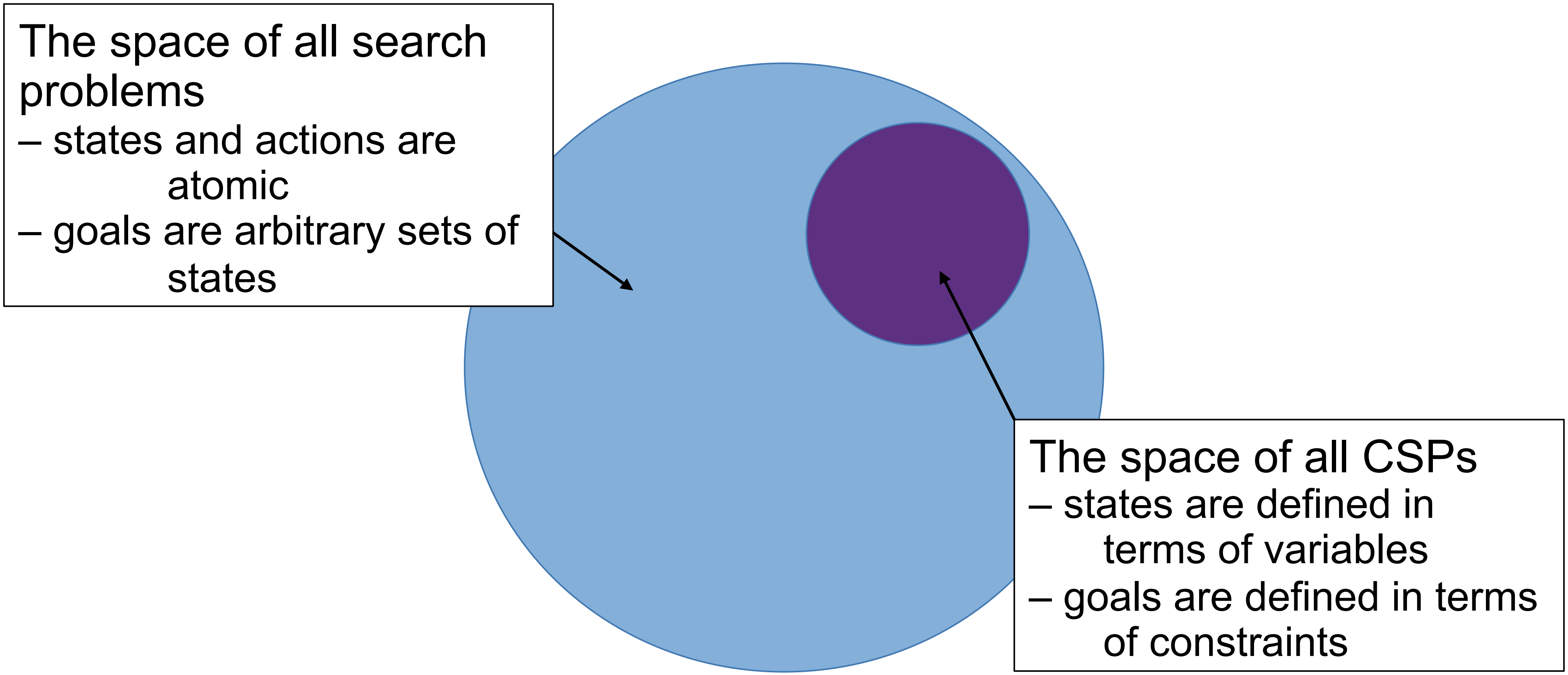


Identification: assignments to variables

- The goal itself is important, not the path
- All paths at the same depth (for some formulatior
- CSPs are specialized for identification problems

What is a CSP?

CSPs $\subseteq$ All search problems



<u>A CSP is defined by:</u>
1. a set of variables and their associated domains.
2. a set of constraints that must be satisfied.

What is a CSP?

Standard search problem:

– *state* is a “black box”—any old data structure that supports goal test, eval, successor

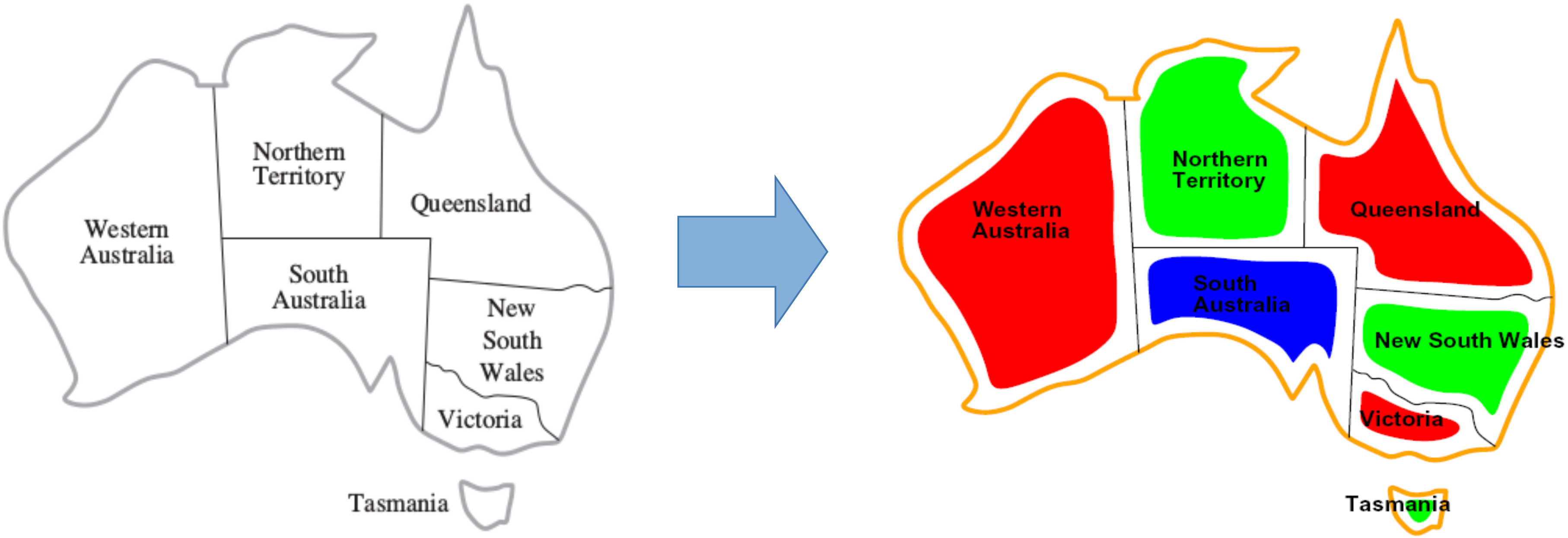
CSP:

– *state* is defined by *variables* X_i with values from *domain* D_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

CSP example: map coloring



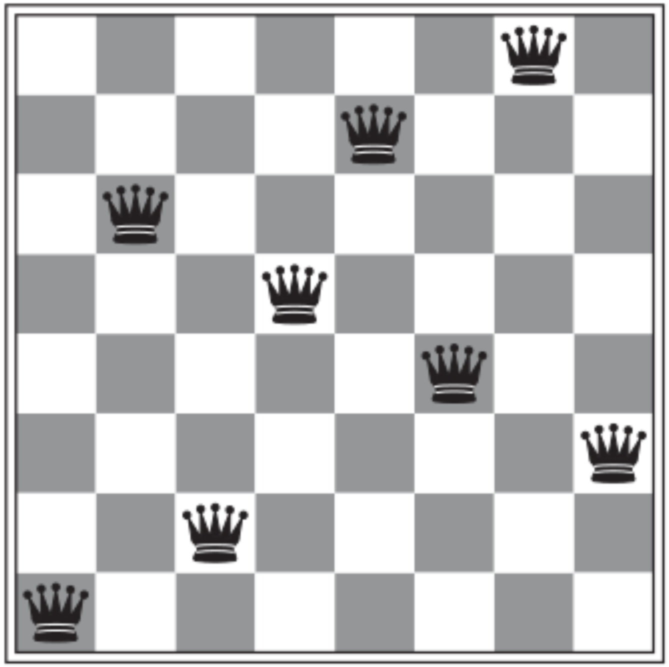
<u>Problem:</u> assign each territory a color such that no two adjacent territories have the same color

Variables: $X = \{WA, NT, Q, NSW, V, SA, T\}$

Domain of variables: $D = \{r, g, b\}$

Constraints: $C = \{SA \neq WA, SA \neq NT, SA \neq Q, \ldots\}$

CSP example: *n*-queens



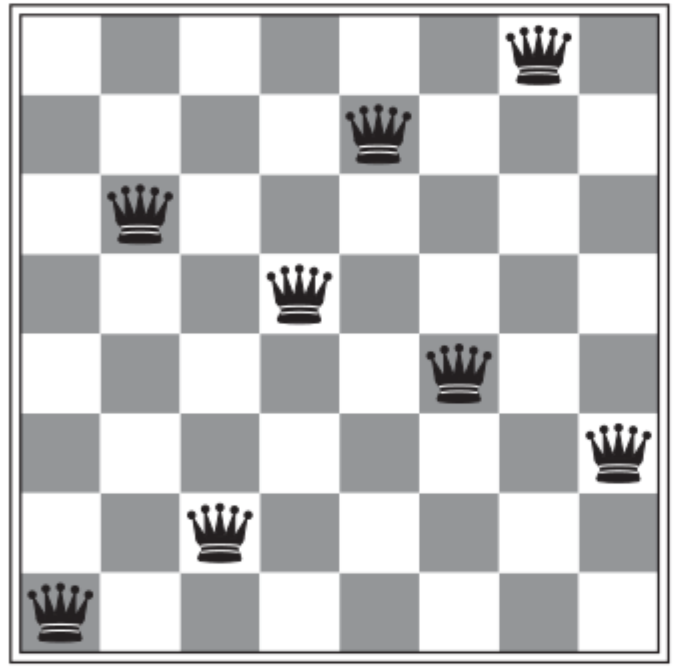
Problem: place *n* queens on an *n*x*n* chessboard such that no two queens threaten each other

Variables: $X = ?$

Domain of variables: $D = ?$

Constraints: $C = ?$

CSP example: *n*-queens



Problem: place *n* queens on an *n*x*n* chessboard such that no two queens threaten each other

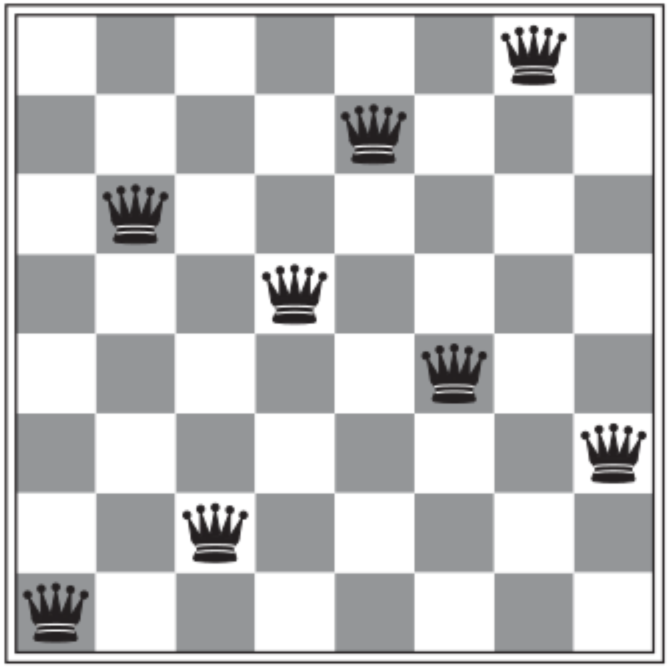
Variables: $X =$ One variable for every square

Domain of variables: $D =$ Binary

Constraints: $C =$ Enumeration of each possible disallowed configuration

– why is this a bad way to encode the problem?

CSP example: *n*-queens



Problem: place *n* queens on an *nxn* chessboard such that no two
queens th

Variables

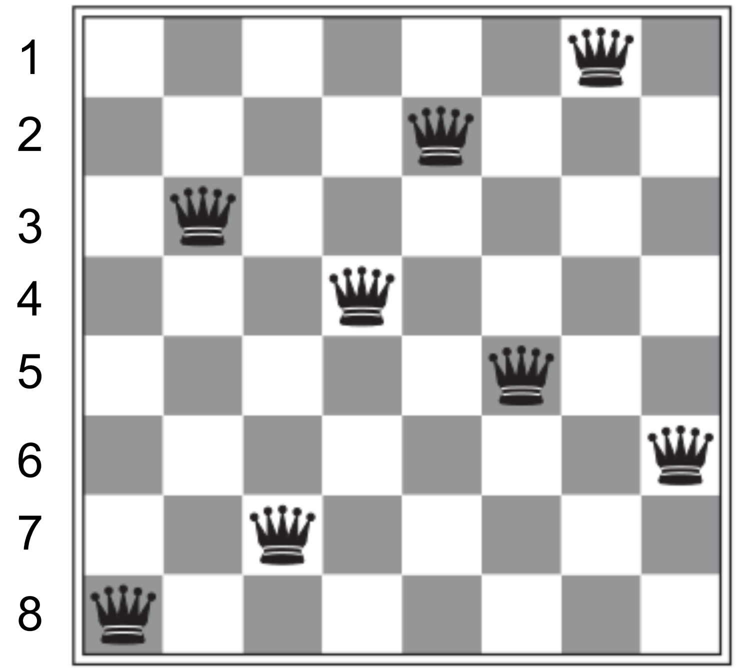
Domain c

Constrair   ved configuration

Is there a better way?

– why is this a bad way to encode the problem?

CSP example: *n*-queens



Problem: place *n* queens on an *n*x*n* chessboard such that no two queens threaten each other

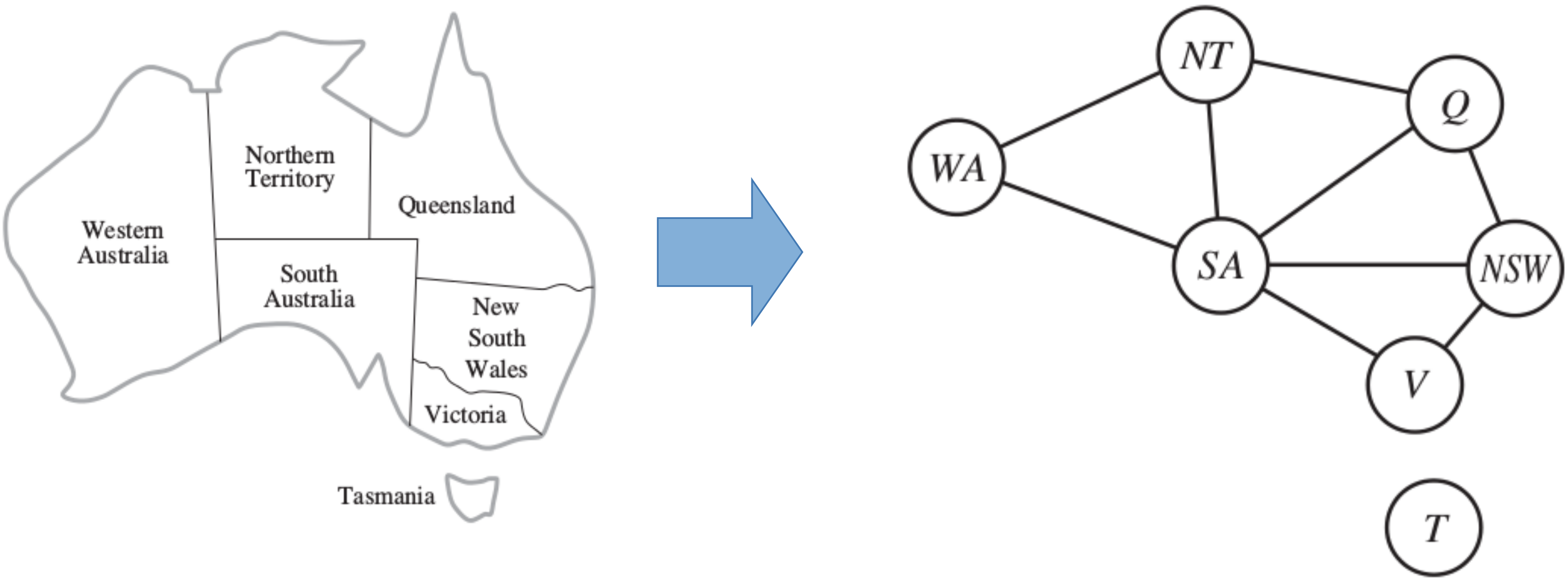
Variables: $X =$ One variable for each row (i.e, each queen)

Domain of variables: $D =$ A number between 1 and 8

Constraints: $C =$ Enumeration of disallowed configurations

– why is this representation better?

The constraint graph



Binary CSP: each constraint relates at most two variables

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!

A harder CSP to represent: Cryptarithmetic

- **Variables:**

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

- **Domains:**

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

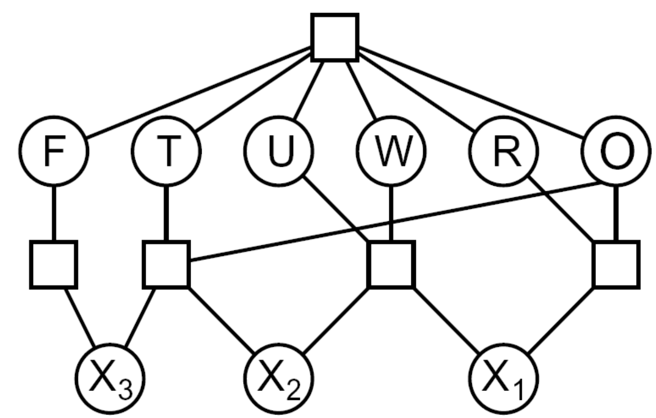
- **Constraints:**

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

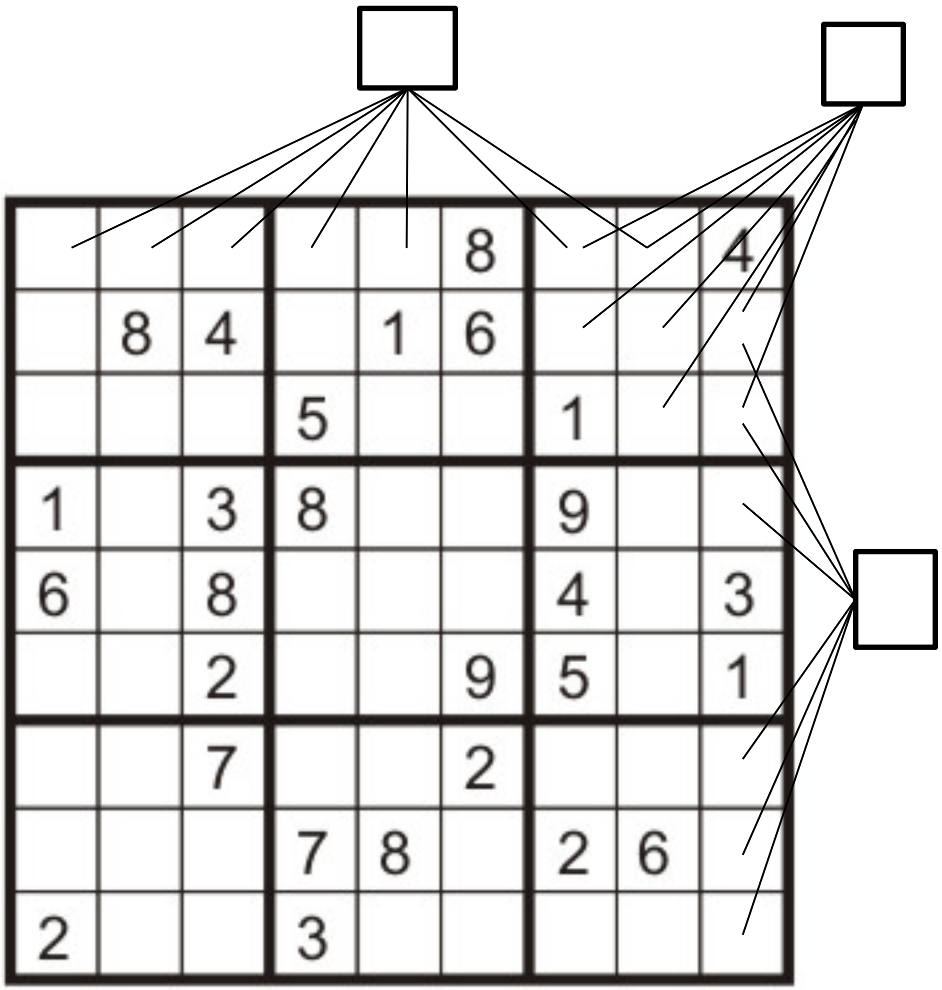
$O + O = R + 10 \cdot X_1$

$\cdots$

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



Another example: sudoku



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

9-way alldiff for each column

9-way alldiff for each row

9-way alldiff for each region

(or can have a bunch of pairwise inequality constraints)

Varieties of CSPs

Discrete Variables

- 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 (e.g., constrained optimization problems)

Real-world CSPs

Assignment problems: e.g., who teaches what class

Timetabling problems: e.g., which class is offered when and where?

Hardware configuration

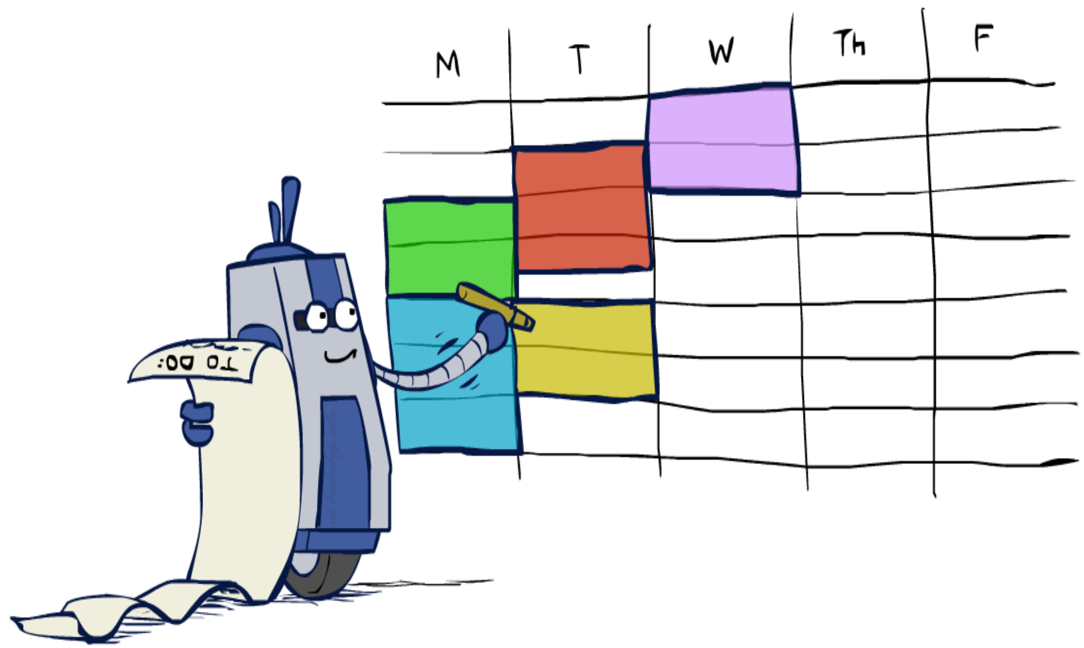
Transportation scheduling

Factory scheduling

Circuit layout

Fault diagnosis

… lots more!



Many real-world problems involve real-valued variables…

Standard search formulation of CSPs

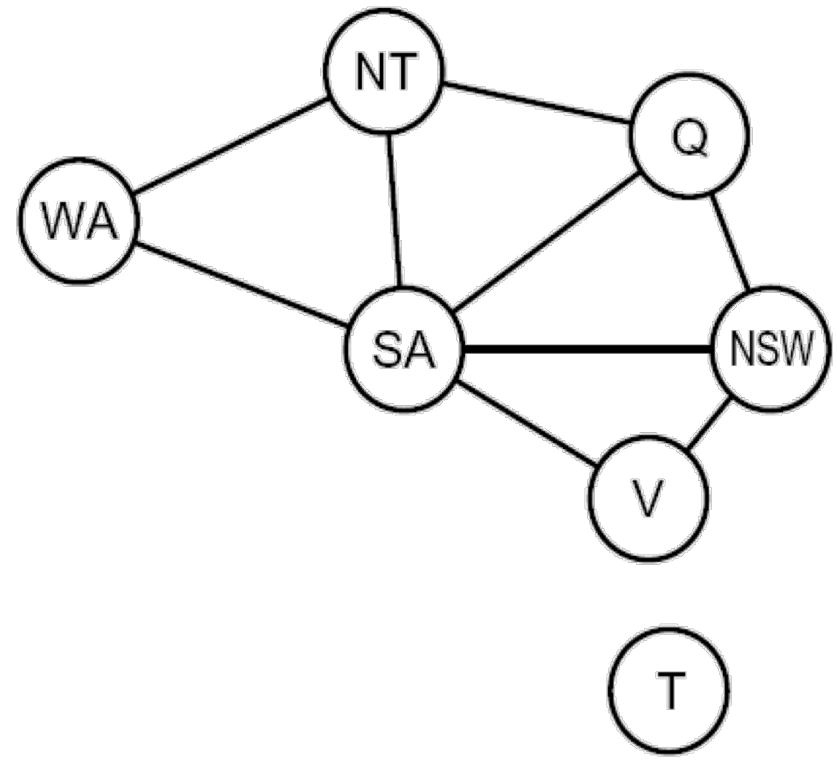
States defined by the values assigned so far (partial assignments)

- 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?

Naive solution: apply BFS, DFS, A*, ...

_ _ _ _ _ _ _

R _ _ _ _ _ _

R G _ _ _ _ _

R G R _ _ _ _

⋮ ⋮

R G R R R R R

How many leaf nodes are expanded in the worst case?

Naive solution: apply BFS, DFS, A*, ...

_ _ _ _ _ _ _

R _ _ _ _ _ _

R G _ _ _ _ _

R G R _ _ _ _

⋮

⋮

R G R R R R R

How many leaf nodes are expanded in the worst case? $3^7 = 2187$

Naive solution: apply BFS, DFS, A*, ...

_ _ _ _ _ _ _

R _ _ _ _ _ _

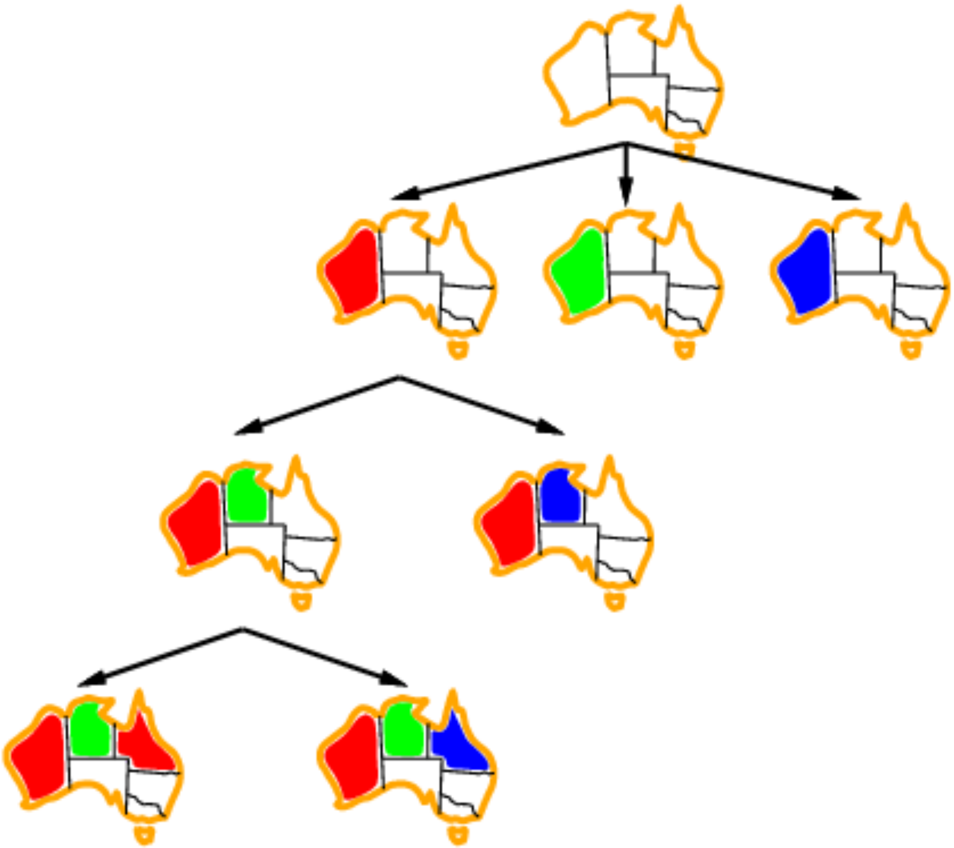
This is bad.
How can we improve it?

R G R R R R R

How many leaf nodes are expanded in the worst case? $3^7 = 2187$

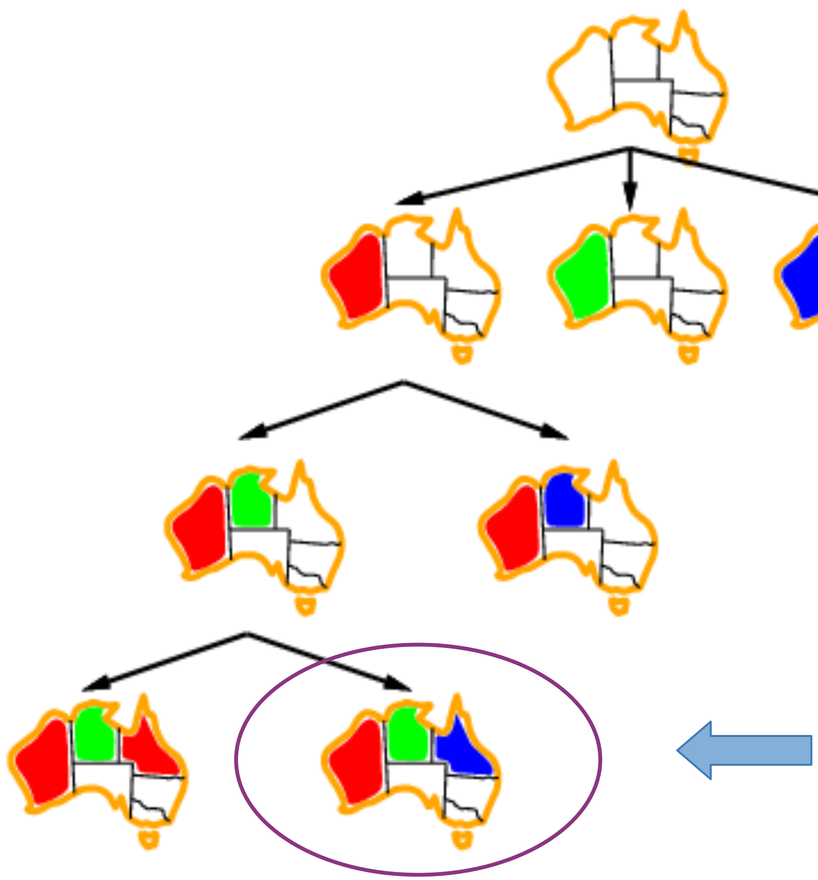
Backtracking search

When a node is expanded, check that each successor state
is consistent before adding it to the queue.



Backtracking search

When a node is expanded, check that each successor state is consistent before adding it to the queue.



Does this state have any valid successors?

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?
– Backtracking enables us the ability to solve a problem as big as 25-queens

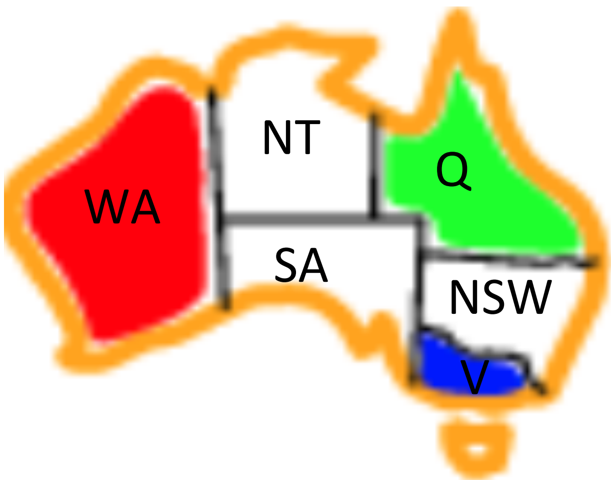
Forward checking

Sometimes, failure is inevitable:

Can we detect this situation in advance?

Forward checking

Sometimes, failure is inevitable:



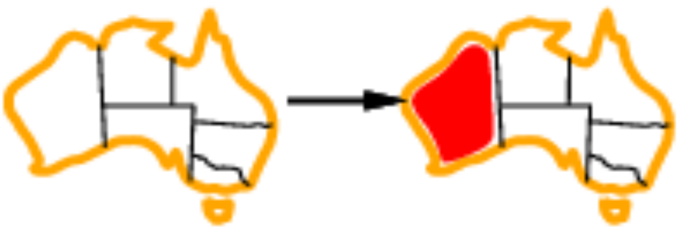
Can we detect this situation in advance?

Yes: keep track of viable variable assignments as you go

Forward checking

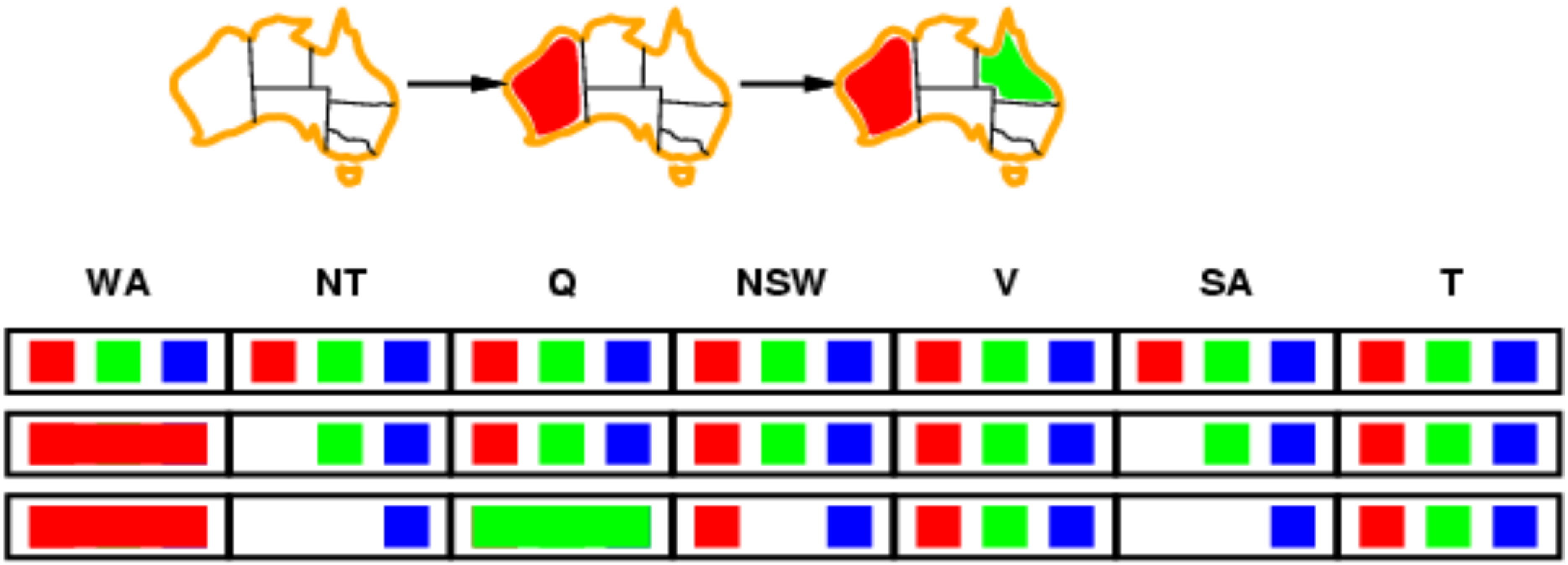
Track domain for each unassigned variable
- initialize w/ domains from problem statement
- each time you expand a node, update domains of all unassigned variables

Forward checking



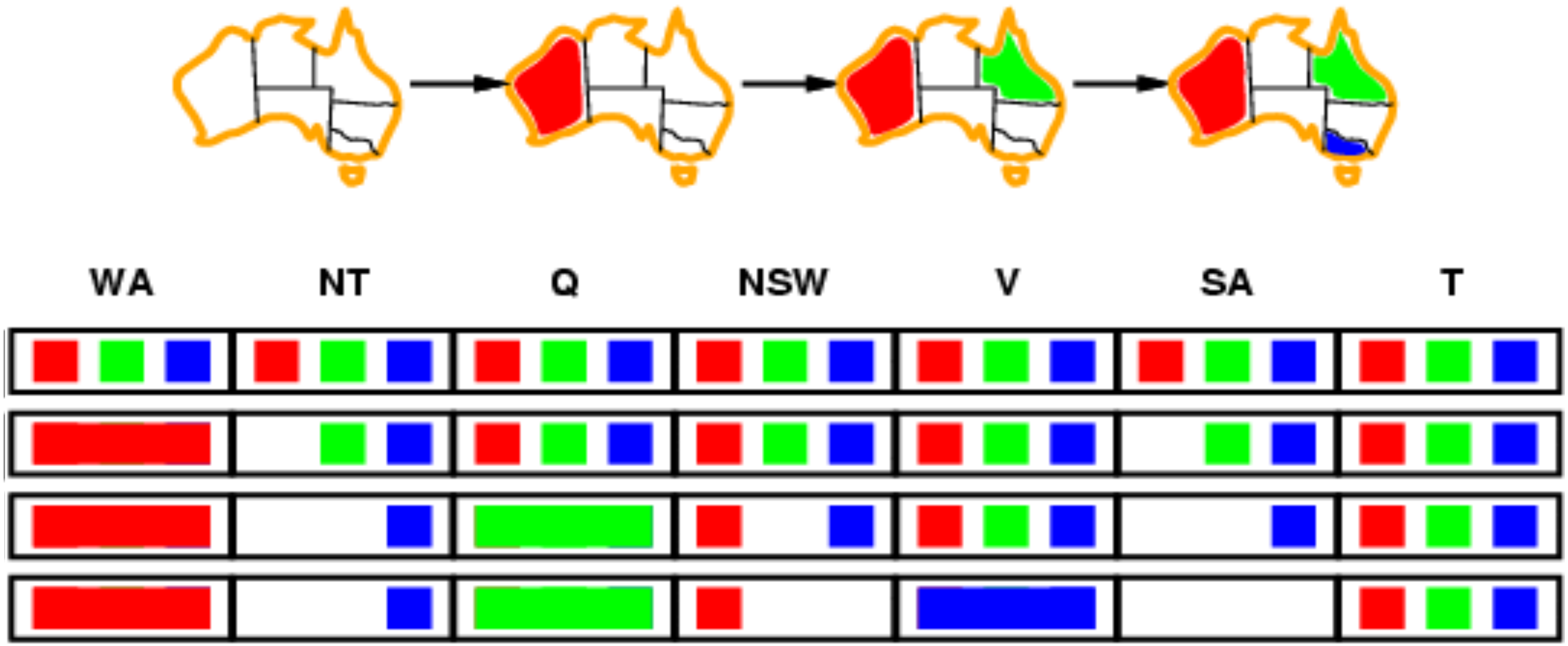
Track domain for each unassigned variable
- initialize w/ domains from problem statement
- each time you expand a node, update domains of all unassigned variables

Forward checking



Track domain for each unassigned variable
- initialize w/ domains from problem statement
- each time you expand a node, update domains of all unassigned variables

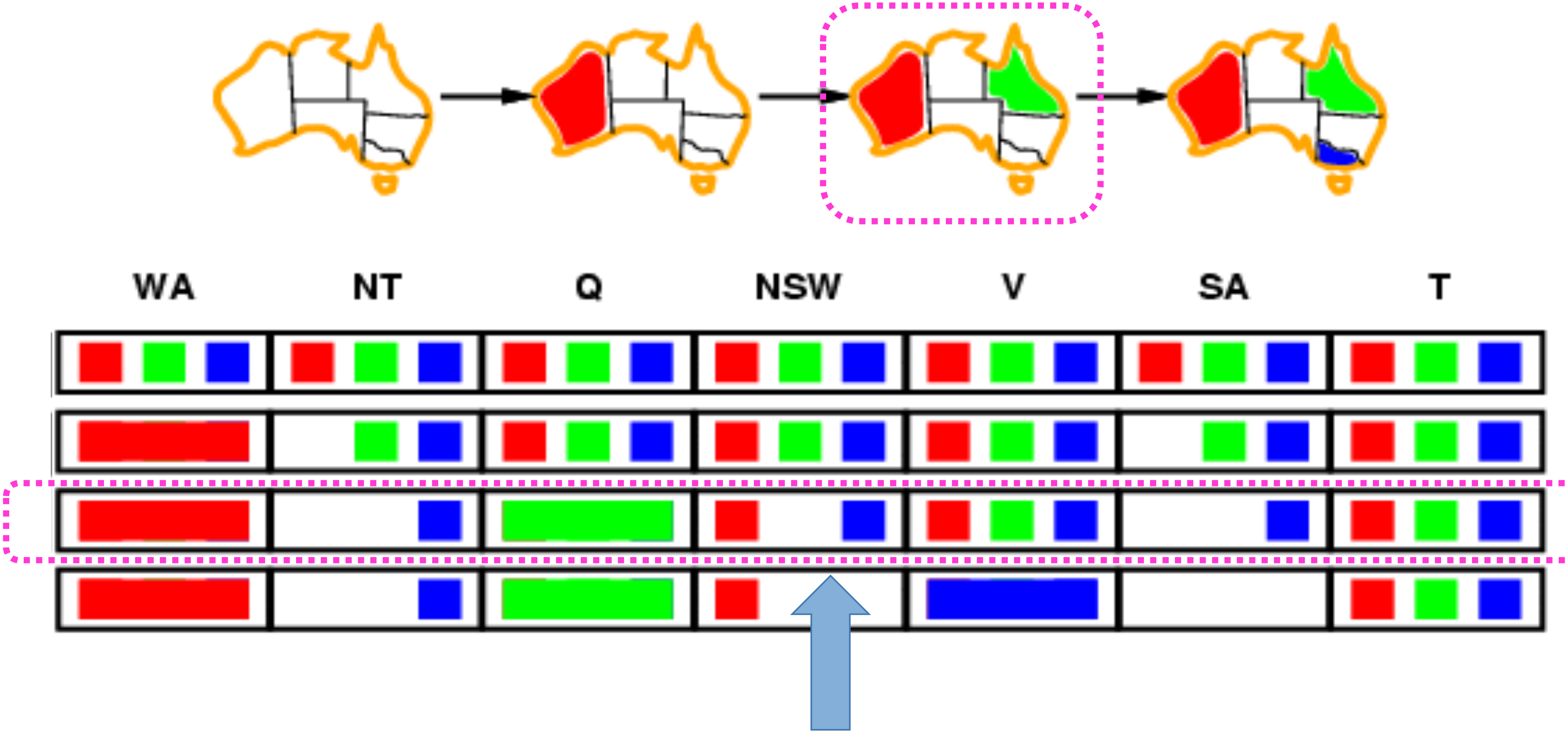
Forward checking



Track domain for each unassigned variable

- initialize w/ domains from problem statement
- each time you expand a node, update domains of all unassigned variables

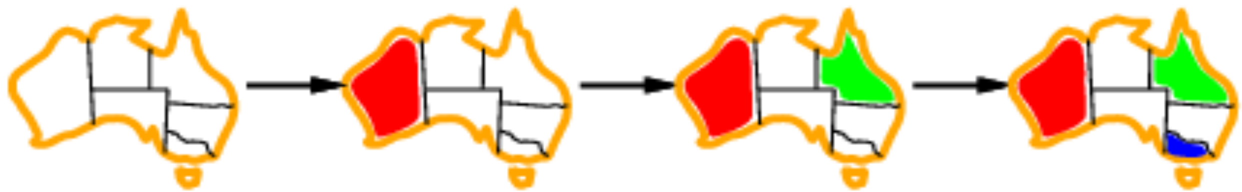
Forward checking



But, failure was inevitable here!
– what did we miss?

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!

Constraint propagation repeatedly enforces constraints locally

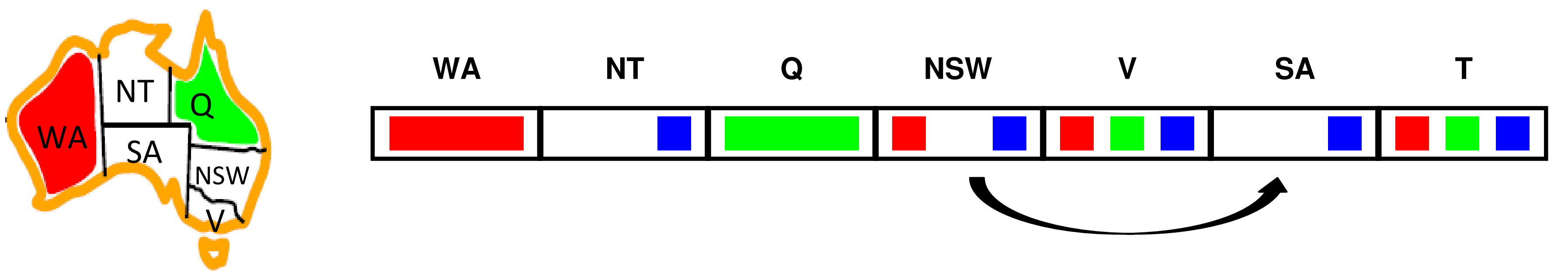
Arc consistency

Simplest form of propagation makes *each arc* consistent

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

Arc consistency: $X \rightarrow Y$ is consistent iff

for every value x of X there is some allowed y



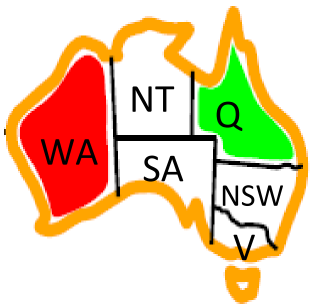
Delete values from tail in order to make each arc consistent

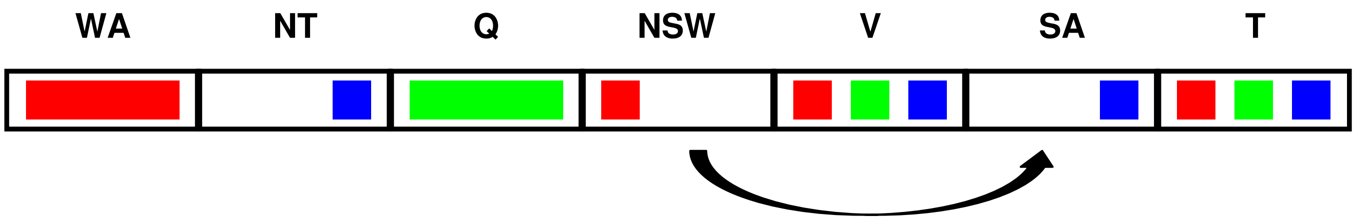
Arc consistency

Simplest form of propagation makes *each arc* consistent

$X \to Y$ is consistent iff:

for every value x of X there is some allowed y





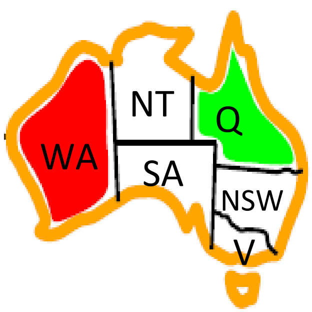
Delete values from tail in order to make each arc consistent

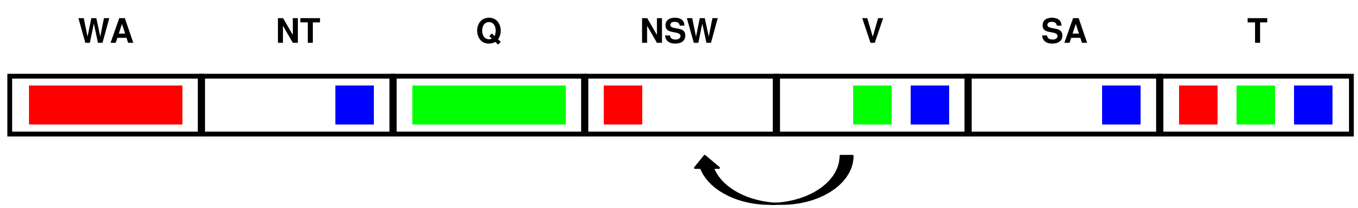
Arc consistency

Simplest form of propagation makes *each arc* consistent

$X \rightarrow Y$ is consistent iff:

for every value x of X there is some allowed y





Delete values from tail in order to make each arc consistent

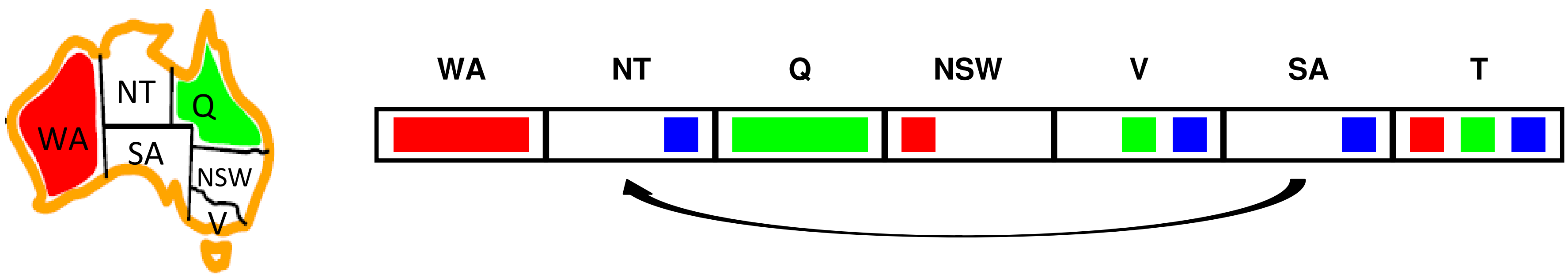
If X loses a value, neighbors of X need to be rechecked!

Arc consistency

Simplest form of propagation makes *each arc* consistent

$X \rightarrow Y$ is consistent iff:

for every value x of X there is some allowed y



Delete values from tail in order to make each arc consistent

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

Arc consistency

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*
  return *true*

---

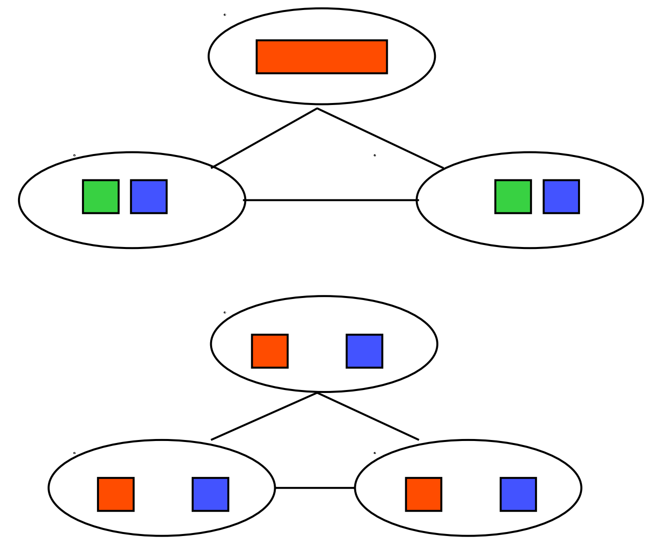
function REVISE(*csp*, $X_i,\ X_j$) **returns** true iff we revise the domain of X_i
  revised $\leftarrow$ *false*
  for each x **in** D_i **do**
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j **then**
      delete x from D_i
      revised $\leftarrow$ *true*
  return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

Why does this algorithm converge?

What’s the downside of enforcing arc consistency?

Arc consistency does not detect all inconsistencies...

- 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!

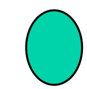


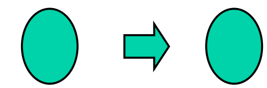
What went wrong here?

K-consistency

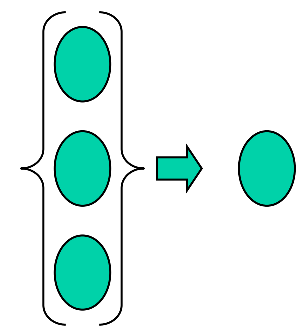
Increasing degrees of consistency

- 1-Consistency (Node Consistency): Each single node's domain has a value which meets that node's unary constraints



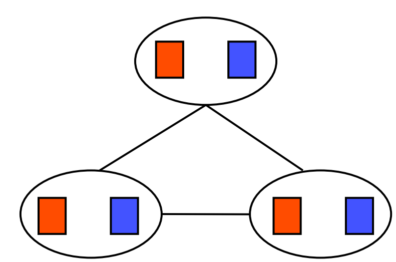


- 2-Consistency (Arc Consistency): For each pair of nodes, any consistent assignment to one can be extended to the other



- K-Consistency: For each k nodes, any consistent assignment to k-1 can be extended to the k^{th} 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)

Improving backtracking efficiency

General-purpose methods can give huge gains in speed:

1. Can we detect inevitable failure early?

2. Which variable should be assigned next?

3. In what order should its values be tried?

4. Can we take advantage of problem structure?

Heuristics for improving CSP performance

Minimum remaining values (MRV) heuristic:

– expand variables w/ minimum size domain first

Heuristics for improving CSP performance

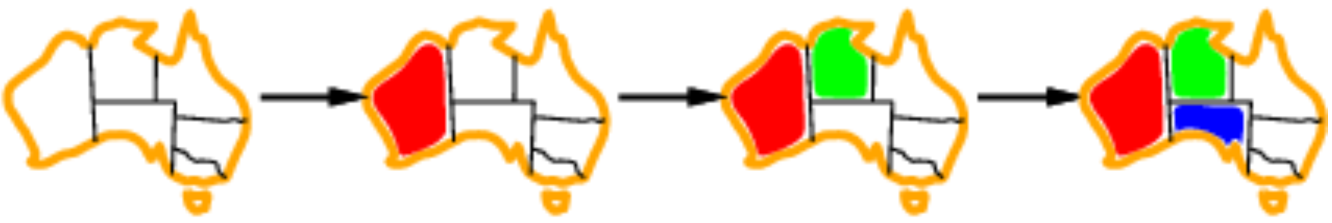
Minimum remaining values (MRV) heuristic:

– expand variables w/ minimum size domain first

Heuristics for improving CSP performance

Minimum remaining values (MRV) heuristic:

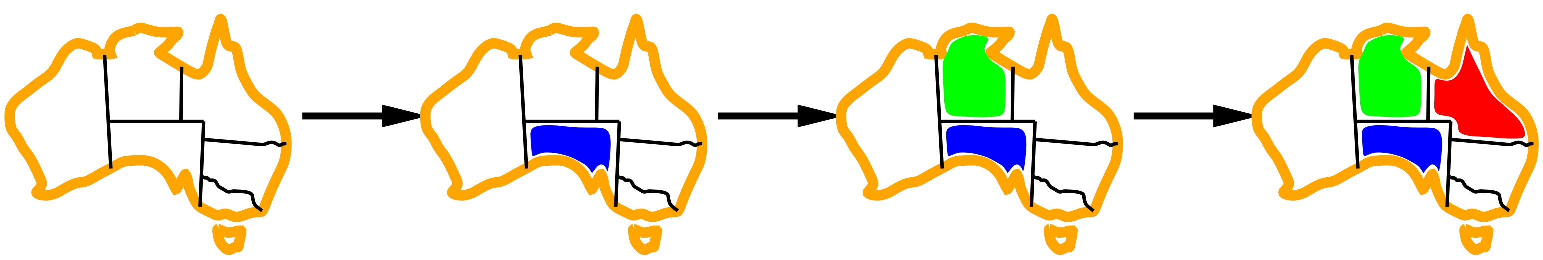
– expand variables w/ minimum size domain first



Heuristics for improving CSP performance

Degree heuristic:

– tie breaker for MRV heuristic
– choose the variable with the most constraints
  on remaining variables

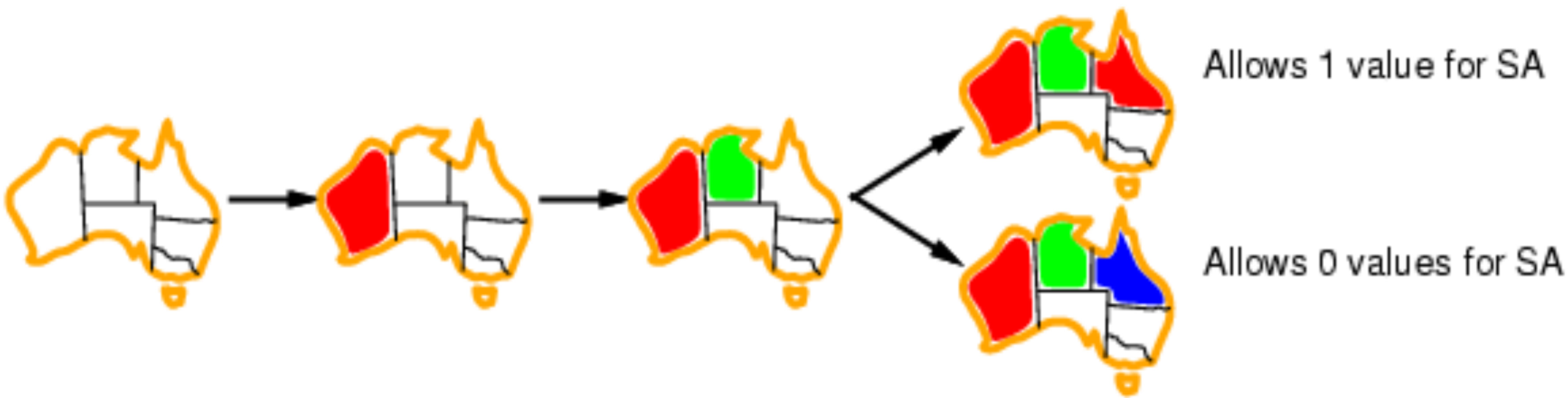


Heuristics for improving CSP performance

Least constraining value (LCV) heuristic:

– consider how domains of neighbors would change

– choose value that contrains neighboring domains the **least**



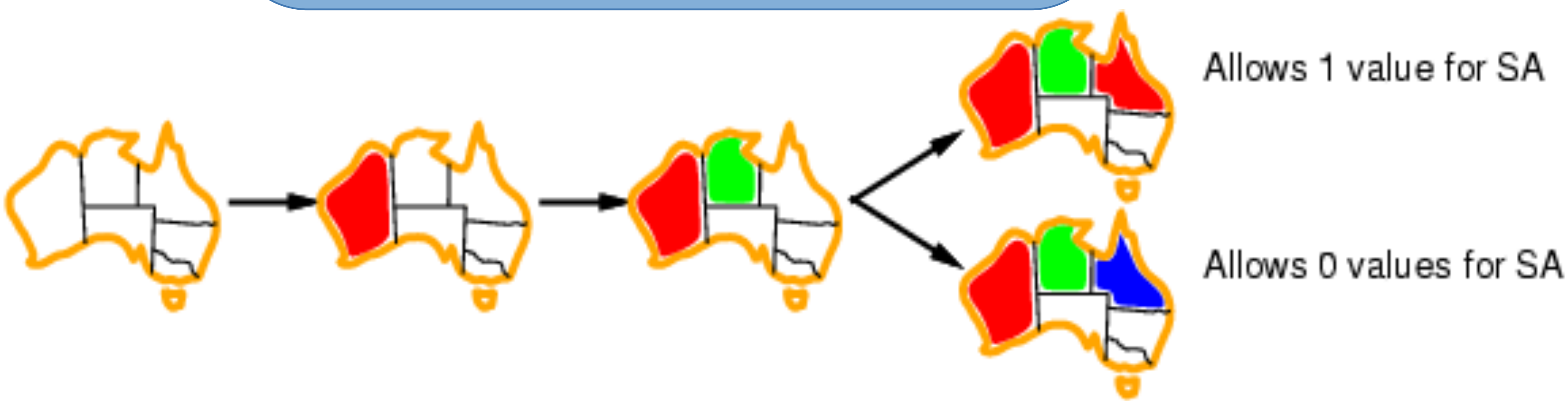
Heuristics for improving CSP performance

Least constraining value (LCV) heuristic:

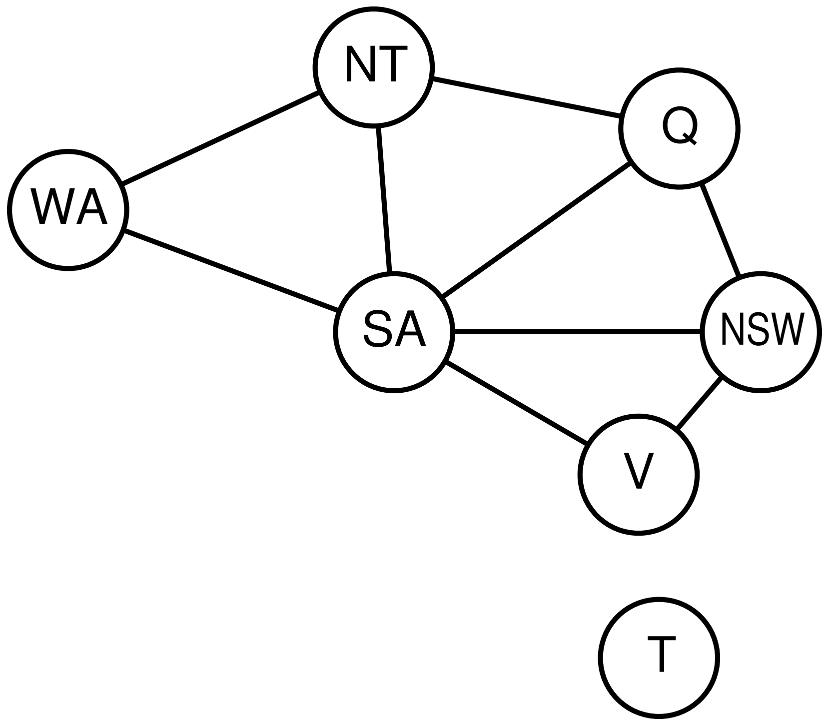
– consid would change

– choos ring domains
the

The combination of MRV and LCV w/ backtracking can solve the 1000-queens problem



Problem structure



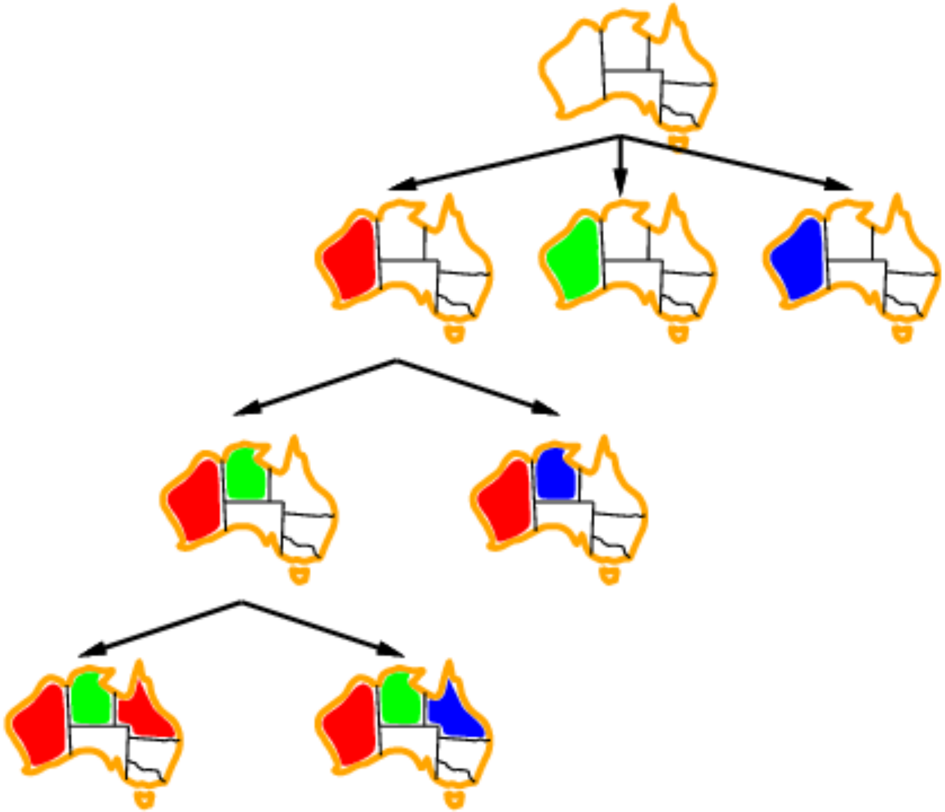
Tasmania and mainland are independent subproblems

Identifiable as connected components of constraint graph

Using structure to reduce problem complexity

In general, what is the complexity of solving a CSP using backtracking?

(in terms of # variables, n, and max domain size, d)

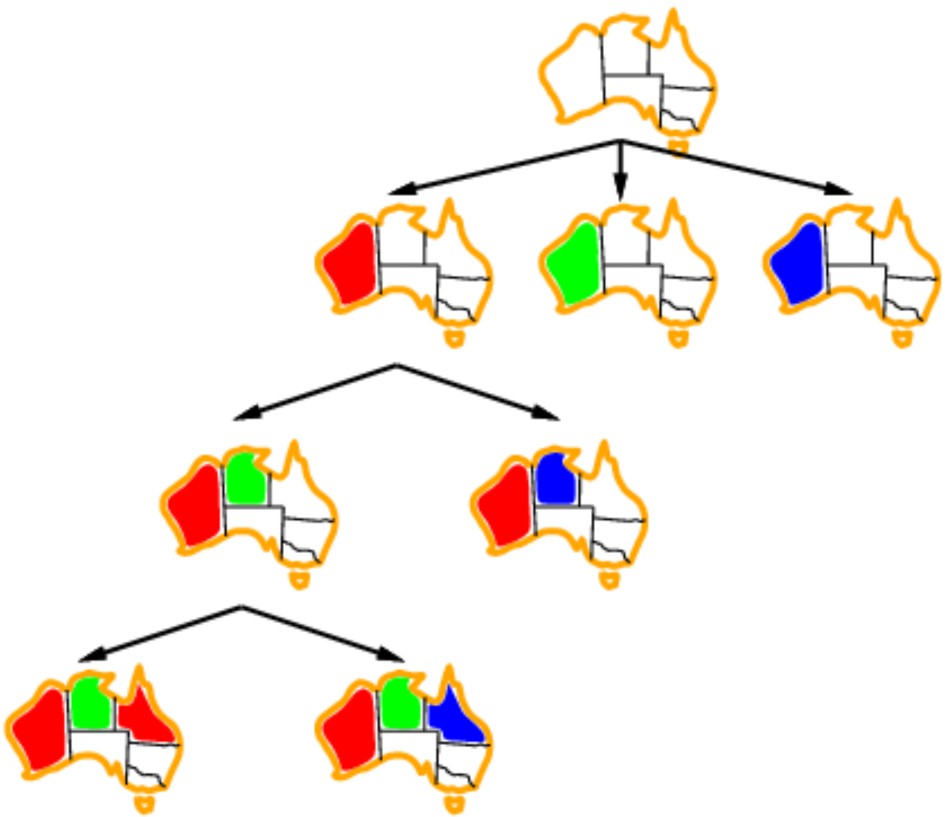


But, sometimes CSPs have special structure that makes them simpler!

Using structure to reduce problem complexity

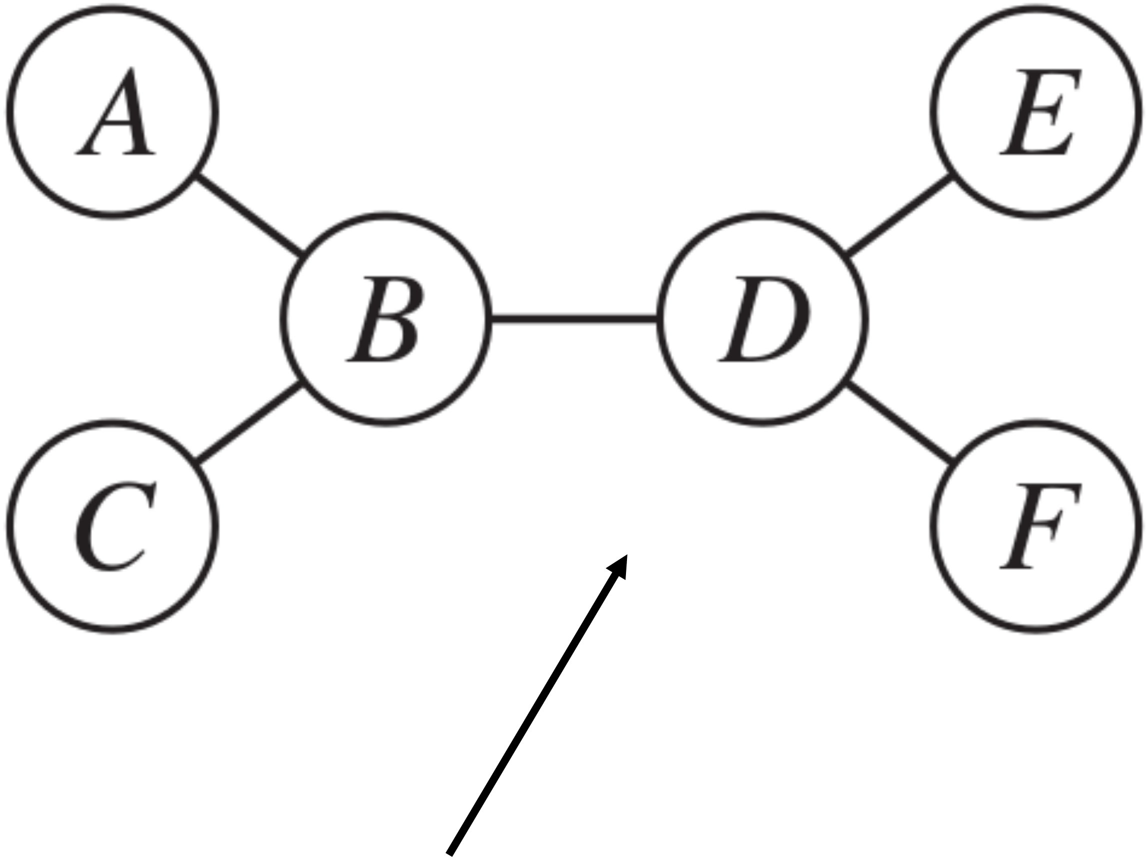
In general, what is the complexity of solving a CSP using backtracking?

(in terms of # variables, n, and max domain size, d) d^n



But, sometimes CSPs have special structure that makes them simpler!

When the constraint graph is a tree

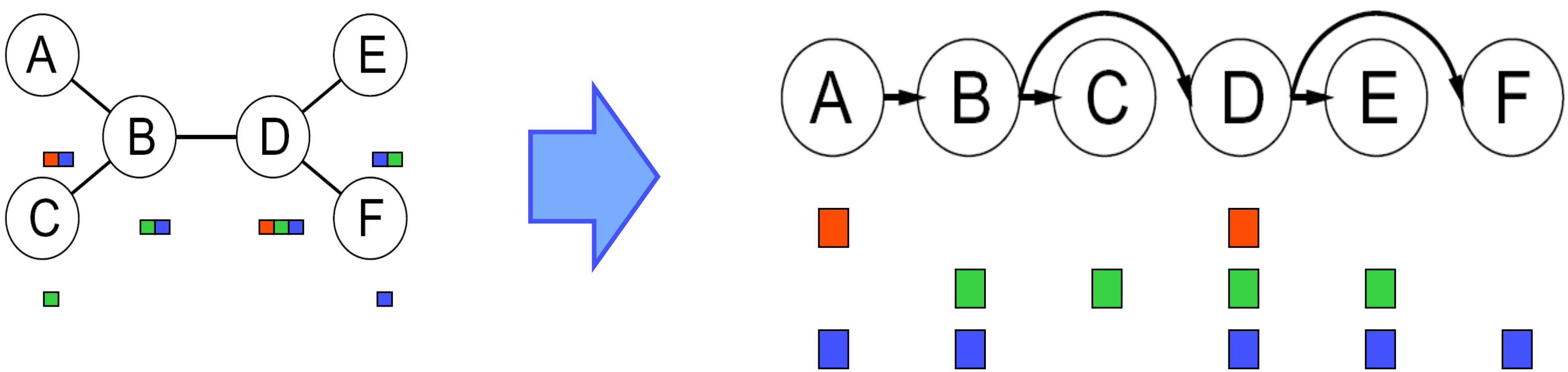


This CSP is easier to solve than the general case...

Tree-structured CSPs

Algorithm for tree-structured CSPs:

Order: Choose a root variable, order variables so that parents precede children



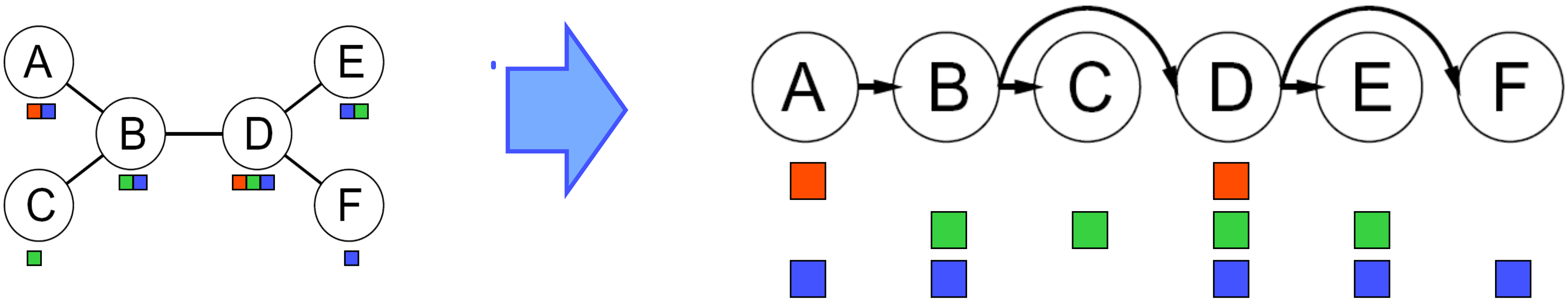
Remove backward: For $i = n : 2$, apply RemInconsistent(Par(X_i), X_i)

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

When the constraint graph is a tree

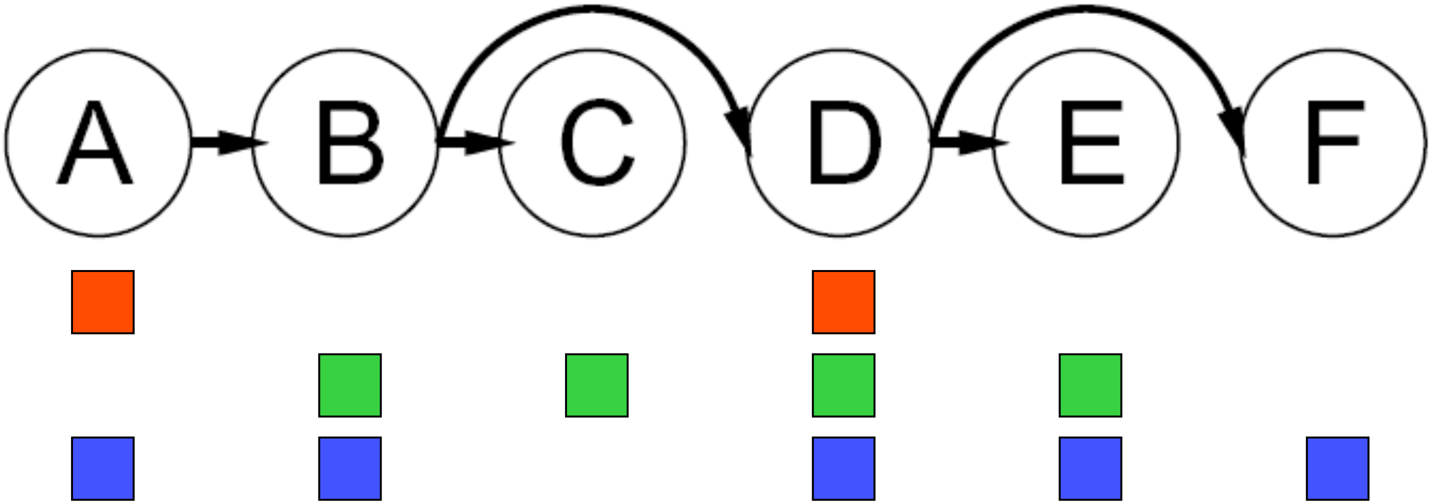
1. Do a *topological sort*
   – a partial ordering over variables

   i. choose any node as the root
   ii. list children after their parents



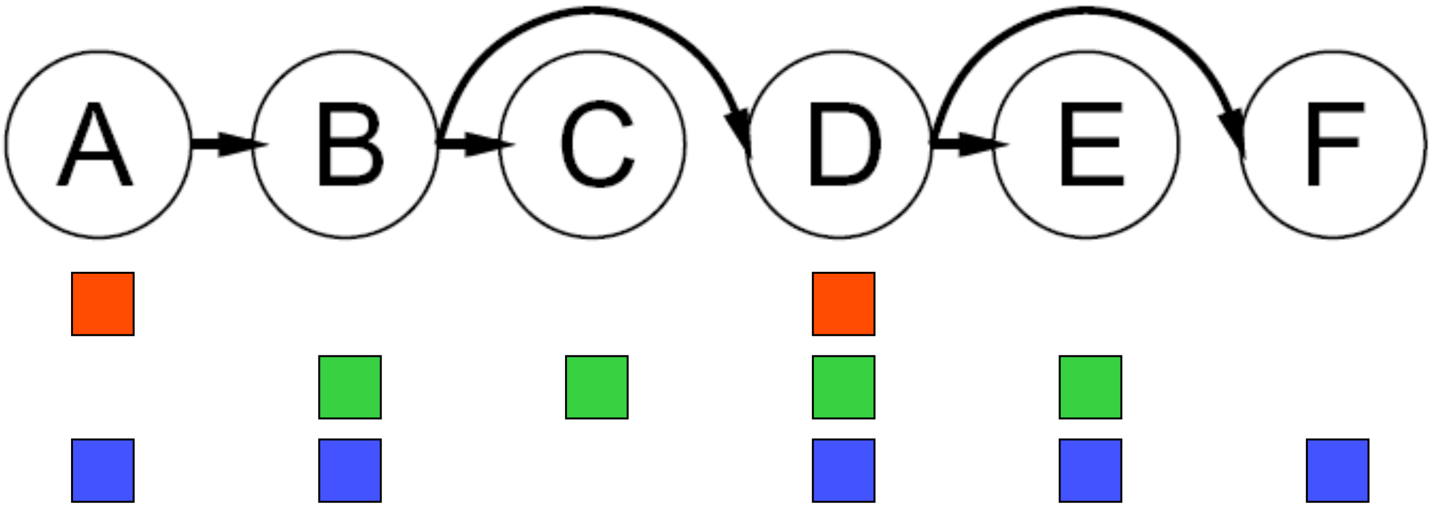
When the constraint graph is a tree

2. make the graph *directed arc consistent*
   – start w/ the tail and make each variable arc consistent wrt its parents



When the constraint graph is a tree

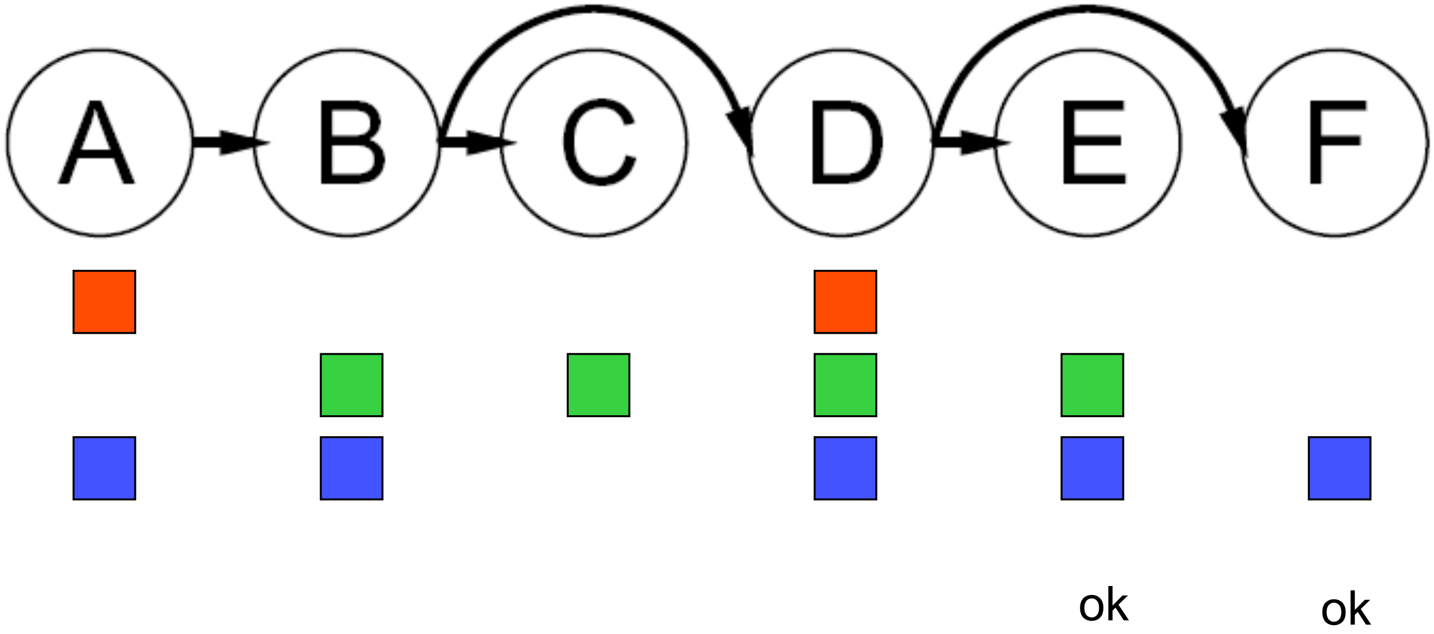
2. make the graph *directed arc consistent*
   – start w/ the tail and make each variable arc consistent wrt its parents



ok

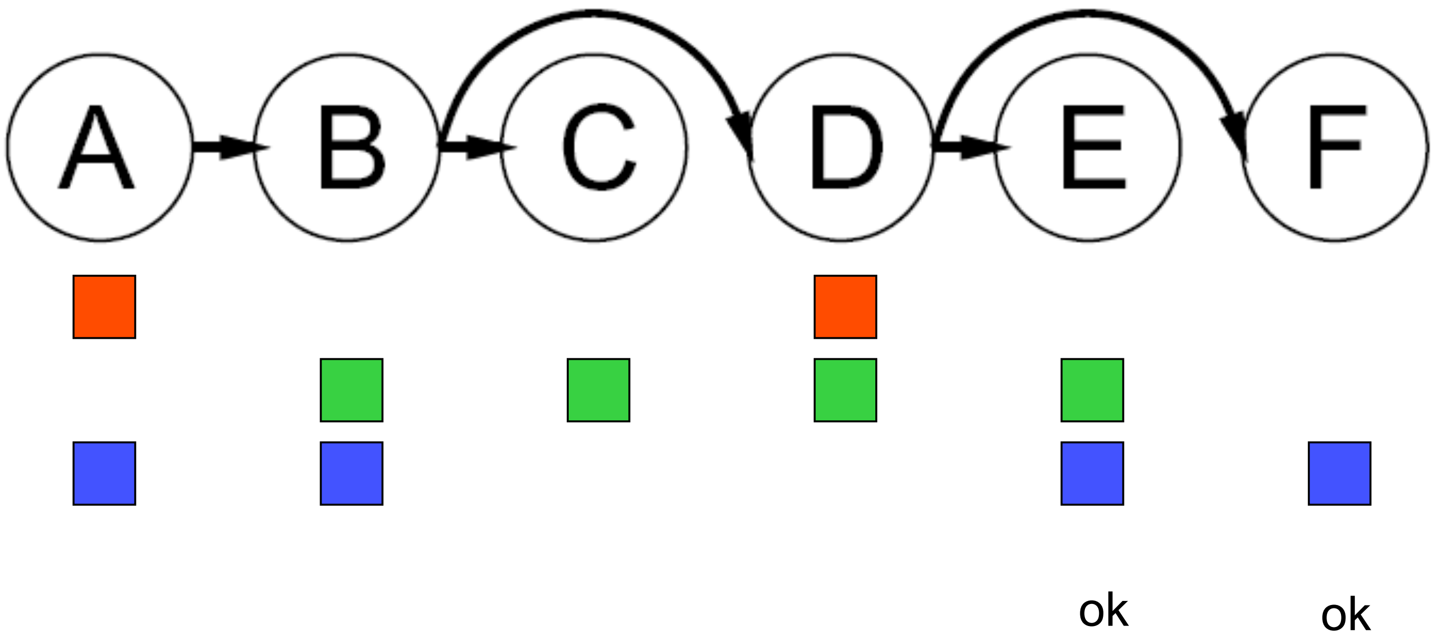
When the constraint graph is a tree

2. make the graph *directed arc consistent*
   – start w/ the tail and make each variable arc consistent wrt its parents



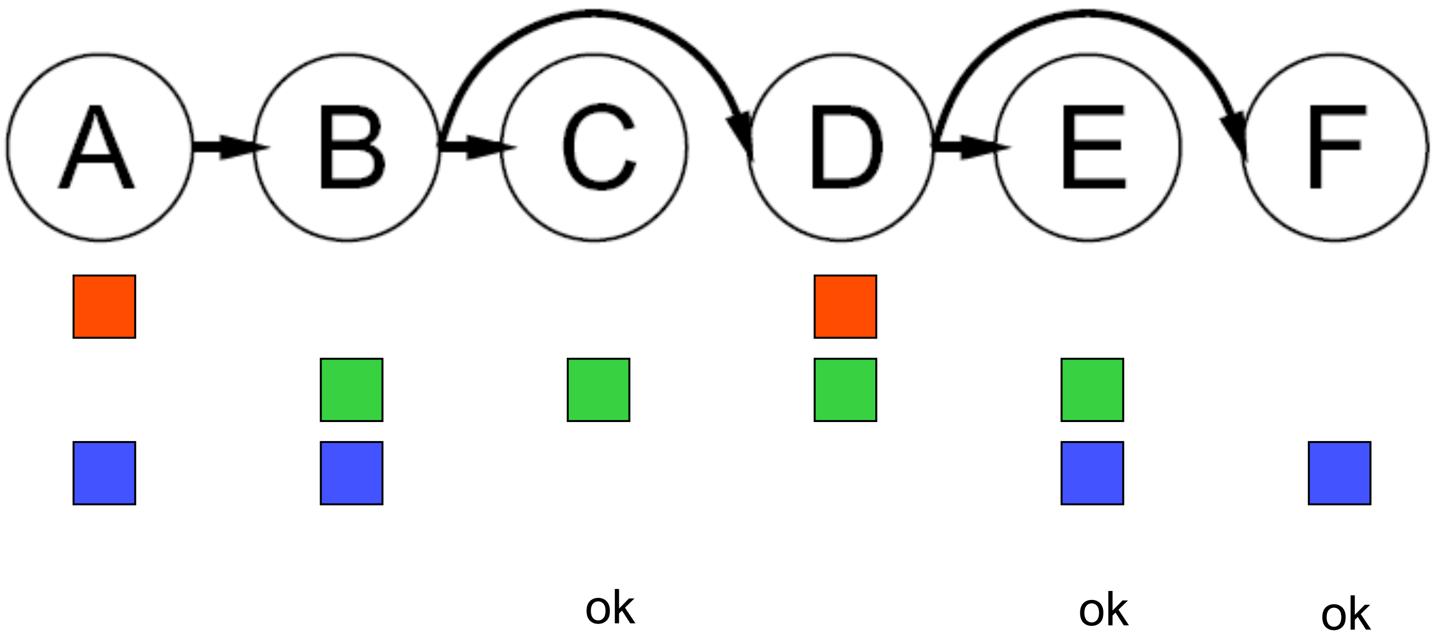
When the constraint graph is a tree

2. make the graph *directed arc consistent*
   – start w/ the tail and make each variable arc consistent wrt its parents



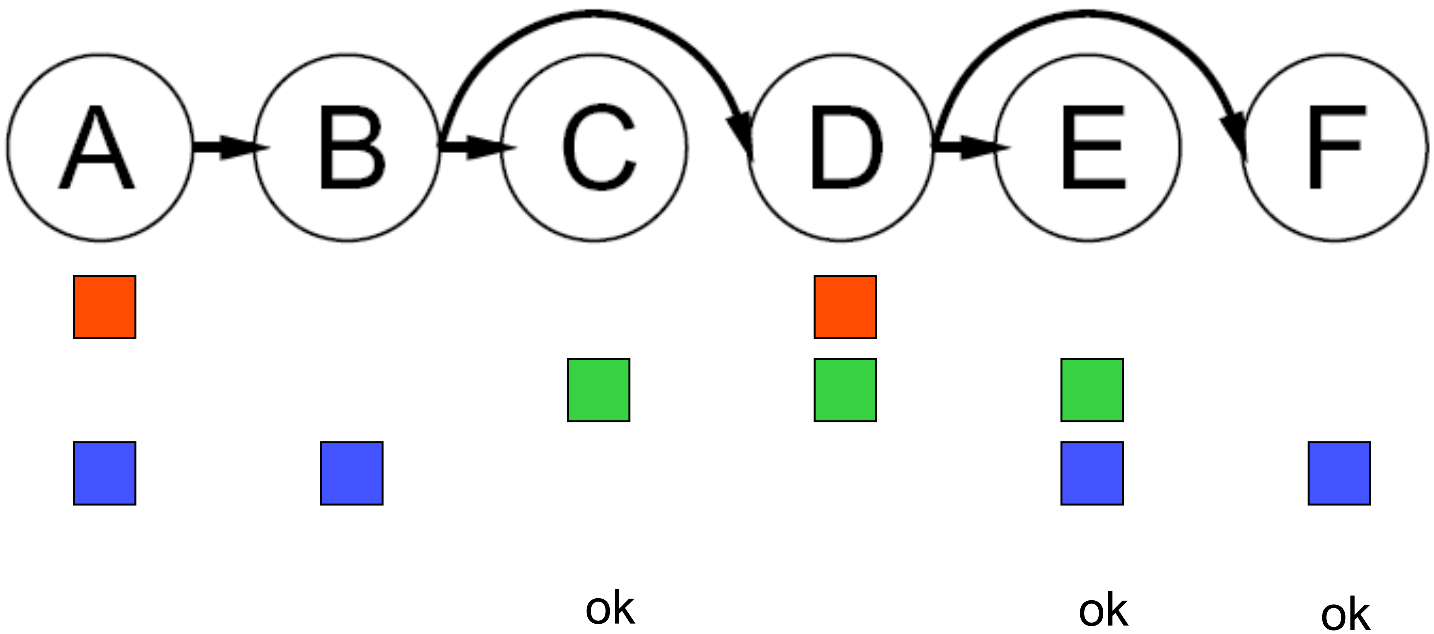
When the constraint graph is a tree

2. make the graph *directed arc consistent*
   – start w/ the tail and make each variable arc consistent wrt its parents



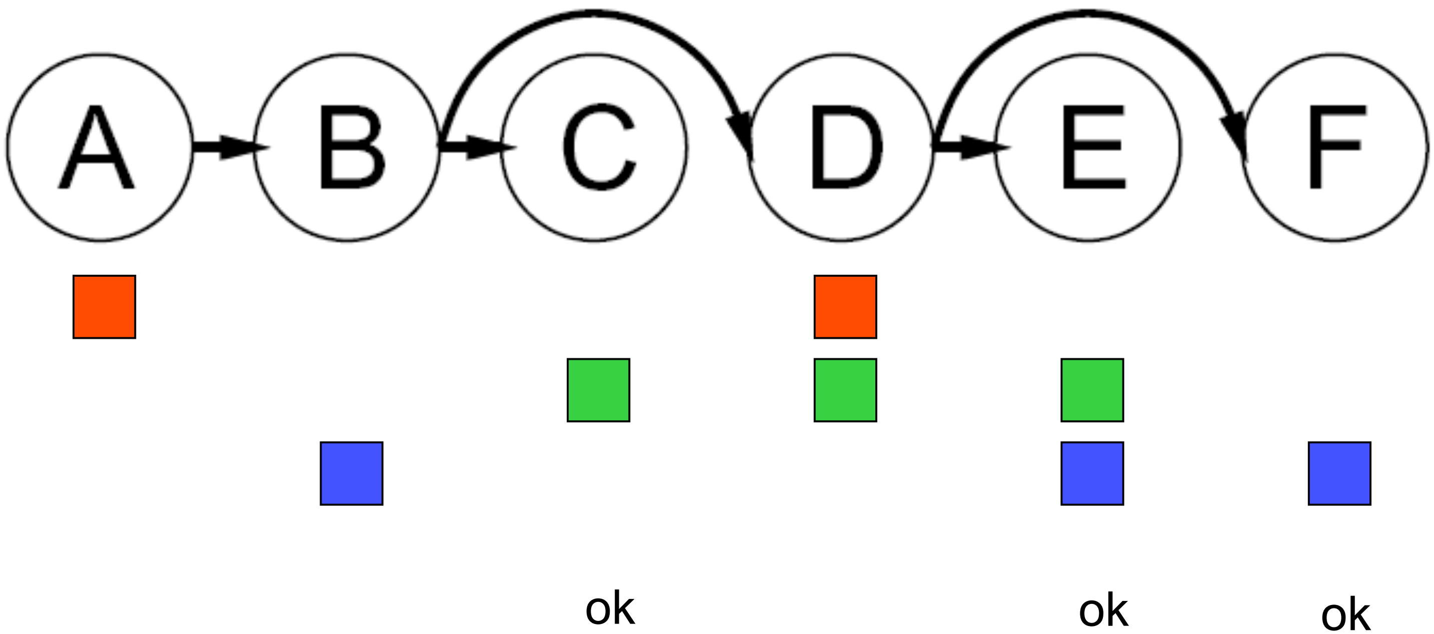
When the constraint graph is a tree

2. make the graph *directed arc consistent*
   – start w/ the tail and make each variable arc consistent wrt its parents



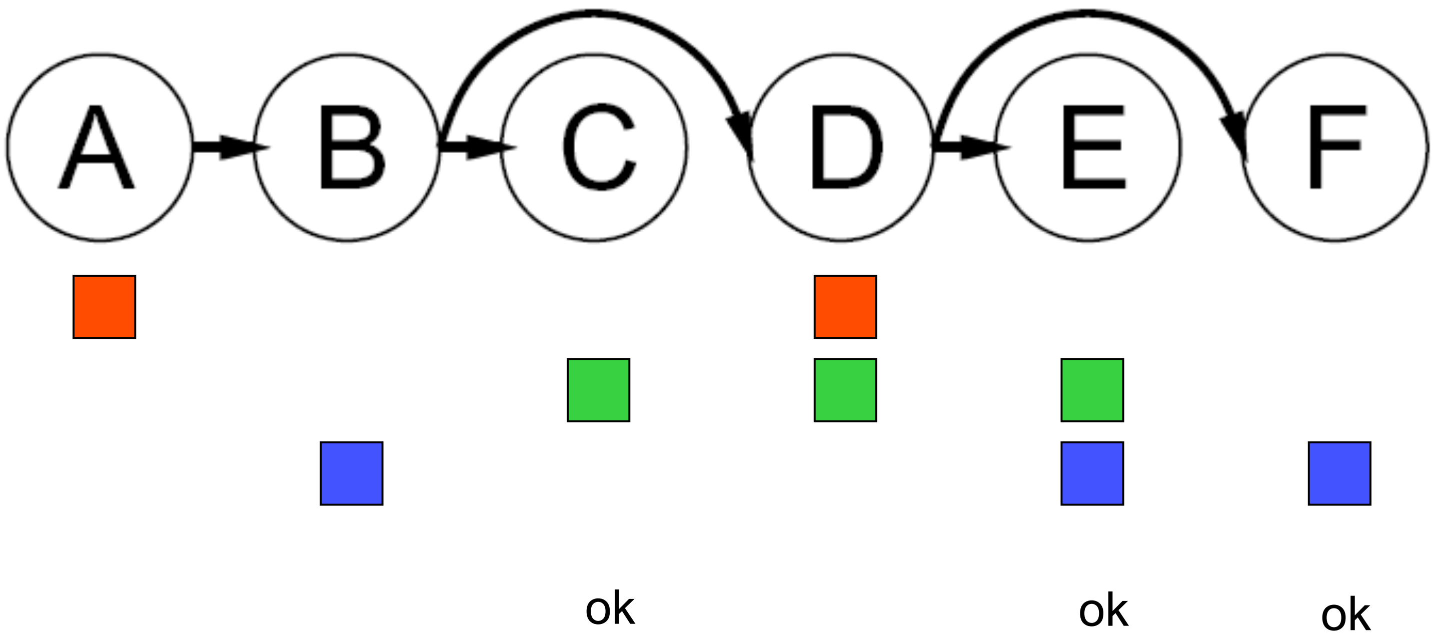
When the constraint graph is a tree

2. make the graph *directed arc consistent*
   – start w/ the tail and make each variable arc consistent wrt its parents



When the constraint graph is a tree

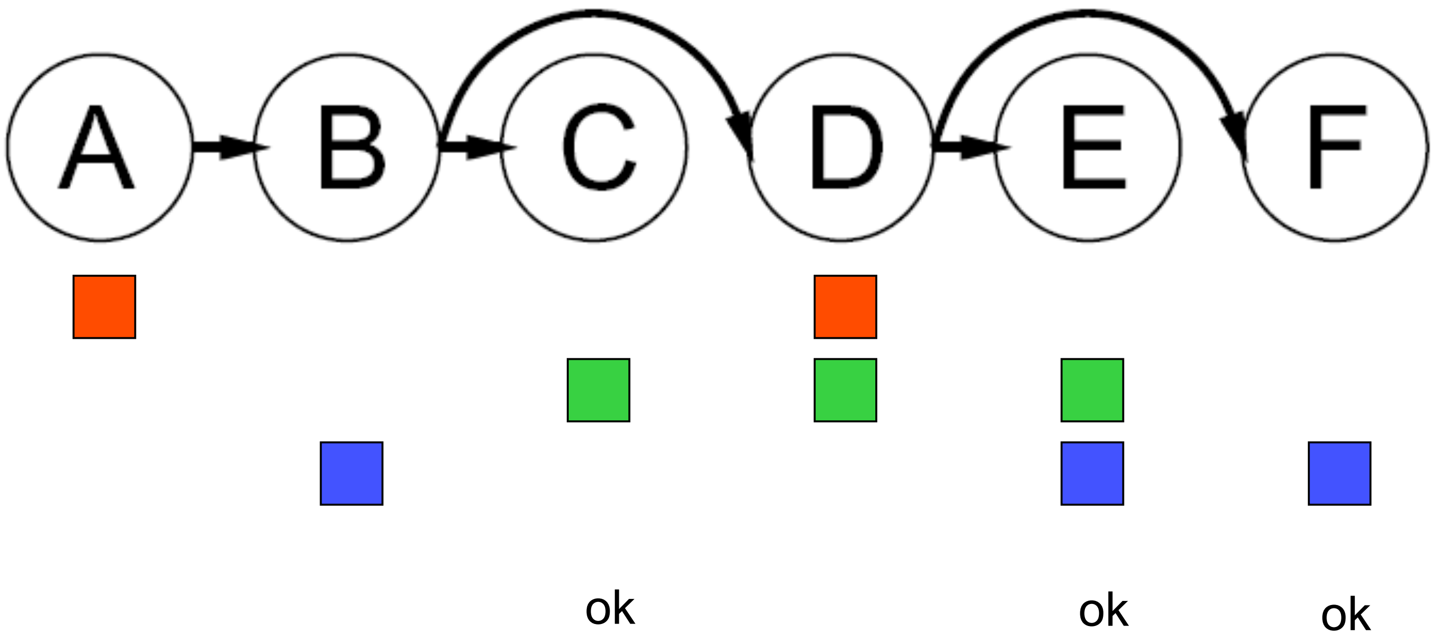
3. Now, start at the root and do backtracking
   – will backtracking ever actually backtrack?



So, what's the time complexity of this algorithm?

When the constraint graph is a tree

3. Now, start at the root and do backtracking
   – will backtracking ever actually backtrack?



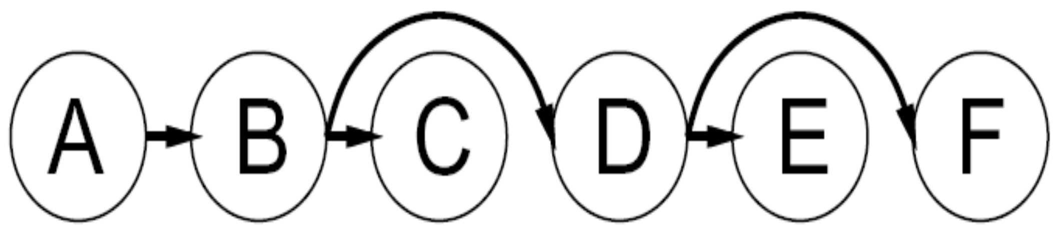
So, what's the time complexity of this algorithm?

Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\ d^2)$ time

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

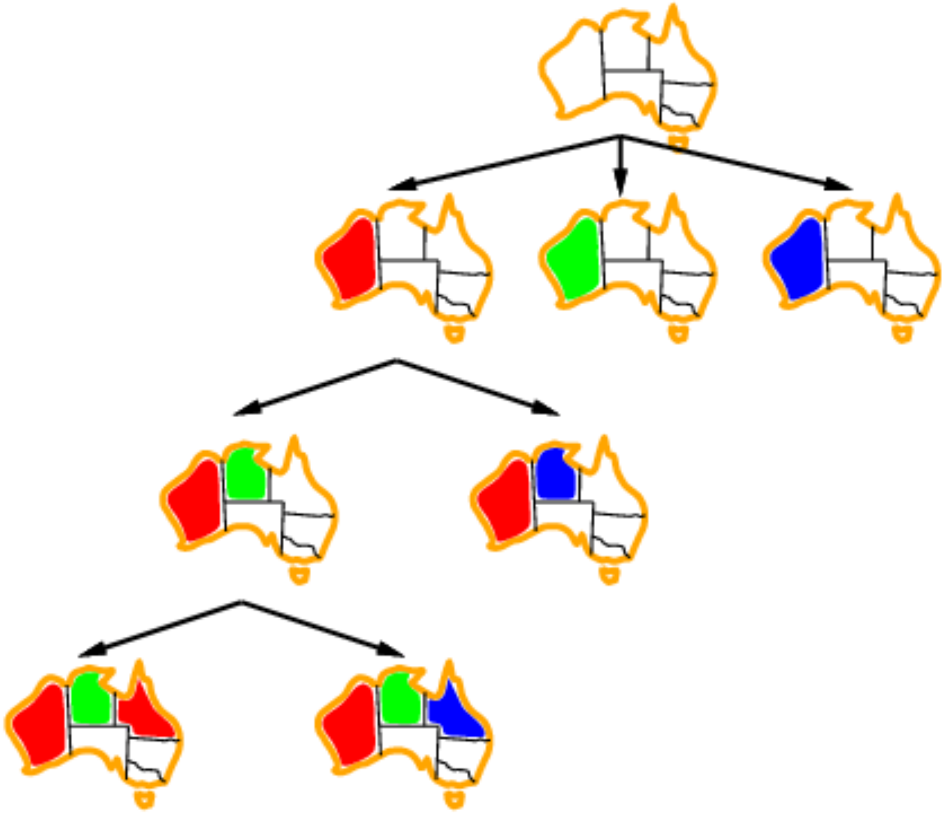
Proof: Induction on position

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

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

Using structure to reduce problem complexity

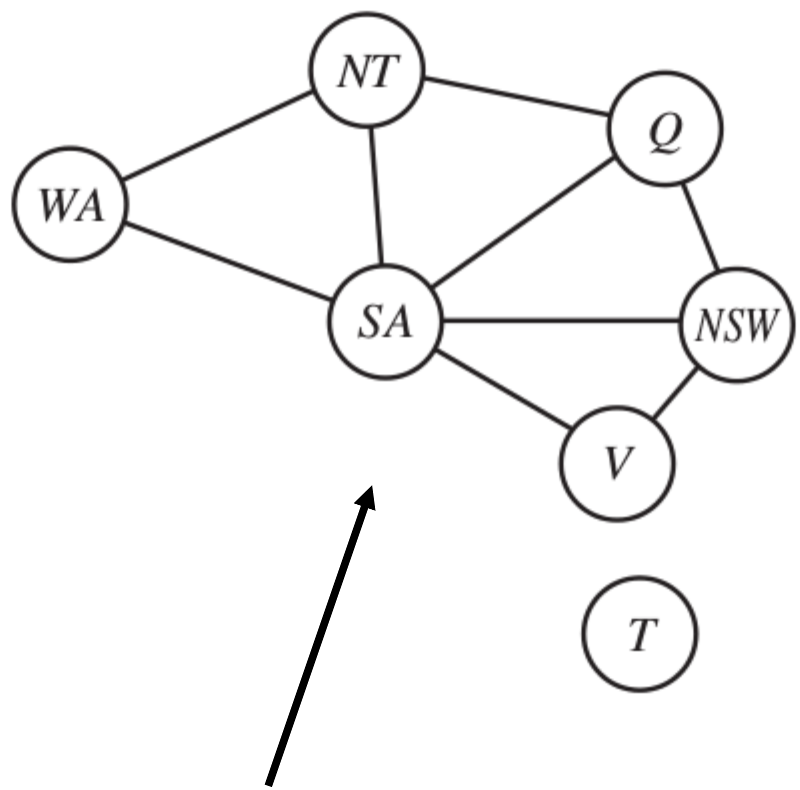
But, what if the constraint graph is not a tree?
– is there anything we can do?



But, sometimes CSPs have special structure that makes them simpler!

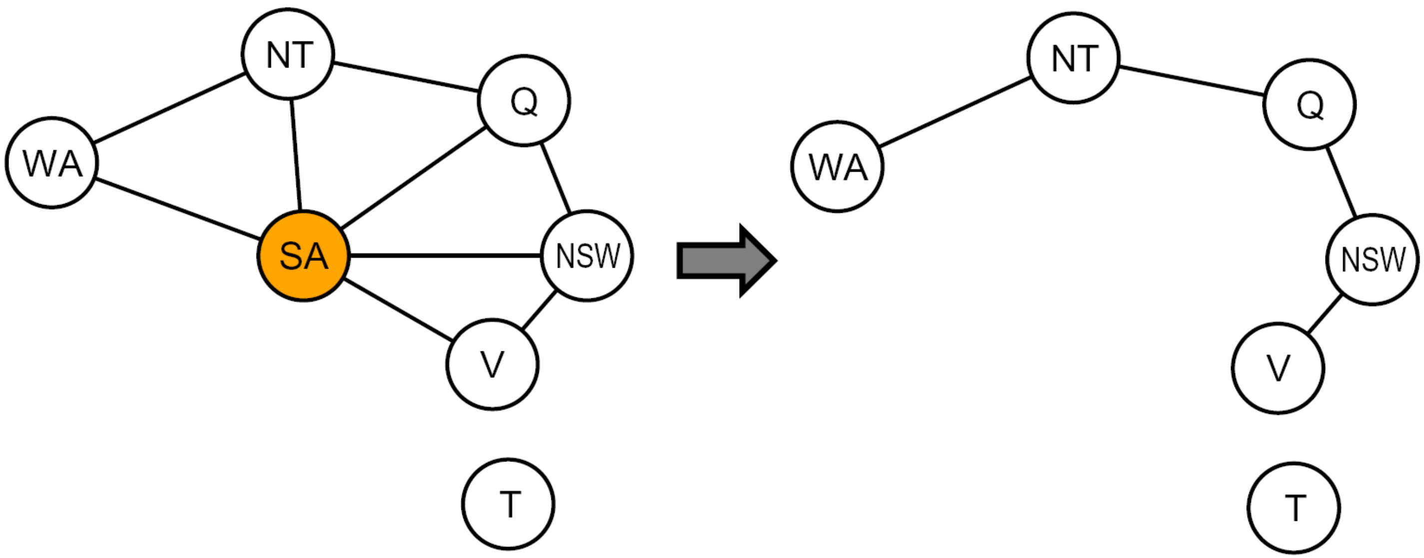
Using structure to reduce problem complexity

But, what if the constraint graph is not a tree?
– is there anything we can do?



This is not a tree...

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

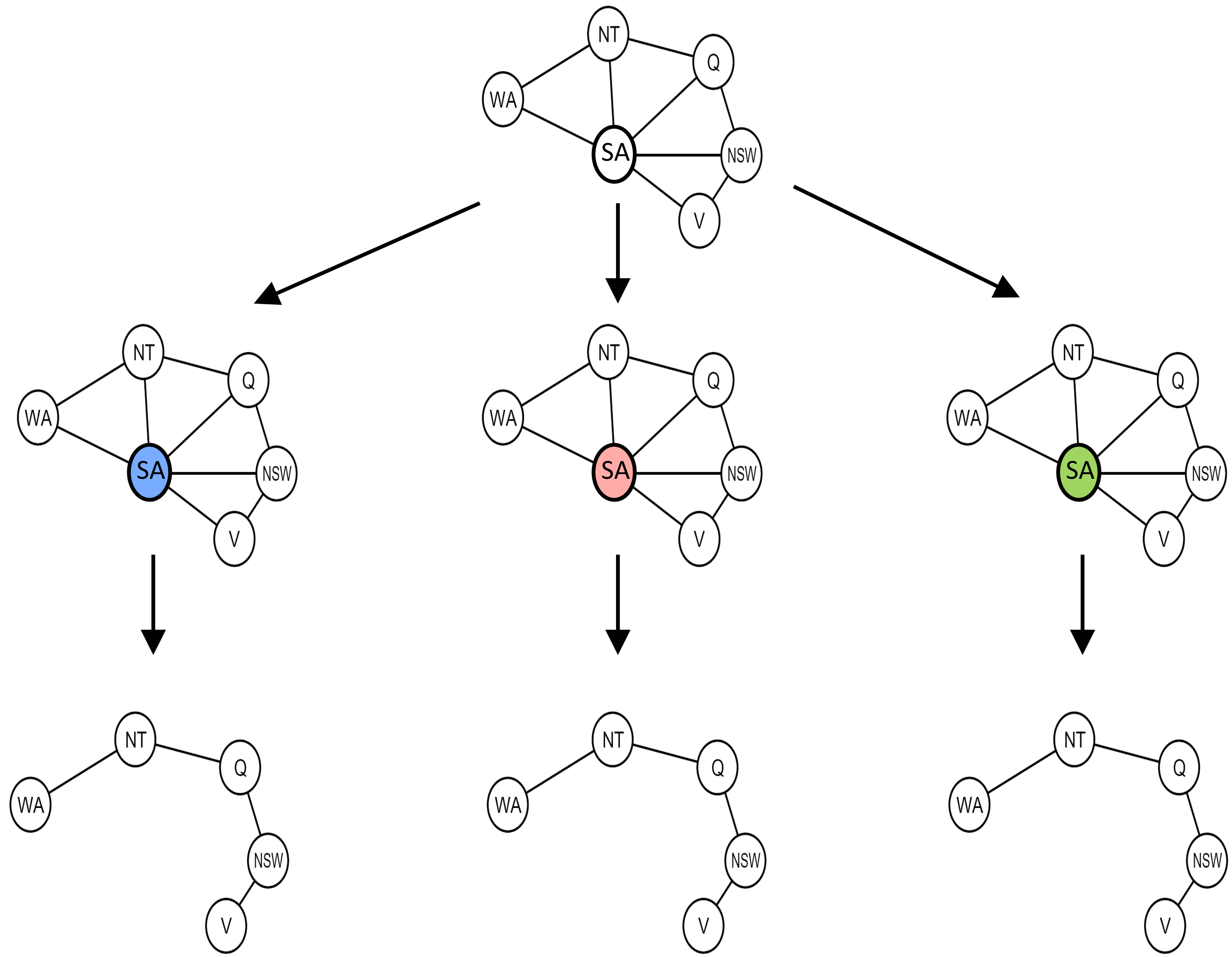
Cutset conditioning

Choose a cutset

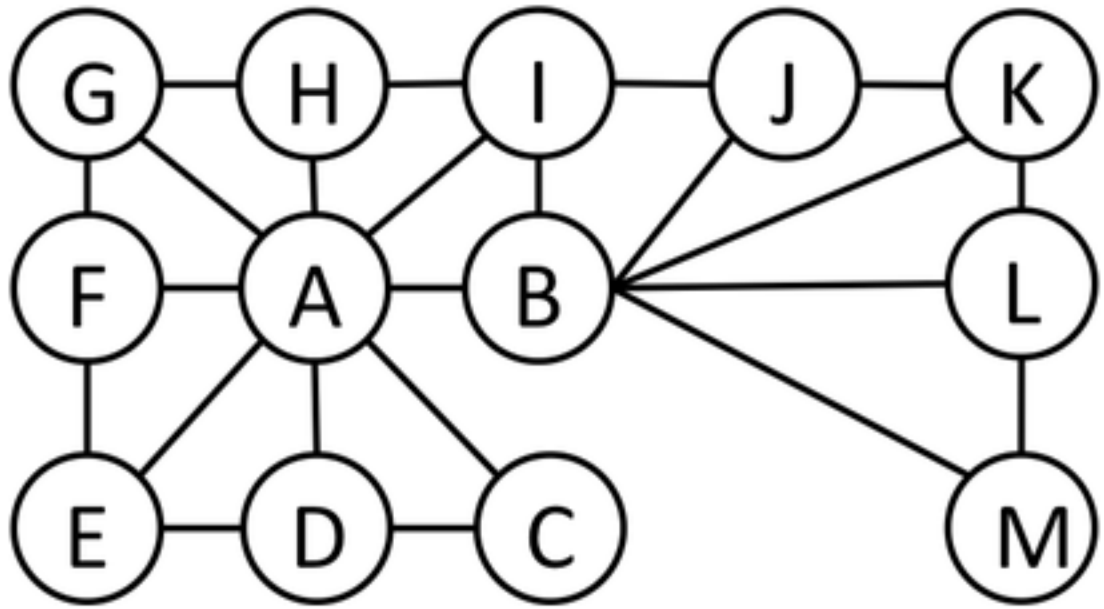
Instantiate the cutset (all possible ways)

Compute residual CSP for each assignment

Solve the residual CSPs (tree structured)



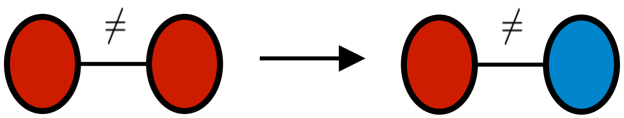
Cutset conditioning



How many variables need to be assigned to turn this graph into a tree?

Iterative algorithms for CSPs

Local search methods typically work with "complete" states, i.e., all variables assigned



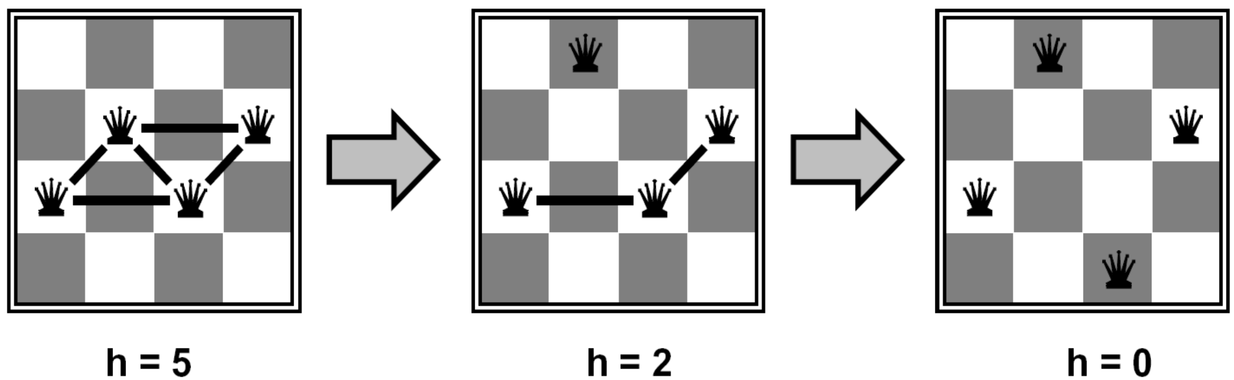
To apply to CSPs:

- 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) = 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

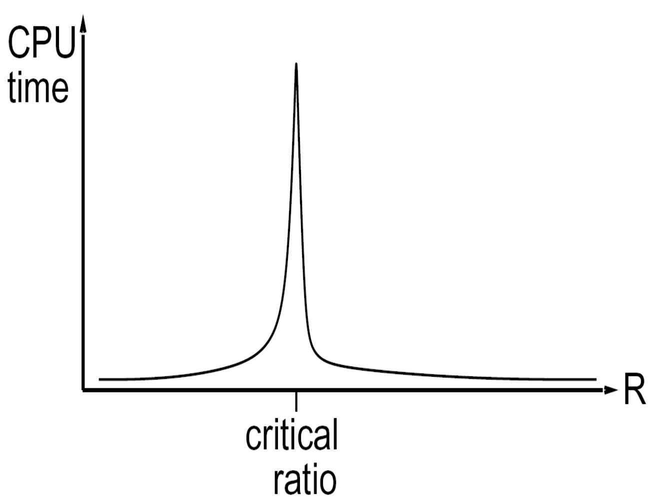
Evaluation: c(n) = number of attacks

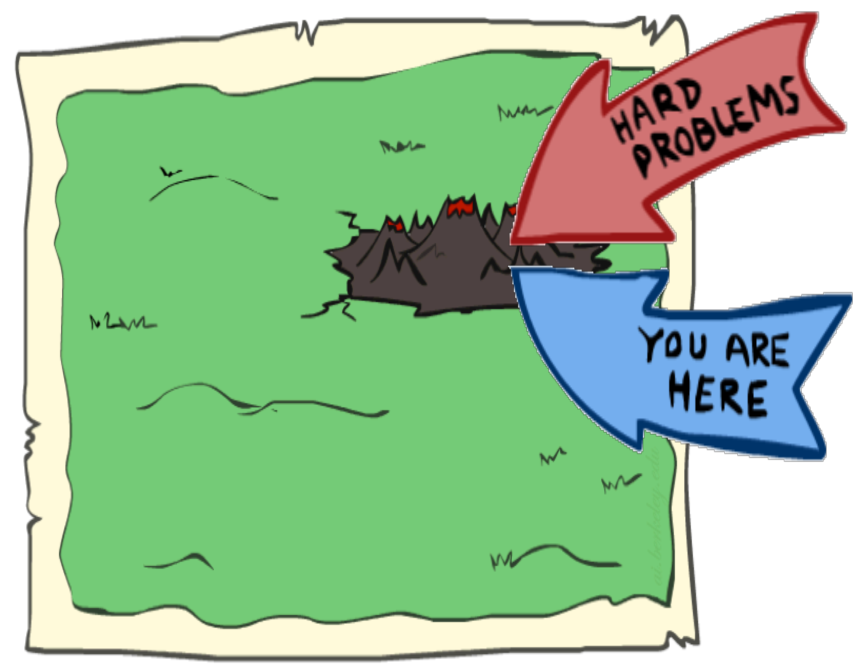
Performance of Min-Conflicts

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

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



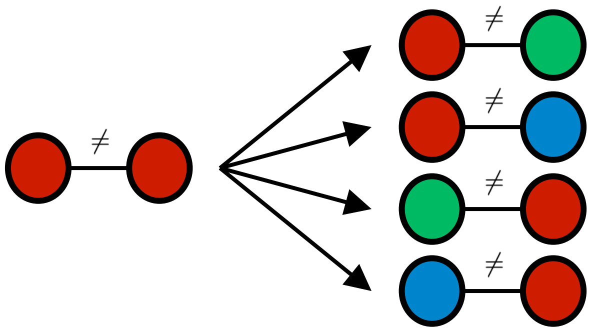


Aside: Local search more generally

Tree search keeps unexplored alternatives on the fringe (ensures completeness)

Local search: improve a single option until you can't make it better (no fringe!)

New successor function: local changes



Generally much faster and more memory efficient (but incomplete and suboptimal)

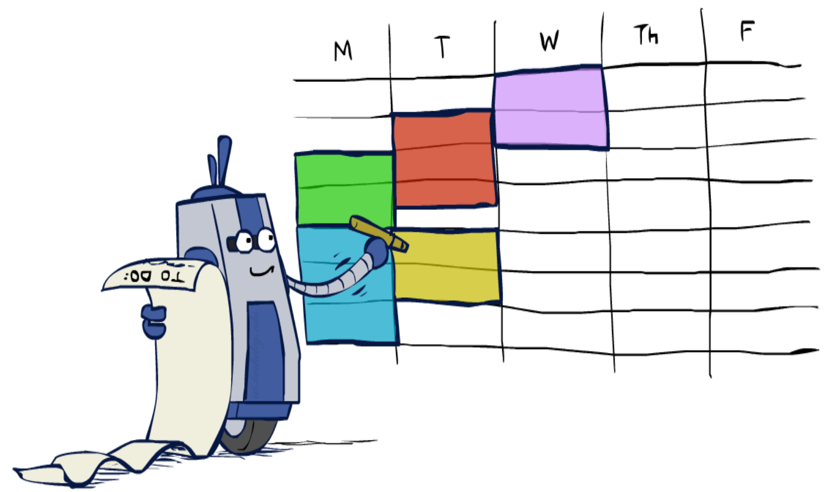
Many local search algorithms (that we won't cover): hill climbing, simulated annealing, genetic algorithms, etc.

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