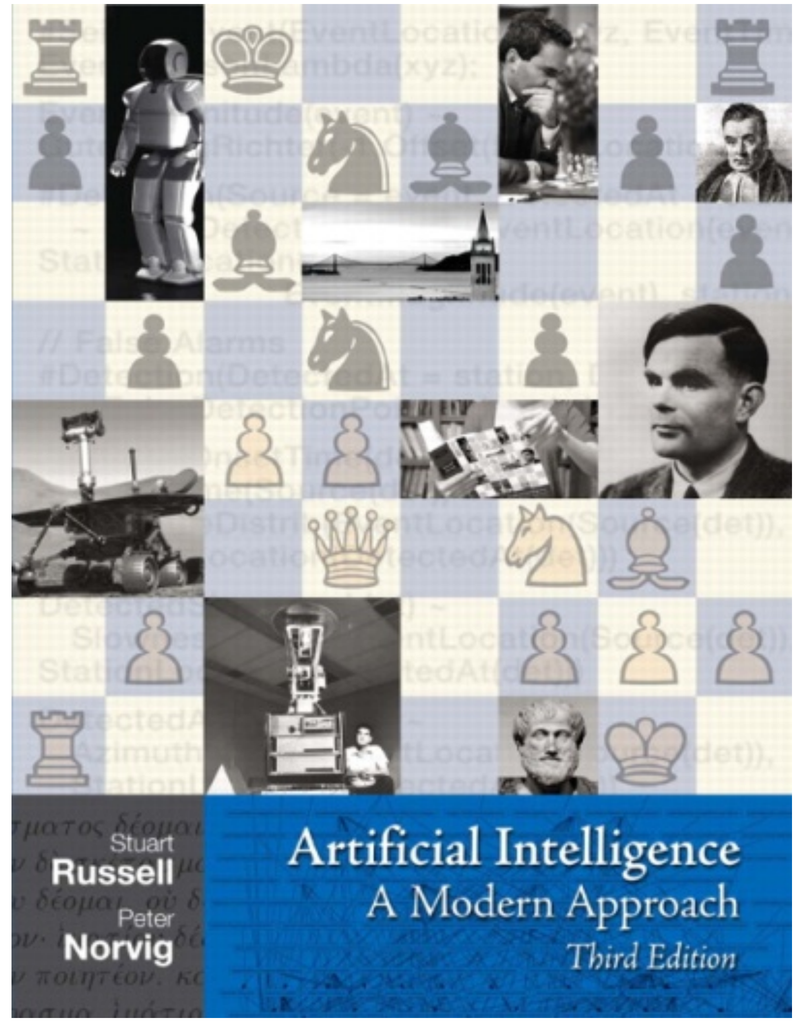
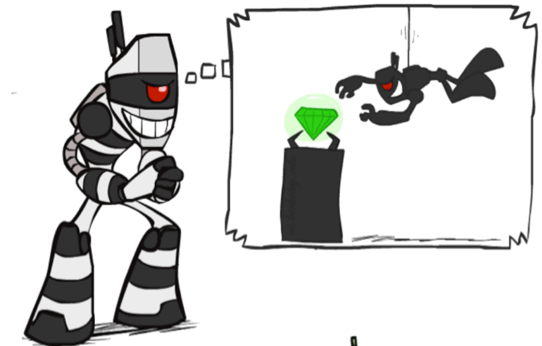
Constraint Satisfaction Problems II



AIMA: Chapter 6

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 specialized for identification problems

Slide credit: Dan Klein and Pieter Abbeel
http://ai.berkeley.edu

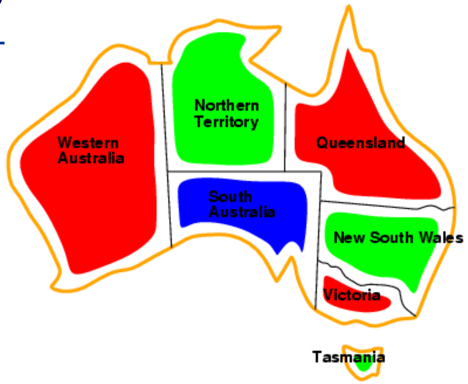
Review: Constraint Satisfaction Problems

A CSP consists of:

- *Finite set of variables* $X_1, X_2, \ldots, X_n$
- *Nonempty* ***domain*** *of possible values* for each variable $D_1, D_2, \ldots D_n$ *where* $D_i = \{v_1, \ldots, v_k\}$
- *Finite set of constraints* $C_1, C_2, \ldots, C_m$
  - —Each *constraint* C_i limits the values that variables can take.
  - —A *state* is defined as an *assignment* of values to some or all variables.

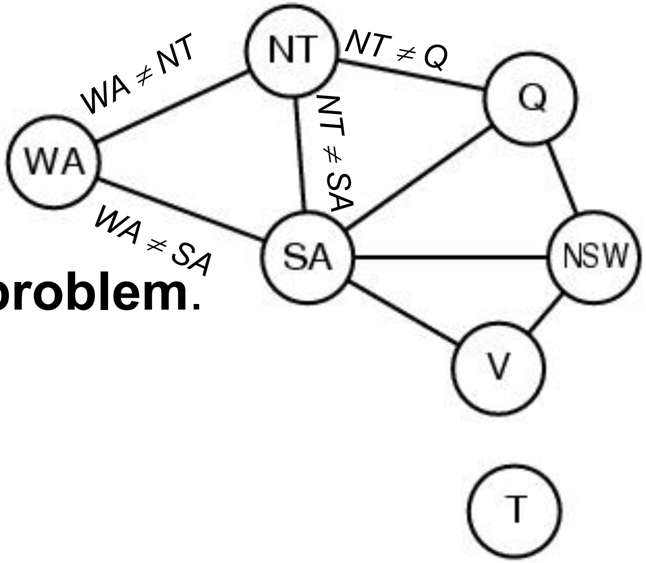
- **A *solution* to a CSP is a *complete, consistent* assignment, where**
  - A *consistent* assignment does not violate the constraints.
  - An assignment is *complete* when every variable is assigned a value.

Review: CSP Representations



- ***Constraint graph:***
  - *nodes* are variables
  - *edges* are (binary) constraints

- ***Standard representation pattern:***
  - variables with values



- ***Constraint graph* simplifies search.**
  - **e.g. Tasmania is an independent subproblem.**

- ***This problem: A binary CSP:***
  - each constraint relates two variables

Idea 1: CSP as a search problem

- **A CSP can easily be expressed as a search problem**
  - *Initial State:* the empty assignment {}.
  - *Successor function:* Assign value to any unassigned variable *provided that there is not a constraint conflict*.
  - *Goal test:* the current assignment is complete.
  - *Path cost:* a constant cost for every step.

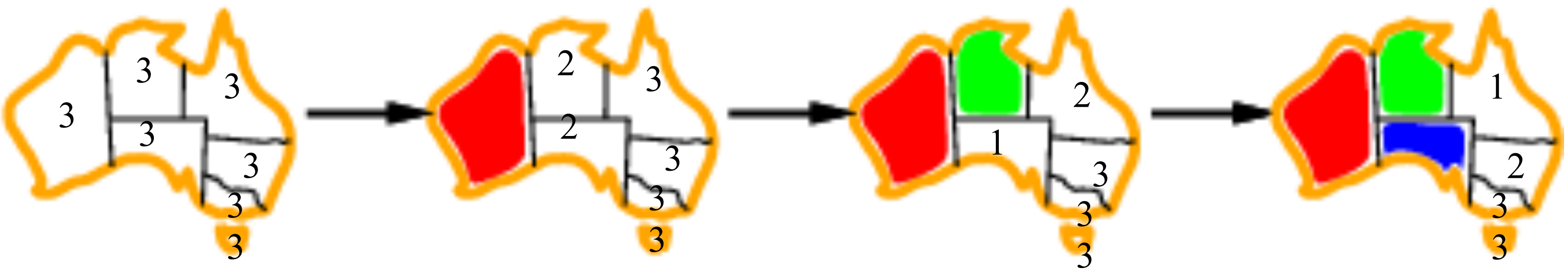
- **Solution is always found at depth *n*, for *n* variables**
  - Hence Depth First Search can be used

Idea 2: Improving backtracking efficiency

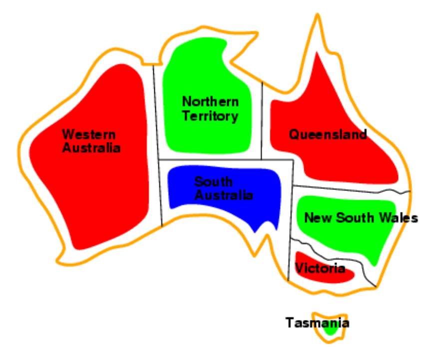
- ***General-purpose*** **methods &** ***general-purpose*** **heuristics can give huge gains in speed,** ***on average***
- **Heuristics:**
  - Q: Which variable should be assigned next?
    1. **Most constrain*ed* variable**
    2. **(if ties:) Most constrain*ing* variable**
  - Q: In what order should that variable's values be tried?
    3. **Least constraining *value***
  - Q: Can we detect inevitable failure early?
    4. **Forward checking**

Heuristic 1: Most constrain*ed* variable

- **Choose a variable with the *fewest legal values***

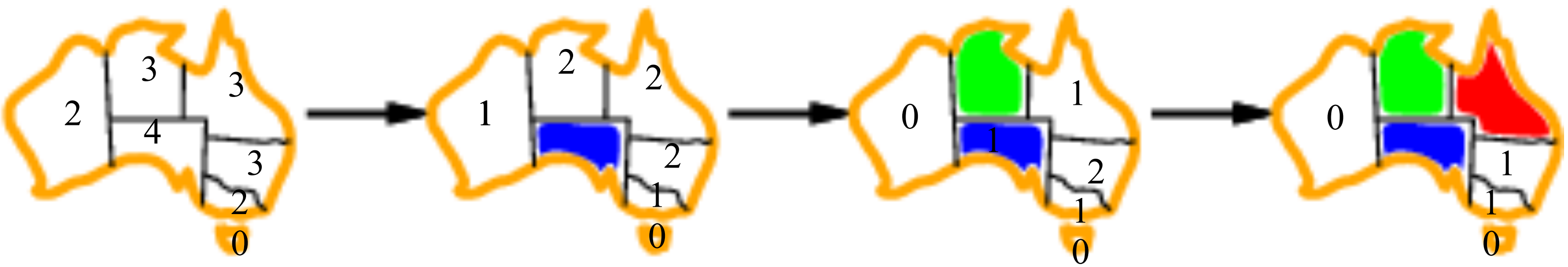


- **a.k.a. *minimum remaining values (MRV)* heuristic**

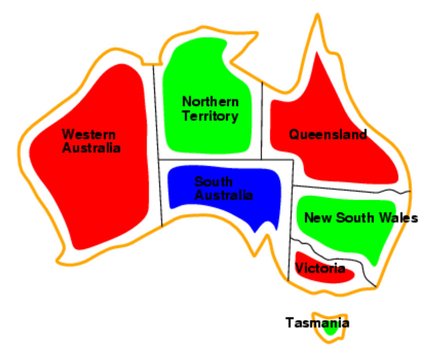


Heuristic 2: Most constrain*ing* variable

- **Tie-breaker among most constrained variables**
- **Choose the variable with the *most constraints on remaining variables***

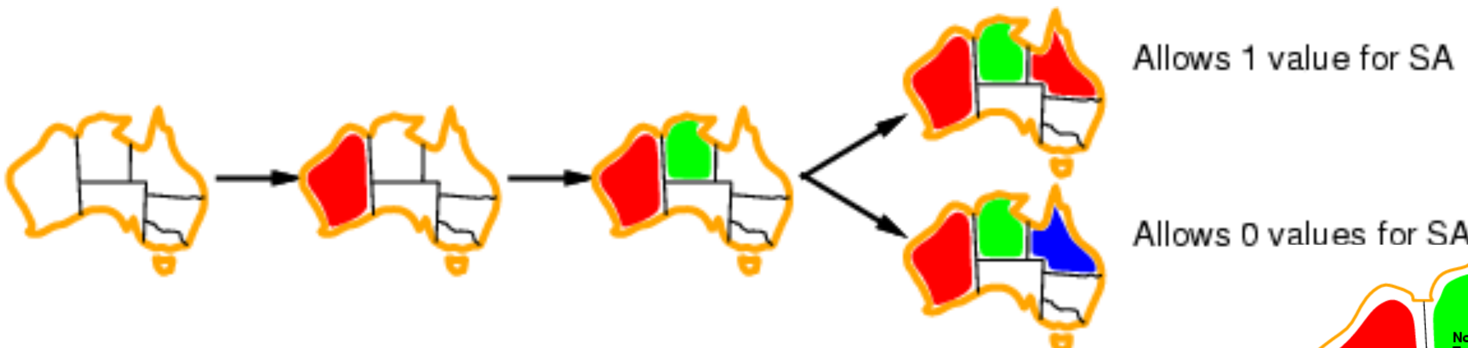


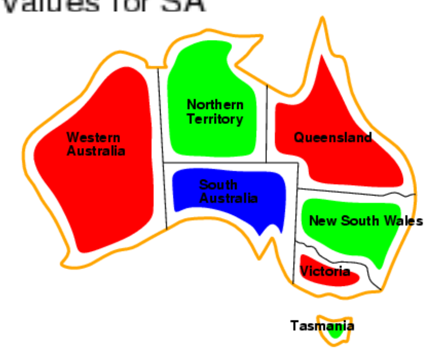
(These two heuristics each lead to immediate solution of our example problem)



Heuristic 3: Least constraining *value*

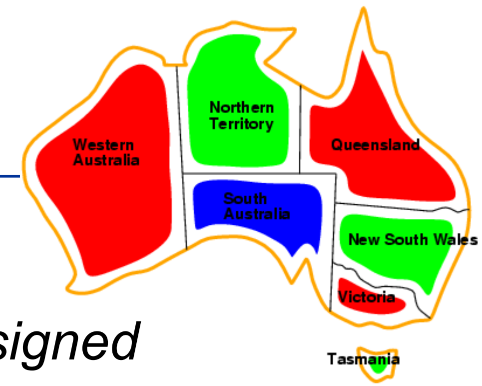
- **Given a variable, *choose the least constraining value:***
  - the one that rules out the fewest values in the remaining variables

- 



Note: demonstrated here independent of the other heuristics

Heuristic 4: Forward checking

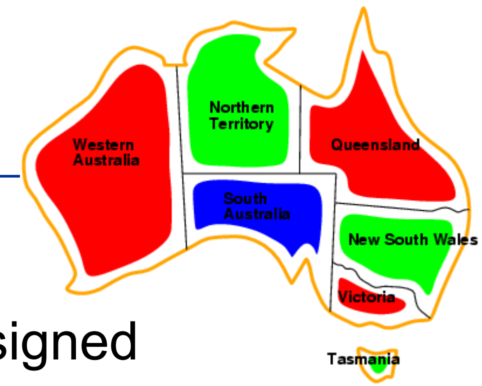


- **Idea**:
  - Keep track of *remaining* legal values for *unassigned* variables
  - Terminate search when any unassigned variable has no remaining legal values

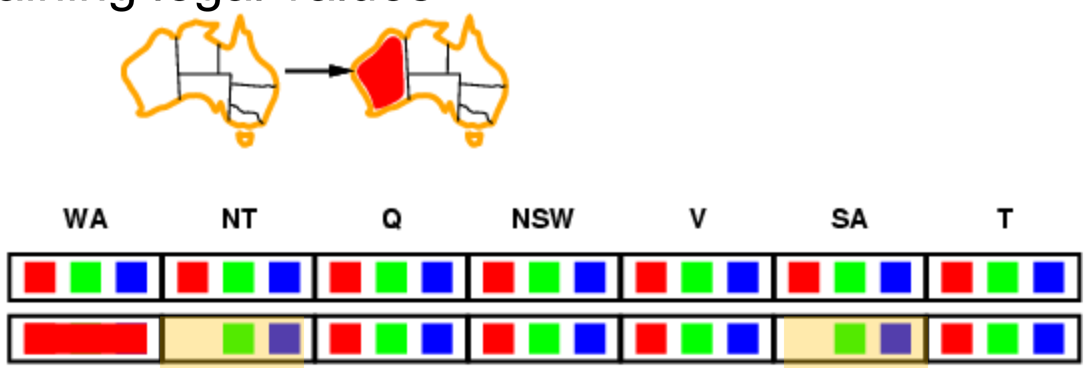
New data structure

(A first step towards Arc Consistency & AC-3)

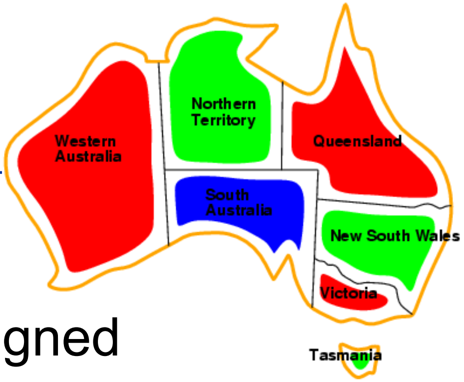
Forward checking



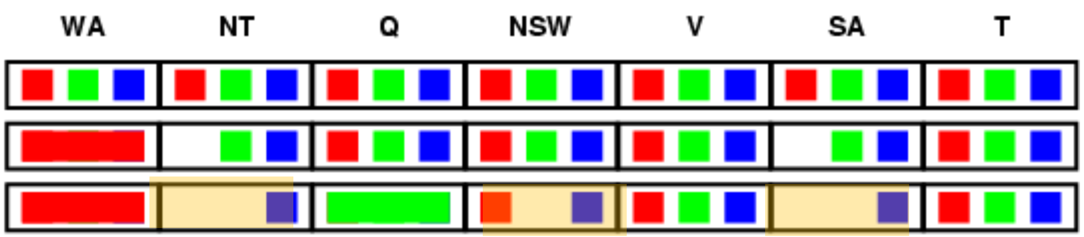
- **Idea**:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any unassigned variable has no remaining legal values



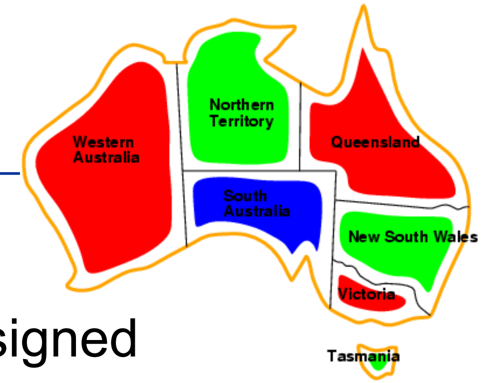
Forward checking



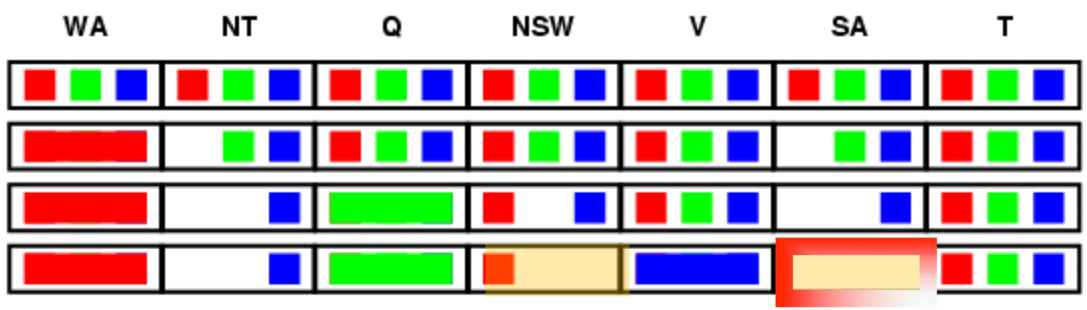
- **Idea**:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any unassigned variable has no remaining legal values



Forward checking

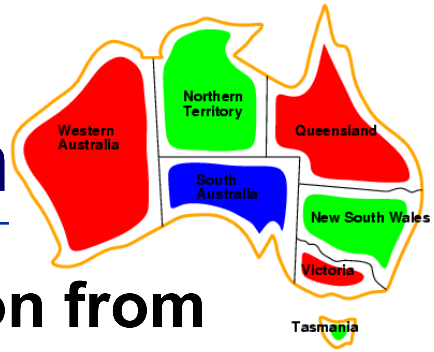


- **Idea**:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any unassigned variable has no remaining legal values

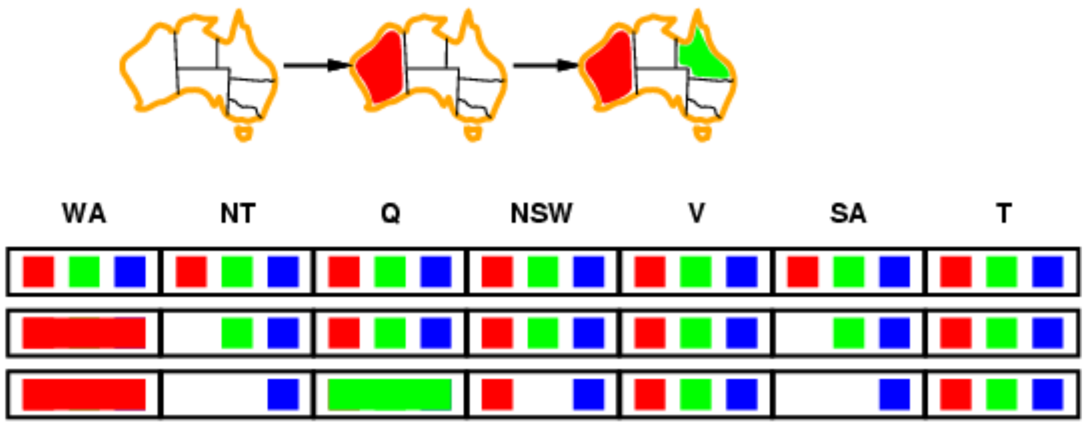


Terminate! No possible value for SA

Towards 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 goes beyond forward checking & repeatedly enforces constraints locally

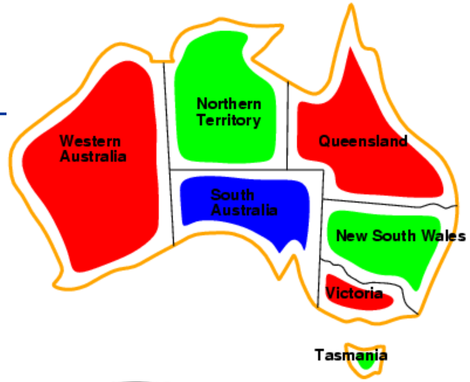
Arc Consistency, Constraint Propagation & AC-3

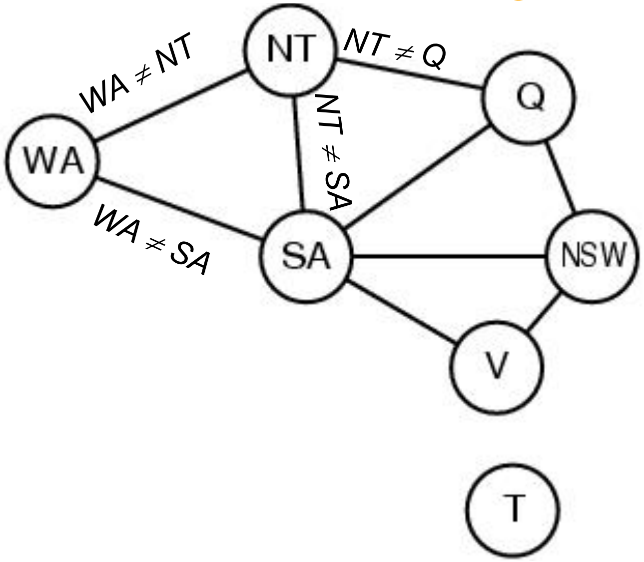
Idea 3 *(big* idea)*: Inference* in CSPs

- **CSP solvers combine search *and inference***
  - Search
    - —assigning a value to a variable
  - *Constraint propagation (inference)*
    - —Eliminates possible values for a variable if the value would violate local consistency
  - *Can do inference first, or intertwine it with search*
    - —You'll investigate this in the Sudoku homework
- **Local consistency**
  - Node consistency: satisfies unary constraints
    - —This is trivial!
  - *Arc consistency*: satisfies binary constraints
    - —(X_i is arc-consistent w.r.t. X_j if for every value v in D_i, there is some value w in D_j that satisfies the binary constraint on the arc between X_i *and* X_j)

Review: CSP Representations

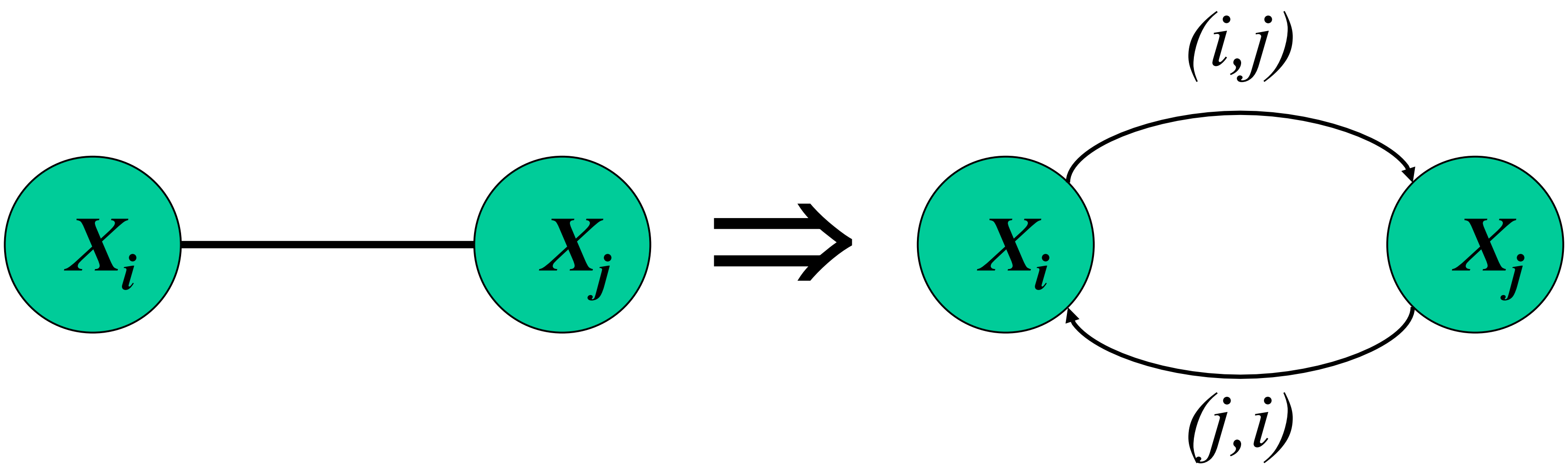
- ***Constraint graph:***
  - *nodes* are variables
  - *edges are constraints*





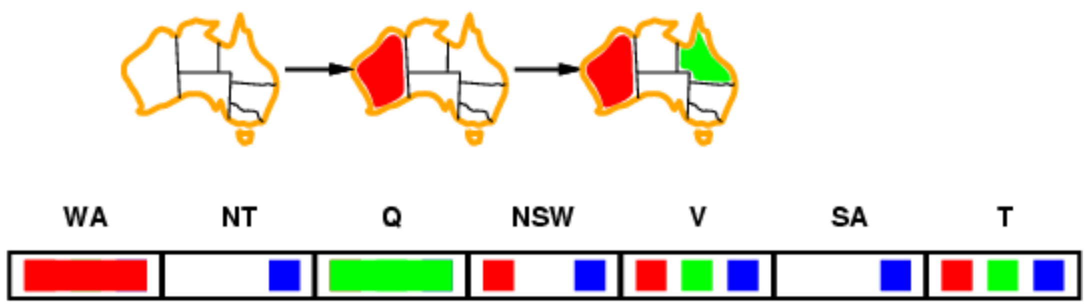
Edges to Arcs: From Constraint Graph to Directed Graph

- **Given a pair of nodes X_i and X_j connected by a constraint *edge*, we represent this not by a single undirected edge, but a *pair of directed arcs*.**
  - **For a connected pair of nodes X_i and X_j, there are *two* arcs that connect them: *(i,j)* and *(j,i)*.**



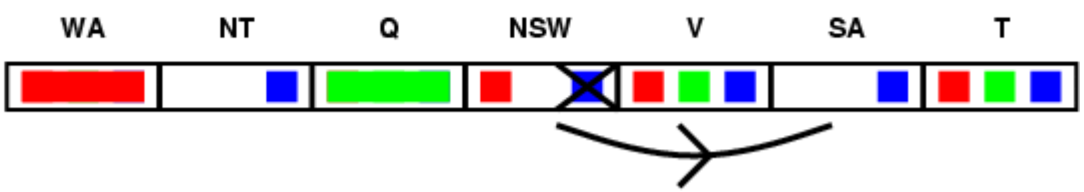
Arc consistency

- **Simplest form of propagation makes each arc consistent**
- ***X* → *Y* is consistent iff**
  for every value *x* of *X* there is some allowed *y*



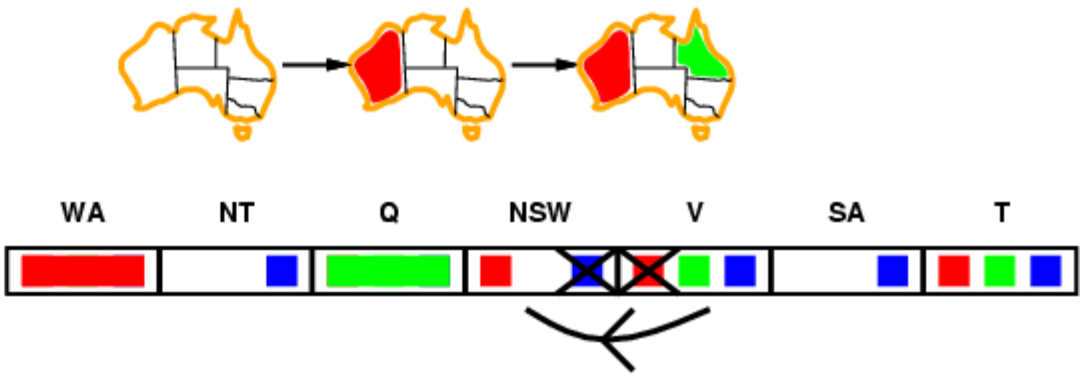
Arc consistency

- **Simplest form of propagation makes each arc consistent**
- ***X* → *Y* is consistent iff**
  for every value *x* of *X* there is some allowed *y*



Arc consistency

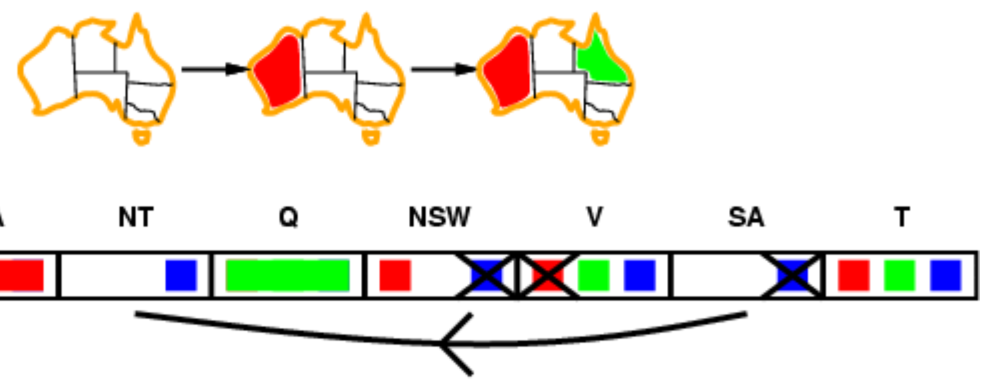
- **Simplest form of propagation makes each arc consistent**
- ***X* → *Y* is consistent iff**
  for every value *x* of *X* there is some allowed *y*



- **If *X* loses a value, recheck neighbors of *X***

Arc consistency

- **Simplest form of propagation makes each arc consistent**
- ***X* → *Y* is consistent iff**
  for every value *x* of *X* there is some allowed *y*



- **If *X* loses a value, we need to recheck neighbors of *X***
- **Detects failure earlier than forward checking**
- **Can be run as a preprocessor or after each assignment**

Arc Consistency

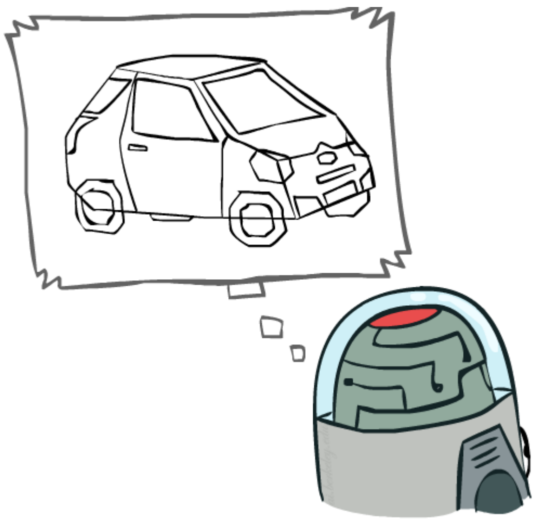
An arc *(i,j) is* arc consistent if and only if every value v on X_i is consistent with some label on X_j.

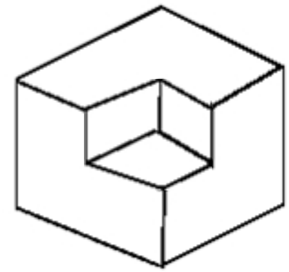
To make an arc *(i,j)* arc consistent,
for each value v on X_i ,
if there is no label on X_j consistent with v
then remove v from X_i

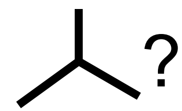
- **Given *d* values, checking arc (i,j) takes *O(d^2)* time worst case**

Example: The Waltz Algorithm

- **The Waltz algorithm is for interpreting line drawings of solid polyhedra as 3D objects**
- **An early example of an AI computation posed as a CSP**



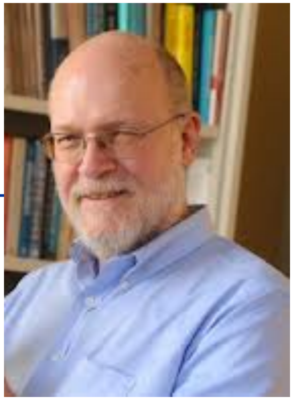




- Approach:
  - Each intersection is a variable
  - Adjacent intersections impose constraints on each other
  - Solutions are physically realizable 3D interpretations

Slide credit: Dan Klein and Pieter Abbeel
http://ai.berkeley.edu

Replacing Search: Constraint Propagation Invented…



Dave Waltz's insight:

- By ***iterating*** **over the graph, the arc-consistency** *constraints* **can be** *propagated* **along arcs of the graph.**

- ***Search*****: Use constraints to** ***add*** **labels to find** ***one solution***

- ***Constraint Propagation*****: Use constraints to** ***eliminate labels*** **to simultaneously find** ***all solutions***

The Waltz/Mackworth Constraint Propagation Algorithm

1. **Assign *every* node in the constraint graph a set of *all* possible values**
2. **Repeat until there is no change in the set of values associated with any node:**
    3. For each node i:
        4. For each neighboring node j in the picture:
            5. Remove any value from i which is not arc consistent with j.

Inefficiencies: Towards AC-3

1. **At each iteration, we only need to examine those** X_i *where at least one neighbor of* X_i *has lost a value* **in the previous iteration.**
2. **If** X_i **loses a value only because of arc inconsistencies with** X_j**, we** *don't need to check* X_j **on the next iteration.**
3. **Removing a value on** X_i **can only make** X_j **arc-inconsistent with respect to** X_i **itself. Thus, we only need to check that** *(j,i)* **is still arc-consistent.**

These insights lead a much better algorithm...

AC-3

function **AC-3(*csp*)** return **the CSP, possibly with reduced domains**

inputs**:** ***csp*, a binary csp with variables** $\{X_1, X_2, \ldots, X_n\}$

local variables: ***queue,*** **a queue of arcs initially the arcs in *csp***

while **queue is not empty** do

(**X_i, X_j**) ← **REMOVE-FIRST(*queue*)**

if **REMOVE-INCONSISTENT-VALUES(X_i, X_j)** then

for each **X_k** in **NEIGHBORS[X_i] – {X_j}** do

add (**X_k, X_i**) **to queue**

function **REMOVE-INCONSISTENT-VALUES(X_i, X_j)** return ***true*** **iff we remove a value**

removed ← ***false***

for each ***x*** in **DOMAIN[X_i]** do

if **no value *y* in DOMAIN[X_j] allows (x,y) to satisfy the constraints between X_i and X_j**

then delete **x from DOMAIN[X_i]**; ***removed*** ← ***true***

return ***removed***

AC-3: Worst Case Complexity Analysis

- **All nodes can be connected to *every* other node,**
  - so each of ***n*** nodes must be compared against ***n-1*** other nodes,
  - so total # of arcs is ***2*n*(n-1), i.e. O(n^2)***
- **If there are *d* values, checking arc (i,j) takes *O(d^2)* time**
- **Each arc (i,j) can only be inserted into the queue *d* times**
- **Worst case complexity: *O(n^2d^3)***

(For *planar* constraint graphs, the number of arcs can only be *linear in N and* the time complexity is only *O(nd^3))*

Local search for CSPs

- **Hill-climbing, simulated annealing typically work with "complete" states, i.e., all variables assigned**

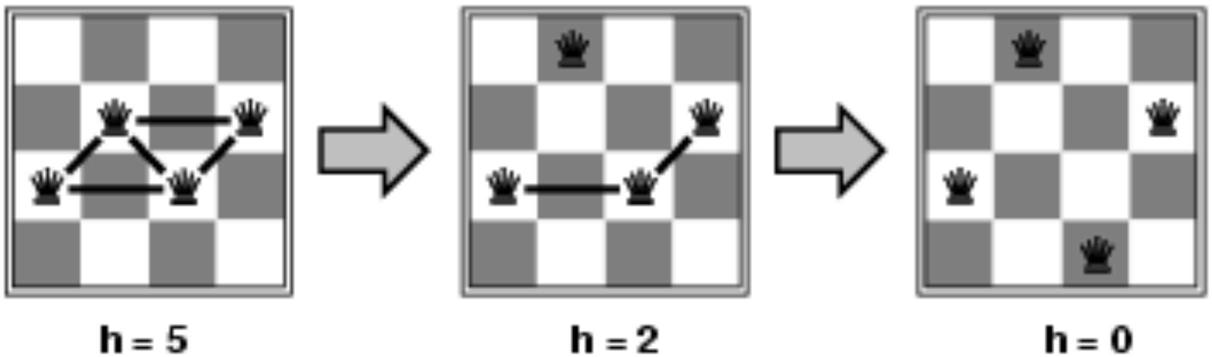
- **To apply to CSPs:**
  - allow states with unsatisfied constraints
  - operators reassign variable values

- **Variable selection: randomly select any conflicted variable**

- **Value selection by min-conflicts heuristic:**
  - choose value that violates the fewest constraints
  - i.e., hill-climb with $h(n)$ = total number of violated constraints

Example: n-queens

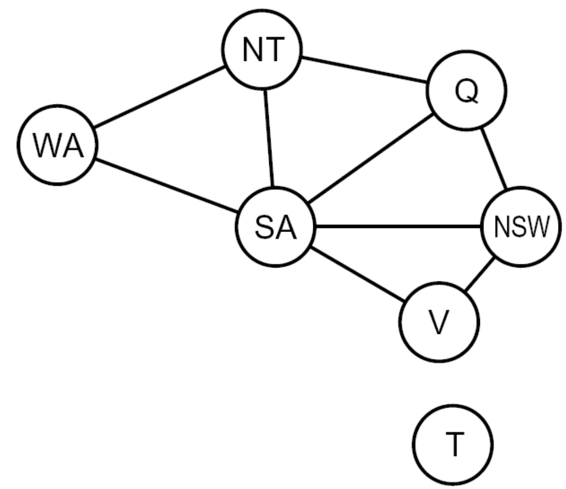
- **States**: 4 queens in 4 columns ($4^4 = 256$ states)
- **Actions**: move queen in column
- **Goal test**: no attacks
- **Evaluation**: $h(n)$ = number of attacks



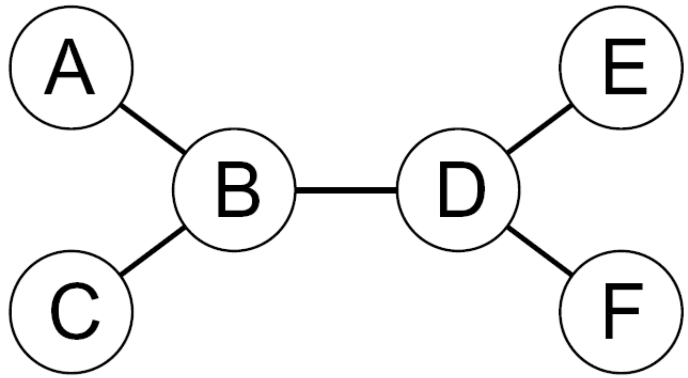
- **Given random initial state, local min-conflicts can solve *n*-queens in almost constant time for arbitrary *n* with high probability (e.g., *n* = 10,000,000)**

Problem Structure

- **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 – could that help us speed up the computation?**



Tree-Structured CSPs

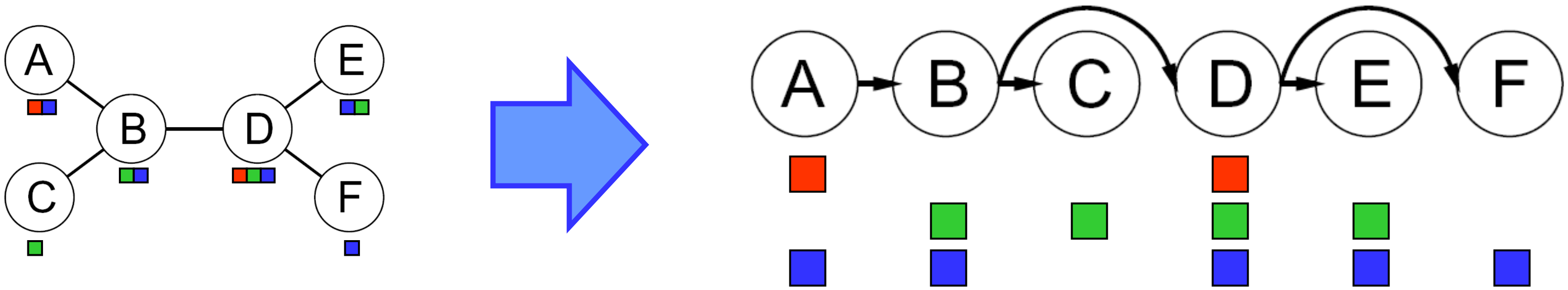


- Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\ d^2)$ time

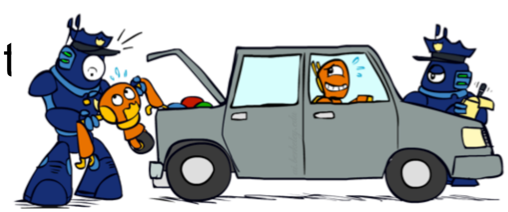
- This property also applies to probabilistic reasoning (later): an example of the relation between syntactic restrictions and the complexity of reasoning

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

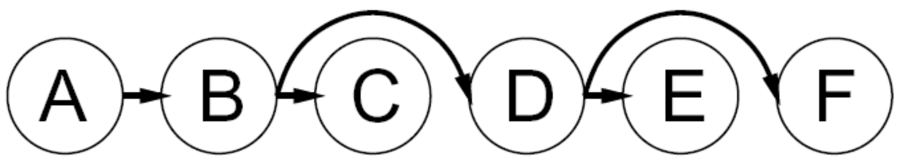


- **Runtime: O(n d^2) (why?)**

Slide credit: Dan Klein and Pieter Abbeel
http://ai.berkeley.edu

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 (because Y's children were processed before Y)



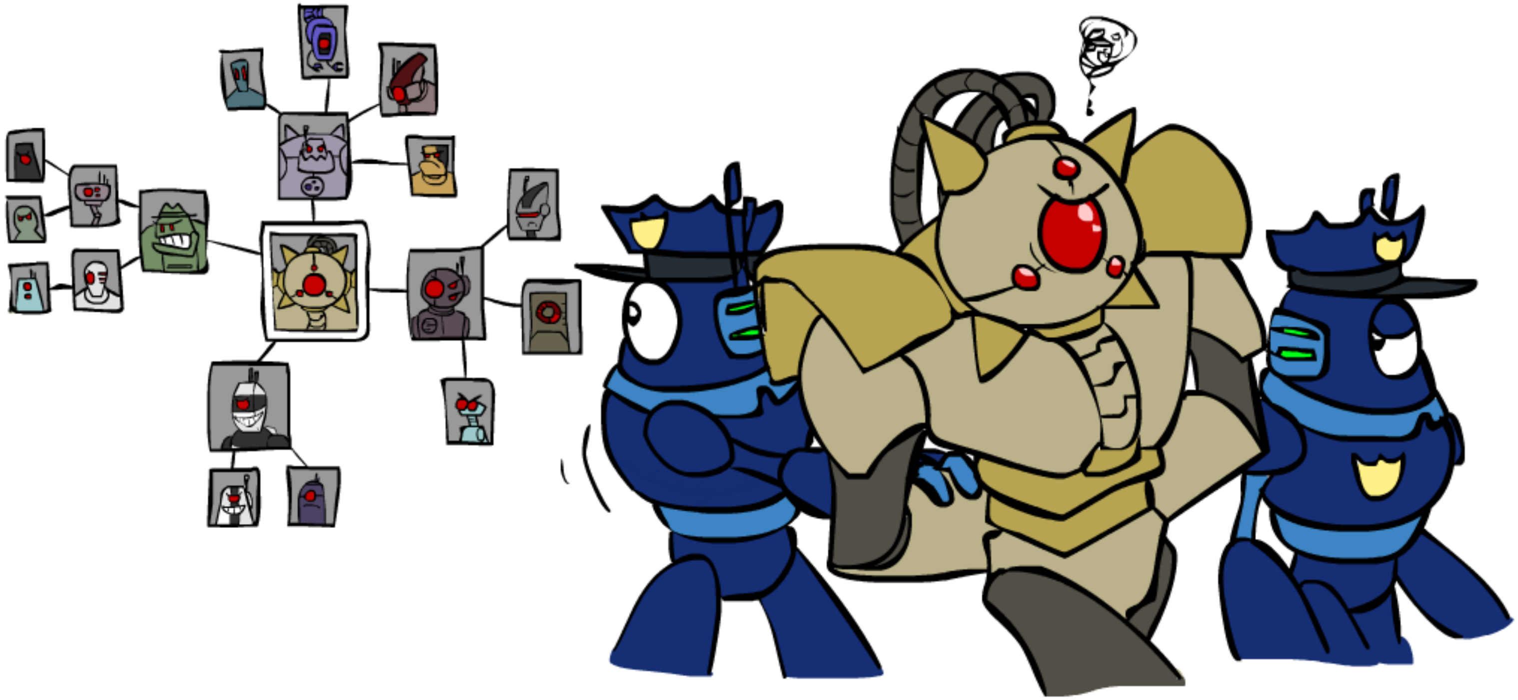
- Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack
- Proof: Induction on position

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

Slide credit: Dan Klein and Pieter Abbeel
http://ai.berkeley.edu

Penn
University of Pennsylvania

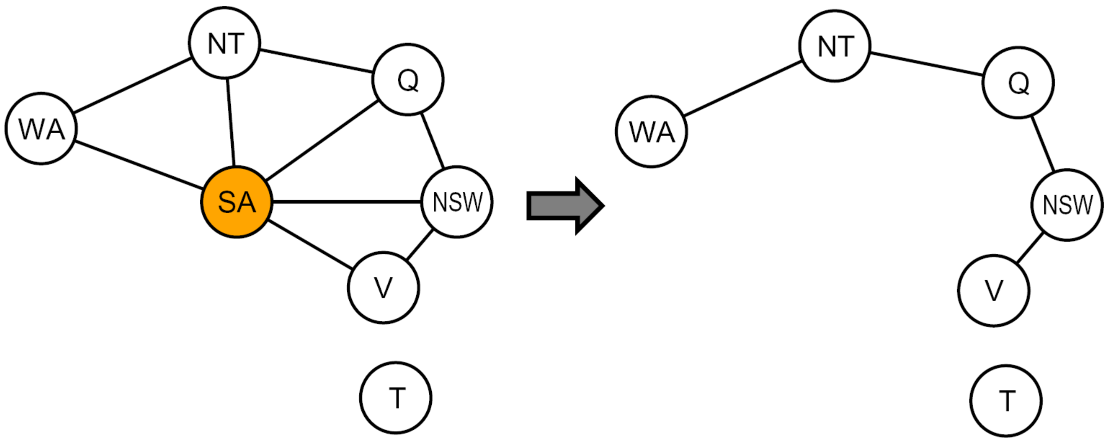
Improving Structure



Slide credit: Dan Klein and Pieter Abbeel
http://ai.berkeley.edu

Penn
UNIVERSITY of PENNSYLVANIA

Nearly Tree-Structured CSPs



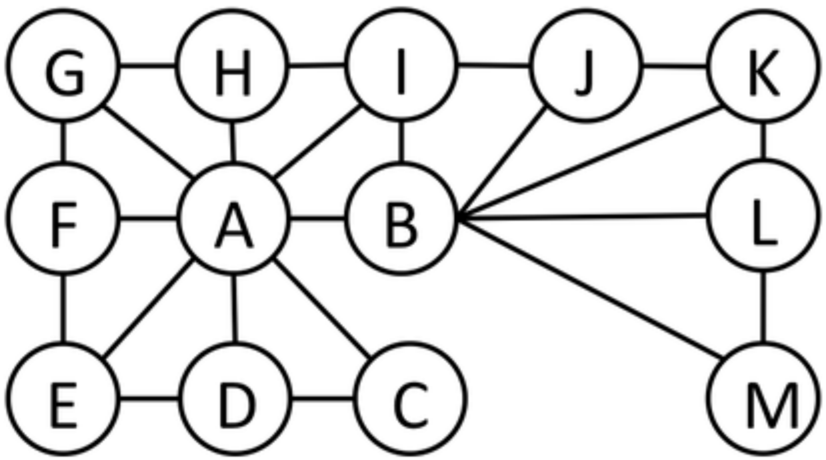
- **Conditioning: instantiate a variable, prune its neighbors' domains**
- **Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree**
- **Cutset size c gives runtime $O((d^c) (n\text{-}c) d^2)$, very fast for small c**

Slide credit: Dan Klein and Pieter Abbeel
http://ai.berkeley.edu

Penn
University of Pennsylvania

Cutset Quiz

- **Find the smallest cutset for the graph below.**



Slide credit: Dan Klein and Pieter Abbeel
http://ai.berkeley.edu

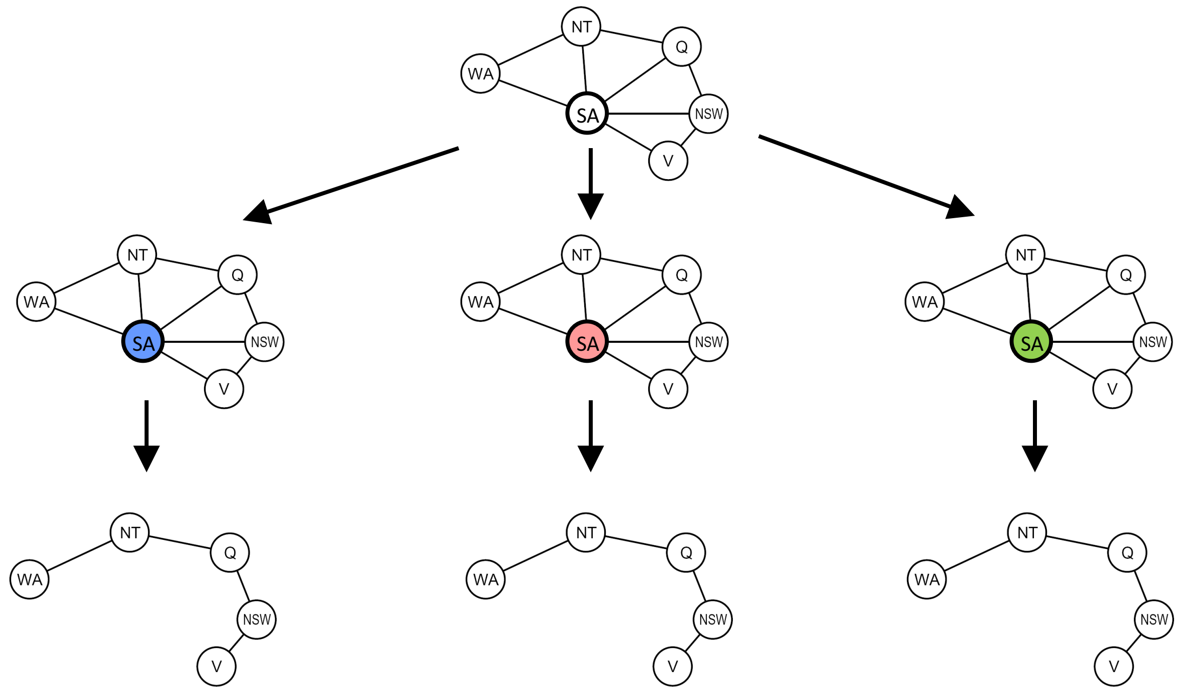
Cutset Conditioning

Choose a cutset

Instantiate the cutset (all possible ways)

Compute residual CSP for each assignment

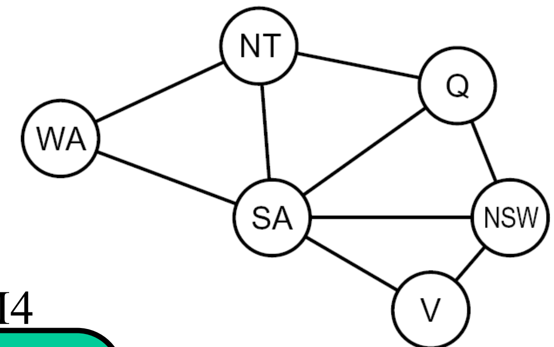
Solve the residual CSPs (tree structured)

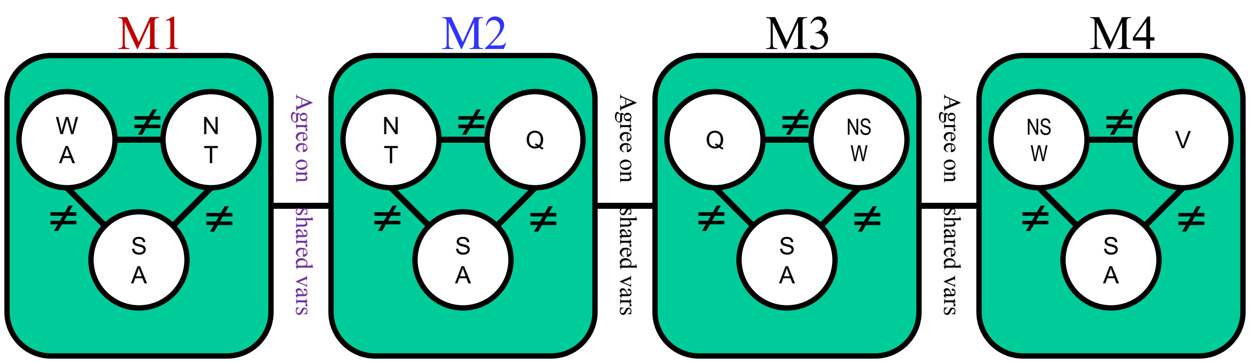


Slide credit: Dan Klein and Pieter Abbeel
http://ai.berkeley.edu

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





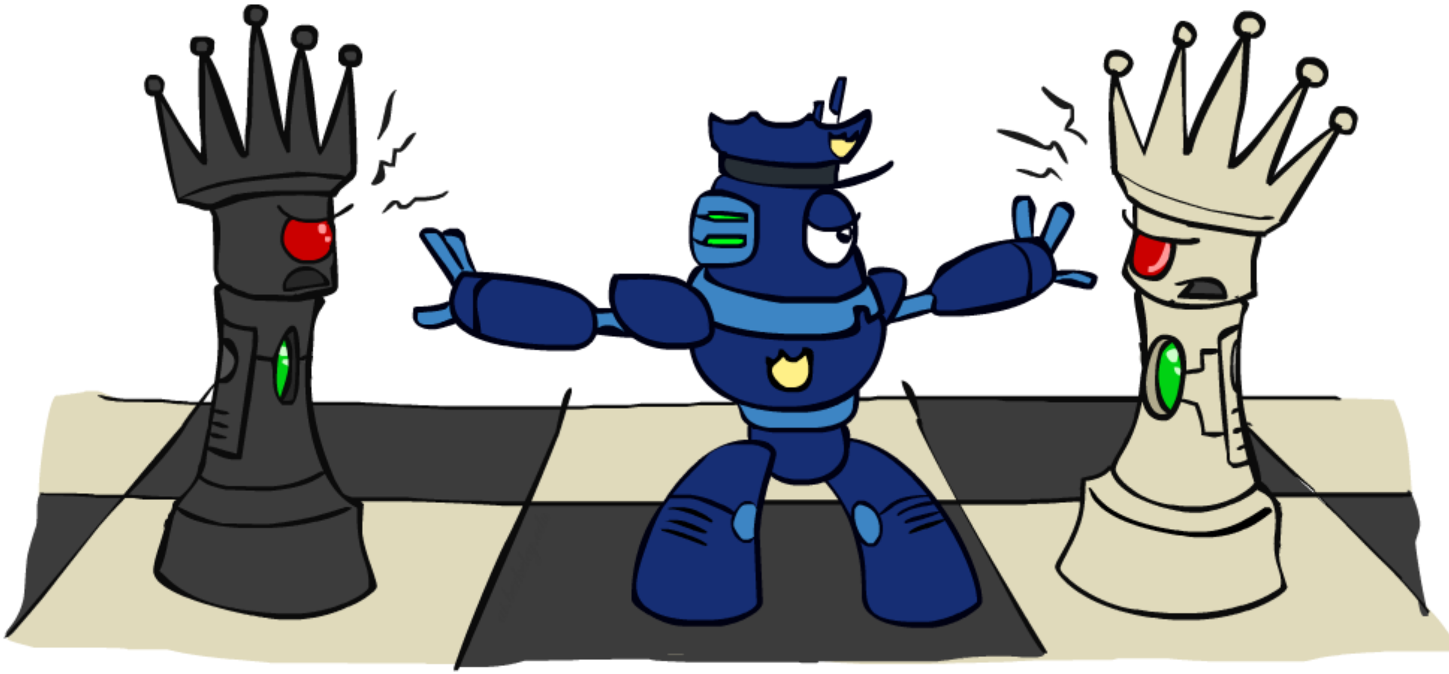
{(WA=r,SA=g,NT=b), (WA=b,SA=r,NT=g), …}

{(NT=r,SA=g,Q=b), (NT=b,SA=g,Q=r), …}

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

Slide credit: Dan Klein and Pieter Abbeel
http://ai.berkeley.edu

Iterative Improvement

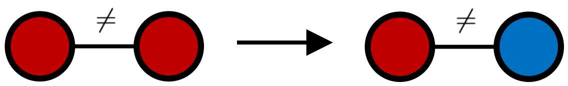


Slide credit: Dan Klein and Pieter Abbeel
http://ai.berkeley.edu

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

Slide credit: Dan Klein and Pieter Abbeel
http://ai.berkeley.edu

Simple CSPs can be solved quickly

1. **Completely independent subproblems**
   - e.g. Australia & Tasmania
   - Easiest
2. **Constraint graph is a tree**
   - Any two variables are connected by only a single path
   - Permits solution in time linear in number of variables
   - Do a topological sort and just march down the list

```
A           E
 \         /
   B –  D
 /         \
C           F
```

A → B → C → D → E → F

Beyond binary constraints: Path consistency

- Generalizes arc-consistency from individual binary constraints to multiple constraints
- A pair of variables X_i, X_j is path-consistent w.r.t. X_m if for every assignment $X_i=a$, $X_j=b$ consistent with the constraints on X_i, X_j there is an assignment to X_m that satisfied the constraints on X_i, X_m and X_j, X_m

• Global constraints

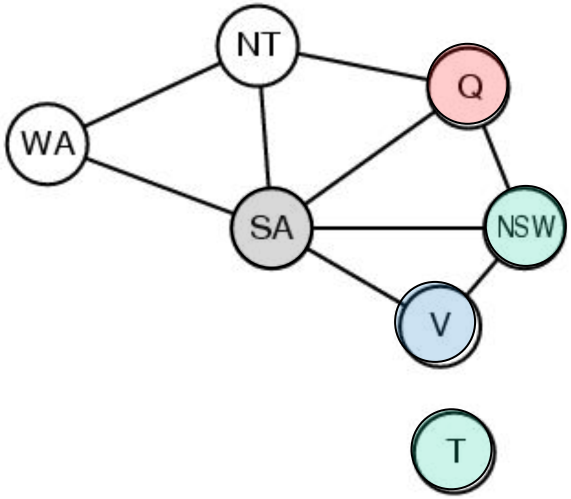
- Can apply to any number of variables
- E.g., in Sudoko, all numbers in a row must be different
- E.g., in cryptarithmetic, each letter must be a different digit
- Example algorithm:
  - —If any variable has a single possible value, delete that variable from the domains of all other constrained variables
  - —If no values are left for any variable, you found a contradiction

Simplifying hard CSPs: Cycle Cutsets

- **Constraint graph can be decomposed into a tree**
  - Collapse or remove nodes
  - *Cycle cutset* S of a graph G: any subset of vertices of G that, if removed, leaves G a tree
- **Cycle cutset algorithm**
  - Choose some cutset S
  - For each possible assignment to the variables in S that satisfies all constraints on S
    - —Remove any values for the domains of the remaining variables that are not consistent with S
    - —If the remaining CSP has a solution, then you have are done
  - For graph size n, domain size d
    - —Time complexity for cycle cutset of size c: $O(d^c * d^2(n-c)) = O(d^{c+2}(n-c))$

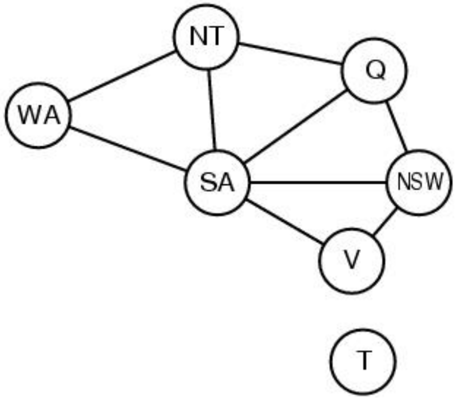
Chronological backtracking

- **DFS does Chronological backtracking**
  - If a branch of a search fails, backtrack to the most recent variable assignment and try something different
  - But this variable may not be related to the failure
- **Example: Map coloring of Australia**
  - Variable order
    - —Q, NSW, V, T, SA, WA, NT.
  - Current assignment:
    - —Q=red, NWS=green, V=blue, T= red
  - SA cannot be assigned anything
  - But reassigning T does not help!



Backjumping: Improved backtracking

- **Find “the conflict set”**
  - Those variable assignments that are in conflict
  - Conflict set for SA: {Q=red, NSW=green, V=blue}
- **Jump back to reassign one of those conflicting variables**
- **Forward checking can build the conflict set**
  - When a value is deleted from a variable’s domain, add it to its conflict set
  - But backjumping finds the same conflicts that forward checking does
  - Fix using “conflict-directed backjumping”
    - —Go back to predecessors of conflict set



When to Iterate, When to Stop?

The crucial principle:

If a value is removed from a node X_i,
then the values on all of X_i's neighbors must be reexamined.

Why? *Removing* **a value from a node may result in one of the neighbors becoming arc** *inconsistent*, **so we need to check…**

(but each neighbor X_j can only become inconsistent with respect to the removed values on X_i)