Constraint Satisfaction Problems

AIMA: Chapter 6

Constraint Satisfaction Problems

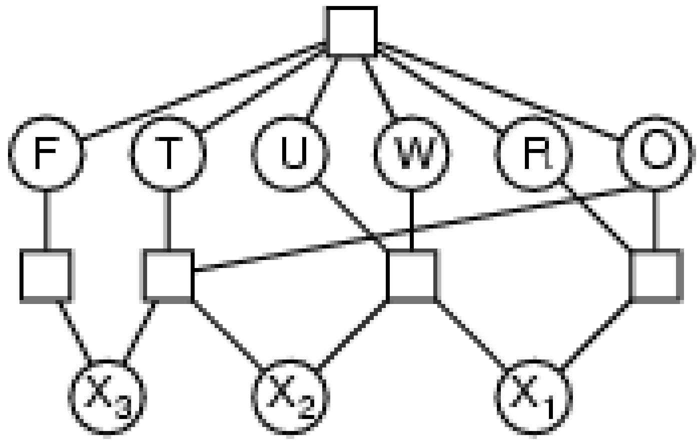
A CSP consists of:

- *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, e.g., $X_1 \neq X_2$ A *state* is defined as an *assignment* of values to some or all variables.

- **A *consistent assignment* does not violate the constraints.**
- **Example: Sudoku**

Example: Cryptarithmetic

$$\begin{array}{ccccc} & X_3 & X_2 & X_1 & \\ & & T & W & O \\ + & & T & W & O \\ \hline & F & O & U & R \end{array}$$

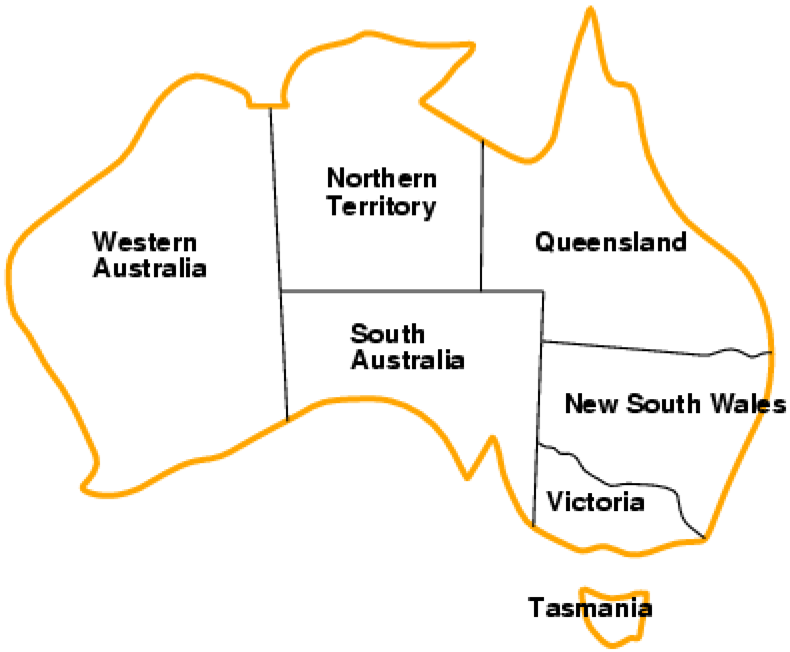


- **Variables**: $F\ T\ U\ W\ R\ O,\ X_1\ X_2\ X_3$
- **Domain**: $\{0,1,2,3,4,5,6,7,8,9\}$
- **Constraints**:
  - $Alldiff\,(F,T,U,W,R,O)$
  - $O + O = R + 10 \cdot X_1$
  - $X_1 + W + W = U + 10 \cdot X_2$
  - $X_2 + T + T = O + 10 \cdot X_3$
  - $X_3 = F,\ T \neq 0,\ F \neq 0$

Constraint satisfaction problems

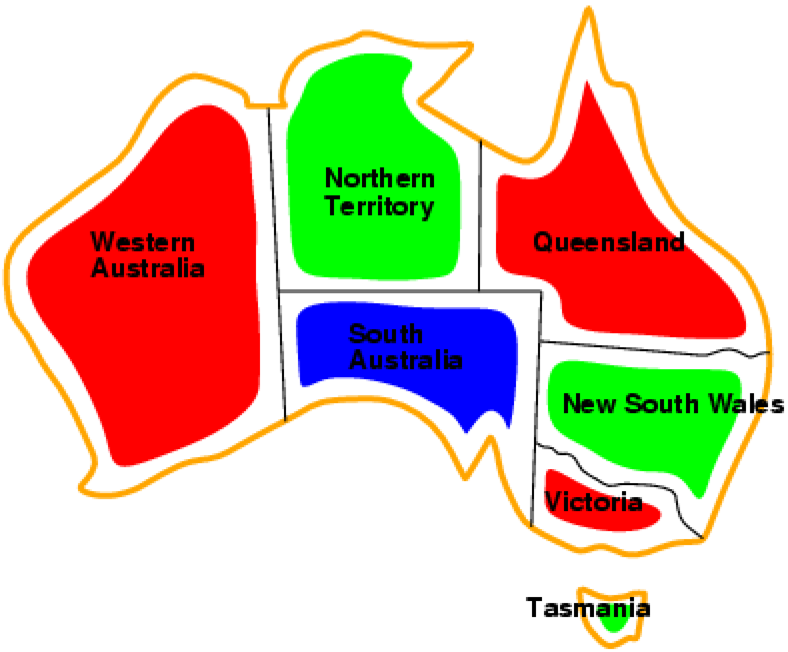
- **An assignment is *complete* when every variable is assigned a value.**
- **A *solution* to a CSP is a complete assignment that satisfies all constraints.**
- **Applications:**
  - Map coloring
  - Line Drawing Interpretation
  - Scheduling problems
    - —Job shop scheduling
    - —Scheduling the Hubble Space Telescope
  - Floor planning for VLSI
- **Beyond our scope: CSPs that require a solution that maximizes an *objective function*.**

Example: Map-coloring



- **Variables:** ***WA, NT, Q, NSW, V, SA, T***
- **Domains:** $\boldsymbol{D_i}$ **= {red,green,blue}**
- **Constraints: adjacent regions must have different colors**
  - e.g., WA ≠ NT
    - —So (WA,NT) must be in {(red,green),(red,blue),(green,red), …}

Example: Map-coloring



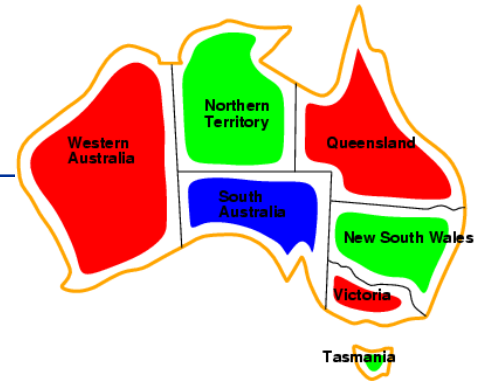
Solutions are complete and consistent assignments,

- e.g., WA = red, NT = green,Q = red,NSW = green,
  V = red,SA = blue,T = green

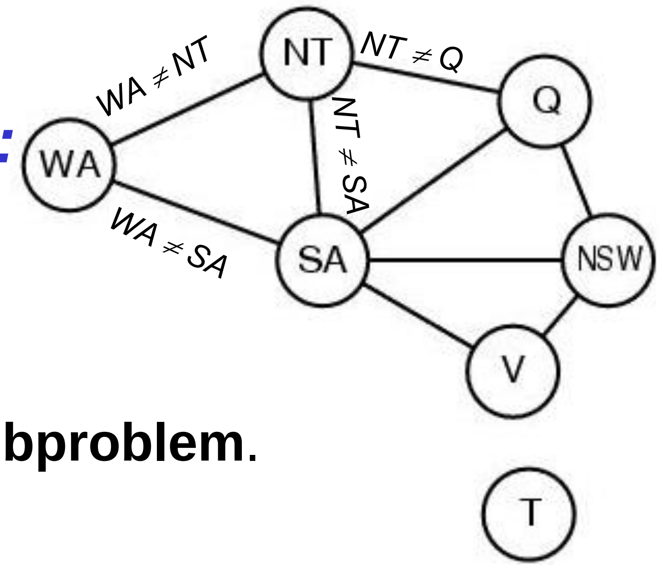
Benefits of CSP

- **Clean specification of many problems, generic goal, successor function & heuristics**
  - Just represent problem as a CSP & solve with general package
- **CSP "knows" which variables violate a constraint**
  - And hence where to focus the search
- ***CSPs:* Automatically prune off all branches that violate constraints**
  - (State space search could do this only by *hand-building constraints into the successor function)*

CSP Representations



- ***Constraint graph:***
  - *nodes* are variables
  - *arcs* are 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

Varieties of CSPs

- ***Discrete variables***
  - finite domains:
    - —*n* variables, domain size *d* → $O(d^n)$ complete assignments
    - —e.g., Boolean CSPs, includes Boolean satisfiability (NP-complete)
    - —***Line Drawing Interpretation***
  - infinite domains:
    - —integers, strings, etc.
    - —e.g., job scheduling, variables are start/end days for each job
    - —need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$
- ***Continuous variables***
  - e.g., start/end times for Hubble Space Telescope observations
  - linear constraints solvable in polynomial time by linear programming

Varieties of constraints

- ***Unary*** **constraints involve a single variable,**
  - e.g., SA ≠ green
- ***Binary*** **constraints involve pairs of variables,**
  - e.g., SA ≠ WA
- ***Higher-order*** **constraints involve 3 or more variables**
  - e.g., crypt-arithmetic column constraints
- ***Preference*** **(soft constraints) e.g. *red* is better than *green* can be represented by a cost for each variable assignment**
  - Constrained optimization problems.

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

Backtracking search

- **Note that variable assignments are *commutative***
  - Eg *[step 1: **WA = red**; step 2: **NT = green**]*
    equivalent to *[step 1: **NT = green**; step 2: **WA = red**]*
  - Therefore, a *tree search,* not a *graph search*

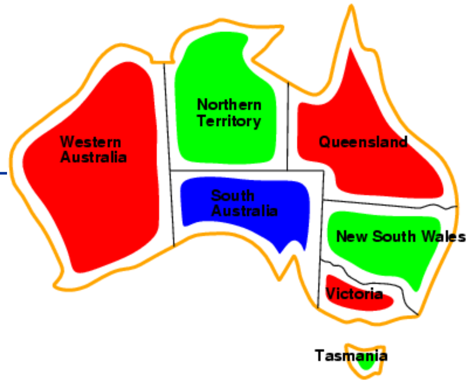
- **Only need to consider assignments to a single variable at each node**
  - $b = d$ and there are d^n leaves

- **Depth-first search for CSPs with single-variable assignments is called *backtracking* search**

- **Backtracking search is the basic *uninformed* algorithm for CSPs**

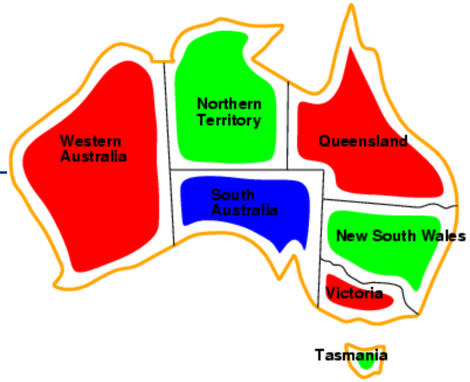
- **Can solve *n*-queens for $n \approx 25$**

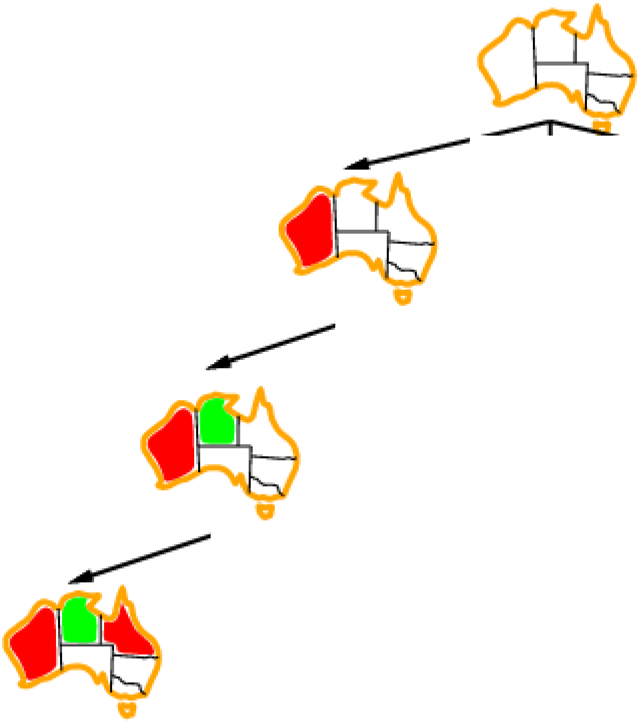
Backtracking example



And so on….

Backtracking example





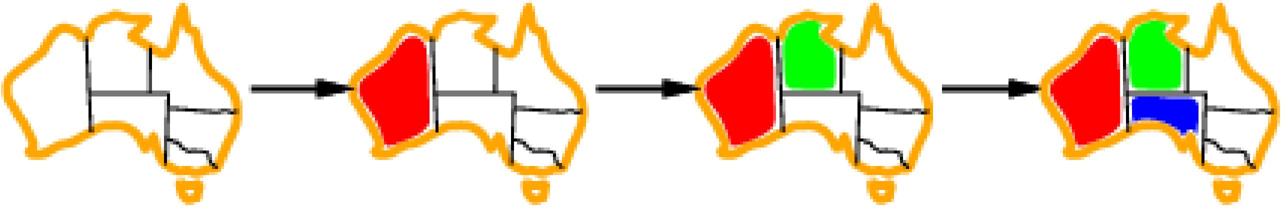
And so on….

Idea 2: Improving backtracking efficiency

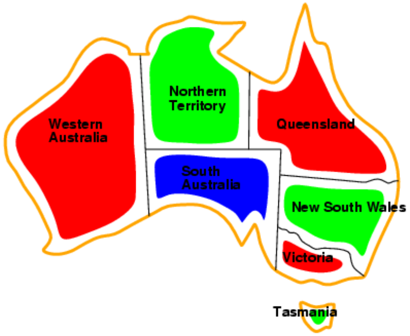
- ***General-purpose*** **methods & heuristics can give huge gains in speed,** ***on average***
- **Heuristics:**
  - Q: Which variable should be assigned next?
    1. **Most constrain*ed* variable**
    2. **Most constrain*ing* variable**
  - Q: In what order should that variable's values be tried?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

Heuristic 1: Most constrained variable

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

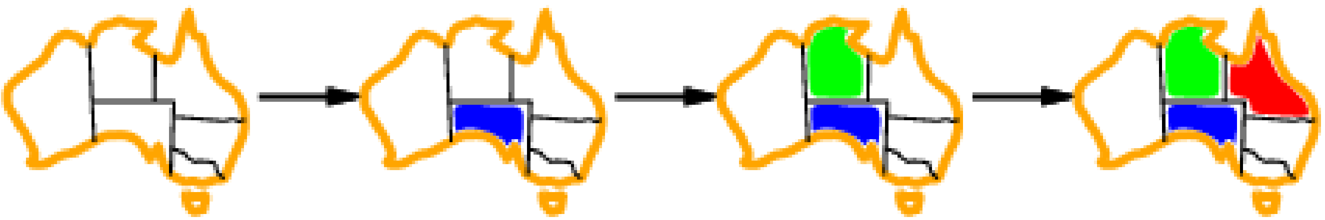


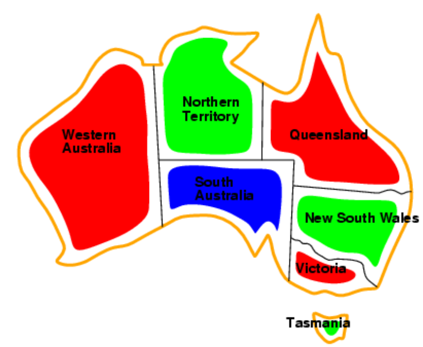
- **a.k.a. *minimum remaining values (MRV)* heuristic**



Heuristic 2: Most constraining variable

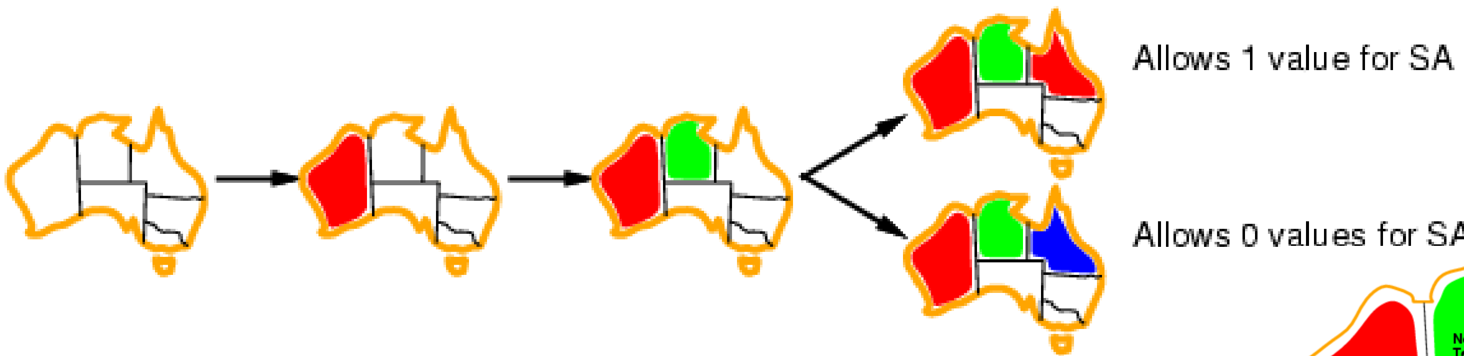
- **Tie-breaker among most constrained variables**
- **Choose the variable with the *most constraints on remaining variables***

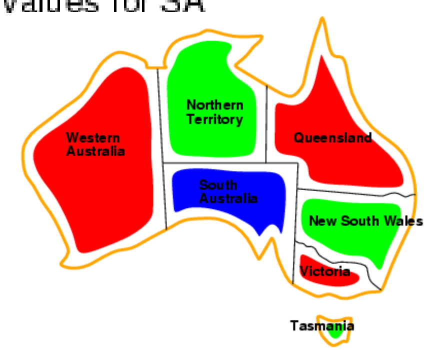




Heuristic 3: Least constraining *value*

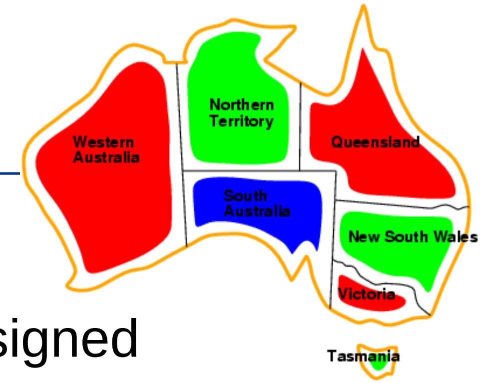
- **Given a variable, *choose the least constraining value:***
  - the one that rules out the fewest values in the remaining variables

- 

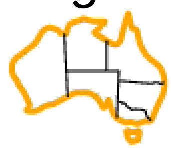


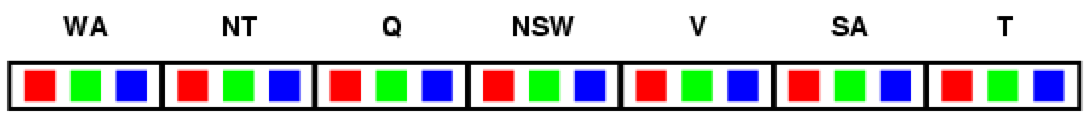
Note: demonstrated here independent of the other heuristics

Heuristic 4: Forward checking



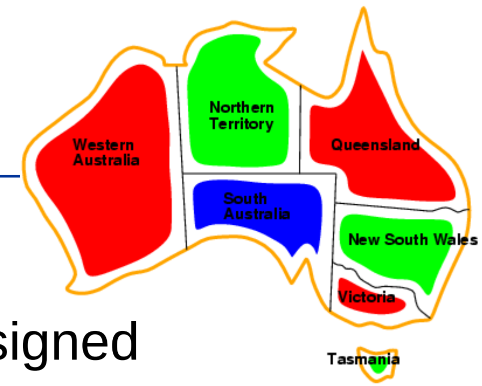
- **Idea**:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values, *given its neighbors*



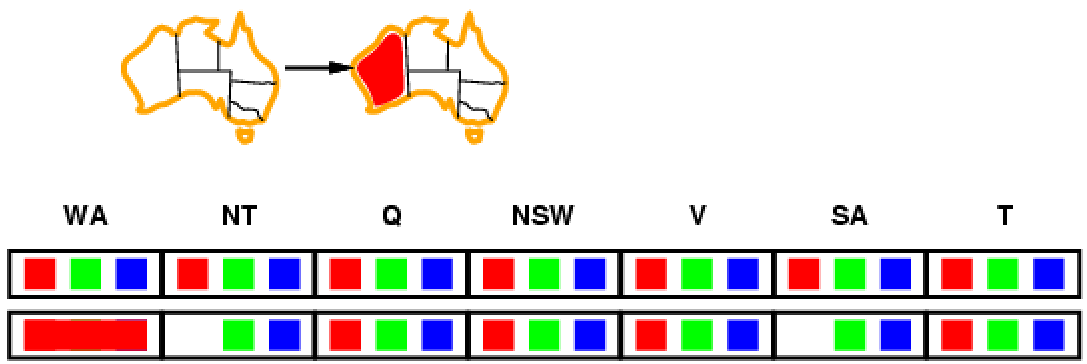


  - (For later: Edge & Arc consistency are variants)

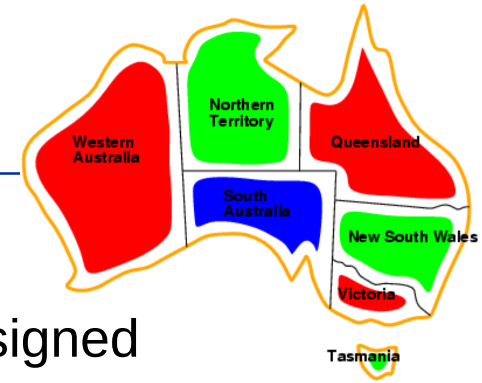
Forward checking



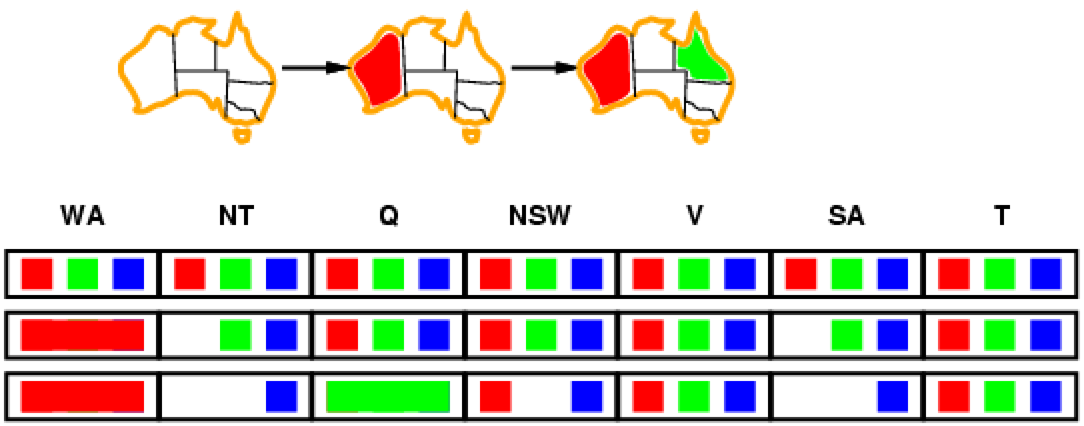
- **Idea**:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values



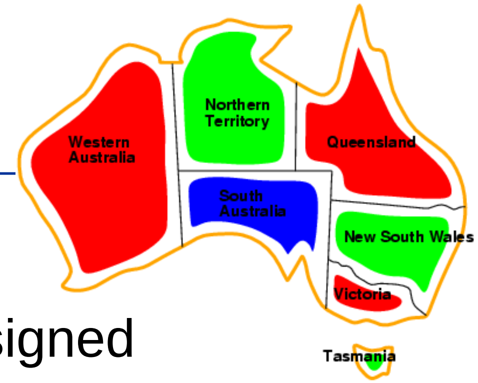
Forward checking



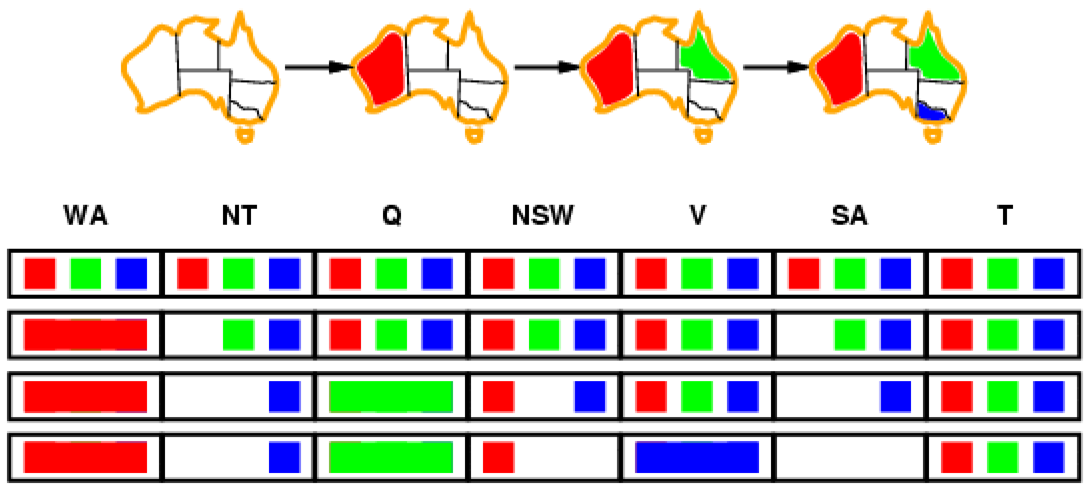
- **Idea**:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values



Forward checking

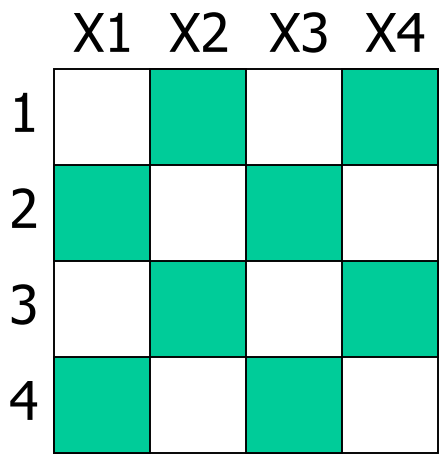


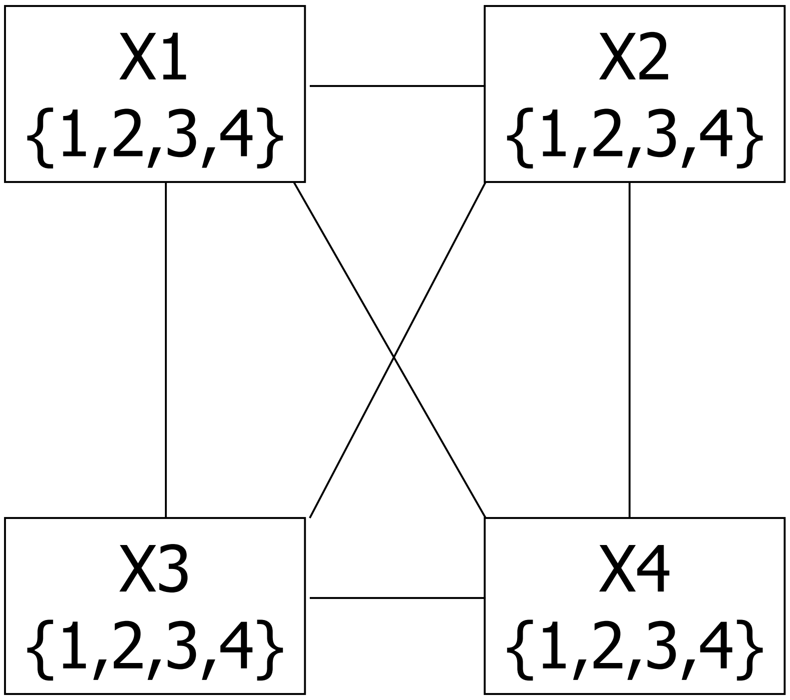
- **Idea**:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values



- **A Step toward AC-3: The most efficient algorithm**

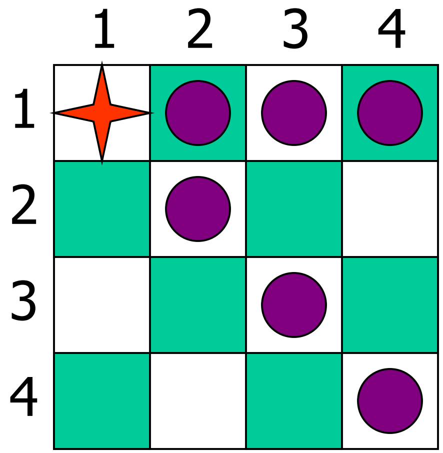
Example: 4-Queens Problem

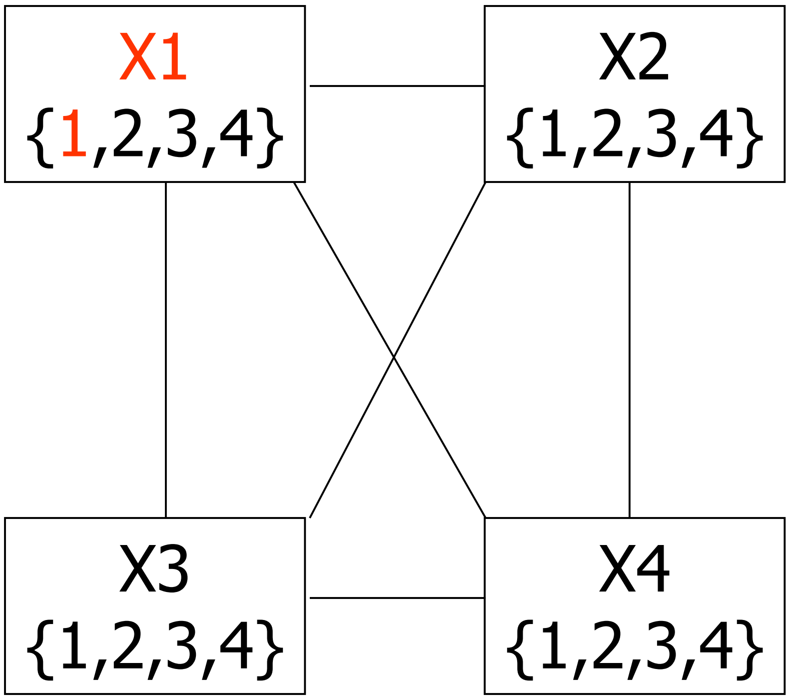




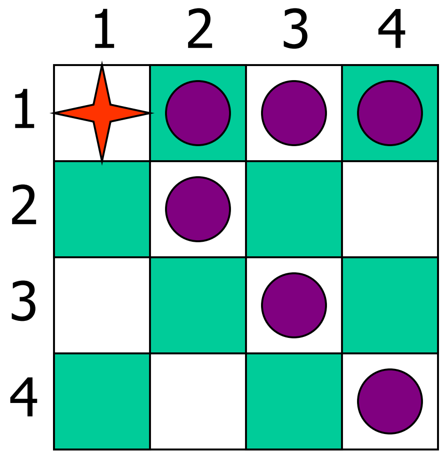
(From Bonnie Dorr, U of Md, CMSC 421)

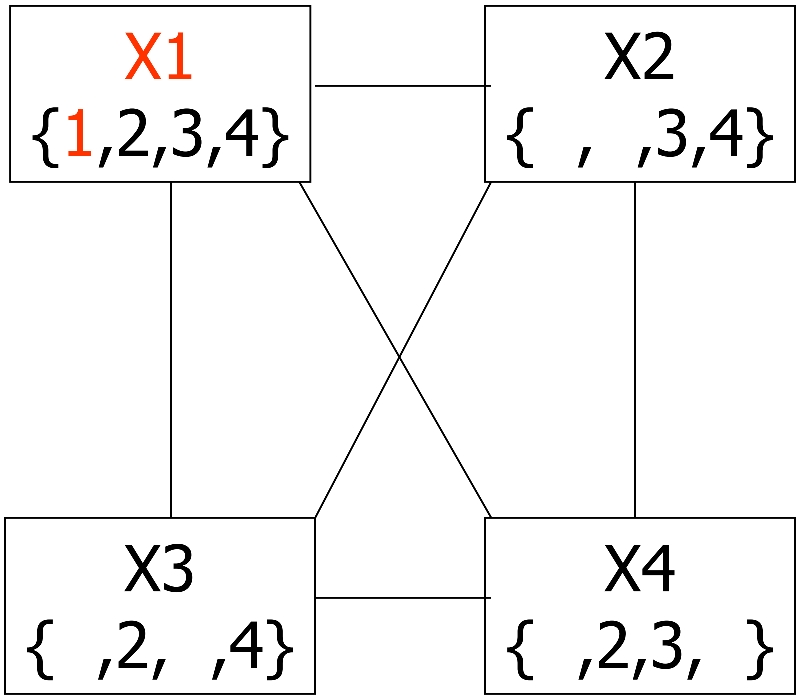
Example: 4-Queens Problem



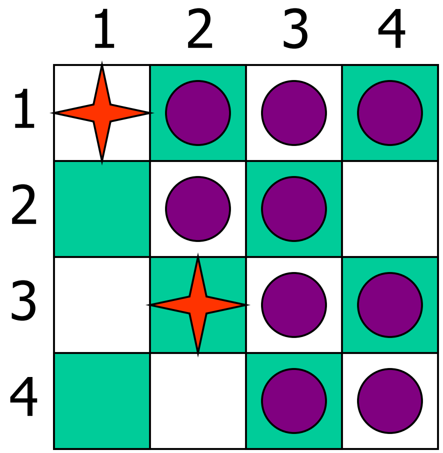


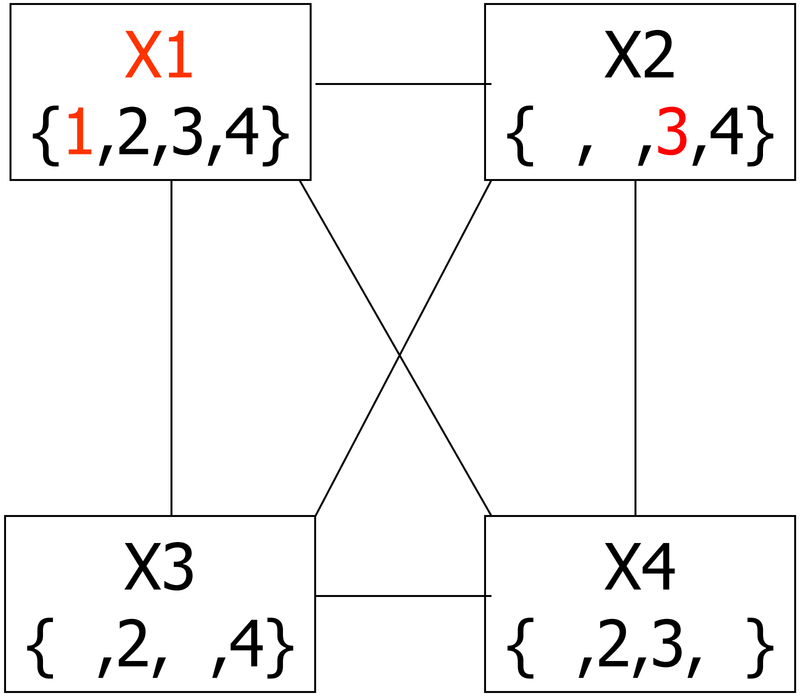
Example: 4-Queens Problem



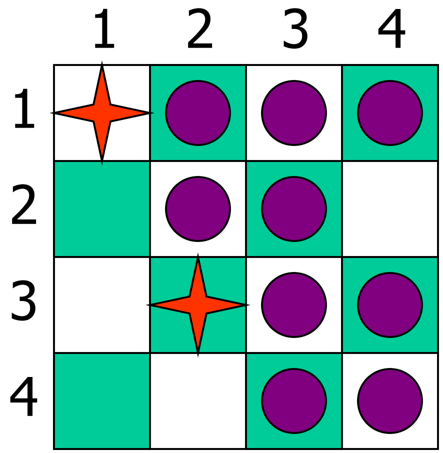


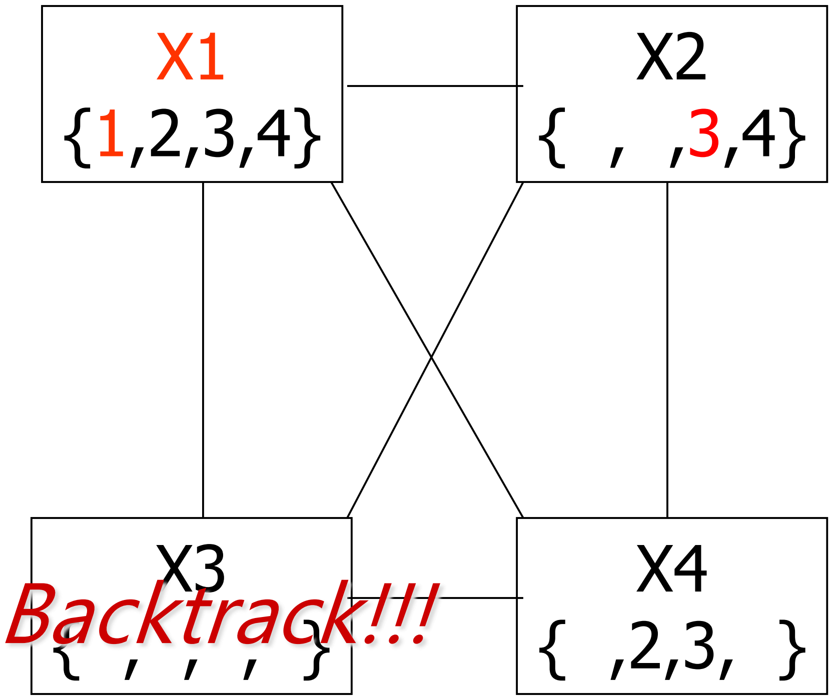
Example: 4-Queens Problem





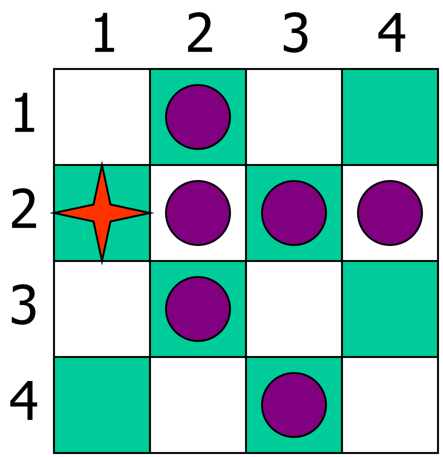
Example: 4-Queens Problem

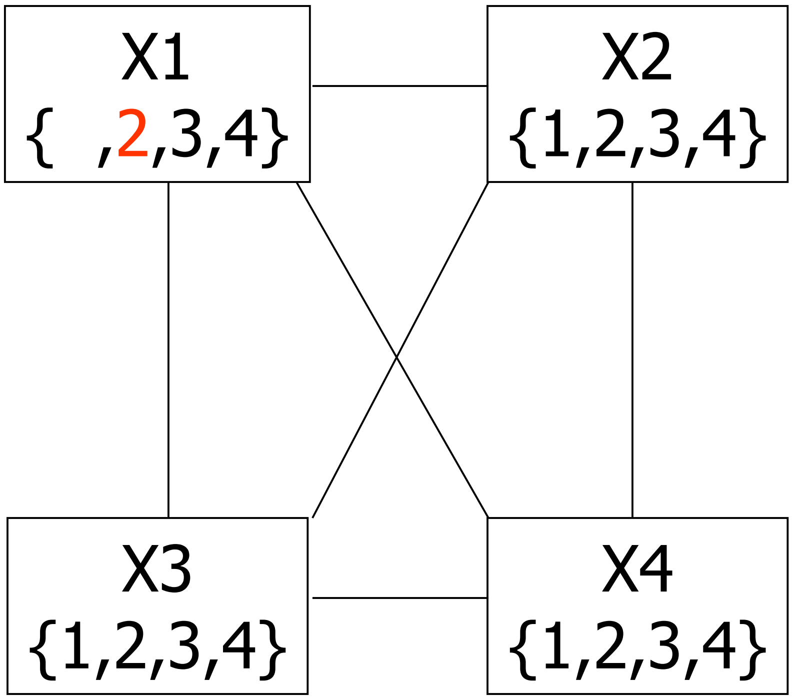




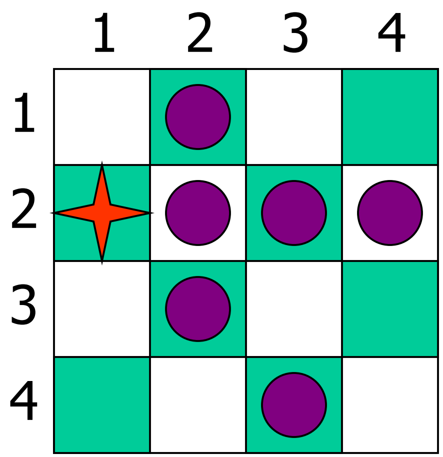
Example: 4-Queens Problem

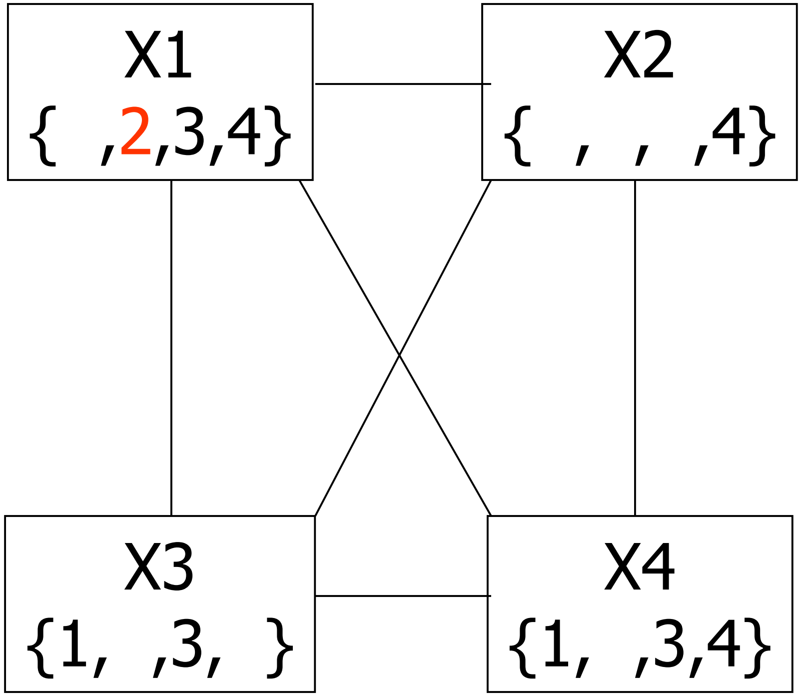
Picking up a little later after two steps of backtracking....



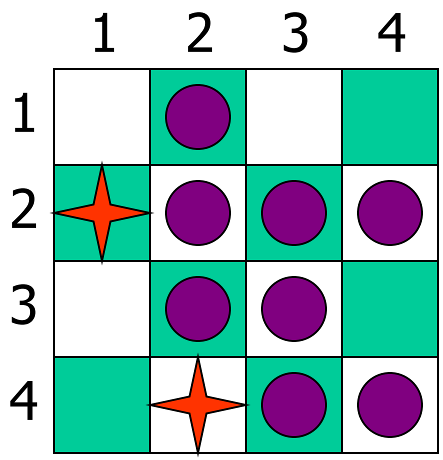


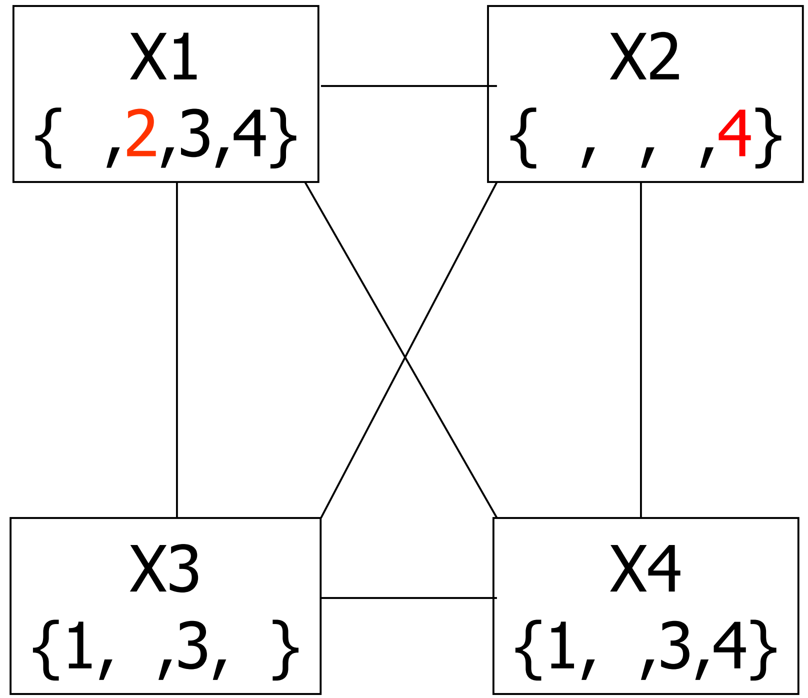
Example: 4-Queens Problem



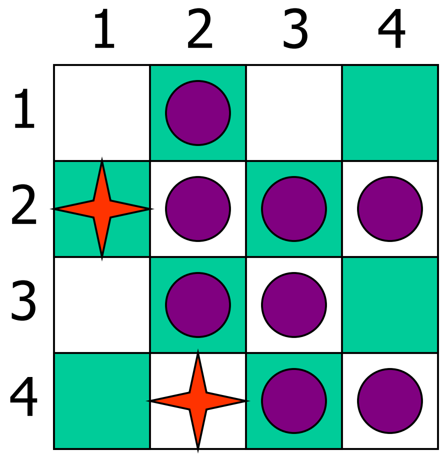


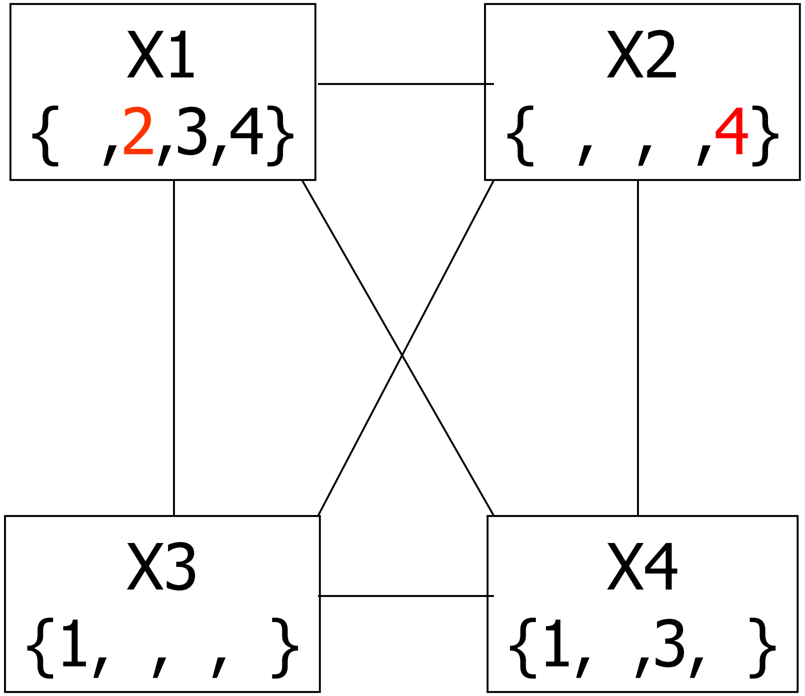
Example: 4-Queens Problem



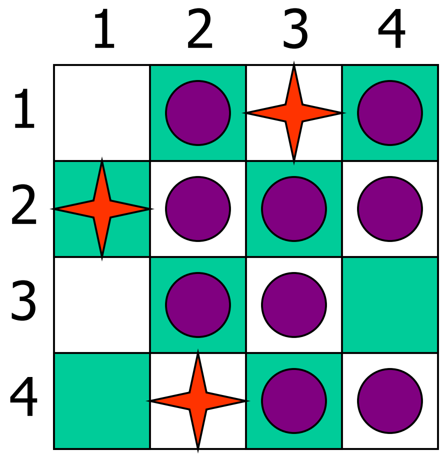


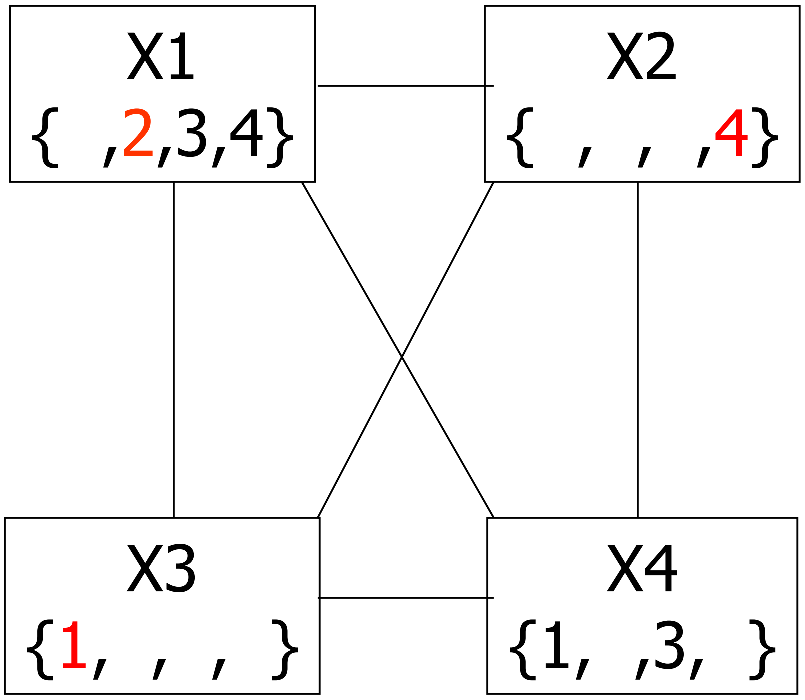
Example: 4-Queens Problem



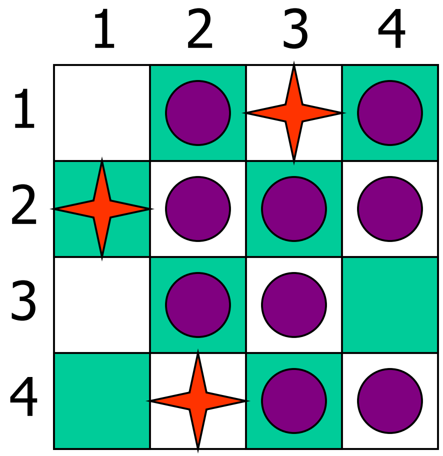


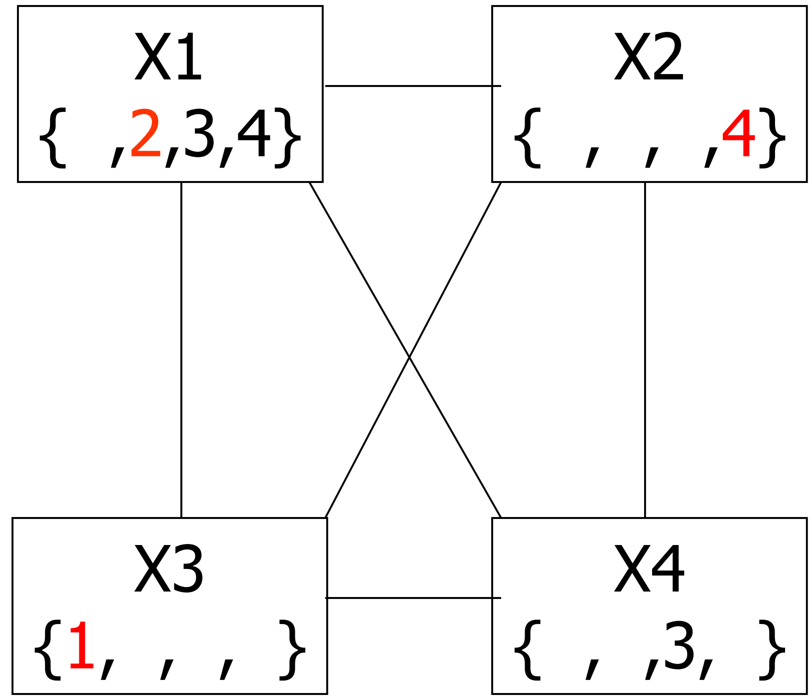
Example: 4-Queens Problem



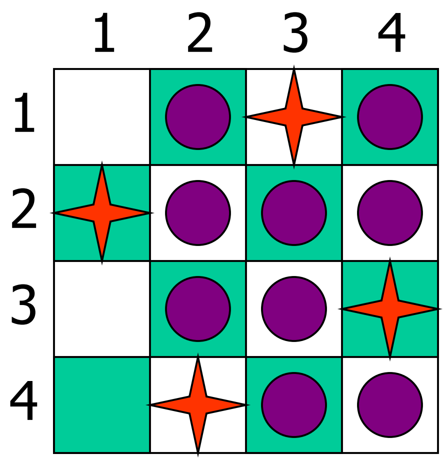


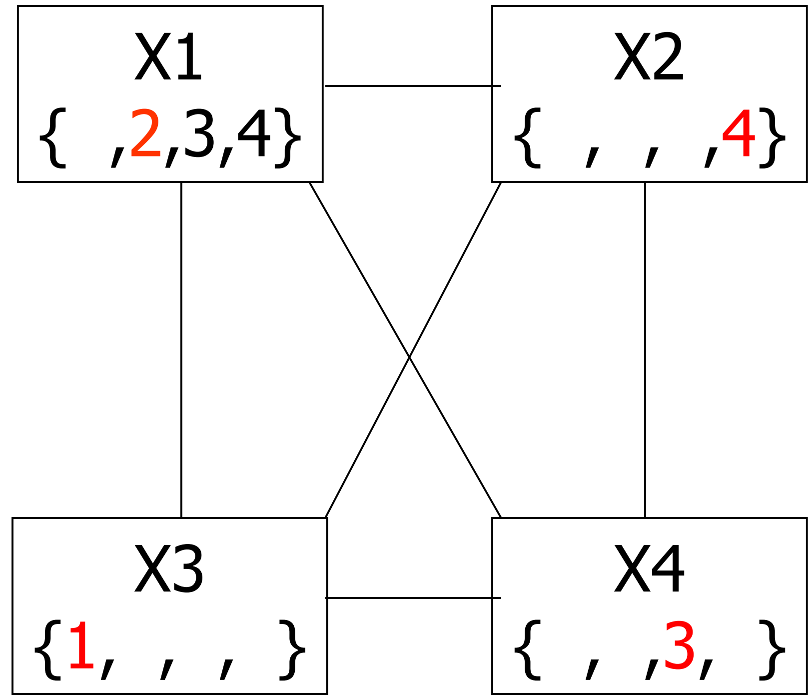
Example: 4-Queens Problem



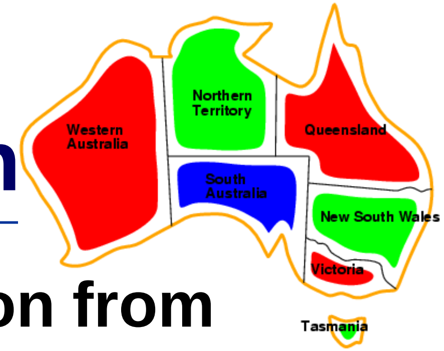


Example: 4-Queens Problem

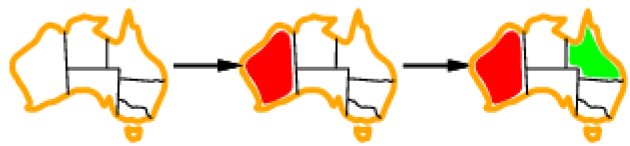




Towards Constraint propagation

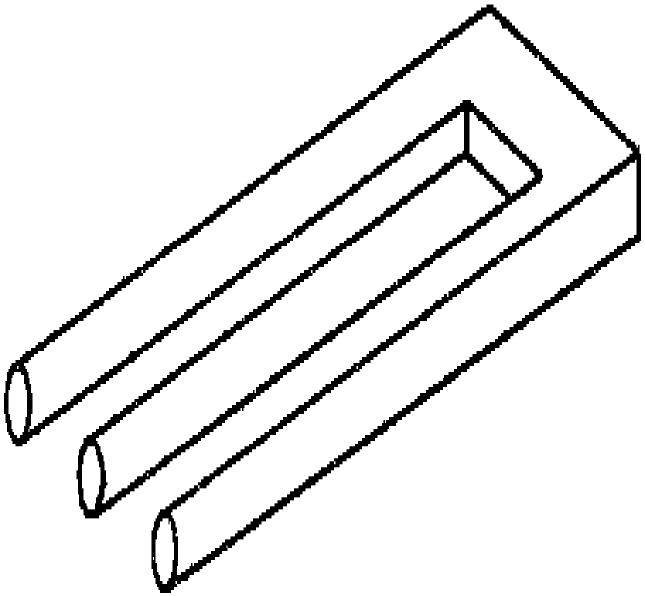


- **Forward checking propagates information from *assigned* to *unassigned* variables, but doesn't provide early detection for all failures:**



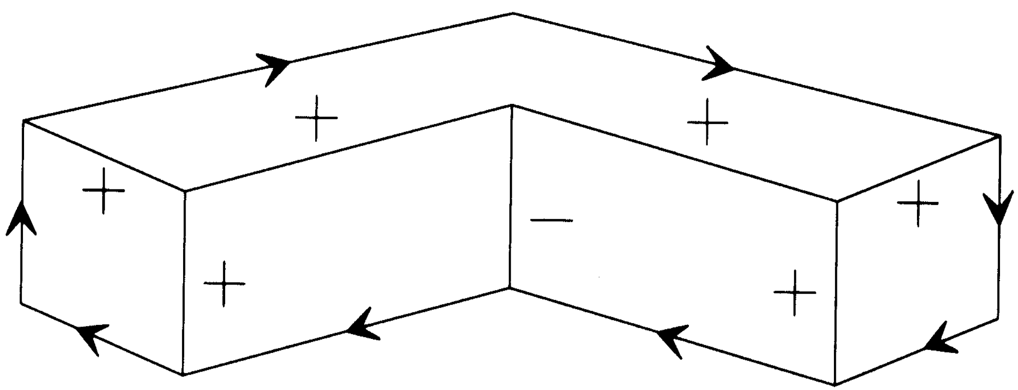
- **NT and SA cannot both be blue!**
- **Constraint propagation goes beyond forward checking & repeatedly enforces constraints locally**

Interpreting line drawings and the invention of constraint *propagation* algorithms



We Interpret Line Drawings As 3D

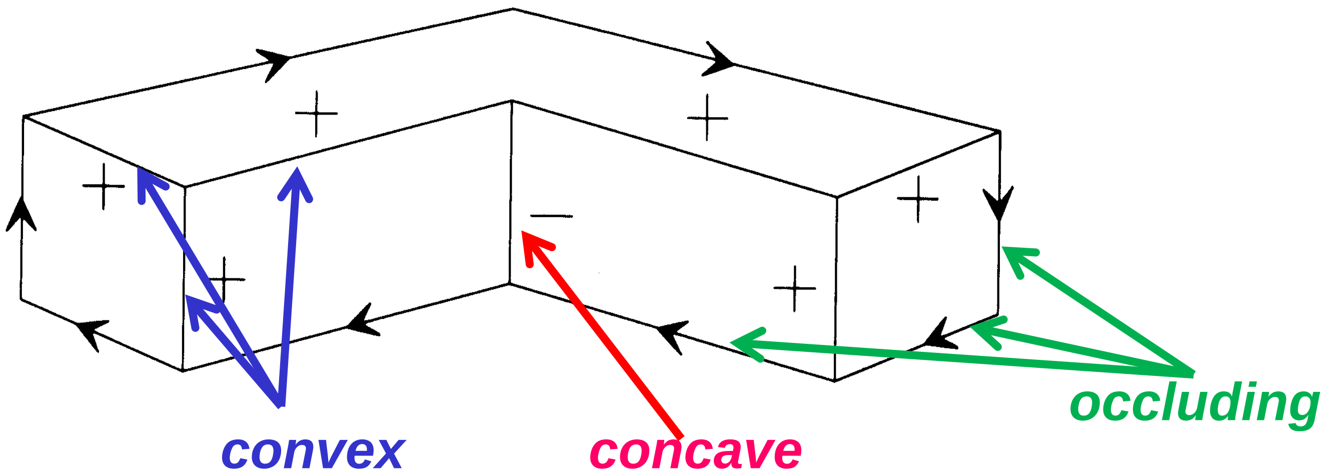
- **We have strong intuitions about line drawings of simple geometric figures:**
- **We naturally interpret 2D line drawings as planar *representations* of *3D* objects.**
- **We interpret each line as being either a *convex, concave or occluding edge* in the actual object.**



Interpretation as Convexity Labeling

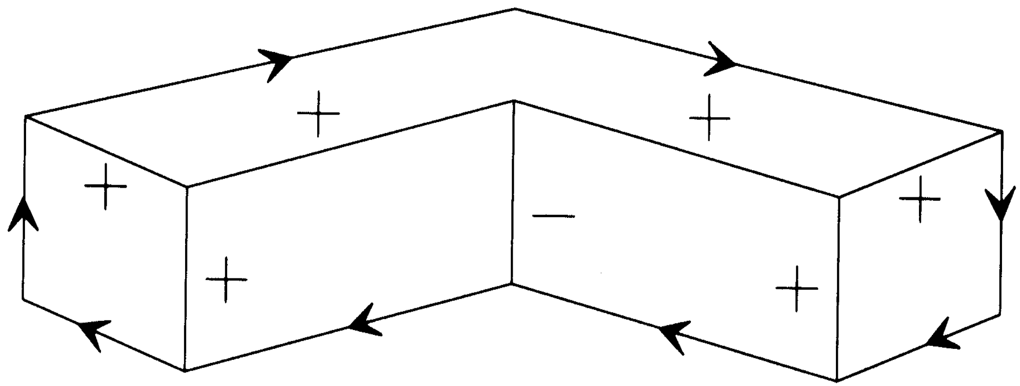
Each edge in an image can be interpreted to be either *a convex edge, a concave edge or an occluding edge:*

- **+ labels a *convex edge* (angled toward the viewer);**
- **- labels a *concave edge* (angled away from the viewer);**
- **→ labels an *occluding edge*. To its right is the body for which the arrow line provides an edge. On its left is space.**



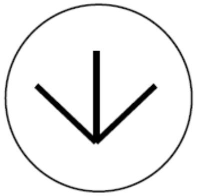
Huffman/Clowes Line Drawing Interpretation

- ***Given:*** **a line drawing of a simple "blocks world" physical image**
- ***Compute*****: a set of junction labels that yields a consistent physical interpretation**



Huffman/Clowes Junction Labels

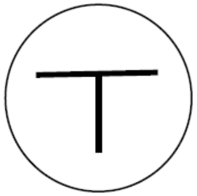
- **A simple trihedral image can be automatically interpreted given only information about each *junction* in the image.**
- **Each interpretation gives *convexity* information for each junction.**
- **This interpretation is based on the *junction type*. (All junctions involve at most *three* lines.)**



arrow junction

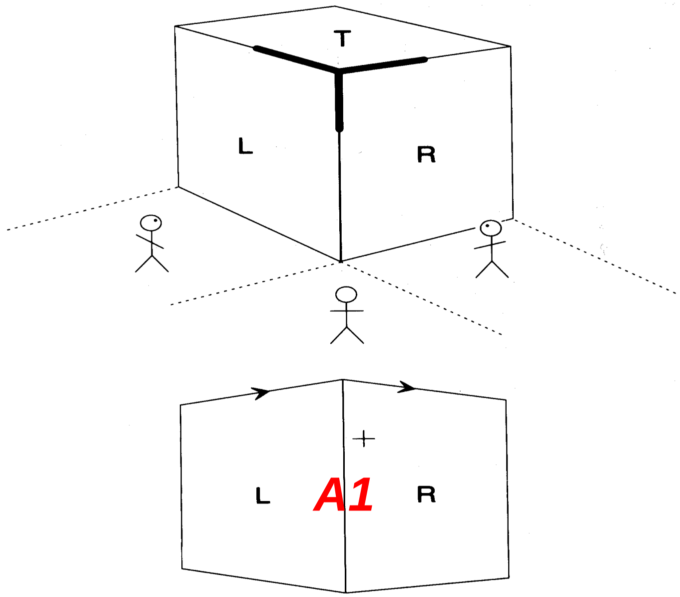
Y junction

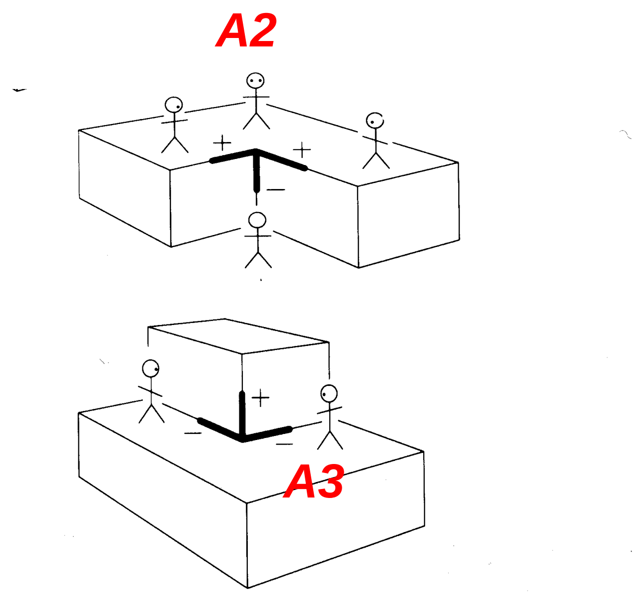
L junction



T junction

Arrow Junctions have *only* 3 interpretations

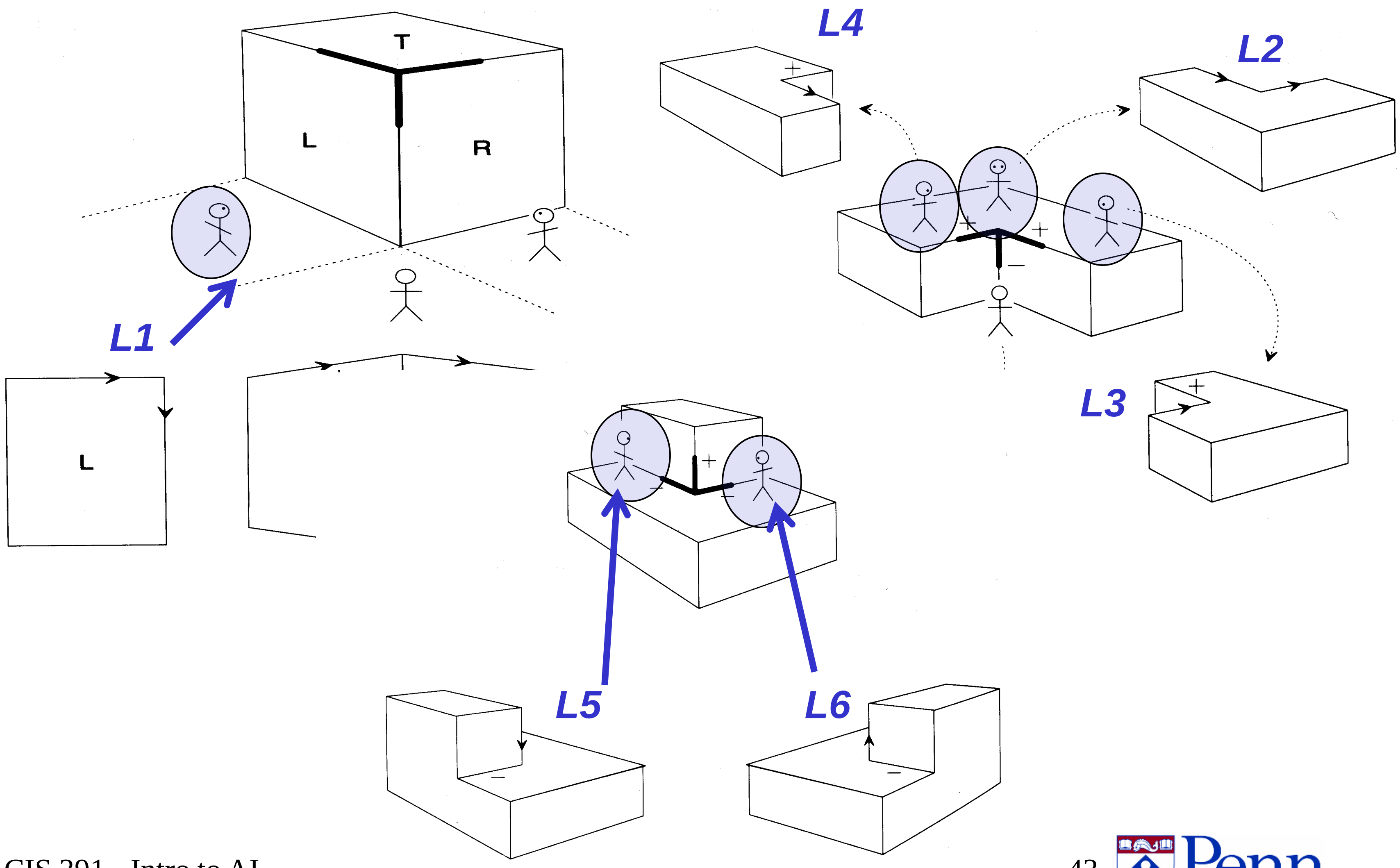




The ***image*** of the ***same*** vertex from a ***different*** point of view gives a ***different*** junction type

(from *Winston, Intro to Artificial Intelligence*)

L Junctions have *only* 6 interpretations!



The world *constrains possibilities*

Type of Junction	*Physically Possible Interpretations*	*Combinatoric Possibilities*
Arrow	**3**	**4x4x4= 64**
L	**6**	**4x4=16**
T	**4**	**4x4x4=64**
Y	**3**	**4x4x4=64**

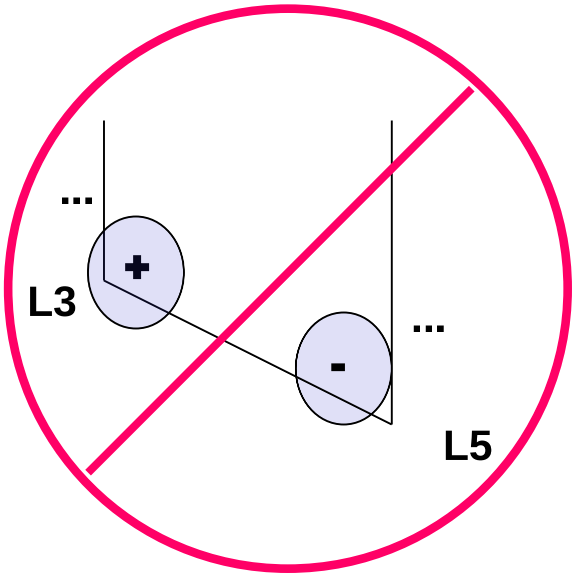
Idea 3 (*big* idea): Inference in CSPs

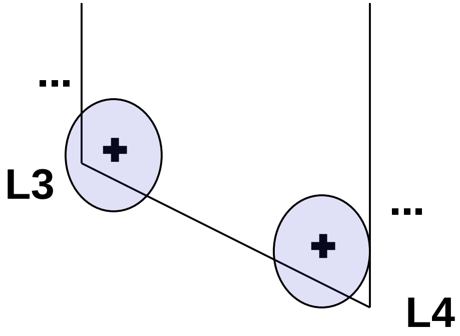
- **CSP solvers combine search *and inference***
  - Search
    - —assigning a value to a variable
  - *Constraint propagation (inference)*
    - —Eliminates possible values for a variable if the value would violate local consistency
  - *Can do inference first, or intertwine it with search*
    - —You'll 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.

An Example Constraint:
The Edge Consistency Constraint

The Edge Consistency Constraint

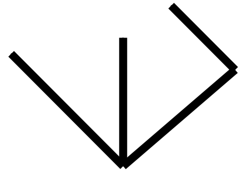
Any *consistent assignment* of labels to the junctions in a picture must assign the *same* line label to any given line.



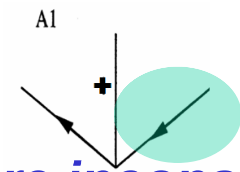


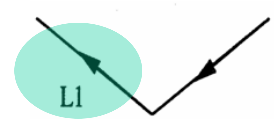
An Example of Edge Consistency

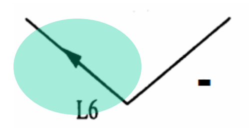
- **Consider an arrow junction with an L junction to the right:**



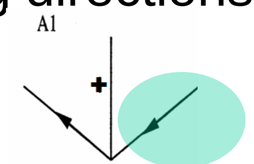
- ***A1 and either L1 or L6 are consistent*** since they both associate the same kind of arrow with the line.

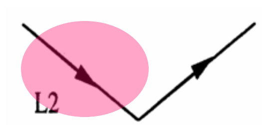




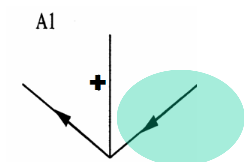


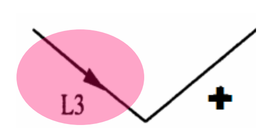
- ***A1 and L2 are inconsistent***, since the arrows are pointed in the opposing directions,



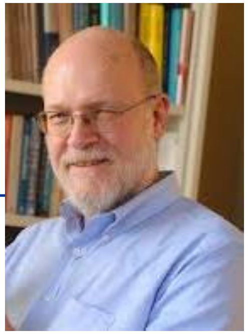


- ***Similarly, A1 and L3 are inconsistent.***





Replacing Search: Constraint Propagation Invented…



Dave Waltz's insight for line labeling:

- **Pairs of** *adjacent* **junctions (junctions connected by a line)** *constrain* **each other's interpretations!**

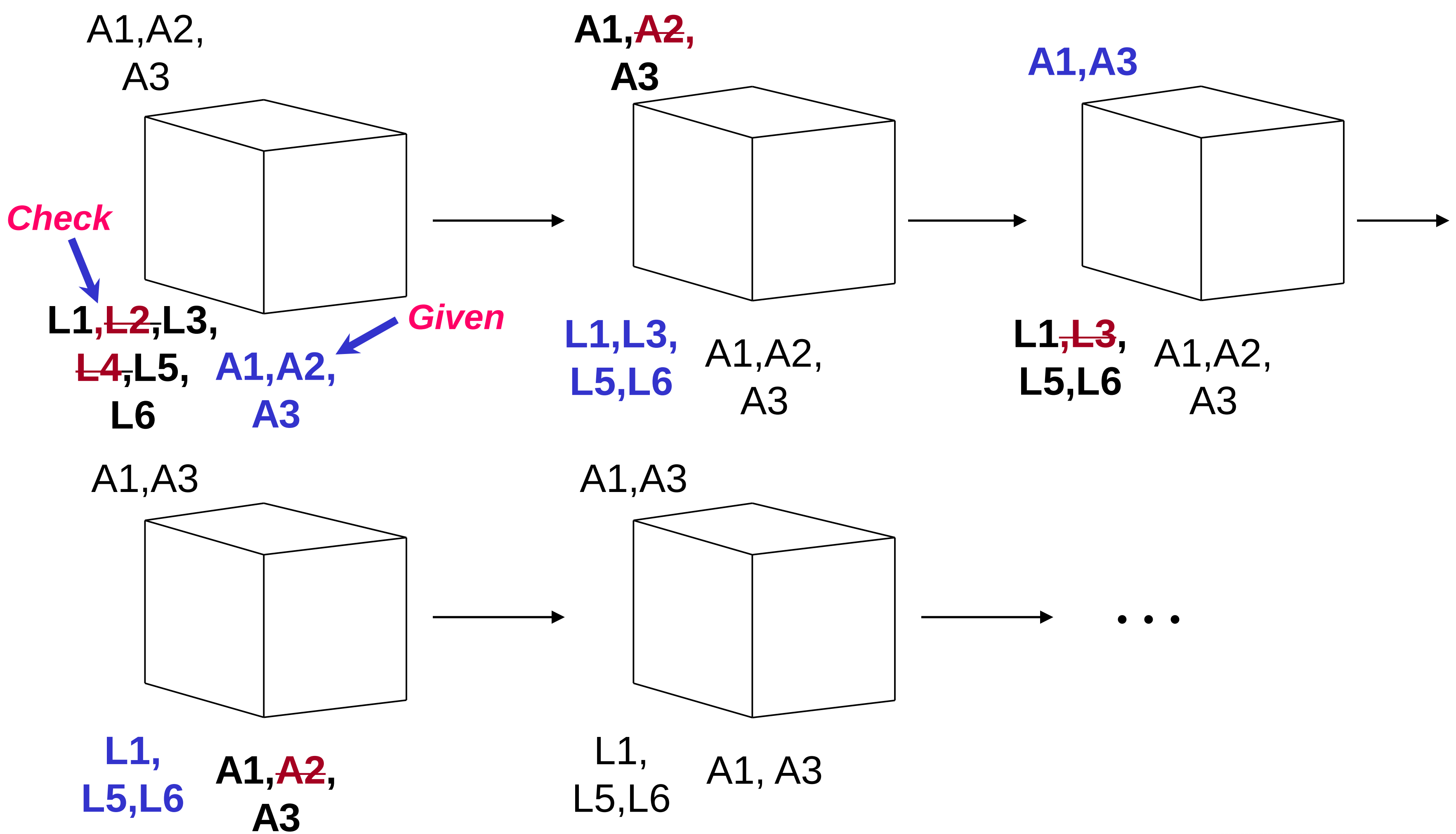
- **By** ***iterating*** **over the graph, the edge-consistency** *constraints* **can be** *propagated* **along the connected edges of the graph.**

- ***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 for line labeling

1. **Assign *every* junction in the picture a set of *all* Huffman/Clowes junction labels for that junction type;**
2. **Repeat until there is no change in the set of labels associate with any junction:**
    3. For each junction i in the picture:
        4. For each neighboring junction j in the picture:
            5. Remove any junction label from i for which there is no edge-consistent junction label on j.

Waltz/Mackworth: An example



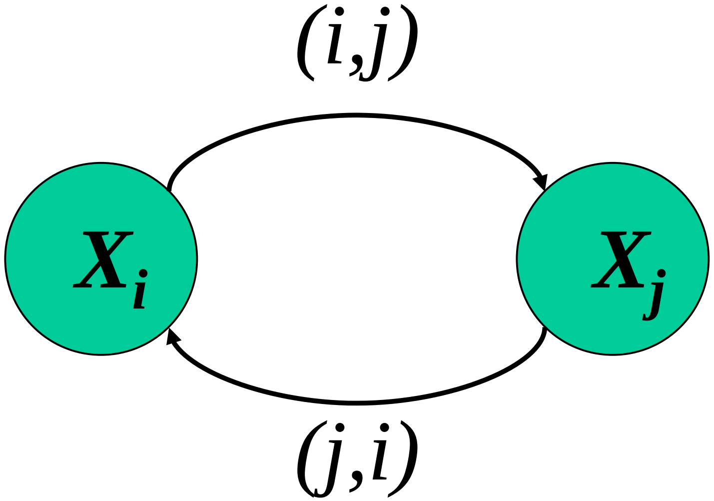
Inefficiencies: Towards AC-3

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 edge 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 edge-inconsistent is with respect to X_i itself. Thus, we only need to check that** *the labels on the pair (j,i)* **are still consistent.**

These insights lead a much better algorithm...

Directed arcs, Not undirected edges

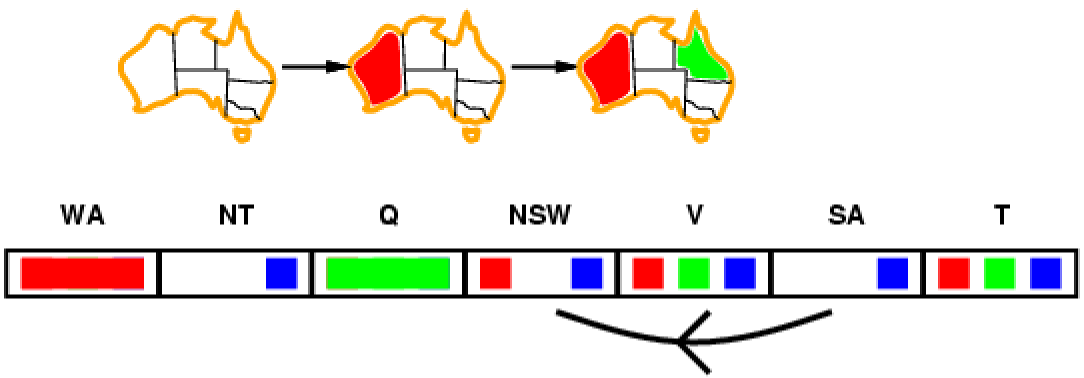
- **Given a pair of nodes X_i and X_j in the constraint graph connected by a constraint *edge*, we represent this not by a single undirected edge, but a *pair of directed arcs*.**
- **For a connected pair of junctions X_i and X_j, there are *two* arcs that connect them: *(i,j)* and *(j,i)*.**



Arc consistency: the general case

- **Simplest form of propagation makes each arc consistent**
- ***X* → *Y* is consistent iff**

  for every value *x* of *X* there is some allowed *y* in *Y*



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

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)

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***, so for our pictures, the time complexity is only** $O(nd^3)$

- *The constraint graph for line drawings is isomorphic to the line drawing itself, so is a planar graph.*

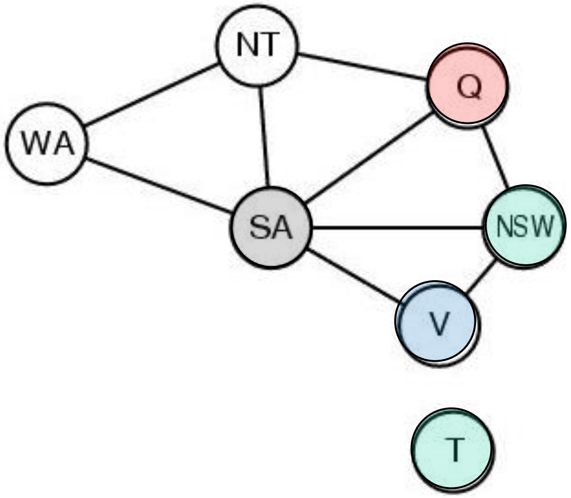
Beyond binary constraints: Path consistency

- 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

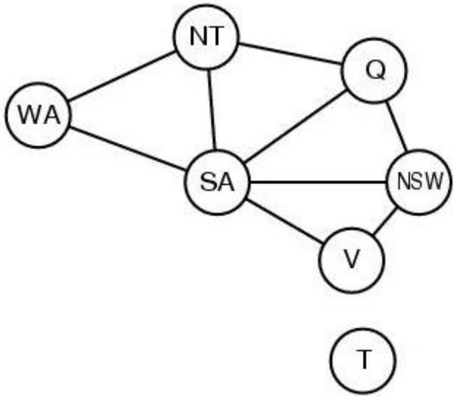
Chronological backtracking

- **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

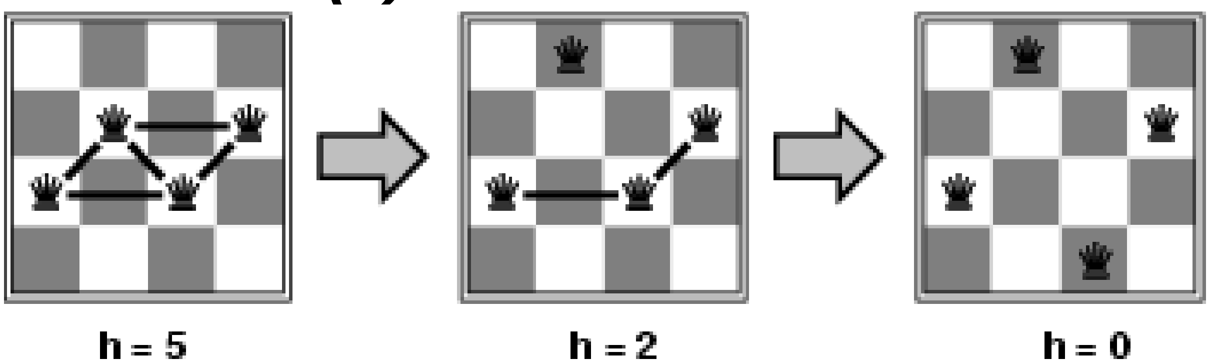


Local search for CSPs

- **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)**

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

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: $O(d^c * d^2(n-c)) = O(d^{c+2}(n-c))$