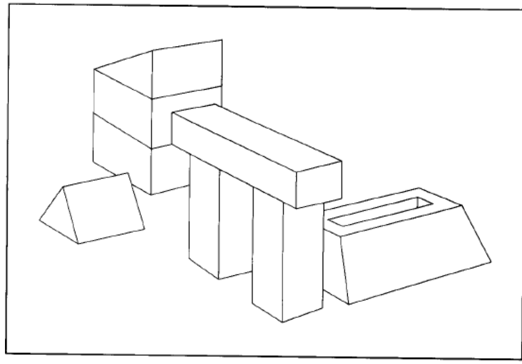
Constraint Satisfaction

CSCE 420 – Fall 2023

read: Ch. 6

Constraint Satisfaction

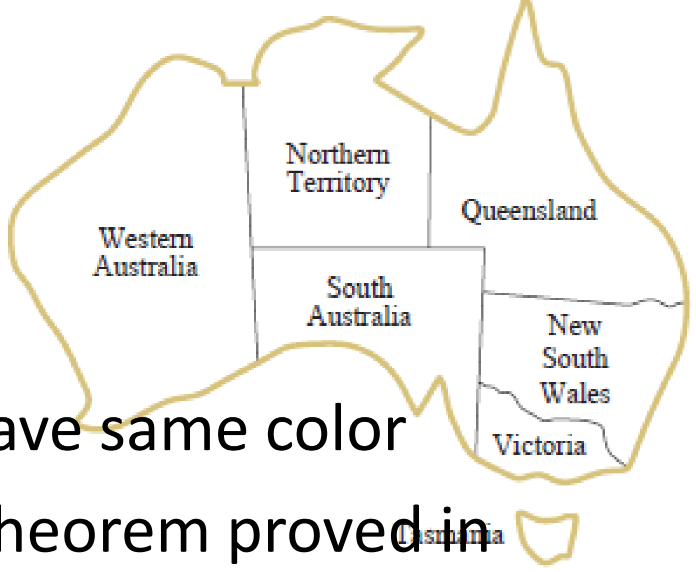
- Constraint Satisfaction Problems (CSPs) are a wide class of problems can be solved with specialized search algorithms
- these types of problems typically required finding a configuration of the world that satisfies some requirements (constraints) which restrict the possible solutions
- examples:
  - limited resources that can only be used one at a time
  - satisfying precedence order constraints (e.g. taking prerequisite classes first)
  - assignments of agents to tasks based on capabilities
  - computer vision: parsing scenes into 3D objects after edge-detection (constraints about possible meetings of edges and corners and faces vs background patches)



Constraint Satisfaction

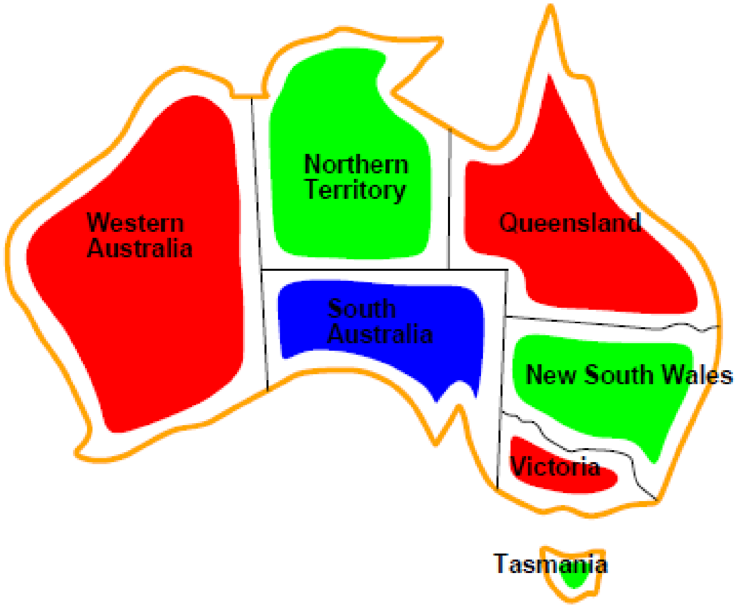
- formal framework:
  - <u>variables</u>: $\{V_i\}$
  - <u>domains</u>: $dom(V_i)=\{a_1...a_n\}$ – a *finite* set of possible values for each variable
  - <u>constraints</u>:
    - the form of constraints can be different for each problem
    - sometimes they are presented as equations
    - examples (binary constraints) : U+V=6; U and V must be opposite parity: $(U\%2)\neq(V\%2)$
    - abstractly, a constraint involving variables can be viewed as a restriction on the allowed set of tuples in the cross-product of domains:
      - constraint $C_j =\{<x_1...x_n>|x_k\in dom(V_k)\}\subset\Pi_{k=1..c}\ dom(V_k)$
      - dom(U)=dom(V)={0,1,2,3,4,5,6,7,8,9}
      - U+V=6: $\{<0,6>,<6,0>,<1,5>,<5,1>,<4,2>,<2,4>,<3,3>\} \subset \{<0,0>,<0,1>,...<0,9>,<1,0>,<1,1>,<1,2>....<9,9>\}$ (100 possible 2-tuples)
  - <u>solution</u>: a *complete variable assignment* that satisfies all constraints
    - for some CSPs, there can be multiple solutions

CSP Example: Map coloring



- no two adjacent states (sharing part of an border) can have same color
- (in general, need at most 4 colors – famous Four Color Theorem proved in 1997 with the help of a computer to enumerate all possible cases)
- Australia:
  - vars = {WA,NT,SA,Q,NSW,V,T}
  - domains: dom(S)={R,G,B}
  - constraints: WA≠NT,WA≠SA,NT≠SA,NT≠Q...

  - solution: {WA=R,NT=G,SA=B,Q=R,NSW=G,V=R,T=G}
  - also: {WA=G,NT=R,SA=B,Q=G,NSW=R,V=G,T=R}
  - and so on



CSP Example: Cryptarithmetic

```
     c2 c1
      T W O          7 6 5
    + T W O        + 7 6 5
    -------        -------
    = F O U R      = 1 5 3 0
```

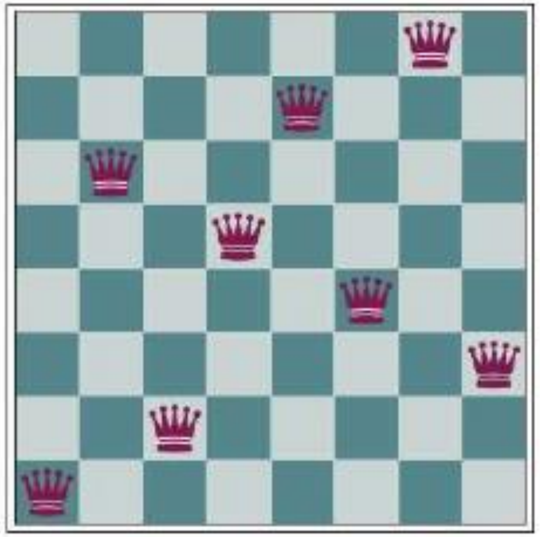
- vars: {F,T,W,O,U,R}
  - and add carry bits {c1,c2}
- domains: dom(var)={0,1,2...9} (digits)
  - domain for c1 and c2 is just {0,1}
- constraints:
  - all var bindings must be distinct: F≠T, F≠W...
  - leading chars can't be 0: T≠0, T≠0
  - the math must add up correctly:
    - O+O=R - what if there is a carry? introduce c1, dom(c1)={0,1}
    - O+O=R-c1*10
    - c1+W+W=U-c2*10
    - C2+T+T=U-F*10

a solution:
F=1
T=7
W=6
O=5
U=3
R=0

are there other solutions?

```
      S E N D
    + M O R E
    ---------
    = M O N E Y
```

CSP Example: 8-queens



- assume there is one queen in each column
- for each column i, what row is the queen in?
- vars: $Q_1..Q_8$
- domains: $Q_i \in \{1..8\}$
- constraints:
  - no 2 queens can be in same row: $Q_i \neq Q_j$ for all $i \neq j$
  - no 2 queens can be in same diagonal: $|Qi\text{-}Qj| \neq |i\text{-}j|$
  - equivalent representation:
    - allowed Q1-Q2 pairs: {(1,3),(1,4),(1,5)...(1,8),(2,4)...(2,8),(3,1),(3,5)...(3.8)...}
    - allowed Q1-Q3 pairs: {(1,2),(1,4),(1,5)...(1,8),(2,1),(2,3),2,5)...}

CSP Example: scheduling

- Job Shop scheduling
  - car assembly tasks: install axles (2), install wheels (4), tighten bolts (4), put on hubcaps(4), inspection (1)
  - variables: time steps for each task (integers): T_{axleF} , T_{axleR} , $T_{wheelFR}$ ...$\in$[1..20] (time limit)
  - precedence constraints: $T_{axleF}<T_{wheelFR}<T_{nutFR}< T_{inspection}$
  - (we could also model task durations)
  - solution: assignment of time slot for each step
    - T_{axleF}=1, $T_{wheelFR}$=2, $T_{wheelFL}$=3, T_{axleR}=4, ...$T_{inspection}$=15
- you can do the same thing with undergrad courses:
  - CSCE 313 is needed to graduate
  - CSCE 312 is a prerequisite for CSCE 313
  - only want to take at most 5 courses per semester
  - can you figure out a solution (assignment of courses to semesters) that satisfies all prereqs and will enable you to graduate in 4 yrs?

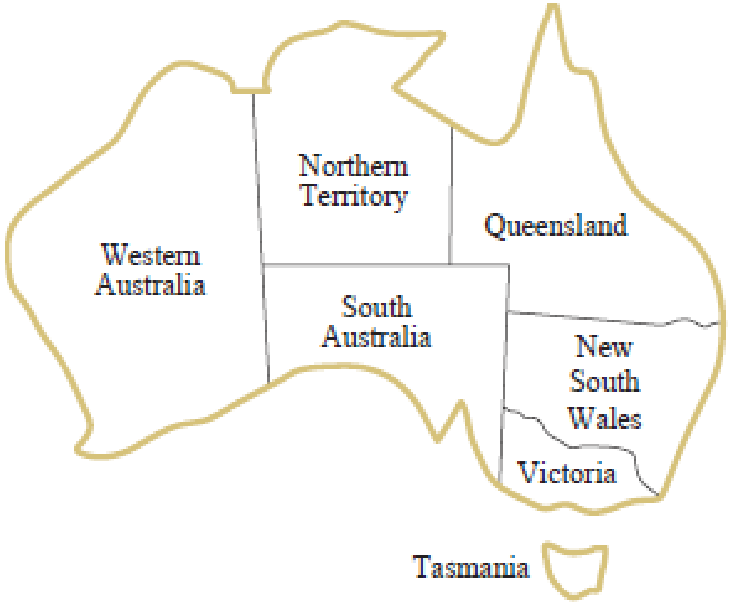
- note: Scheduling is a big field of computer science, and there are many variants of scheduling problems
- often, we want to know more that just whether there is a feasible solution: we want to find a schedule of minimum length (make-span)
- this goes beyond CSPs

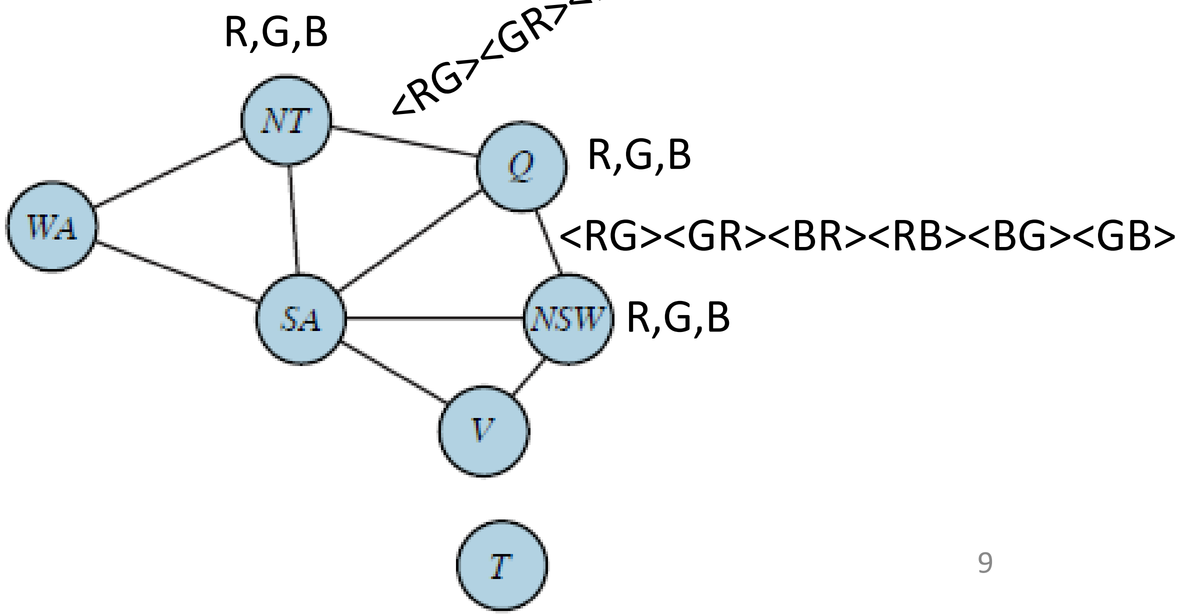
CSP Example: Jobs Puzzle

- There are four people: Roberta, Thelma, Steve, and Pete.
- Among them, they hold eight different jobs.
- Each holds exactly two jobs.
- The jobs are chef, guard, nurse, clerk, police officer (gender not implied), teacher, actor, and boxer.
- The job of nurse is held by a male.
- The husband of the chef is the clerk.
- Roberta is not a boxer.
- Pete has no education past the ninth grade.
- Roberta, the chef, and the police officer went golfing together.
- Who has what jobs?

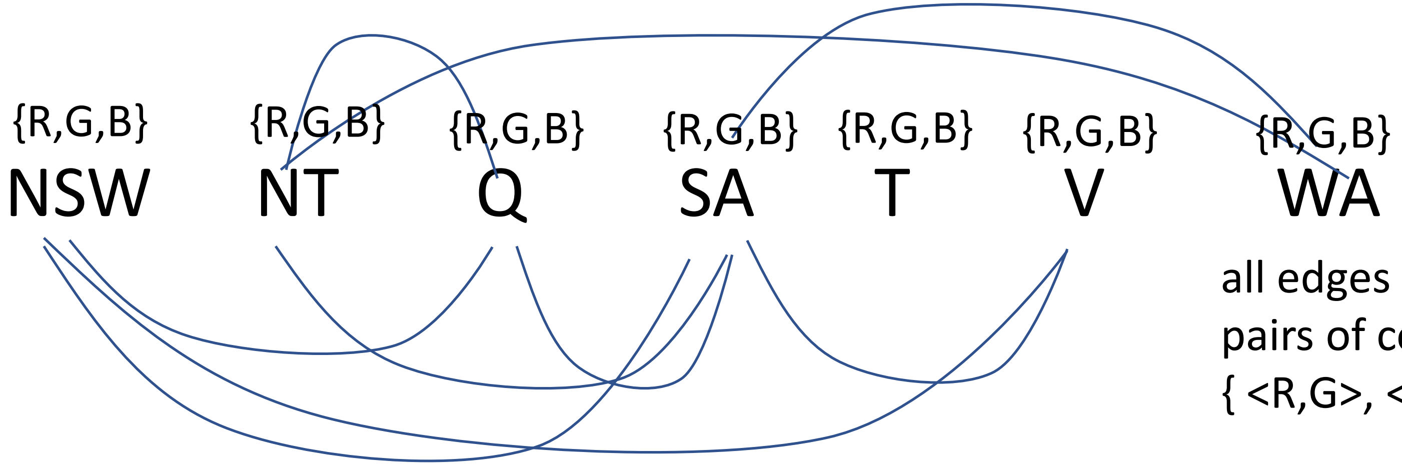
Constraint Graphs

- nodes=vars (label with domain, possible values)
- edges=constraints
  - easy for binary constraints
  - <u>label edges with pairs of consistent values from each domain</u>





- realistically, a computer would only process variables in given order (e.g. alphabetically): NSW, NT, Q, SA, T, V, WA
- it does not “know” the order that would be most useful
- the constraint graph really looks like this:

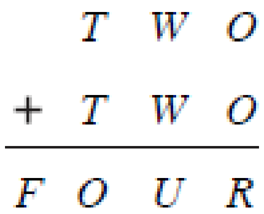


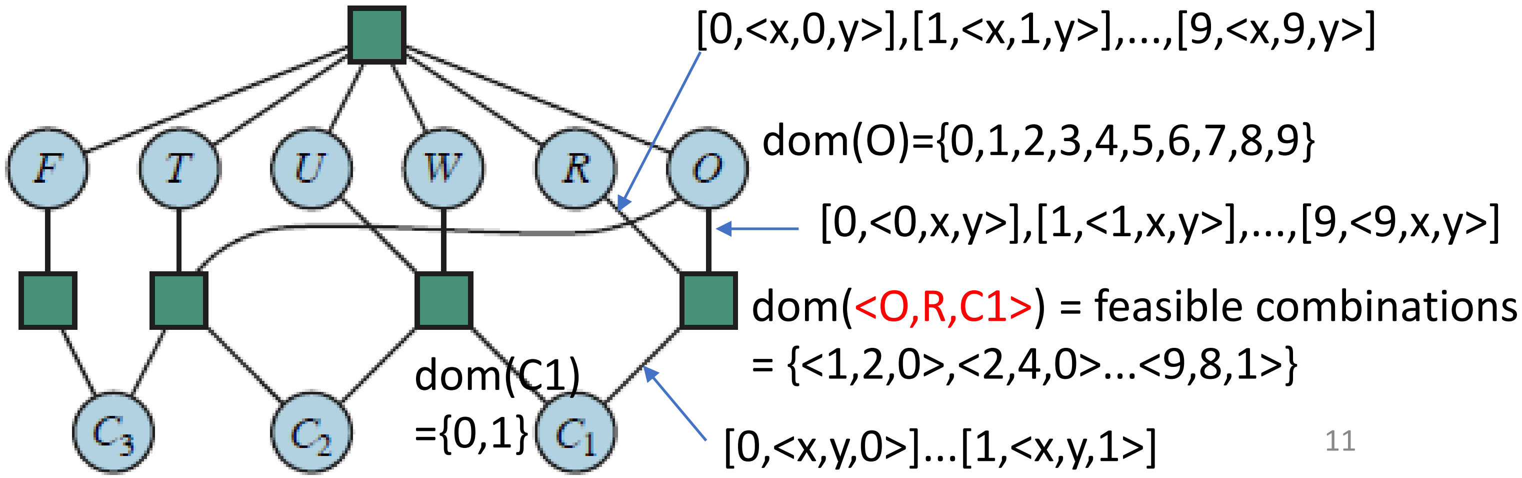
all edges are initially labeled with pairs of consistent values: e.g. { <R,G>, <R,B>, <G,R>, <G,B>, ...}

- would have to choose color for NSW first, then choose NT (no constraints to check), then choose Q
- then check consistency by looking at back-edges between Q-NSW, and Q-NT
- and so on...

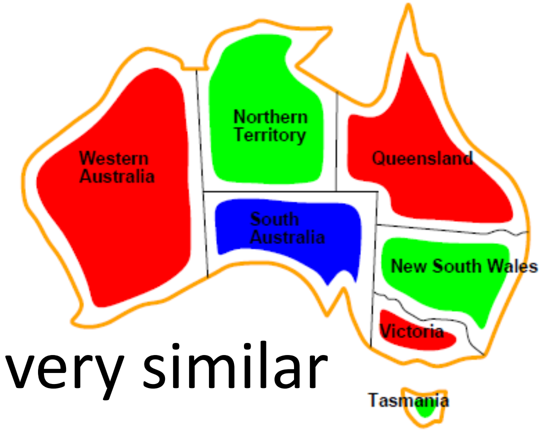
Constraint Graphs

- for ternary constraints (3 or more variables), e.g. O+O=R-c1*10
  - creates a "hypergraph" with special edges that connect ≥3 nodes (hard to draw)
  - convert to a binary graph:
    - create new nodes (green) for each constraint
    - label the new nodes with all possible tuples based on cross-product of domains
    - connect the new nodes to the constrained variables
    - label the edges to enforce consistency of variable assignment with position in tuple



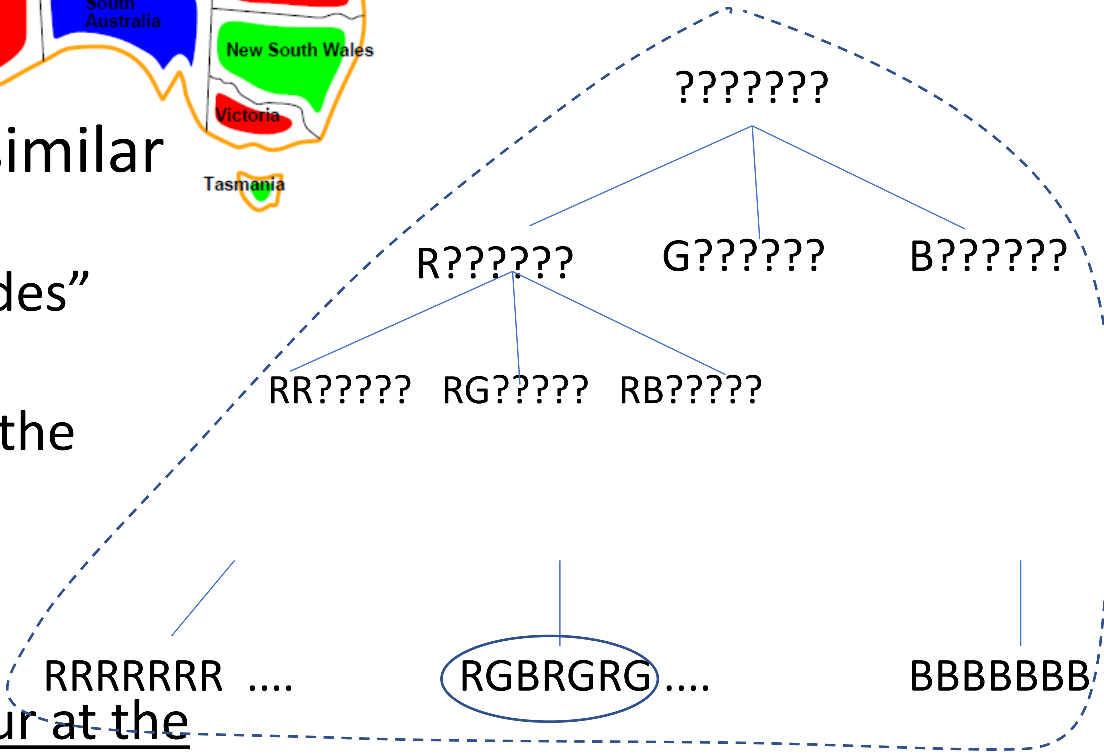


Back-tracking



vars: WA,NT,SA,Q,NSW,V,T
states: <c1,c2,c3,c4,c5,c6,c7>
where $c_i \in \{R,G,B,?\}$

- the basic search algorithm for CSPs is very similar to DFS
  - variable assignments represent "states" or "nodes"
  - the root node is the empty assignment
  - for a selected variable, the branches represent the choices from the domain
  - each level assigns one more variable
- there are two important differences:
  - tree depth is uniform (# vars), and all goals occur at the fringe
  - as soon as assigning any variable at an internal node causes inconsistency with a constraint, prune that subtree, and try next value in the domain
  - when a domain runs out of values, must backtrack to most recent choice-point



how many leave are there?

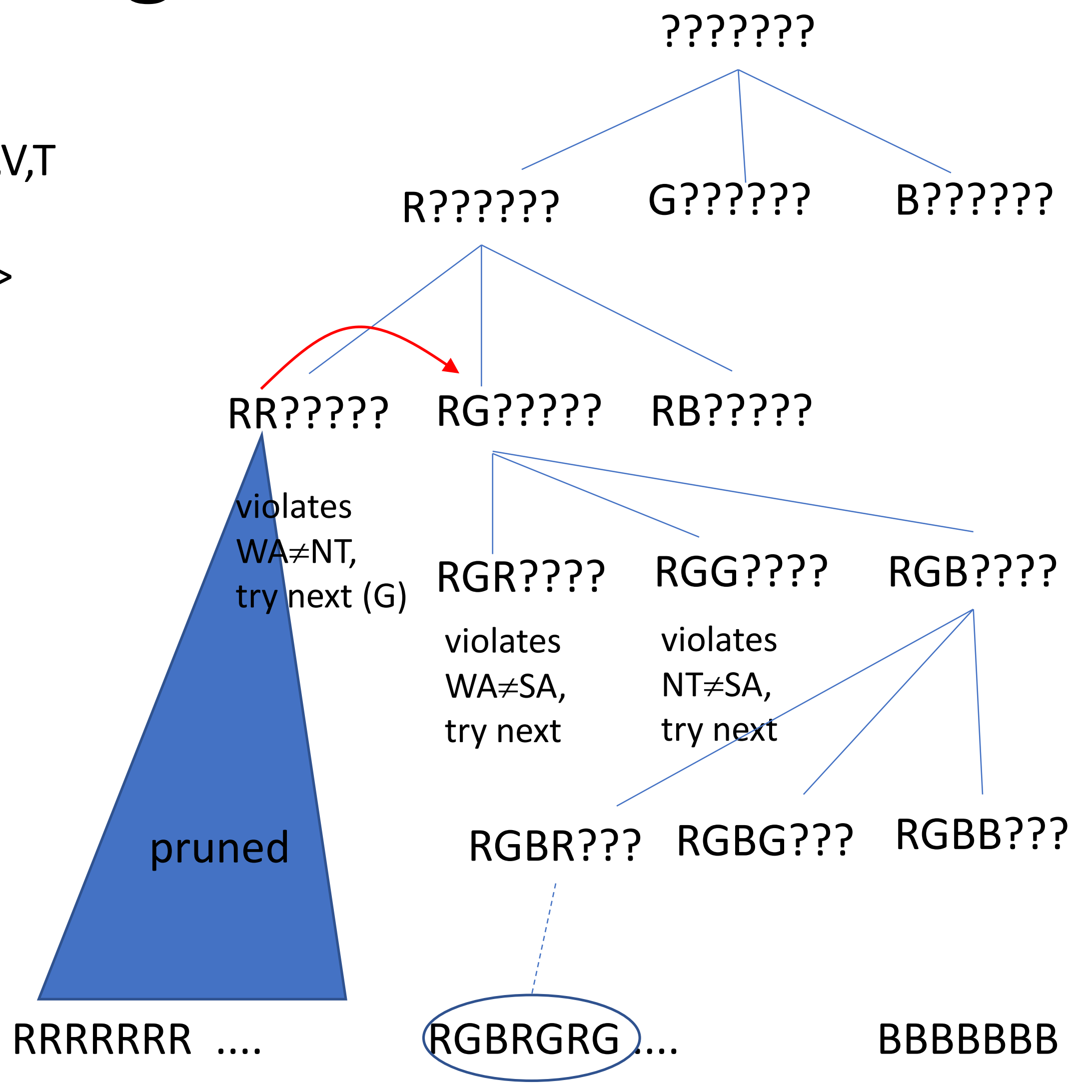
Back-tracking

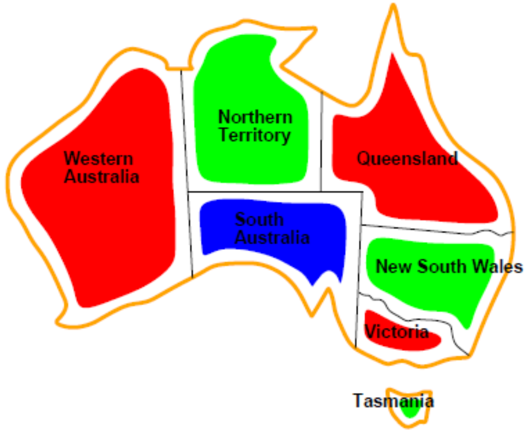
vars: WA,NT,SA,Q,NSW,V,T

state representation:

<c1,c2,c3,c4,c5,c6,c7>

where ci∈{R,G,B,?}





function BACKTRACKING-SEARCH(*csp*) **returns** a solution or *failure*
  return BACKTRACK(*csp*, { })

function BACKTRACK(*csp*, *assignment*) **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(*csp*, *var*, *assignment*) **do**
    if *value* is consistent with *assignment* **then**
      add {*var* = *value*} to *assignment*
      [redacted]
        result ← BACKTRACK(*csp*, *assignment*)
        if *result* ≠ *failure* **then return** *result*
      [redacted]
      remove {*var* = *value*} from *assignment*
  return *failure*

think of consistent(assignment) as a function you call on partial assignments to check if bound variables satisfy all known constraints

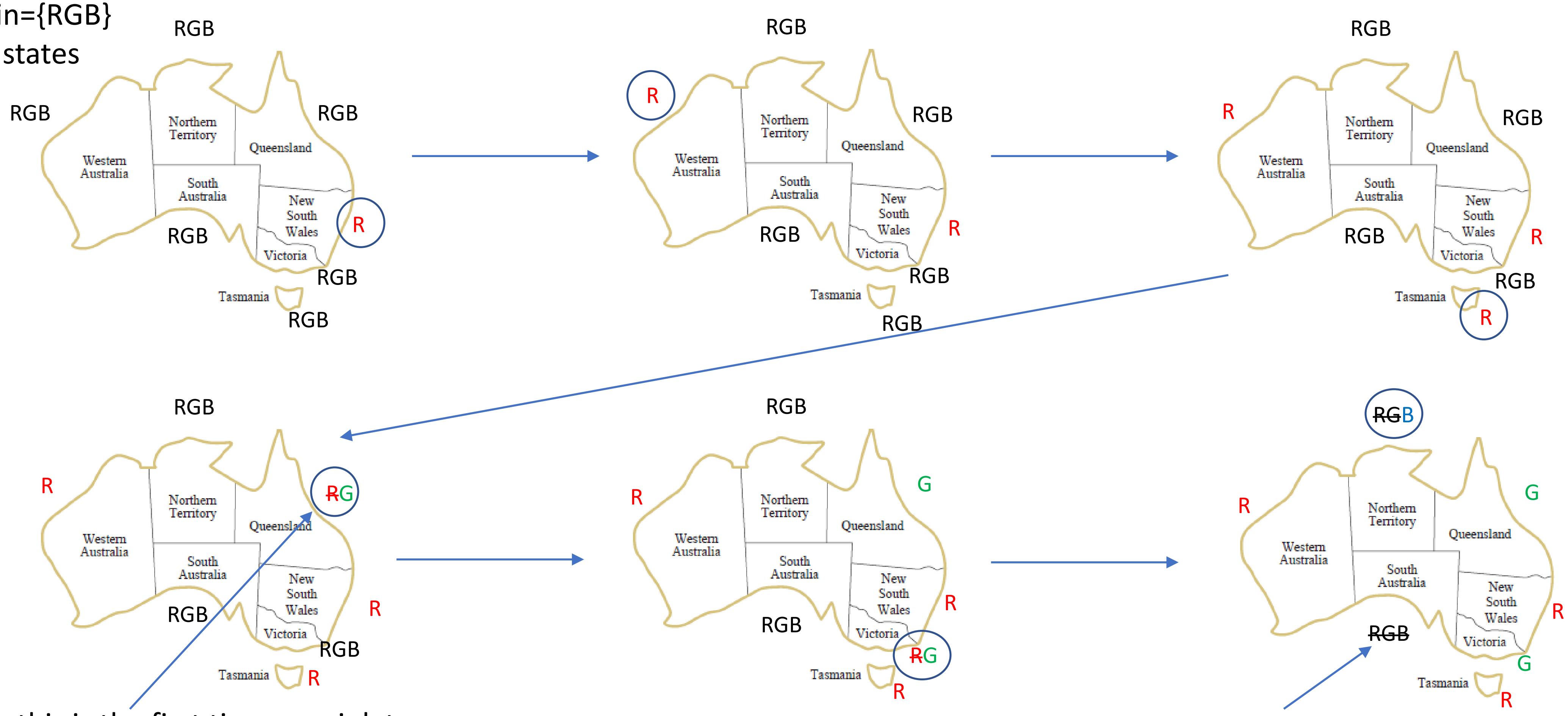
ignore inferences for now

recursion: bind more variables...

Tracing Backtracking

suppose the order of vars is given as: NSW, WA, T, Q, V, NT, SA

initially,
domain={RGB}
for all states

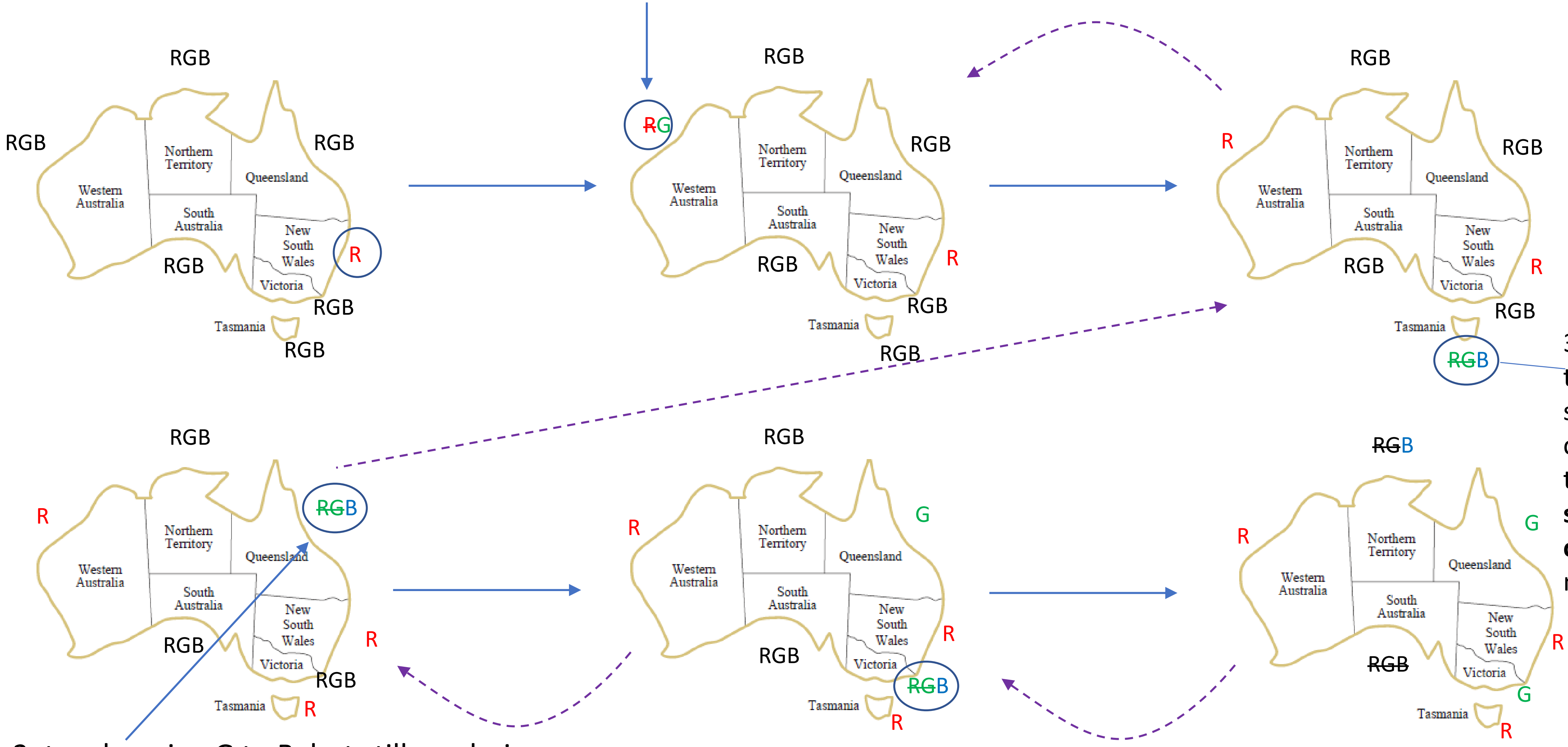


this is the first time we violate a, constraint, but only change R to G

crisis: no values remain for SA;
must back-track to WA (ultimately) and change it to G,
after trying all combinations of V, Q, and T

Tracing Backtracking

4. ultimately have to change this to G, and resume search

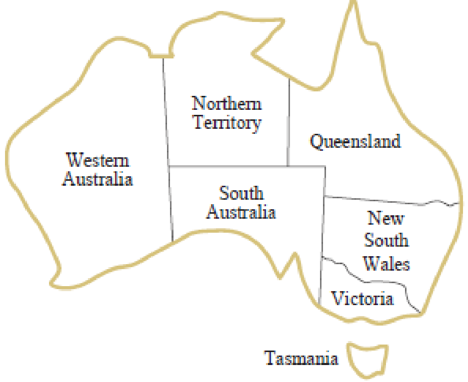


3. try G... then retry all subsequent choices; then try B **and all subsequent choices**; still no choices for SA

2. try changing G to B, but still no choices remain that lead to a consistent solution

1. no other choices remain for NT, so back track to V and try changing G to B; but NT is still B and SA still has no values

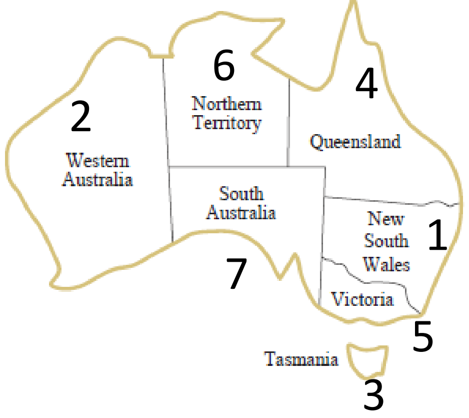
Alternative ways to Trace BT



suppose the order of vars is given as: NSW, WA, T, Q, V, NT, SA

step	NSW	WA	T	Q	V	NT	SA	explanation
	R							
	R	R						
	R	R	R					
	R	R	R	G				choose G because Q!=NSW
	R	R	R	G	G			choose G because V!=NSW
	R	R	R	G	G	B		
	R	R	R	G	G	B		back-track, no choices for SA are consistent
	R	R	R	G	B			change previous choice: V->B
	R	R	R	G	B	B		back-track again, no more choices for SA
	R	R	R	B				no more choices for V, so go back to Q->B
	R	R	R	B	G	G		back-track, no choices for SA (WA=R, NT=G, V=B)
	R	R	R	R	B			
	R	R	R	B				back up to Q and change to B
	...							

Alternative ways to Trace BT



- or you could write out the steps using indentation...
- suppose the order of vars is given as: NSW, WA, T, Q, V, NT, SA

```
try NSW=R
  try WA=R
    try T=R
      try Q=G (can't be red because of NSW)
        try V=G (can't be read because of NSW)
          try NT=B (because WA=R and Q=G)
            back-track; no consistent choices left for SA
          back-track; no choices left for NT
        change V->B
          try NT=B
            back-track, no choices left for SA
          back-track, no choices left for NT
        back-track, no choices left for V
      change V->B
        try V=G ...
```

instead of choosing next **var** arbitrarily (in order given), or we could use **MRV heuristic** to choose more intelligently...

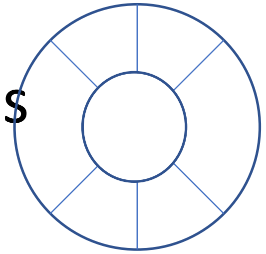
```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
  return BACKTRACK(csp, { })

function BACKTRACK(csp, assignment) 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(csp, var, assignment) do
    if value is consistent with assignment then
      add {var = value} to assignment

        result ← BACKTRACK(csp, assignment)
        if result ≠ failure then return result

      remove {var = value} from assignment
  return failure
```

instead of choosing next **value** arbitrarily (in domain order), or we could use **LCV heuristic** to choose more intelligently...

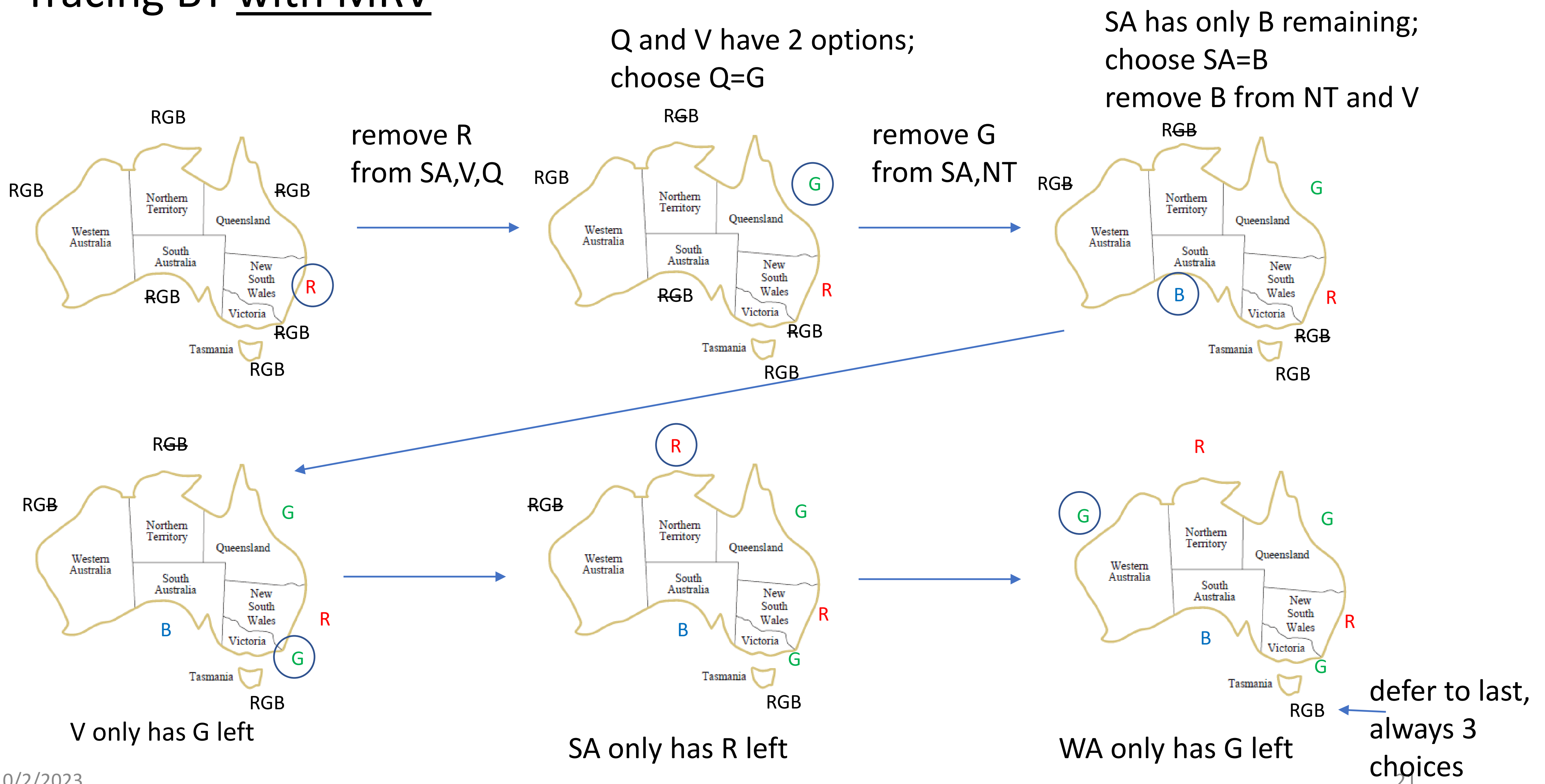
CSP Heuristics

- MRV – select var based on Minimum Remaining Values
  - in current partial assignment, some variable bindings might preclude choices in domains for unbound variables based on constrains
  - for each unbound variable, rule out values that are inconsistent with curr. assignment
  - choose variable with fewest choices
    - the best case: if there is a variable with just 1 choice left, choose it!
    - forces back-tracking to happen sooner
- LCV – select value for var based on Least Constraining Value
  - once a var is chosen, can we try the values in an intelligent order?
  - pick value that would remove the fewest (leave the most) choices for
  - this will tend to delay back-tracking to happen later
- degree heuristic: if all domains are equal-sized, choose the variable that is involved in the most constraints (connected to the most other vars)

<u>Food for thought:</u>
How much would MRV help in coloring the map of USA, compared to doing BT on 50 states in alphabetical order?



- Tracing BT with MRV



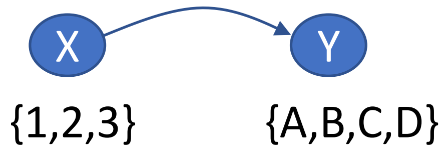
No back-tracking! notice how choices tend to *propagate* to neighbors

Forward-checking (FC)

- MRV is very similar to forward-checking
  - technically, MRV is passive; in each iteration, it re-calculates how many consistent values remain in domain of each unbound var
  - FC is active: every time you choose a value for a var, you remove inconsistent values in domains of other vars (like “propagation”)
  - almost identical, except... if making a choice at var X causes domain for var Y to become empty, back-track immediately and try another value for X (don’t have to wait till Y is selected to see that it’s domain is empty)

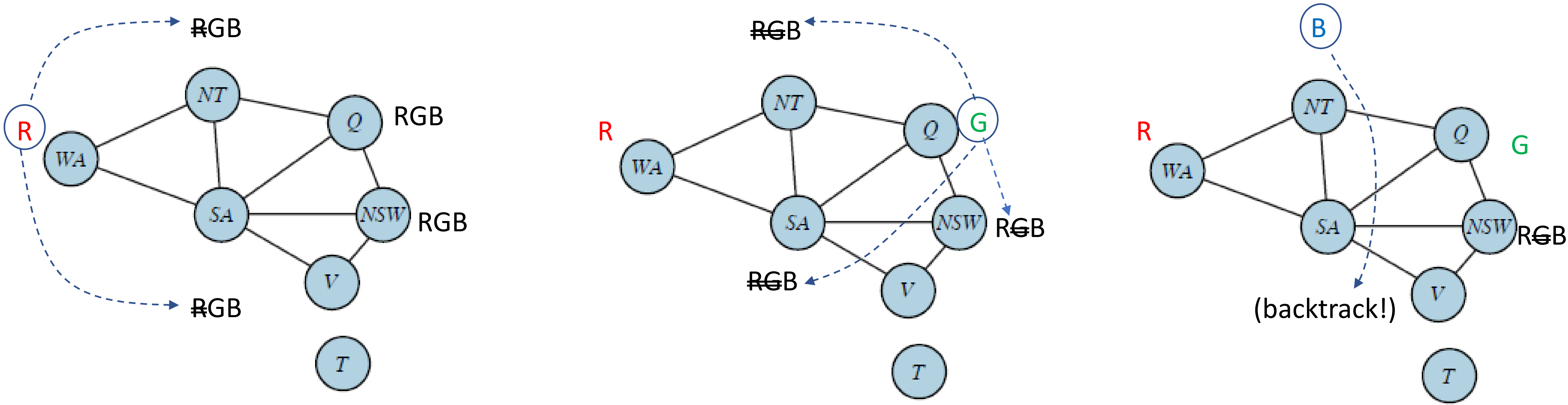
Constraint Propagation

- we can generalize the idea of FC
- whenever we make a choice at one node in the constraint graph, propagate the consequences to neighboring nodes
  - remember, edges are determined by constraints
- sometimes, a choice has <u>no effect</u> on domains of neighbors
- sometimes, choice at node X <u>removes some options</u> from domain of neighbor Y
- sometimes, choice at X <u>removes all but one option</u> at Y
  - if so, make this choice at Y, and propagate consequences to its neighbors...
- sometimes, choice at X <u>reduces the domain of neighbor Y to empty</u>, forcing back-tracking



Constraint Propagation

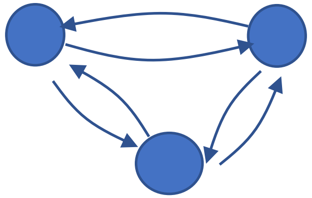
suppose we assign WA=R, and then Q=G,
and we are doing Forward checking...



why shouldn't we be able to propagate one *more* step and see that NT is forced to be B, leaving no choices for SA? (or vice versa)

AC-3

- formalization of constraint propagation as a *graph algorithm*
- let (V,E) be the constraint graph (assume all constraints are binary)
- define *arc-consistency*:
  - an edge X→Y is arc-consistent if, for every value *a* in the domain of X, there is a value *b* for Y that is consistent with X=a (i.e. satisfies the constraint)
  - for all edges (X,Y), $\forall$ a$\in$dom(X) $\exists$ b$\in$dom(Y) s.t. X=a and Y=b are consistent
  - a graph is arc-consistent if every edge is arc-consistent (bi-directionally for each constraint)
- ensure the initial graph is arc-consistent
- after making a choice for an initial var, it might rule out some choices in domains of neighbors, so must check that its neighbors are arc-consistent...
- put *edges* to be checked in a *queue*



```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  queue ← a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (Xi, Xj) ← POP(queue)
    if REVISE(csp, Xi, Xj) then
      if size of Di = 0 then return false
      for each Xk in Xi.NEIGHBORS - {Xj} do
        add (Xk, Xi) to queue
  return true
```

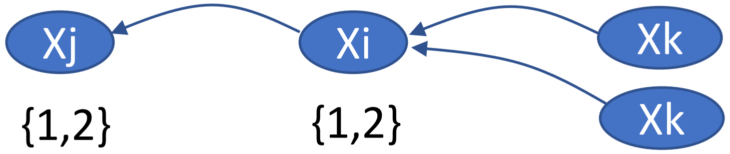
initialize queue with all directed edges between nodes

Revise() returns true if dom(Xi) was updated

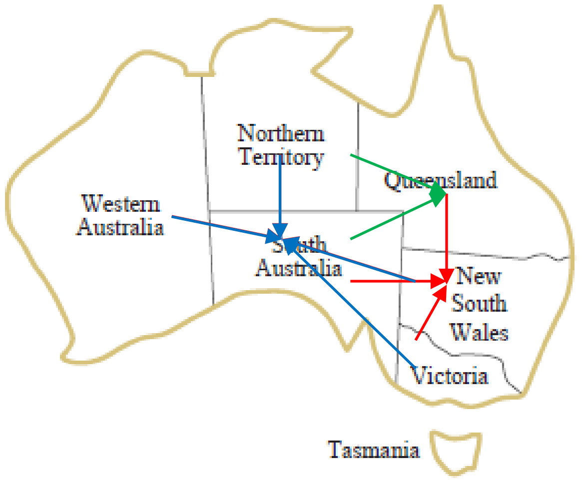
every time we delete a value from the domain of Xi, put the connected edges in the queue; note the reverse order: (X_k, X_i) – list the neighbors first

```
function REVISE(csp, Xi, Xj) returns true iff we revise the domain of Xi
  revised ← false
  for each x in Di do
    if no value y in Dj allows (x,y) to satisfy the constraint between Xi and Xj then
      delete x from Di
      revised ← true
  return revised
```

suppose the sum of Xi and Xj must be odd, and we remove 2 from dom(Xj)



Tracing AC-3



- suppose we start by choosing NSW=R
  - all edges connected to NSW must be checked for arc-consistency
- queue: {<Q,NSW>,<SA,NSW>,<V,NSW>}
  - pop <Q,NSW>,
  - R∈dom(Q) has no consistent value in dom(NSW)={R} so remove R from dom(Q);
  - but G,B∈dom(Q) each are consistent with R∈dom(NSW)
  - push neighbors of Q: <NT,Q>,<SA,Q> // note the reverse order of each pair
- queue: {<SA,NSW>,<V,NSW>, <NT,Q>,<SA,Q> }
  - pop <SA,NSW>, check each choice in dom(SA)={RGB} for a consistent choice in dom(NSW)={R}; remove R from dom(SA)
  - push neighbors of SA: <WA,SA>,<NT,SA>,<V,SA>,<Q,SA>
- queue: {<V,NSW>, <NT,Q>,<SA,Q>, <WA,SA>,<NT,SA>,<V,SA>,<Q,SA>}

Maintaining Arc Consistency (MAC)

- often, the initial graph is arc-consistent, so nothing to do
- after making first choice, run AC-3 till it quiesces
- usually the problem is not solved
    - a problem is solved when every node has just 1 value remaining
    - if some vars still have multiple values in their domains, we must make more choices
    - if any domain is empty, must back-track to previous choice point and try another value, followed by calling AC-3 to propagate consequences by reducing domains
- thus MAC is a *wrapper algorithm* around AC-3 that iteratively makes another choice and calls AC-3, till one of these two conditions is met

Maintaining Arc Consistency

```
MAC(graph G)
 if every node has exactly 1 val: return solution (complete assignment)
 if some node has no val, return fail (backtrack)
 choose a node V that still has multiple values in its domain
 for each value a in dom(V):
   G′ = G{V=a} // set node V to the value a
   G′′ = AC3(G′) // make graph arc-consistent based on this choice
   result = MAC(G′′) // recurse, try to extend this to a complete solution
   if result!=fail: return result
 return fail
```

path-consistency and *k*-consistency

- the concept of arc-consistency can be generalized to *path-consistency*
  - mutually consistent choice of values for 3 variables related by two constraints
  - suppose variables A, B, and C are involved in constraints, connected by edges A→B and B→C
  - $\forall$ a$\in$dom(A) and $\forall$ b$\in$dom(B), $\exists$ c$\in$dom(C) s.t. A=a, B=b, C=c are consistent
- can be generalized further to *k*-consistency (any sequence of k nodes)
  - however, the number of paths increases exponentially with *k*
  - so ensuring *k*-consistency in a graph takes more processing time
- in the limit: *n*-consistency (for all *n* nodes in graph) means every node has at least 1 choice consistent with some choice at every other node
  - a) if there is 1 value at every node, this is a unique solution for the CSP
  - b) if some nodes have multiple value, there might be multiple solutions; still have to run MAC to make some choices
  - c) it is possible there are no choices left for some vars: CSP is *infeasible* (has no solutions)

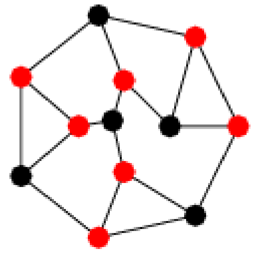
Complexity of AC-3

- what is the time-complexity of AC-3?
- assume there are 2*c* edges (num. of constraints, $c \leq n^2$), and d is the max domain size: $d = max(|dom(V_i)|)$
- an edge is only put in the queue whenever a value is deleted from the domain of a var
- so all edges will be processed at most *cd* times in total (calls to Revise())
- Revise() takes up to d^2 loop iterations to check for arc-consistency
- so AC-3 is $O(cd^3) = O(n^2d^3)$

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
   queue ← a queue of arcs, initially all the arcs in csp

   while queue is not empty do
      (Xi, Xj) ← POP(queue)
      if REVISE(csp, Xi, Xj) then
         if size of Di = 0 then return false
         for each Xk in Xi.NEIGHBORS - {Xj} do
            add (Xk, Xi) to queue
   return true

function REVISE(csp, Xi, Xj) returns true iff we revise the domain of Xi
   revised ← false
   for each x in Di do
      if no value y in Dj allows (x,y) to satisfy the constraint between Xi and Xj then
         delete x from Di
         revised ← true
   return revised
```

Computational Complexity of CSPs

- Theorem: Solving CSPs is NP-hard.
  - one can check whether a given variable assignment satisfies all constraints in polynomial time
- Theorem: Determining whether CSPs have a solution is NP-complete.
  - Proof: Graph Coloring can be reduced to CSP (CSP $\leftarrow$ graph 3-coloring $\leftarrow$ graph clique $\leftarrow$ 3-Sat)
  - we have already shown that graph-coloring can be transformed into a CSP in polynomial size
- thus *many* discrete problems can be encoded as CSPs
- *food for thought*: how would you encode Vertex Cover as a CSP?
  - does there exists a subset of k nodes that touches every edge?



Computational Complexity of CSPs

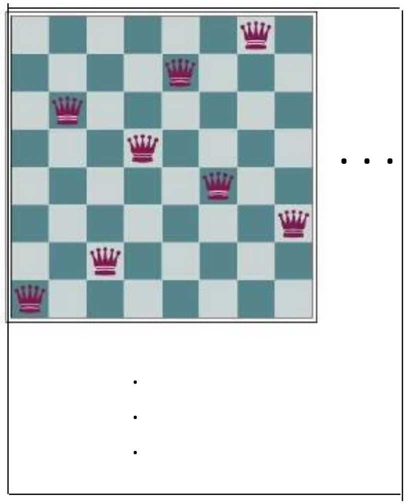
- how can CSPs be NP-complete if AC-3 runs in polynomial time, $O(n^2d^3)$?
  - we might have to call it an exponential number of times from MAC before we find a complete and consistent solution
- relation to Linear Programming (LP)
  - Linear Programs are like CSPs except they use continuous variables instead of discrete domains, and linear constraints
  - example:

    maximize $5x+3y-z$
    subject to $8x-7y \le 12$, $y+2z \le 1$, $0 \le x \le 2$, $0 \le y \le 10$, $0 \le z \le 2$
  - there exist polynomial time algorithms for LPs (e.g. Simplex Algorithm)
  - Mixed Integer-Linear Programs (MIPs): some variables are restricted to integers
  - Integer Programs (IPs) have all discrete values and can encode CSPs: IPs ↔ CSPs
  - discrete values makes solving constraints HARDER computationally
    - Linear Programming is in P
    - Mixed Integer Programming is in NP (actually NP-hard)

Min-Conflicts Algorithm

```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
          max_steps, the number of steps allowed before giving up

  current ← an initial complete assignment for csp
  for i = 1 to max_steps do
     if current is a solution for csp then return current
     var ← a randomly chosen conflicted variable from csp.VARIABLES
     value ← the value v for var that minimizes CONFLICTS(csp, var, v, current)
     set var = value in current
  return failure
```

- Local Search for CSPs
  - start by choosing a random variable assignment (which probably violates lots of constraints)
  - pick a variable at random and change its values to something that causes less conflicts
  - repeat until it "plateaus" (number of conflicts stops decreasing)
  - note: this is NOT guaranteed to find a complete and consistent solution!
  - but it works surprisingly well in practice
  - MinConflicts can solve the ***million-queens*** problem (on a 10^6x10^6 chess board) in a few minutes (!)

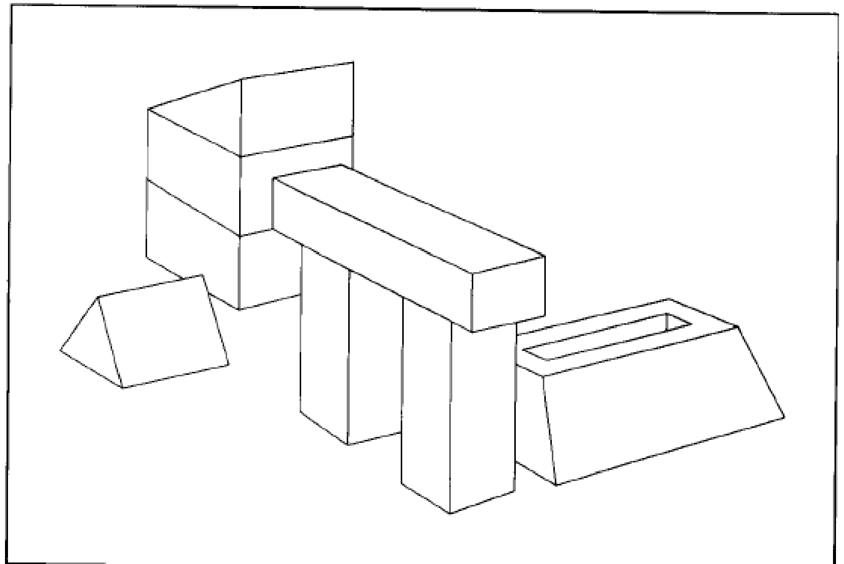


Application of CSP to Computer Vision

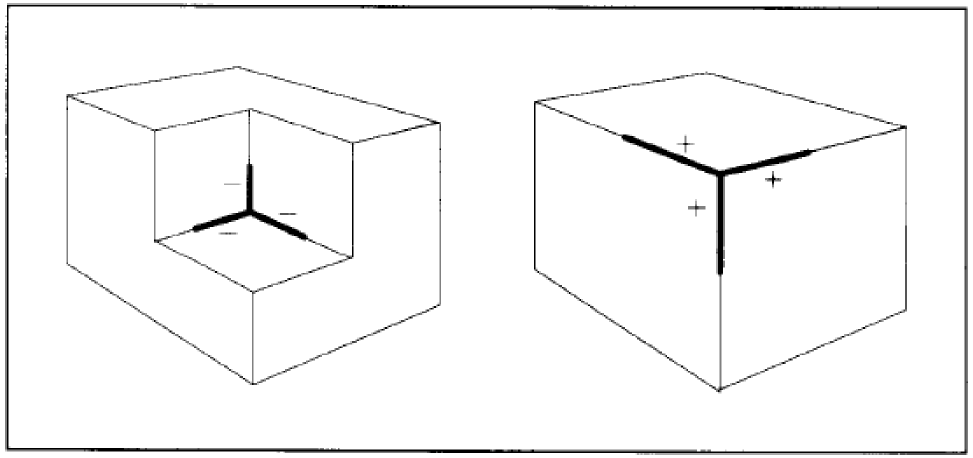
- 2D edge-detection → 3D object interpretation
- Waltz Constraint Propagation algorithm
- edges can be ambiguous – which side is part of object, vs background (or another object behind, i.e. occluded?)
- for any intersection of edge, there are only a finite number of possible labeling (for realistic 3D images)
- some 3-way intersections can be interpreted as corners

3D Image Interpretation and Waltz Propagation

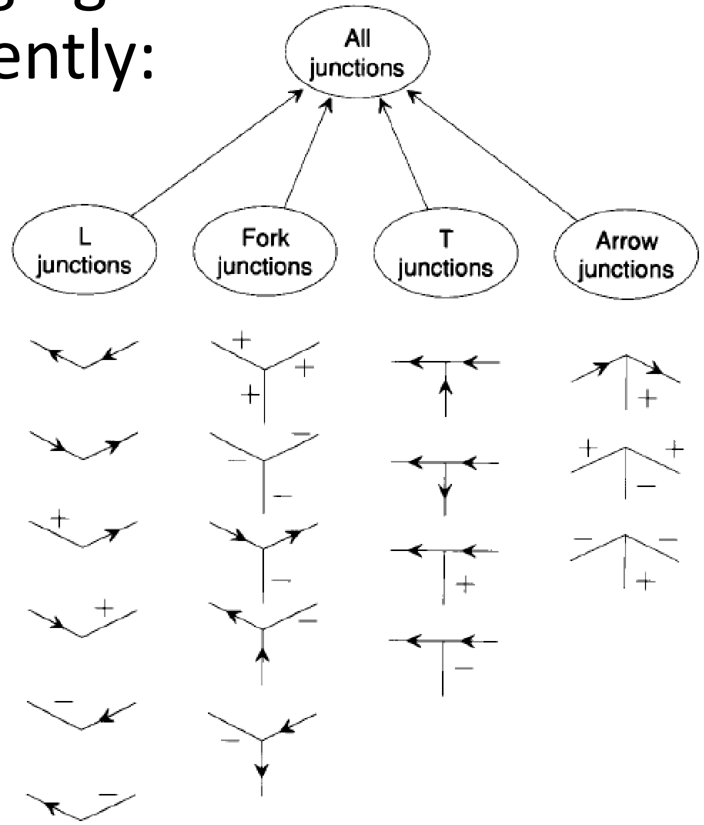
- from Ch. 12 in Patrick Winston (1984). Artificial Intelligence.
  - http://courses.csail.mit.edu/6.034f/ai3/ch12.pdf
- 2D image pre-processing: edge detection
  - Gaussian filter + segmentation
- how can you infer the 3D objects from line segments?
- how many object are there in this image?

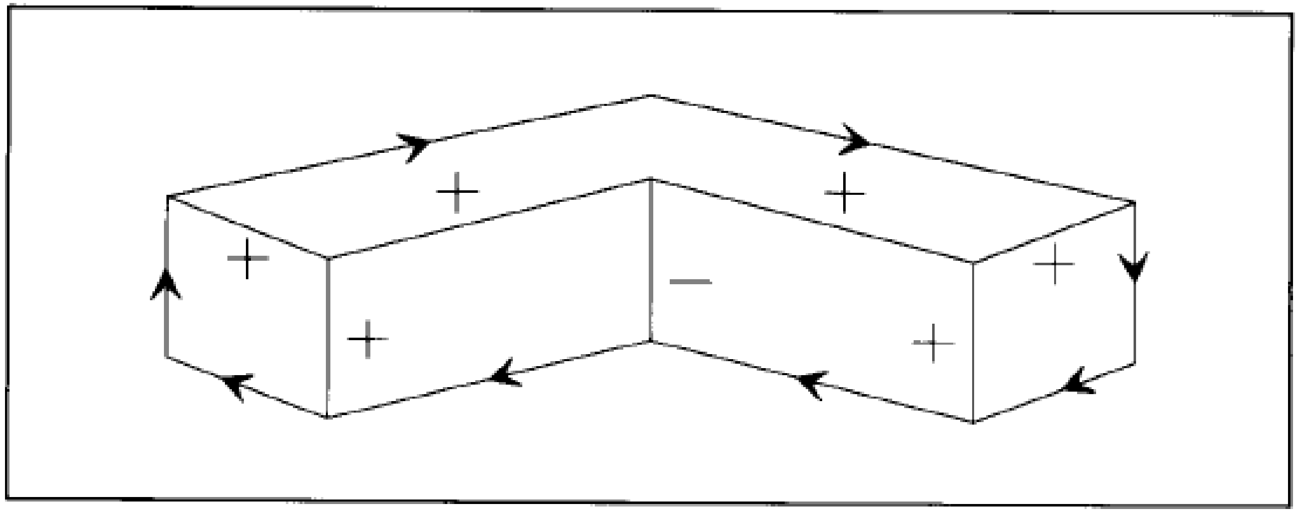


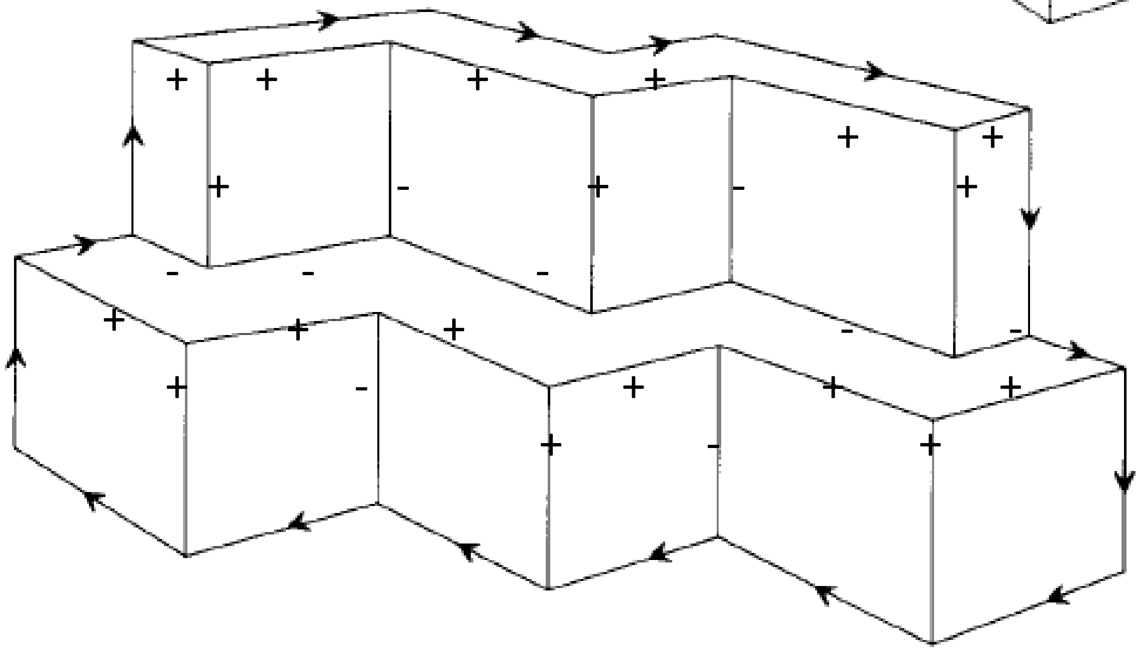
- lines are CSP variables with discrete labels:
  - + = convex
  - - = concave
  - ->- = boundary (between foreground and background; right-hand rule)

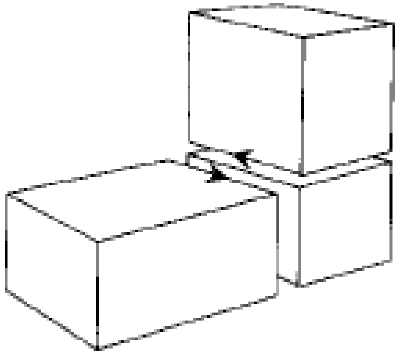


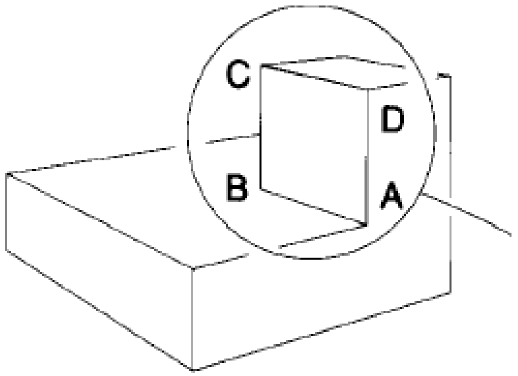
- junctions acts as constraints; converging lines must be labeled consistently:

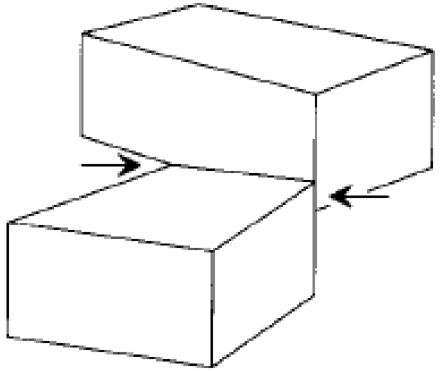


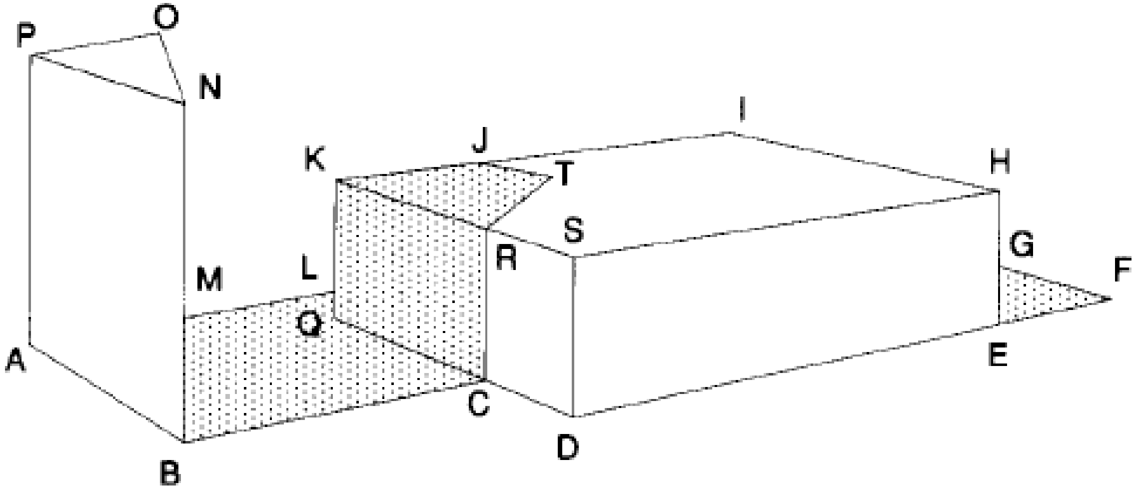












- The Waltz Propagation algorithm is a predecessor of modern Constraint Propagation, which can label these diagrams and extract 3D objects.
- Shadows, cracks, and coincident boundaries are challenges.