Lecture 11

Backtracking Algorithm II

Lusi Li
Department of Computer Science
ODU

Reading for This Class:
Chapter 6, Russell and Norvig

Review

- Last Class
  - Backtracking Algorithm I
  - Basic algorithms to solve CSPs

- This Class
  - Backtracking Algorithm II
  - Pruning space through propagating information

- Next Class
  - Games

Backtracking Search

- **Idea 1: Only consider a single variable at each point**
- **Idea 2: Only consider values which do not conflict with assignment made so far**
- **Depth-first search for CSPs with these two improvements is called backtracking search**

Backtracking search

function BACKTRACKING-SEARCH(*csp*) return a solution or failure

- **return BACKTRACK (*{}* , *csp*)**

function BACKTRACK (*assignment, csp*) return a solution or failure

- **if *assignment* is complete then return *assignment***
- ***var*** $\leftarrow$ **SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*],*assignment*,*csp*)**
- **for each *value* in ORDER-DOMAIN-VALUES(*var, assignment, csp*) do**
  - **if *value* is consistent with *assignment* according to CONSTRAINTS[*csp*] then**

    add *{var=value}* to assignment

    result $\leftarrow$ **BACKTRACK(*assignment, csp*)**

    if *result* $\neq$ ***failure*** **then return *result***

    else **remove {*var=value*} from assignment**
  - **end if**
- **end for**
- **return *failure***

Improving Backtracking Efficiency

- **For CSPs, general-purpose heuristic methods can give large gains in speed, e.g.,**
  - **Ordering:**
    - **Which variable should be assigned next?**
      - **minimum remaining values (MRV) heuristic**
      - **degree heuristic**
    - **In what order should its values be tried?**
      - **least constraining value heuristic**
  - **Inference (constraint propagation):**
    - **Can we detect inevitable failure early?**
  - **Structure:**
    - **Can we take advantage of problem structure?**

  - **They can be used in combination.**

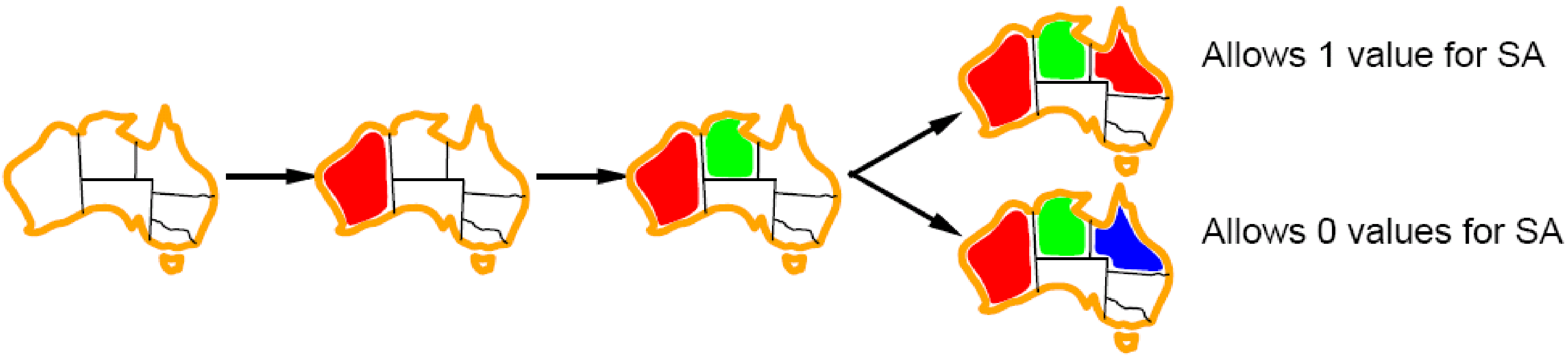
Variable Ordering

- **Static ordering**

- **Dynamic ordering**
  - **minimum remaining values (MRV) heuristic**
    - **choose variable with the fewest legal moves**
    - **“fail first”**
  - **degree heuristic**
    - **choose variable that is involved in the largest number of constraints on other unassigned variables**
    - **“tie-breaker”**

Value Ordering
Least constraining value (only)

(WA = red, NT = green, Q=red)



(WA = red, NT = green, Q=blue)

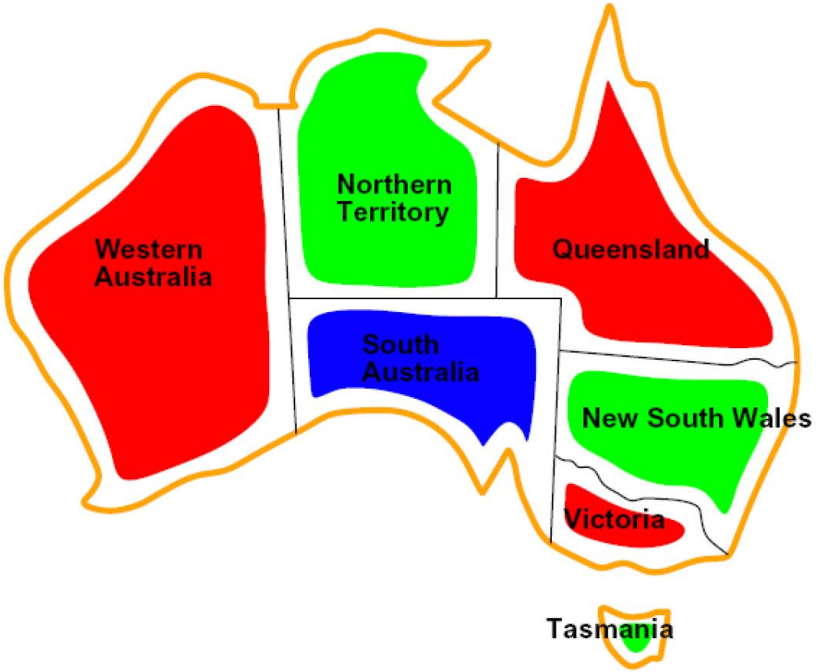
- **Heuristic for selecting what value to try next**
- **Heuristic Rule: given a variable, choose the least constraining value**
  - **the one that rules out the fewest values in the remaining variables**
  - **leaves the maximum flexibility for subsequent variable assignments**
- **"Fail last": make it more likely to find a solution early**
- **Implementation: keep track of remaining legal values for other unassigned variables**

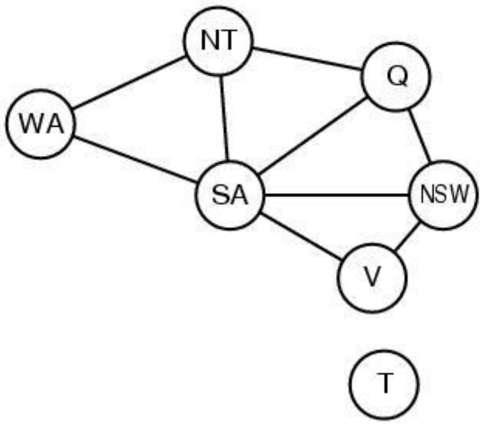
Ordering

- **Why fail first when selecting variables?**
  - Prune large portions of tree early on

- **Why fail last when selecting values?**
  - Only need one solution, so examine probable values first

Propagate Information



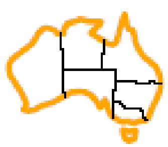


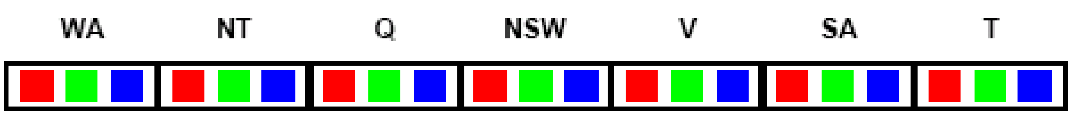
- If we choose a value for one variable, that affects its neighbors
- And then potentially those neighbors…
- We can use this inference to prune the search space by removing inconsistent values

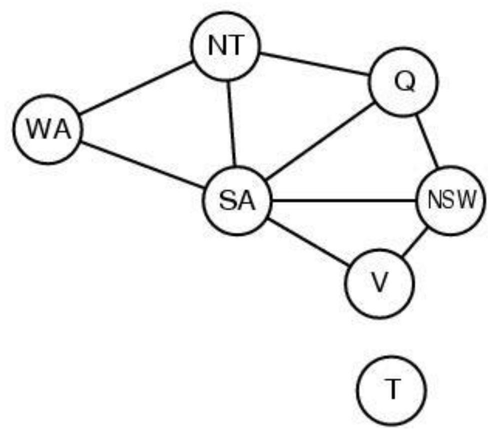
Inference: forward checking (only)

Idea:

- **keep track of remaining legal values for unassigned variables.**
- **ONLY check neighbors of most recently assigned variable after each assignment and remove any inconsistent values from its neighbors**
- **Backtrack when any variable has no legal values**





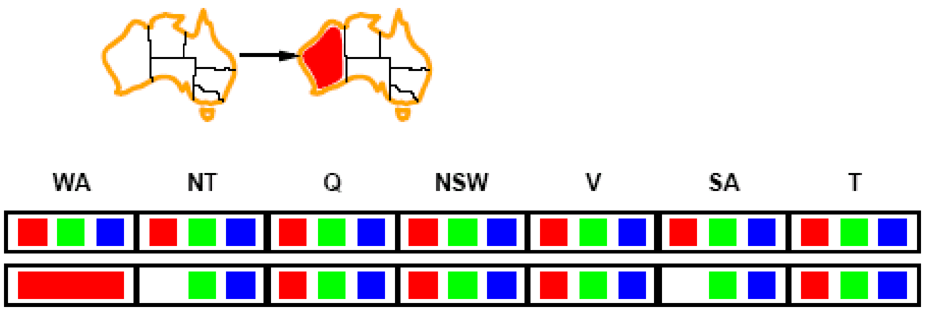


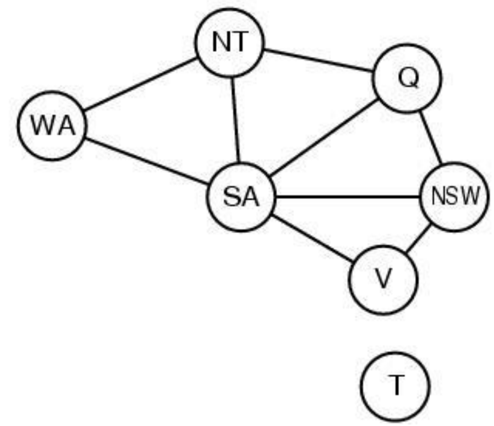
For the map coloring problem, suppose a variable and its value are selected randomly at each step.

Inference: forward checking (only)

Idea:

- **keep track of remaining legal values for unassigned variables.**
- **ONLY check neighbors of most recently assigned variable after each assignment and remove any inconsistent values from its neighbors**
- **Backtrack when any variable has no legal values**



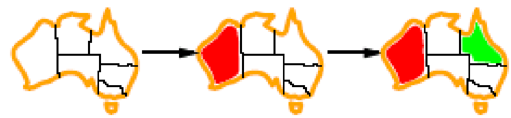


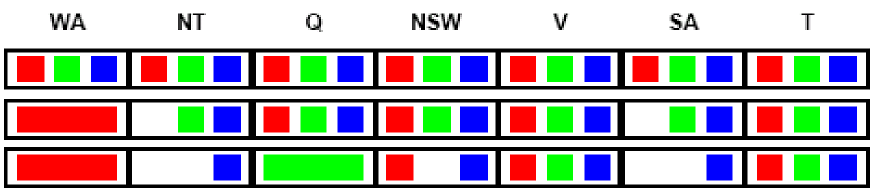
- **Assign *{WA=red}***
- **Effects on neighbors of WA**
  - ***NT can no longer be red***
  - ***SA can no longer be red***

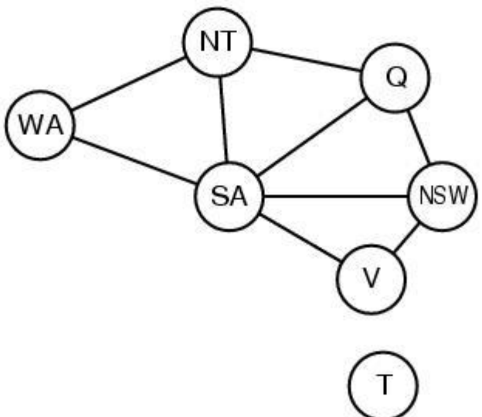
Inference: forward checking (only)

Idea:

- **keep track of remaining legal values for unassigned variables.**
- **ONLY check neighbors of most recently assigned variable after each assignment and remove any inconsistent values from its neighbors**
- **Backtrack when any variable has no legal values**







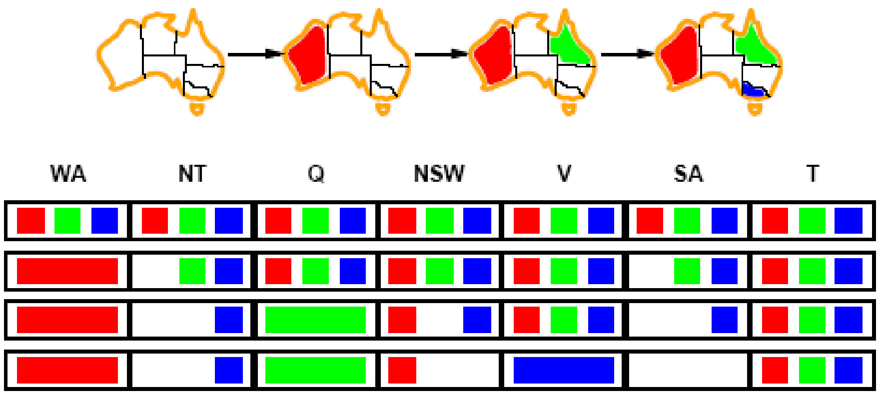
- **Assign *{Q=green}***
- **Effects on other variables connected by constraints with Q**
  - ***NT can no longer be green***
  - ***SA can no longer be green***
  - ***NSW can no longer be green***

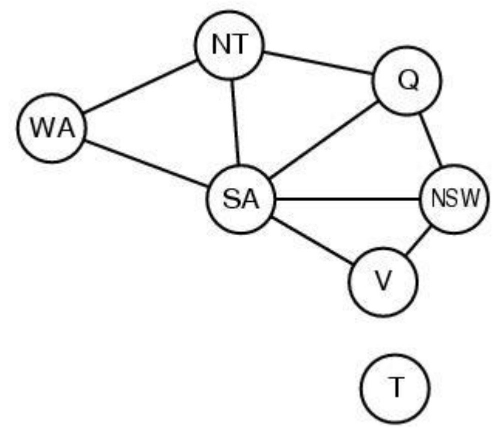
(We already have a failure, but FC is too simple to detect it now)

Inference: forward checking (only)

Idea:

- **keep track of remaining legal values for unassigned variables.**
- **ONLY check neighbors of most recently assigned variable after each assignment and remove any inconsistent values from its neighbors**
- **Backtrack when any variable has no legal values**



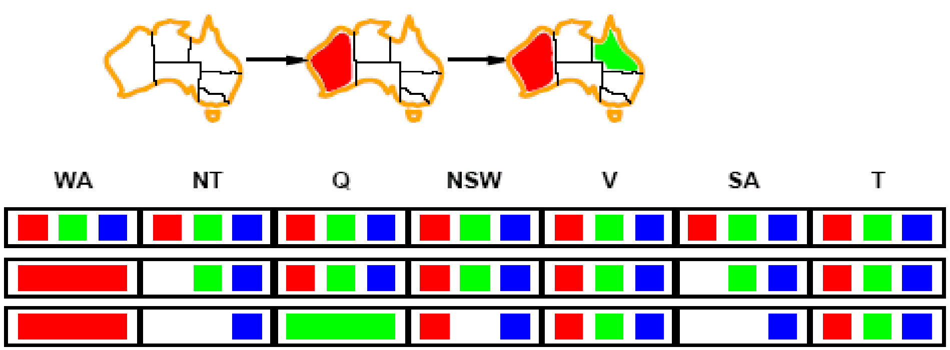


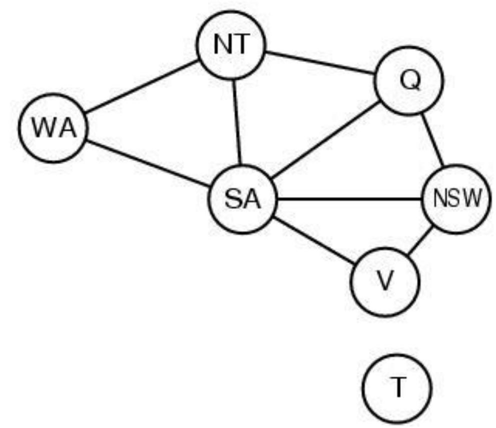
- **Assign *{V=blue}***
- **Effects on other variables connected by constraints with V**
  - ***NSW can no longer be blue***
  - ***SA is empty (no possible values!)***
- **Forward-checking has detected that SA has no legal values and backtracking can occur (restore the deleted values).**

Inference: forward checking (only)

Forward checking

- Solving CSPs with combination of heuristics plus forward checking is more efficient than either approach alone.
- Propagates information from most recent assigned to unassigned neighbors
- But, doesn't provide early detection for all failures
  - NT and SA cannot both be blue!





Constraint Propagation

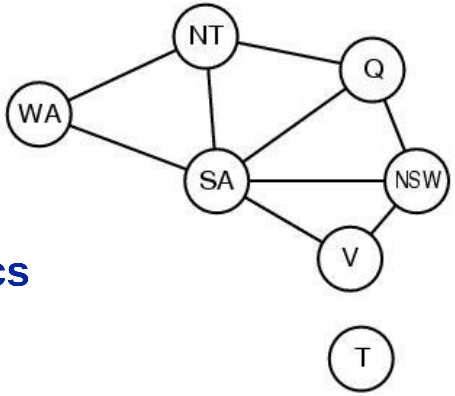
- Constraint propagation goes further than FC by repeatedly (recursively) enforcing constraints for all variables.
  - can detect failure earlier

- Main idea:
  - When you delete a value from a variable's domain, check all variables connected to it.
  - If any of them change, delete all inconsistent values connected to them, etc.

- Arc-consistency (AC) is a systematic procedure for constraint propagation

Arc consistency (only)

- An arc X→Y (connection between two variables X, Y in constraint graph) is consistent iff (iff = if and only if)
  - for every value x of X there is some value y of Y that is consistent with x
  - i.e., there is at least 1 value y of Y that allows (x ,y) to satisfy the constraint between X and Y



- AC-3 Algorithm for a binary CSP
  - Push all arcs X → Y on a queue
    - Each undirected constraint graph arc is two directed arcs
    - Undirected X−Y becomes directed X → Y and Y → X
    - X → Y and Y → X both go on queue, separately
  - Pop one arc X → Y and remove any inconsistent values from X
  - If any change in X, put all arcs Z → X back on queue, where Z is any neighbor of X that is not equal to Y
  - (If X has no legal values, AC detects failure and backtrack!!) during search
  - Continue until queue is empty

- Simplest form of propagation makes each arc consistent

Arc consistency algorithm

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

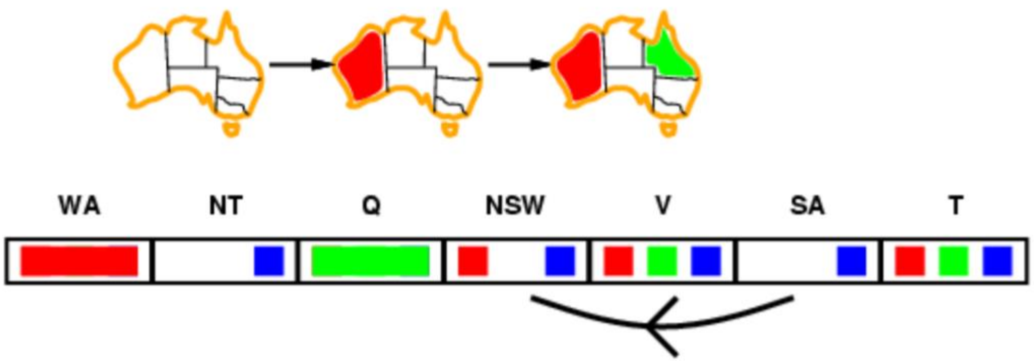
  while $queue$ is not empty **do**
    $(X_i,\ X_j) \leftarrow$ REMOVE-FIRST($queue$)
    if REVISE(csp, $X_i,\ X_j$) **then**
      if size of $D_i\ =\ 0$ **then return** $false$
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
        add $(X_k,\ X_i)$ to $queue$
  return $true$

---

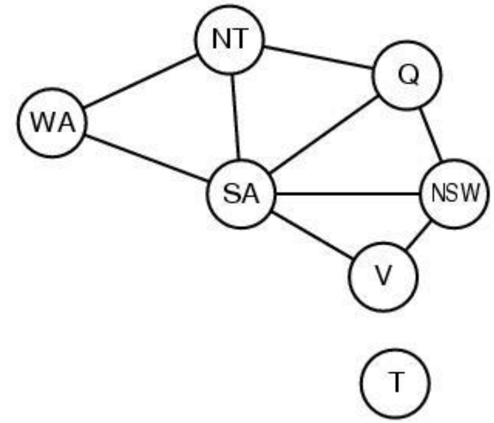
function REVISE(csp, $X_i,\ X_j$) **returns** true iff we revise the domain of X_i
  $revised \leftarrow false$
  for each x **in** D_i **do**
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j **then**
      delete x from D_i
      $revised \leftarrow true$
  return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

Arc consistency (only)

- **X → Y is consistent iff**
  - **for every value x of X there is some allowed y of Y**



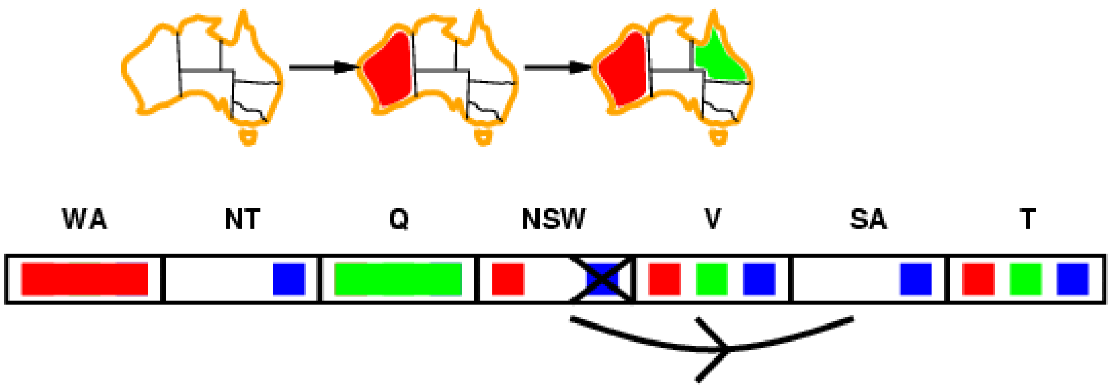
***X* = SA, *Y* = NSW**



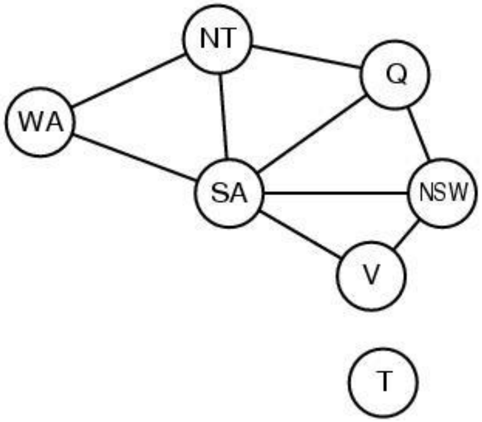
Consider the state after {WA = red, Q=green}
SA → NSW?
Consistent: because SA = blue and NSW = red satisfies all constraints on SA and NSW

Arc consistency (only)

- X → Y is consistent iff
  - for every value x of X there is some allowed y of Y
- Pop one arc X → Y and remove any inconsistent values from X



X = NSW, Y = SA



Consider state after {WA = red, Q=green}

NSW → SA?

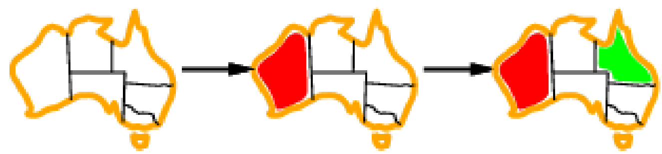
Inconsistent because when NSW = blue, SA has no valid value

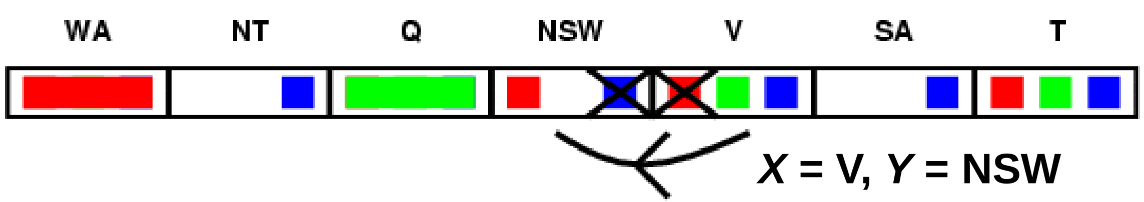
then remove blue from NSW's domain, **thus NSW → SA is consistent**

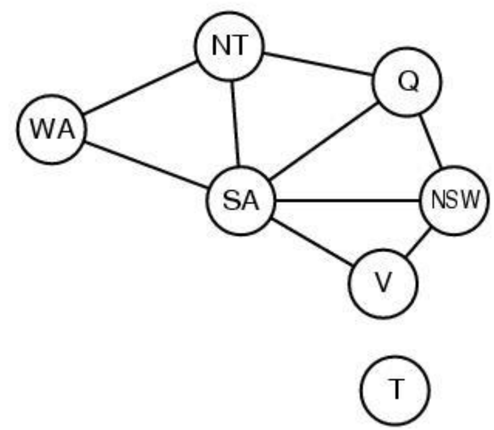
If X loses a value, neighbors of X need to be rechecked: put V → NSW on queue

Arc consistency (only)

- **X → Y is consistent iff**
  - **for every value x of X there is some allowed y of Y**
- **Pop one arc X → Y and remove any inconsistent values from X**
  - **If any change in X, put all arcs Z → X back on queue, where Z is any neighbor of X that is not equal to Y**



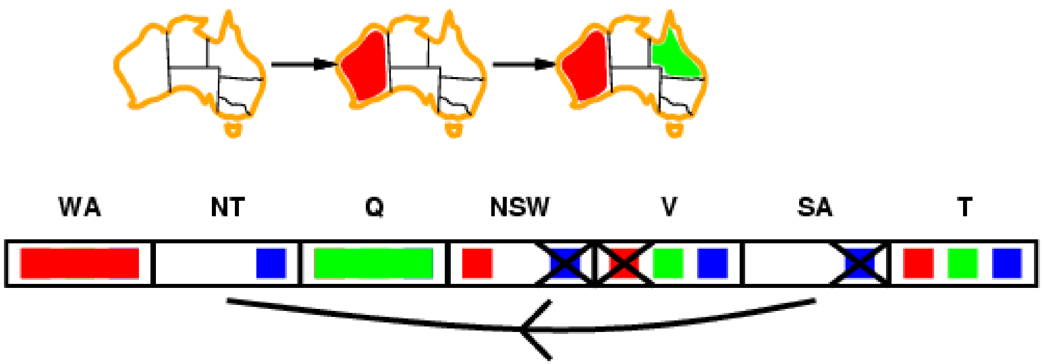




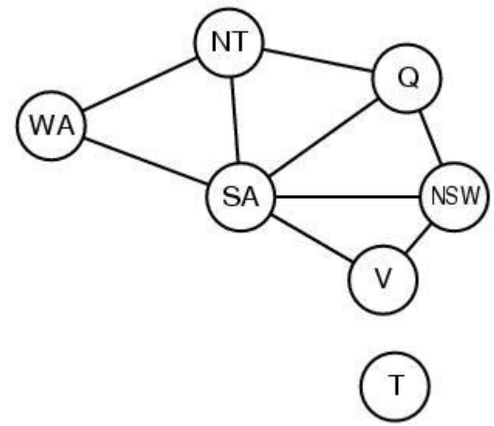
Consider state after {WA = red, Q=green}
V → NSW?
Inconsistent because when V = red, NSW has no valid value
then remove red from V's domain, thus V → NSW is consistent
If X loses a value, neighbors of X need to be rechecked: put SA → V on queue

Arc consistency (only)

- **X → Y is consistent iff**
  - **for every value x of X there is some allowed y of Y**
- **Pop one arc X → Y and remove any inconsistent values from X**
  - **If any change in X, put all arcs Z → X back on queue, where Z is any neighbor of X that is not equal to Y**



***X* = SA, *Y* = NT**



Consider state after {WA = red, Q=green}
SA → NT?
Inconsistent because when SA = blue, NT has no valid value
then remove blue from SA's domain, and SA has no legal value! Failure! (Backtrack)
AC detects failure earlier than FC

Arc consistency checking

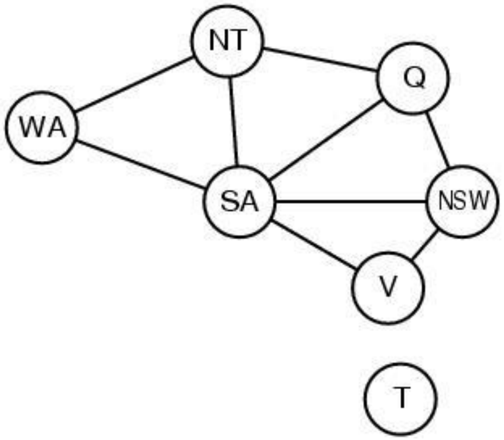
- Can be run as a preprocessor, or after each assignment
  - As preprocessor before search: removes obvious inconsistencies
  - After each assignment: reduces search cost but increases step cost

- AC is run repeatedly until no inconsistency remains
  - Like Forward Checking, but exhaustive until quiescence

- Trade-off
  - Requires overhead to do; but usually better than direct search
  - In effect, it can successfully eliminate large (and inconsistent) parts of the state space more effectively than can direct search alone

- Need a systematic method for arc-checking
  - If X loses a value, neighbors of X need to be rechecked:
    - i.e., incoming arcs can become inconsistent again (outgoing arcs stay consistent).

Problem Structure



- **Tasmania and mainland are independent subproblems.**
- **Any solution for the mainland combined with any solution for Tasmania yields a solution for the whole map.**
- **Independence can be ascertained by finding connected components of the constraint graph.**

Summary

- Constraint satisfaction problems
- Backtracking Algorithm
  - Minimum remaining values
  - Degree heuristics
  - Least constraining values
  - Forward checking
  - Constraint propagation
  - Problem structure

What I want you to do

- Review Chapter 6
- Next week (No in-person Sessions)
  - 02/20 Games I in Zoom Meeting (Links in “Course Collaboration Tool” on Canvas)
  - 02/22 Cancelled
  - 02/27 Games II and midterm exam review
  - 02/29-03/01 Midterm Exam
    - Time duration: 3 hours
    - Open-book and open-note
    - Five Problems (100 points)
    - 1) Search problem: BFS, DFS, UCS, Greedy, A*
    - 2) Search Problem
    - 3) Local search- simulated annealing
    - 4) Constraint satisfaction problem
    - 5) Games