21CS2213RA
AI for Data Science

Session -11

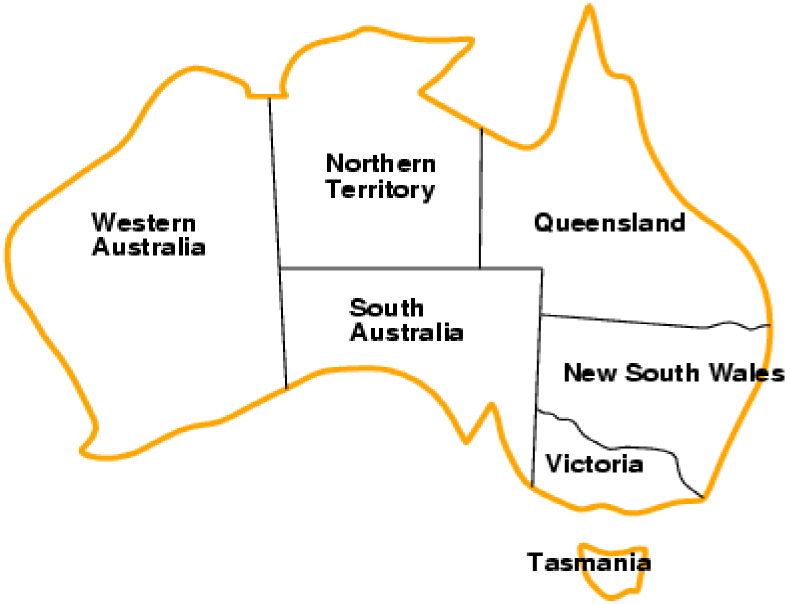
Contents: *Constraint Satisfaction Problem*

(DEEMED TO BE UNIVERSITY)

Constraint satisfaction problems (CSPs)

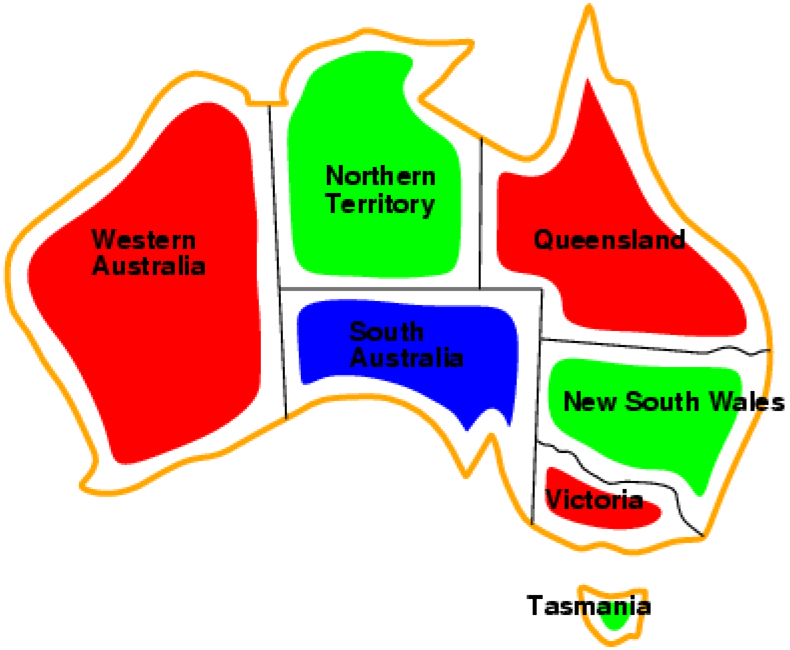
- Standard search problem: state is a "black box“ – any data structure that supports successor function and goal test
- CSP:
  - state is defined by variables X_i with values from domain D_i
  - goal test is a set of constraints specifying allowable combinations of values for subsets of variables
- Simple example of a formal representation language
- Allows useful general-purpose algorithms with more power than standard search algorithms

Example: Map-Coloring



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

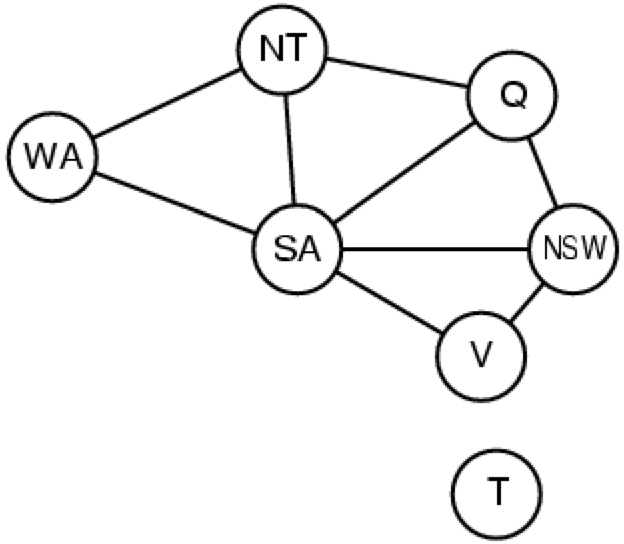
Example: Map-Coloring



- Solutions are complete and consistent assignments
- e.g., WA = red, NT = green, Q = red, NSW = green,V = red,SA = blue,T = green
-

Constraint graph

- Binary CSP: each constraint relates two variables
- Constraint graph: nodes are variables, arcs are constraints



Varieties of CSPs

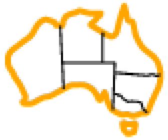
- Discrete variables
  - finite domains:
    - *n* variables, domain size *d* → $O(d^n)$ complete assignments
    - e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)
  - infinite domains:
    - integers, strings, etc.
    - e.g., job scheduling, variables are start/end days for each job
    - need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$

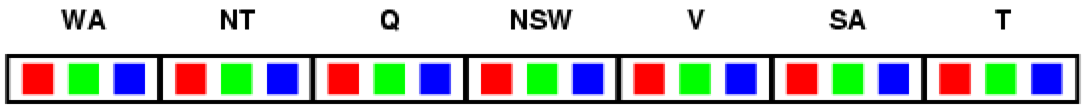
- Continuous variables
  - e.g., start/end times for Hubble Space Telescope observations
  - linear constraints solvable in polynomial time by LP

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., cryptarithmetic column constraints

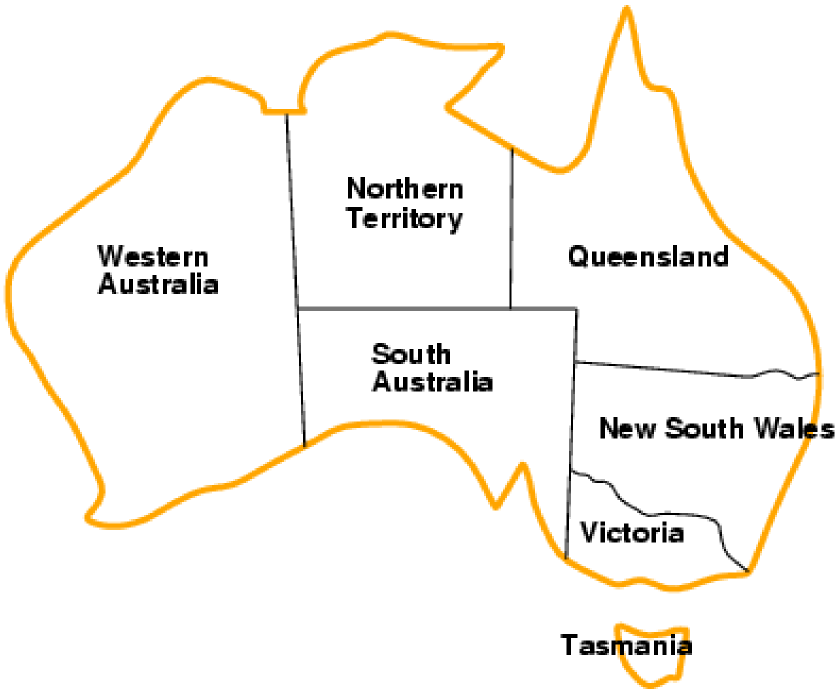
Forward checking

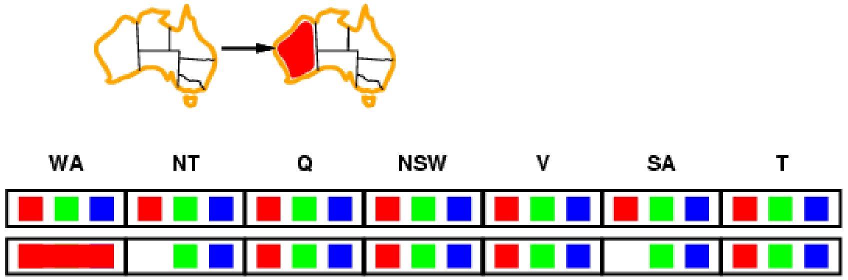
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values
  - 



Forward checking

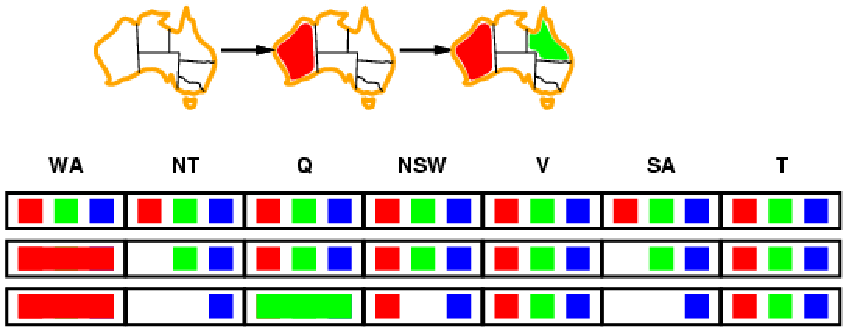
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values
  -





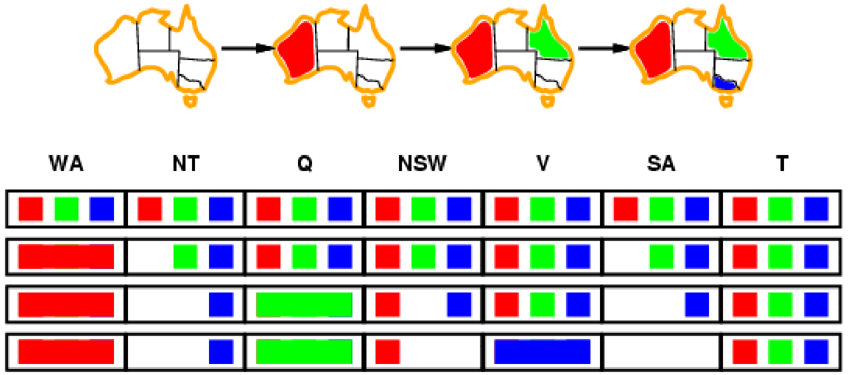
Forward checking

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



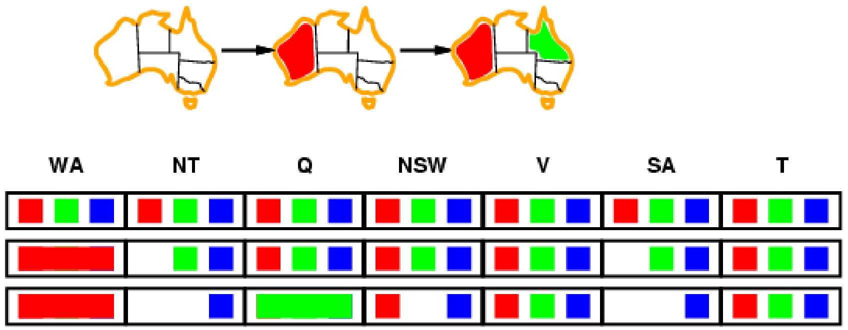
Forward checking

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



Constraint propagation: Inferences in CSPs

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

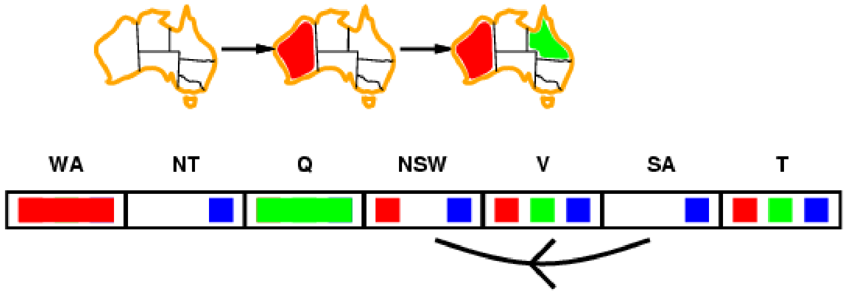


- NT and SA cannot both be blue!
- Constraint propagation algorithms repeatedly enforce constraints locally…

Arc consistency

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

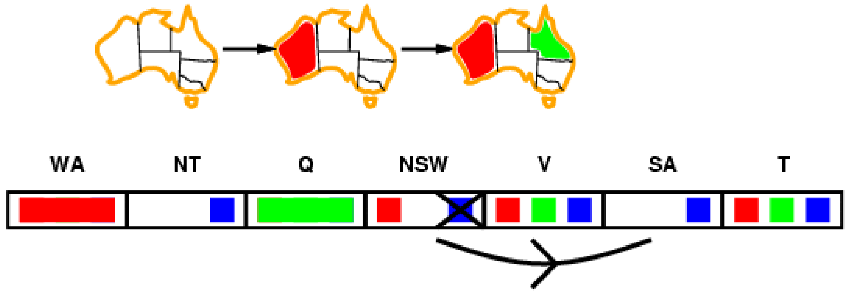
  for every value *x* of *X* there is some allowed *y*



Arc consistency

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

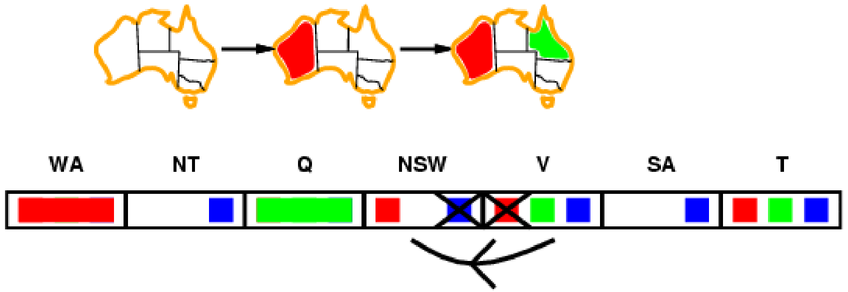
  for every value *x* of *X* there is some allowed *y*



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*

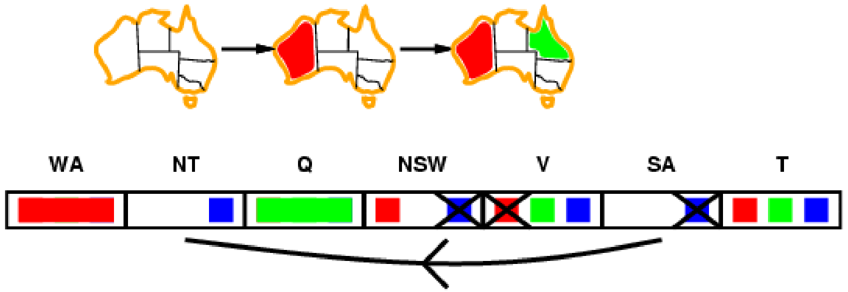


- If *X* loses a value, neighbors of *X* need to be rechecked

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, neighbors of *X* need to be rechecked
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment

Arc consistency algorithm AC-3

```
function AC-3( csp) returns the CSP, possibly with reduced domains
    inputs: csp, a binary CSP with variables {X_1, X_2, ..., X_n}
    local variables: queue, a queue of arcs, initially all the arcs in csp

    while queue is not empty do
        (X_i, X_j) ← REMOVE-FIRST(queue)
        if RM-INCONSISTENT-VALUES(X_i, X_j) then
            for each X_k in NEIGHBORS[X_i] do
                add (X_k, X_i) to queue
```

```
function RM-INCONSISTENT-VALUES( X_i, X_j) returns true iff 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 constraint(X_i, X_j)
            then delete x from DOMAIN[X_i];  removed ← true
    return removed
```

- Time complexity: O(#constraints |domain|^3)

Checking consistency of an arc is O(|domain|^2)

k-consistency

- A CSP is *k-consistent* if, for any set of k-1 variables, and for any consistent assignment to those variables, a consistent value can always be assigned to any kth variable
- 1-consistency is node consistency
- 2-consistency is arc consistency
- For binary constraint networks, 3-consistency is the same as *path consistency*
- Getting k-consistency requires time and space exponential in k
- *Strong k-consistency* means k'-consistency for all k' from 1 to k
  - Once strong k-consistency for k=#variables has been obtained, solution can be constructed trivially
- Tradeoff between propagation and branching
- Practitioners usually use 2-consistency and less commonly 3-consistency

End of Session

Thank you