COSC76/276 Artificial Intelligence
Fall 2022
Constraint Satisfaction Problems

Soroush Vosoughi
Computer Science
Dartmouth College
Soroush@Dartmouth.edu

Reminders

- SA4 due on Oct 21st
- PA3 due Oct 24th

Additional Late Days

- 5 more free late days for everyone!

Overview

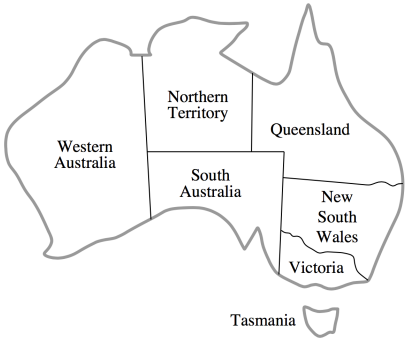
- Model problems as Constraint Satisfaction Problems
- Solve them through “backtracking” (depth-first search)

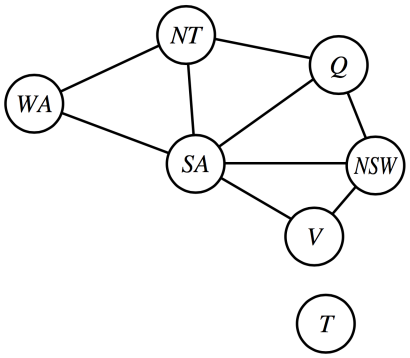
Outline

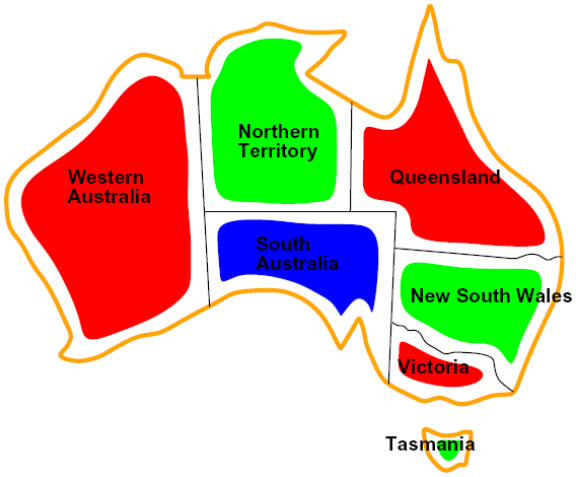
- CSP model
- CSP search

Example: Map coloring

- Variables: WA, NT, SA, Q, NSW, V, T
- Domains: Di: r, g, b
- Constraints:
  - Explicit
    - (WA,NT)∈{(r,g),(r,b),(g,r),(g,b),(b,r),(b,g)}
    - (NT,SA)∈{(r,g),(r,b),(g,r),(g,b),(b,r),(b,g)}
    - …
  - Implicit
    - WA != NT
    - WA != SA
    - …
- Assignment:
  {WA=r, NT=g, Q=r,
  NSW=g, V=r, SA=b, T=g}





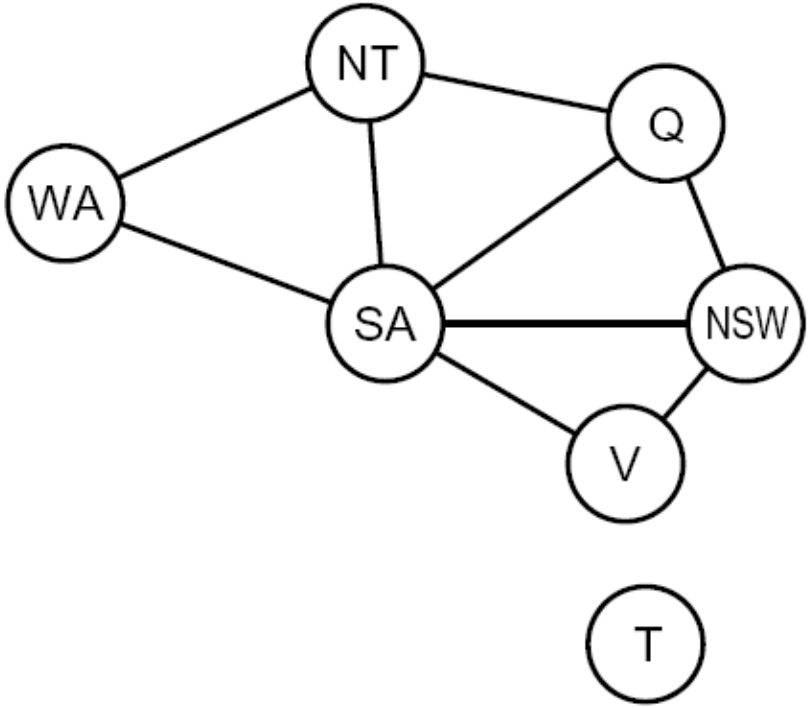


CSP definition

- n variables: $X_1, \ldots, X_n$
- For each variable, a domain Di of possible values. Example: $D_1 : X_1 \in v_1, v_2, v_3$
- m constraints $C_1, \ldots, C_n$, each specifying allowable combinations of values for some set of variables
- An assignment of values to variables is the state of the problem.
  - We want to find a **complete** assignment (all variables with a value) **consistent** with the constraints

Constraint graphs

- Binary CSP: each constraint relates at most two variables
- Nodes are variable, arcs show constraints

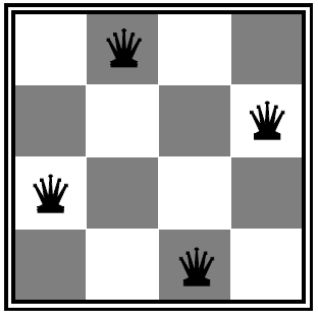


Varieties of constraints

- Unary constraints involve a single variable (equivalent to reducing domains)
  - E.g., in the map coloring problem SA != g
- Binary constraints involve pairs of variables, e.g.:
  - E.g., in the map coloring problem SA != WA
- Higher-order constraints involve 3 or more variables (alldiff):
  - e.g., Sudoko
- Preferences (soft constraints):
  - E.g., red is better than green
  - Often representable by a cost for each variable assignment
  - Gives constrained optimization problems
  - We won't include them in the CSP

Example: 4-Queens

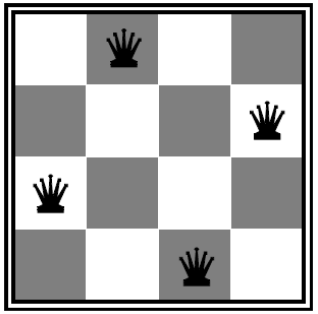
- Formulation 1:
  - Variables: Q1, Q2, Q3, Q4
  - Domains: Qi = {(x,y) | x in [0,3] and y in [0,3]}
  - Constraints
    - Any queen not in the same x
      - x1!=x2, x1!=x3, …
    - Any queen not in the same y
      - y1!=y2 …
    - Any queen not on the diagonal
      - abs(y2-y1) != abs (x2-x1) …



Discussion

Example: N-Queens

- Formulation 2:
  - Variables: X_{ij}
  - Domains: $\{0, 1\}$
  - Constraints



$$\forall i, j, k \quad (X_{ij}, X_{ik}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \quad (X_{ij}, X_{kj}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \quad (X_{ij}, X_{i+k,j+k}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \quad (X_{ij}, X_{i+k,j-k}) \in \{(0,0), (0,1), (1,0)\}$$

$$\sum_{i,j} X_{ij} = N$$

Discussion

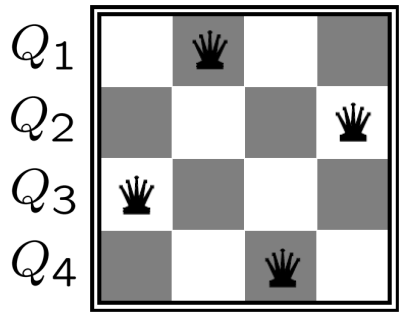
Example: N-Queens

- Formulation 3:
  - Variables: Q_k
  - Domains: $\{1, 2, 3, \ldots N\}$
  - Constraints:

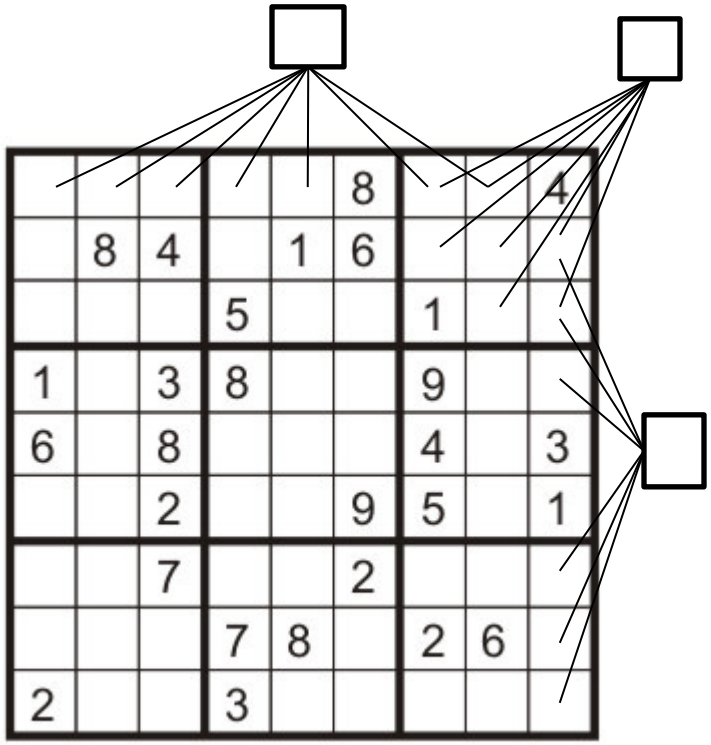
Implicit: $\forall i, j$ non-threatening(Q_i, Q_j)

Explicit: $(Q_1, Q_2) \in \{(1, 3), (1, 4), \ldots\}$

$\ldots$



Example: Sudoku



- Variables:
  - Each (open) square
- Domains:
  - {1,2,…,9}
- Constraints:

9-way alldiff for each column

9-way alldiff for each row

9-way alldiff for each region

(or can have a bunch of pairwise inequality constraints)

Example: Assignment problem

- Assign four workers W1,W2,W3,W4 to four products such that each worker works on one product and each product is produced by one worker.
- Effectiveness of production is given by the following table (e.g., worker W1 produces P1 with effectiveness 7) and the total effectiveness must be 19 at least

	P1	P2	P3	P4
W1	7	1	3	4
W2	8	2	5	1
W3	4	3	7	2
W4	3	1	6	3

Discussion

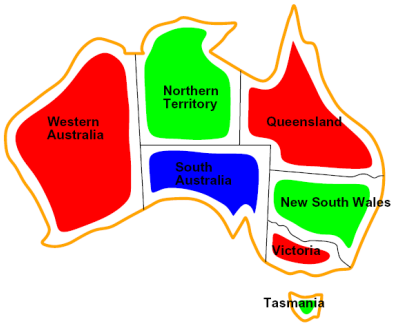
Example: assignment problem

- Variables
  - W1, W2, W3, W4
- Domains:
  - {P1, P2, P3, P4}
- Constraints
  - All_diff(W1, W2, W3, W4)
  - E1(W1)+E2(W2)+E3(W3)+E4(W4)>=19

	P1	P2	P3	P4
W1	7	1	3	4
W2	8	2	5	1
W3	4	3	7	2
W4	3	1	6	3

How can we write general code?

Implementation notes



We want to write general code.

An example of assignment:

- state = {0, 1, 1, 2, 0, 1, 0}

- variables order: WA, NT, SA, Q, NSW, V, T

- Domains can be represented by sets of numbers
  - So WA=0 (red); NT has the value 1 (green); Q has value 2 (blue)

- The CSP model should be separate from the solver, as we have seen for the search problems

Outline

- CSP model
- CSP search

What are the elements needed for the search?

Standard search formulation

- Standard search formulation of CSPs
- States defined by the values assigned so far (partial assignments)
  - Initial state: the empty assignment, {}
  - Successor function: assign a value to an unassigned variable
  - Goal test: the current assignment is complete and satisfies all constraints
- We'll start with the straightforward, naïve approach, then improve it

Search Methods

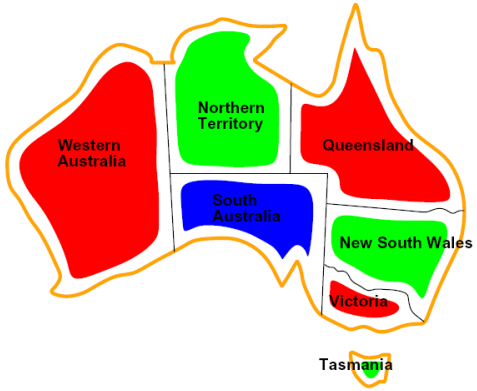
- What would BFS do?

$\{\}$

$\{WA=g\}$ $\{WA=r\}$ … $\{NT=g\}$ …

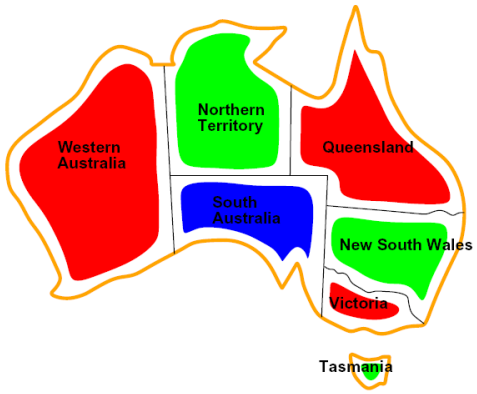
$\{WA=g, NT=g\}$ $\{WA=g, NT=r\}$ …

…



Search Methods

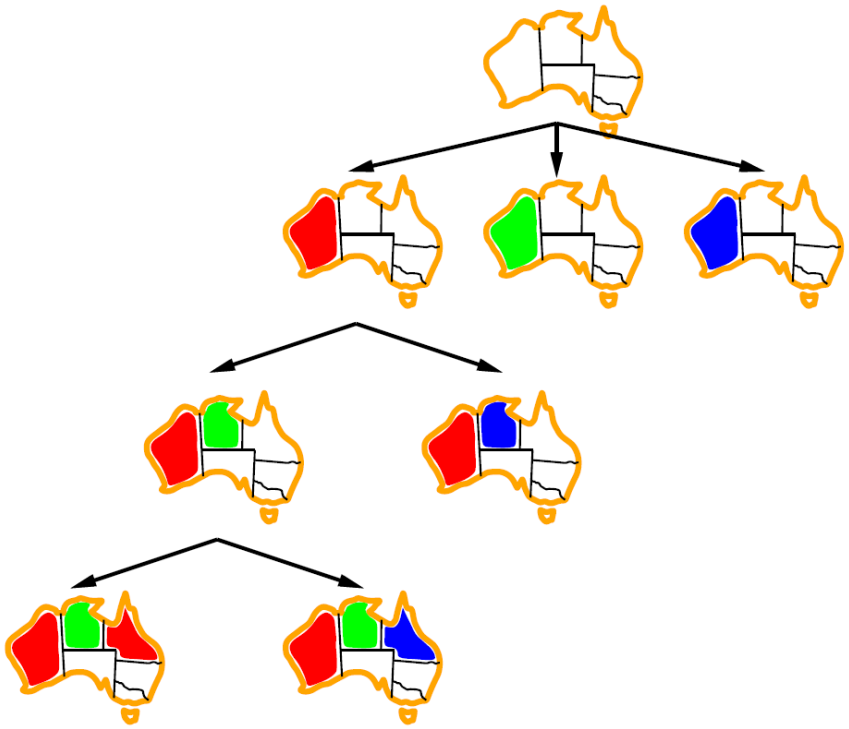
- What about DFS?
  - Expand in depth
- What problems does naïve search have?
  - With n number of variables and d domain size, the branching factor nd at the first level, (n-1)d at the second level, … and so on for n levels
    - n!d^n leaves however, d^n assignments



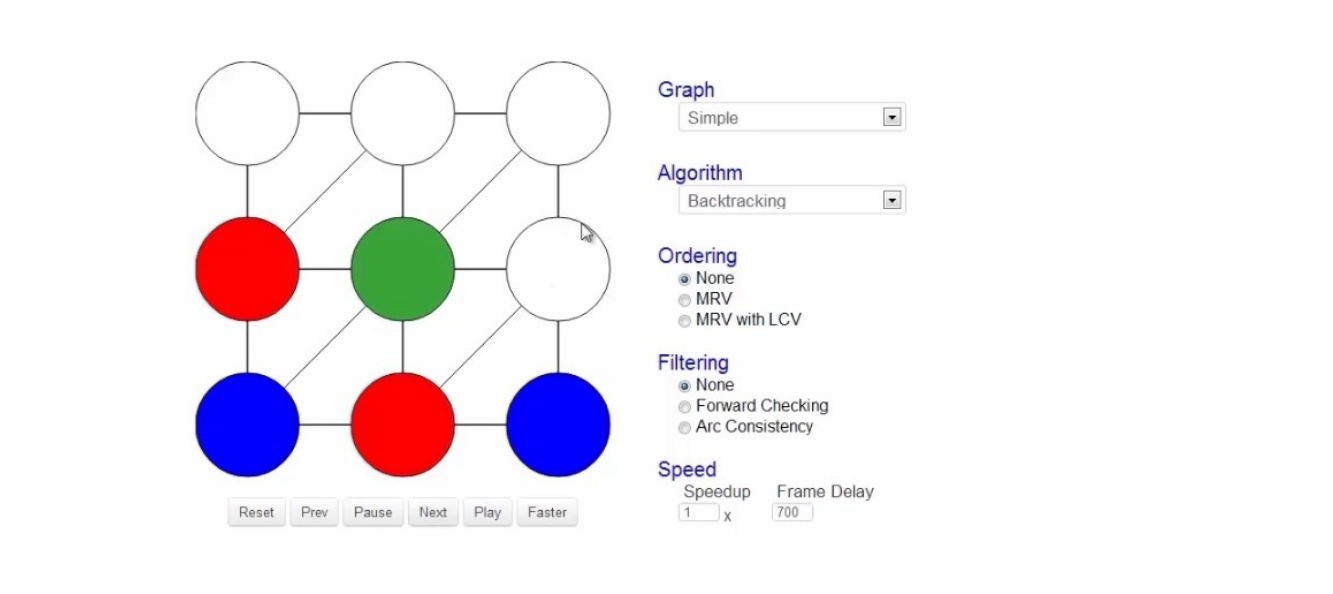
Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
- Idea 1: One variable at a time
  - Variable assignments are **commutative**, so fix ordering -> better branching factor!
    - i.e., [WA = red then NT = green] same as [NT = green then WA = red]
  - Only need to consider assignments to a single variable at each step
  - Then there are only d^n paths. We have eliminated the n! redundancy by arbitrarily choosing an order in which to assign variables.
- Idea 2: Check constraints as you go
  - i.e. consider only values which do not conflict previous assignments
  - Might have to do some computation to check the constraints
- Depth-first search with these two improvements is called *backtracking search*
- Can solve n-queens for $n \approx 25$

Backtracking Example



Video of Demo Coloring – Backtracking



[video: AI Berkeley]

Backtracking Search

Discussion

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

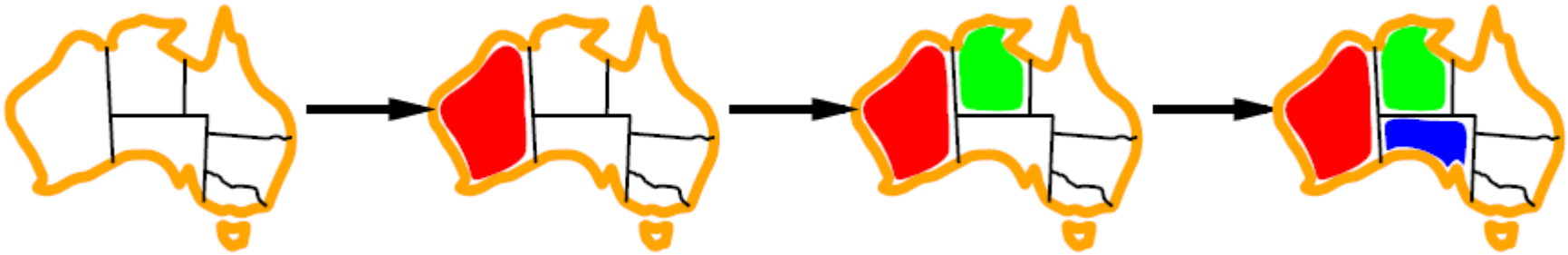
- What are the choice points?

Heuristics for making backtracking search more efficient

- Heuristics that choose the next variable to assign:
  - Minimum Remaining Values (MRV)
  - Degree heuristic
- Heuristic that chooses a value for that variable:
  - Least Constraining Value (LCV)

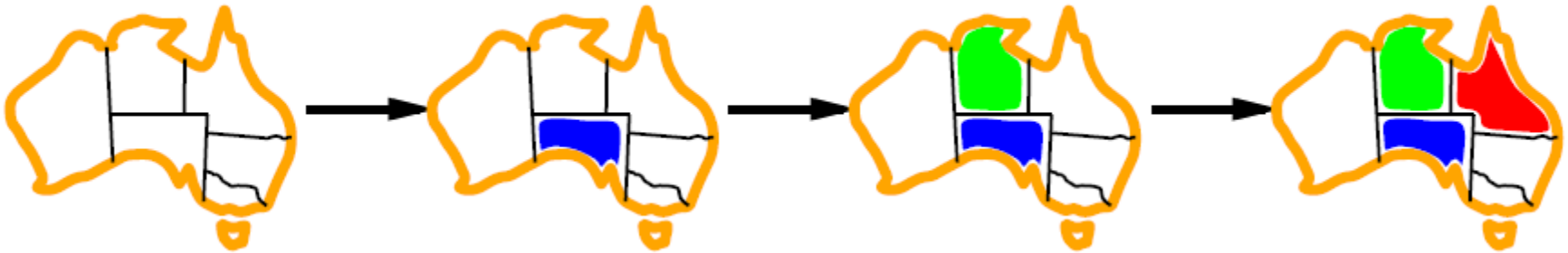
Which variable should be assigned next?

- Minimum Remaining Values (MRV) Heuristic:
  - Choose the variable with the fewest legal values



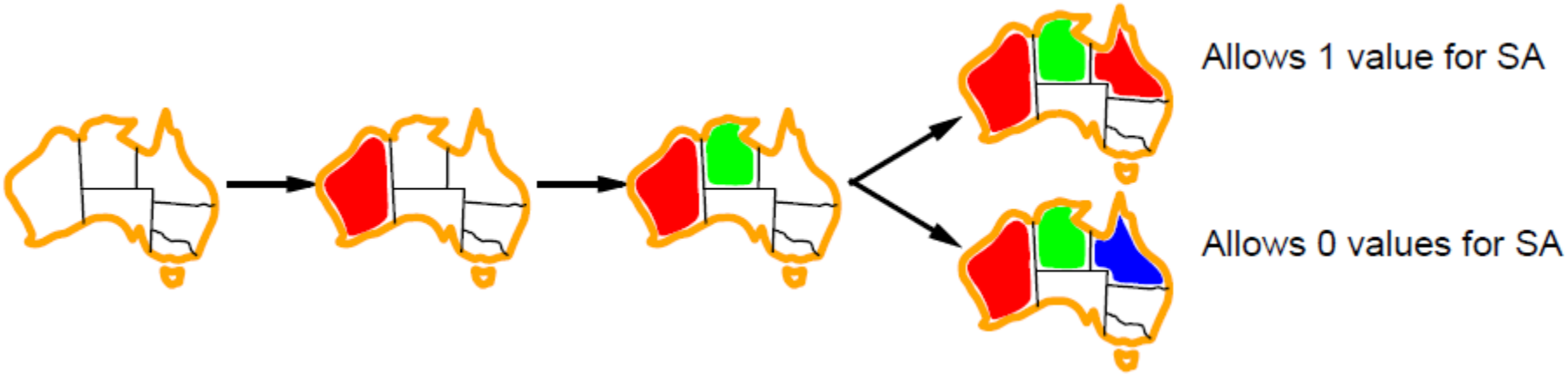
Degree heuristic

- Tie-breaker among MRV variables
- Degree heuristic:
  - choose the variable with the most constraints on remaining variables



Given a variable, in which order should its values be tried?

- Least Constraining Value (LCV) Heuristic:
  - Try the following assignment first: to the variable you're studying, the value that rules out the fewest values in the remaining variables



Summary

- CSP model
  - Variables, domains, constraints
- Formulated as search problem
  - Standard formulation though high complexity
- Backtracking search:
  - Fix order of variables at specific level
  - Check constraints
- Heuristics to improve backtracking search:
  - Variable selection: Minimum Remaining Value (MRV) and Degree heuristics
  - Value selection: Least Constraining Value (LCV)

Next

- What can we do to be able to further improve CSP?
  - Early detection of failure

```
function RECURSIVE-BACKTRACKING(assignment, csp)
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp)
      if value is consistent with assignment given CONSTRAINTS[csp]
          add {var = value} to assignment
          result ← RECURSIVE-BACKTRACKING(assignment, csp)
          if result ≠ failure then return result
          remove {var = value} from assignment
  return failure
```

Apply *inference* to reduce the space of possible assignments and detect failure early

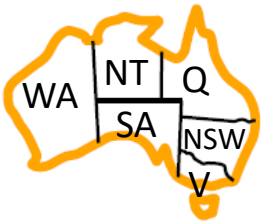
Early detection of failure: forward checking

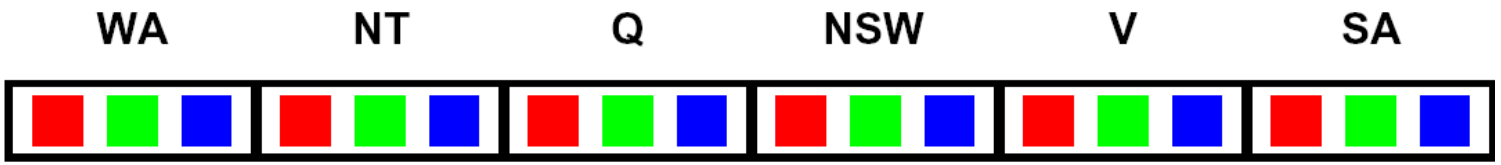
- Idea: Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values
- Forward Checking:
  - Check to make sure that every variable still has at least one possible assignment

Early detection of failure: forward checking

Example

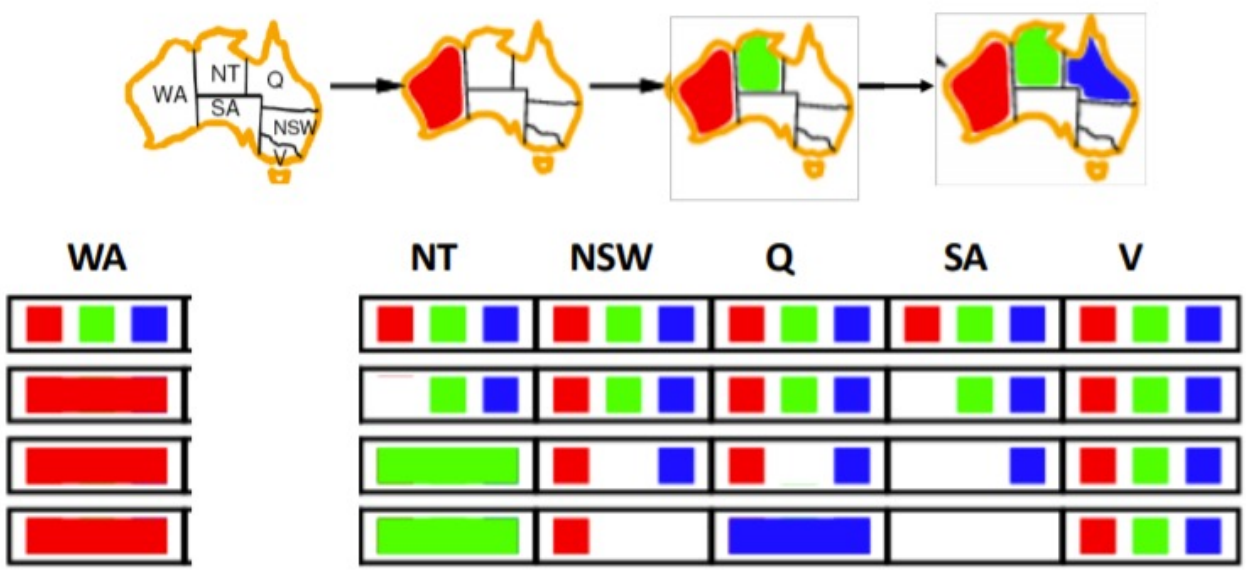
- Forward Checking:
  - Check to make sure that every variable still has at least one possible assignment





Early detection of failure: forward checking

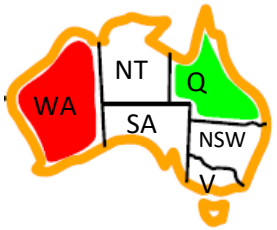
- Forward Checking:
  - Check to make sure that every variable still has at least one possible assignment



Early detection of failure: forward checking problem

Example

- 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!
- Forward checking does not detect that

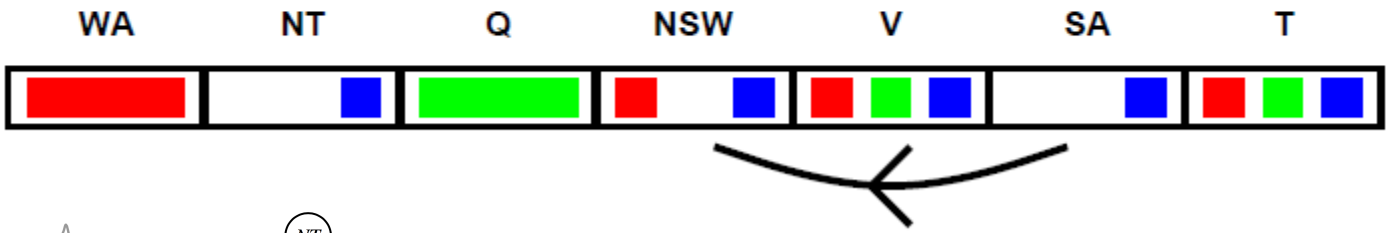
Early detection of failure: Constraint propagation

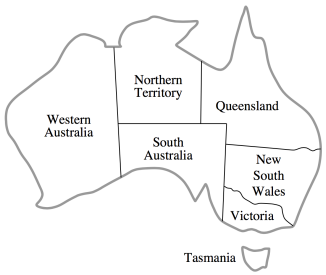
- Constraint propagation
  - Check to make sure that every PAIR of variables still has a pair-wise assignment that satisfies all constraints
  - Apply inference to detect failure early and reduce the space of possible assignments and

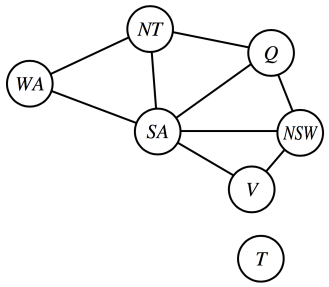
Constraint propagation algorithm: Consistency of A Single Arc

Example

- Simplest form of propagation makes each pair of variables consistent:
  - An arc X $\rightarrow$ Y is consistent iff for every value of X there is some allowed value of Y

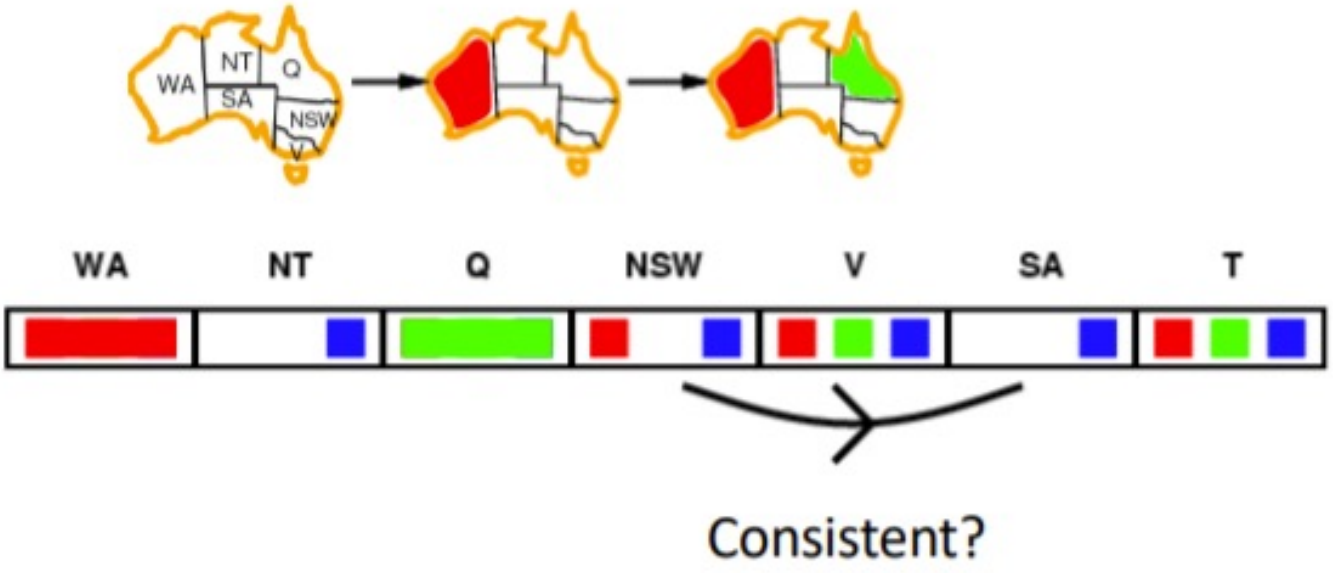






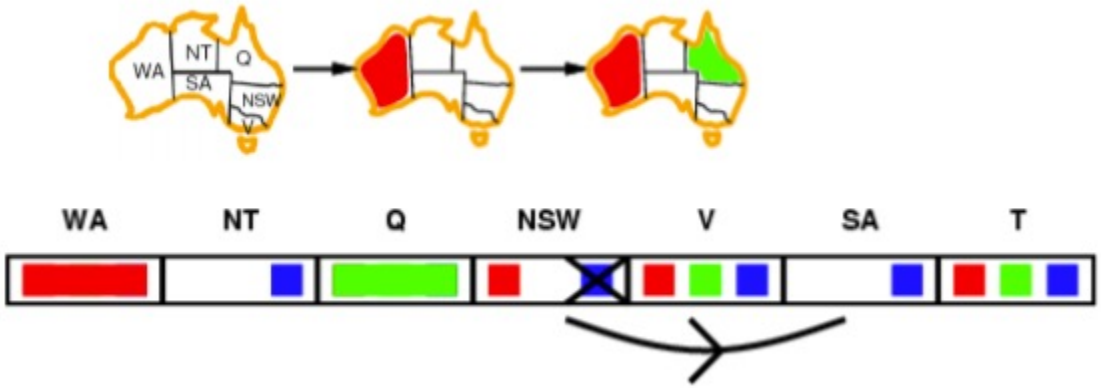
Constraint propagation algorithm: Consistency of A Single Arc

- Simplest form of propagation makes each pair of variables consistent:
  - An arc X $\rightarrow$ Y is consistent iff for every value of X there is some allowed value of Y



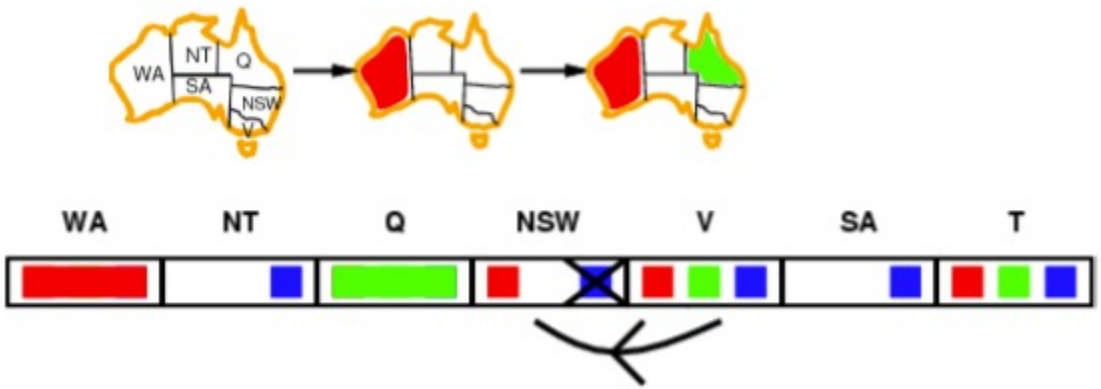
Constraint propagation algorithm: Consistency of A Single Arc

- Simplest form of propagation makes each pair of variables consistent:
  - An arc X $\rightarrow$ Y is consistent iff for every value of X there is some allowed value of Y
  - When checking X ->Y, throw out any values of X for which there isn't an allowed value of Y



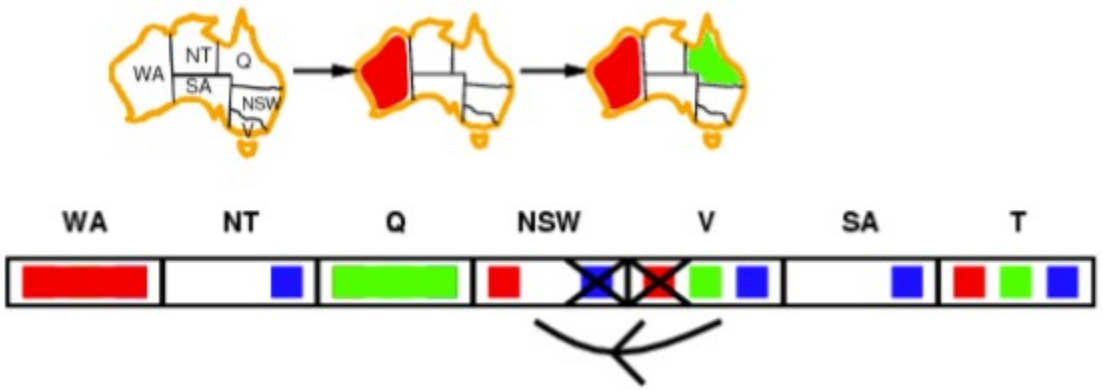
Constraint propagation algorithm: Consistency of A Single Arc

- Simplest form of propagation makes each pair of variables consistent:
  - An arc X $\rightarrow$ Y is consistent iff for every value of X there is some allowed value of Y
  - When checking X ->Y, throw out any values of X for which there isn't an allowed value of Y
  - If X loses a value, all pairs Z -> X need to be rechecked



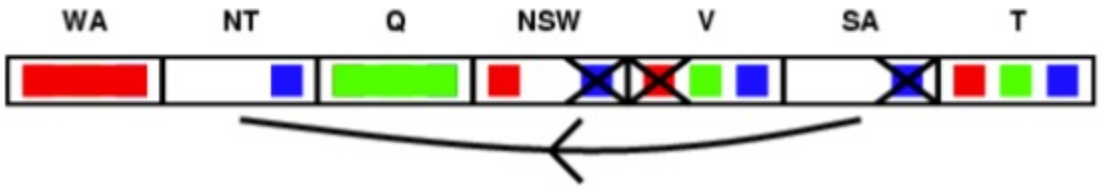
Constraint propagation algorithm: Consistency of A Single Arc

- Simplest form of propagation makes each pair of variables consistent:
  - An arc X $\rightarrow$ Y is consistent iff for every value of X there is some allowed value of Y
  - When checking X ->Y, throw out any values of X for which there isn't an allowed value of Y
  - If X loses a value, all pairs Z -> X need to be rechecked



Constraint propagation algorithm: Consistency of A Single Arc

- Simplest form of propagation makes each pair of variables consistent:
  - An arc X $\rightarrow$ Y is consistent iff for every value of X there is some allowed value of Y
  - When checking X ->Y, throw out any values of X for which there isn't an allowed value of Y
  - If X loses a value, all pairs Z -> X need to be rechecked



- Arc consistency detects failure earlier than forward checking
- Can be run before or after each assignment

Enforcing Arc Consistency in a CSP

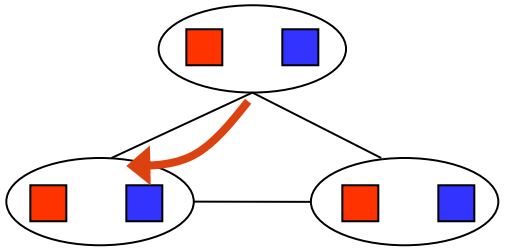
```
function AC-3( csp) returns the CSP, possibly with reduced domains
    inputs: csp, a binary CSP with variables {X_1, X_2, ..., X_n}
    local variables: queue, a queue of arcs, initially all the arcs in csp

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

```
function REMOVE-INCONSISTENT-VALUES( X_i, X_j) returns true iff succeeds
    removed ← false
    for each x in DOMAIN[X_i] do
        if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint X_i ↔ X_j
            then delete x from DOMAIN[X_i]; removed ← true
    return removed
```

Time complexity?

Limitations of Arc Consistency

- After enforcing arc consistency:
  - Can have one solution left
  - Can have multiple solutions left
  - Can have no solutions left (and not know it)



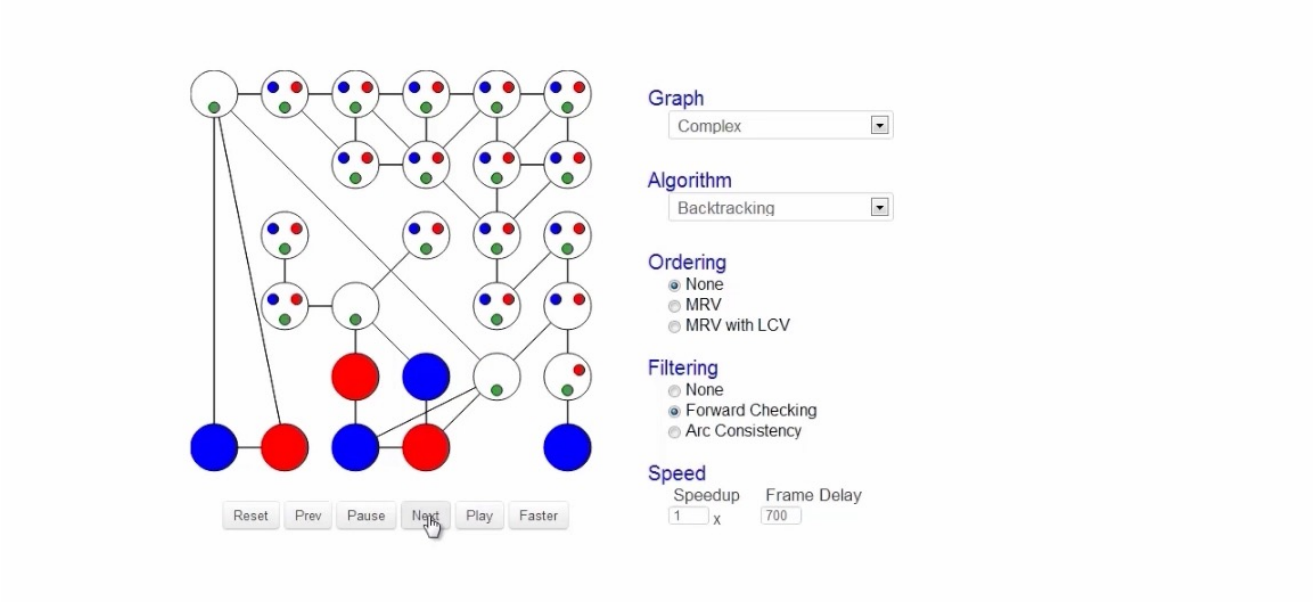
- Arc consistency still runs inside a backtracking search!

Backtracking-search with inference

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
  return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment given CONSTRAINTS[csp] then
      add {var = value} to assignment
      result ← RECURSIVE-BACKTRACKING(assignment, csp)
      if result ≠ failure then return result
      remove {var = value} from assignment
  return failure
```

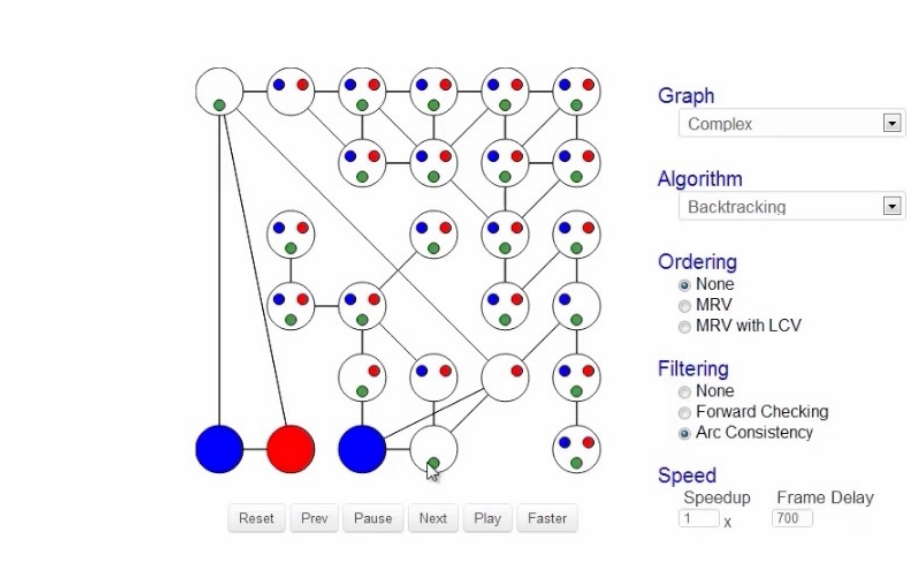
```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, value)
      if inferences ≠ failure then
        add inferences to assignment
        result ← BACKTRACK(assignment, csp)
        if result ≠ failure then
          return result
    remove {var = value} and inferences from assignment
  return failure
```

Video of Demo Coloring – Backtracking with Forward Checking – Complex Graph



[video: AI Berkeley]

Video of Demo Coloring – Backtracking with Arc Consistency – Complex Graph



[video: AI Berkeley]

Summary

- Early detection of failure to improve backtracking
  - Forward checking
  - Constraint propagation
  - Still some failure cannot be detected early

Next

- Logical agents