Intelligent Systems

Unit 1
Constraint Satisfaction Problems

Dra. Elisa Guerrero Vázquez
elisa.guerrero@uca.es
University of Cadiz - Spain

Contents

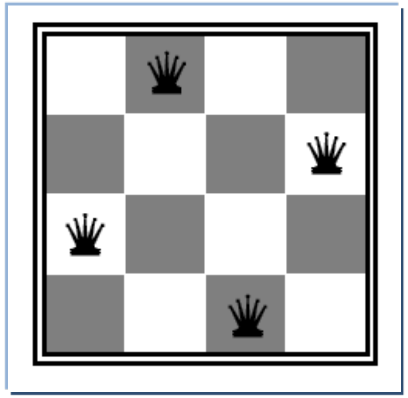
1.1 CSP definition

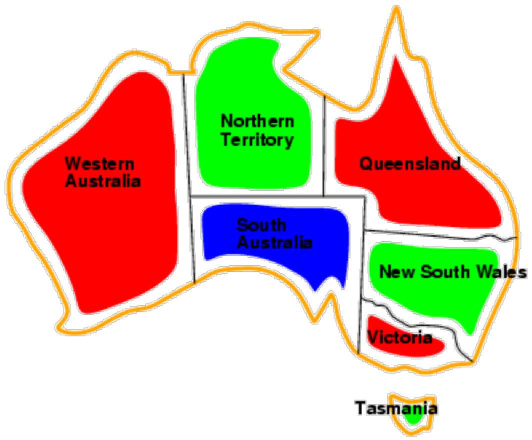
Constraint Satisfaction Problem (CSP)

the solution is a correct assignment of values to each variable according to a set of constraints that must be satisfied

1.2 Toy Problems

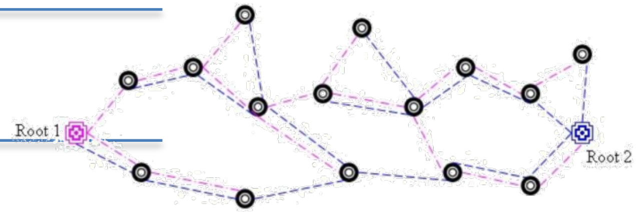
- N-Queens
- Map Coulouring
- Cryptography
- Sudoku



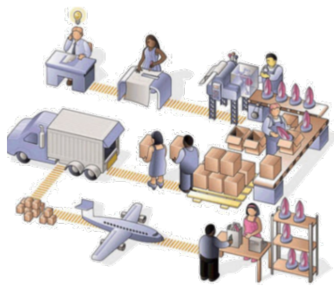


	T	W	O	
	T	W	O	+
F	O	U	R	

1.2 Real World CSP Problems



- Shortest path between several points
- Optimal routing of data in Internet
- Minimal cost planning for product shipping
- Optimal sequencing in process manufacturing
- Task scheduling
- Optimal aircrew selection



https://www.youtube.com/watch?v=y4RAYQjKb5Y

2 CSP formulation

Set of Variables $X_1 \ldots X_n$

- whose values belong to a domain D_i

Set of Constraints $C_1 \ldots C_m$

- the set of allowed values
- rules or properties that each variable must satisfied

State

- assignment of values to variables

Solution

- complete assignment that satisfies all the constraints

2.1 Assignments

Consistent

an assignment that does not violate any constraints

Complete

every variable is mentioned in the assignment and all the constraints are satisfied

Solution

2.2 CSP classification

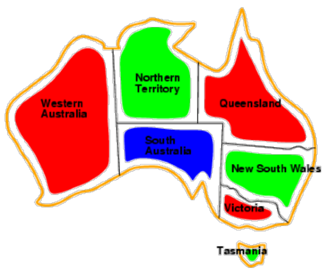
- Domain
  - Discrete
  - Continuous
- Constraints
  - Hard (Obligation)
  - Soft (Preference)
- Number of variables (in the constraints)
  - Binary
  - Multiple

2.2 CSP formulation example: Graph-colouring

GOAL: Assign different colours to adjacent regions

- **Variables**: R_1 .. R_7 each region
- **Domain**: set of colours {red, green, blue}
- **Constraints**:
  - Two adjacent regions must have different colours
    - $R_i \neq R_j$ If R_i and R_j are adjacent
- **State:** variables with some assigned value {R_1=red, R_2=, R_3=, R_4=, R_5=, R_6=, R_7=}
- **Solution**: Consistent and complete assignment

{R_1=red, R_2=green, R_3=red, R_4=blue,
R_5=green, R_6=red, R_7=green}



2.2 CSP formulation example: Graph-colouring

GOAL: assign different colours to adjacent regions

- **Variables**: R_1 .. R_7 each region
- **Domain**: set of colours {red, green, blue}
- **Constraints**:
  - Two adjacent regions must have different colo
    - $R_i \neq R_j$ If R_i and R_j are adjacent
- **State**: variables with some assigned value {R_1=red ... }
- **Solution**: Consistent and complete assigment

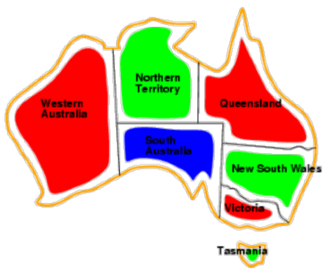
{R_1=red, R_2=green, R_3=red,

R_5=green, R_6=

Discrete Domain

Hard Constraints

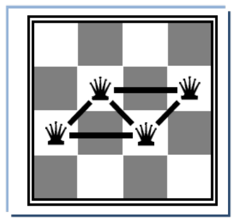
Binary Constraints



2.2 CSP formulation example: N-Queens

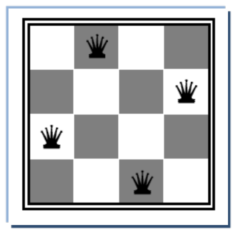
GOAL: Place N queens on an NxN chess board so that no queen can attack any other queen

- **Variables:** $Q_1, Q_2, .. Q_N$ representing each queen position (Queen 1 is always in column 1, Queen 2 in column 2, ...)
- **Domain:** row numbers {1, 2, .. N}
- **Constraints**:
  - Different Row: $Q_i \neq Q_j$
  - Different Diagonal: $|Q_i - Q_j| \neq |i - j|$



- **State**: Any assignment
- **Solution for N=4**: {3, 1, 4, 2} that is

Q_1=3 Q_2=1 Q_3=4 Q_4=2



2.2 CSP formulation example: N-Queens

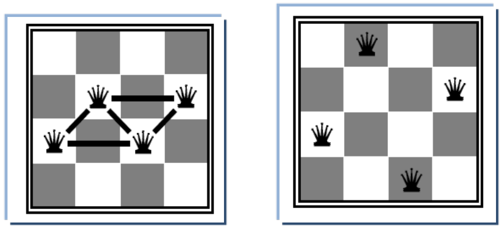
GOAL: Place N queens on an NxN chess board so that no queen can attack any other queen

- **Variables:** $Q_1, Q_2, .. Q_N$ representing each queen position (Queen 1 is always in column 1, Queen 2 in column 2, …)
- **Domain:** row numbers {1, 2, .. N}
- **Constraints**:
  - Different Row: $Q_i \neq Q_j$
  - Different Diagonal: $|Q_i - Q_j| \neq |i - j|$
- **State**: Any assignment
- **Solution for N=4**: $Q_1=3$ $Q_2=1$ $Q_3=4$ $Q_4=2$

Discrete Domain

Hard Constraints

Binary Constraints



2.2 CSP formulation example: Criptarithmetic

GOAL: assign different digits to the letters

- **Variables**: each letter is a different variable, and two more variables are needed:

  $\{T, W, O, F, U, R, X_1, X_2\}$

- **Domain** for the letters: values from 0 to 9, for the carrying variables, values from 0 to 1
- **Constraints**:
  - Sum1: $O+O=R + 10^* X_1$
  - Sum2: $X_1 + W + W = U + 10 * X_2$
  - Sum3: $X_2 + T + T = O + 10^*F$
- **State:** assignments
- **Solution:** F=1, R=2,W=3, O=6, U=7, T=8

	X_2	X_1		
	T	W	O	+
	T	W	O	
F	O	U	R	

	0	1		
	8	3	6	+
	8	3	6	
1	6	7	2	

	0	0		
	7	3	4	+
	7	3	4	
1	4	6	8	

2.2 CSP formulation example: Criptarithmetic

GOAL: assign different digits to the letters

- **Variables**: each letter is a different variable, and two more variables are needed:

  $\{T, W, O, F, U, R, X_1, X_2\}$

- **Domain** for the letters: values from 0 to 9, for the … om 0 to 1
- **Constraints**:
  - Sum1: $O+O=R + 10^* X_1$
  - Sum2: $X_1 + W + W = U + 10 * X_2$
  - Sum3: $X_2 + T + T = O + 10^*F$
- **State:** assignments
- **Solution:** F=1, R=2,W=3, O=6, U=7, T=8

Discrete Domain

Hard Constraints

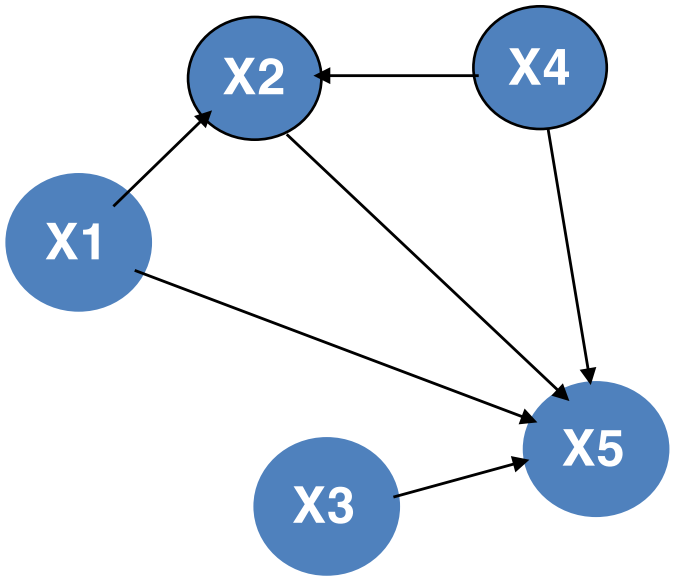
Multiple Constraints

	T	W	O	
				+
	T	W	O	
F	O	U	R	

	0	1		
	8	3	6	
				+
	8	3	6	
1	6	7	2	

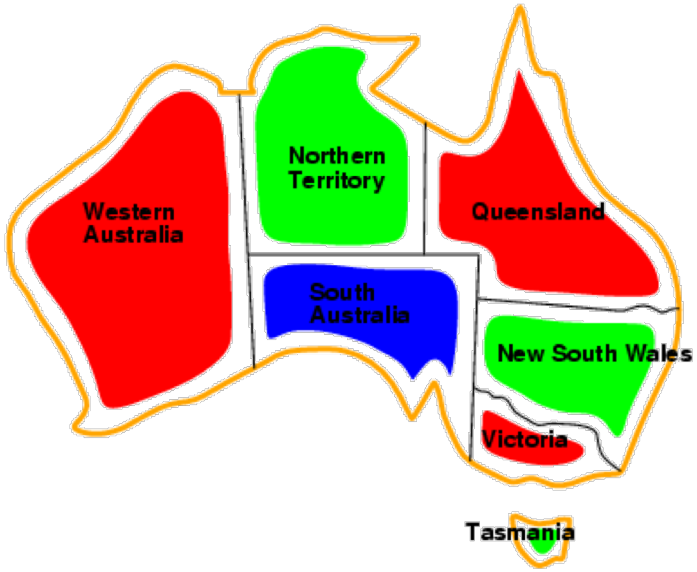
2.3 CSP Representation

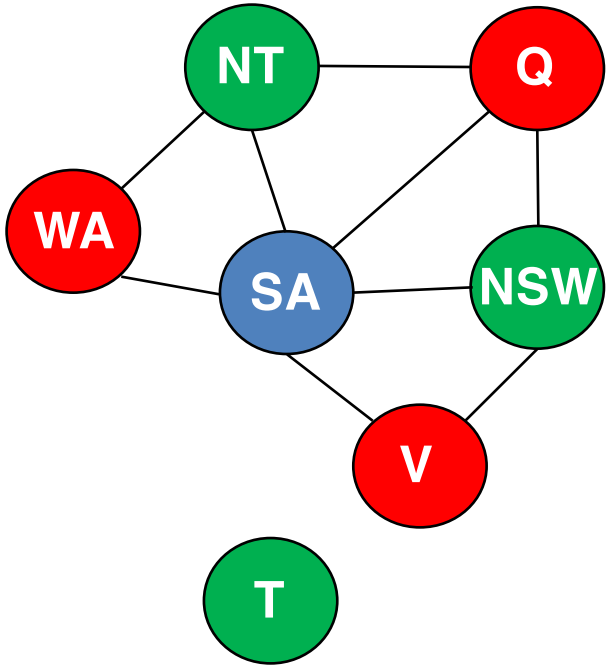
- Binary constraint graph:
  - **Nodes or Vertices :** Variables
  - **Arcs or edges:** Binary relations between variables



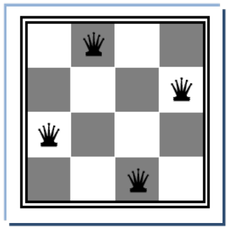
2.3 CSP Representation

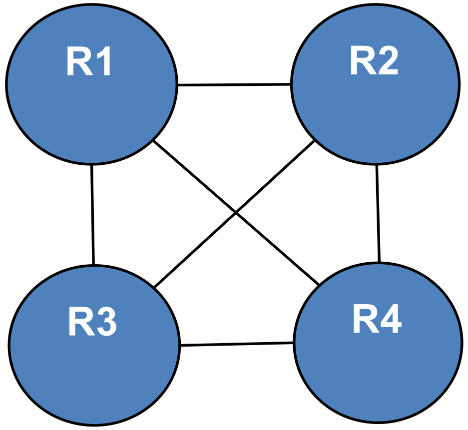
- Binary constraint graph:





2.3 N-Queens Example





3. CSP solving

- **Search strategies**

  Systematic search: Exploration of the state space

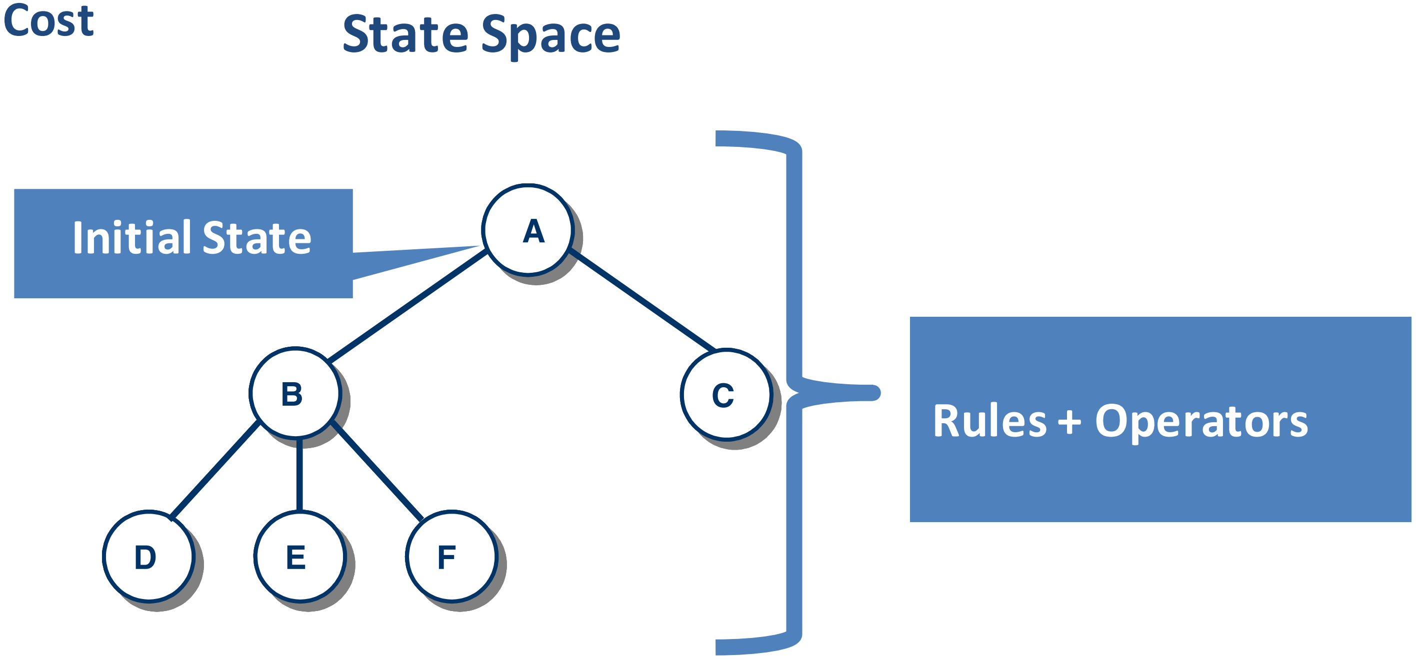
- **Consistency approaches**

  Inconsistent values are removed from variables domains

  Help to reduce the state space

3.1 Search Strategies

- Goal Test
- Path or Solution
- Solution Cost



3.1 Example

- Graph colouring problem

3.1 CSP as State Space Search

Incremental formulation as a standard search problem:

- **Initial State**: empty assignment {R_1=, R_2=, R_3=, R_4=, etc.}

  { , , , }

- **List of Actions:** Assign a color to a variable: Red, Green or Blue
- **Successor Function**: assignment of a value *v* to an unassigned variable when this action does not conflict with previous assignments
  - **isSafe** function to guarantee consistent assignments

  {**Red** , , , }
  { Red , **Green**, , }

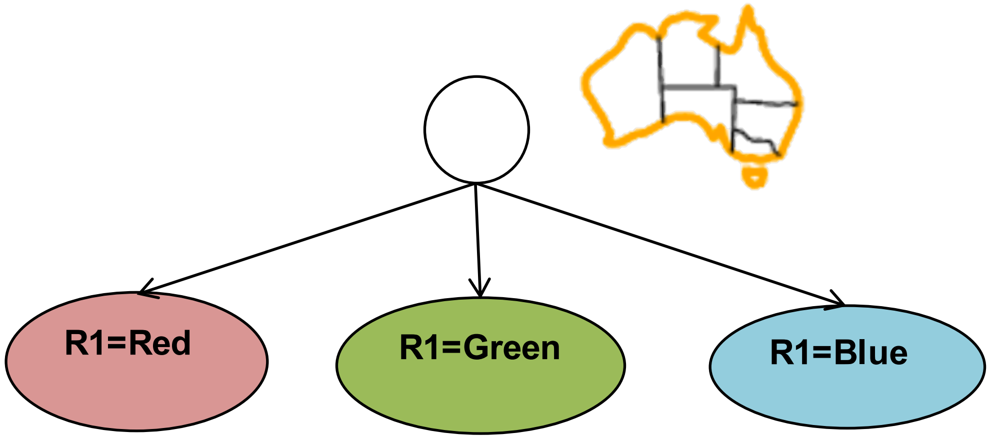
- **Goal Test**: the current assignment is complete

3.1 CSP as state space search

Incremental formulation as a standard search problem:

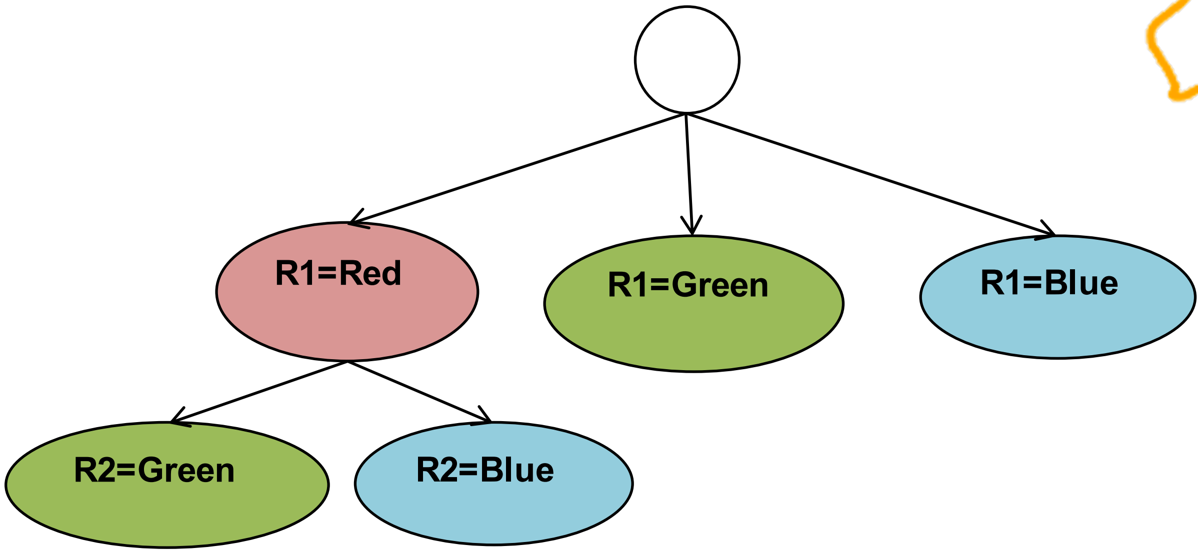
3.1 CSP as state space search

Incremental formulation as a standard search problem:



3.1 CSP as state space search

Incremental formulation as a standard search problem:



¿Depth or Breadth Search?

3.1 Some considerations

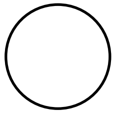
- FINITE DEPTH: the number of variables determines the solution depth
- CONMUTATIVITY: assignment order is irrelevant
  - Depth-first search for CSPs with single-variable assignments is called backtracking search
- CONTROL OF REPEATED STATES is unnecessary

3.2 Backtracking

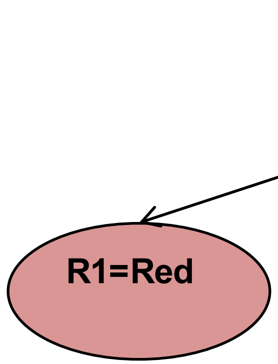
Special depth search …

- Consider the variables in some order
- Pick an unassigned variable and give it a provisional value such that it is consistent with all of the constraints
  - If no such assignment can be made, we've reached a dead end and we need to backtrack to the previous variable
- Continue this process until a solution is found or all possible assignments have been exhausted

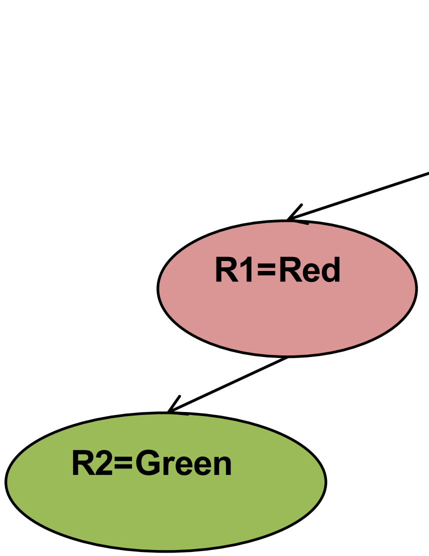
3.2 Backtracking



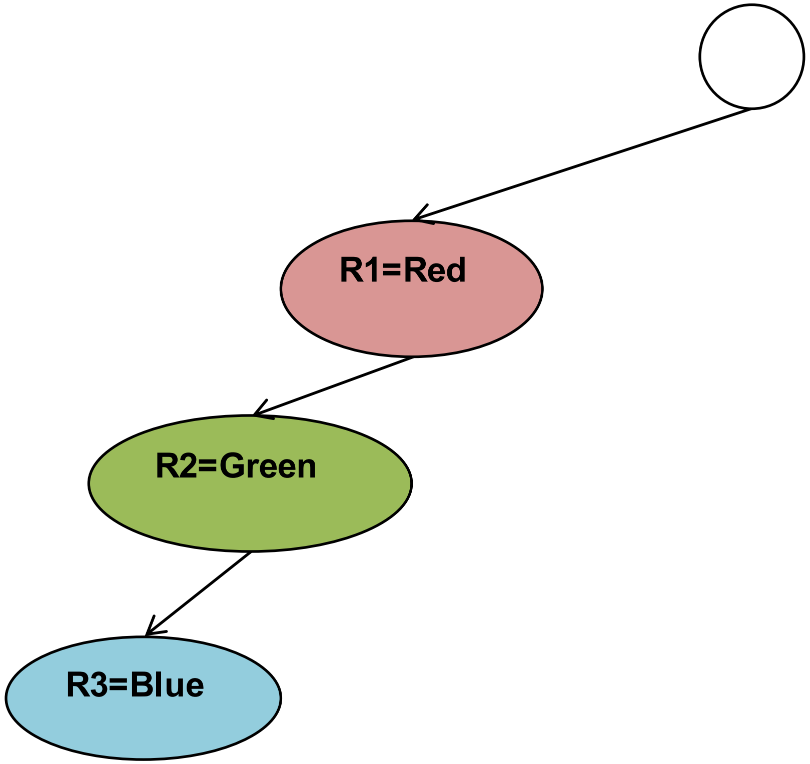
3.2 Backtracking



3.2 Backtracking



3.2 Bactracking



Etc.

Drawbacks of Backtracking

Universidad de Cádiz

Advantages

Simple to implement

Intuitive approach of trial and error

Code size is usually small

Disadvantages

Multiple function calls are expensive

Inefficient

- there is lots of branching from one state
- explore areas of the search space that aren't likely to succeed

3.2 Backtracking Algorithm

```
function [solution, domains] = backtracking(solution, domains)
   variable = SELECT-UNASSIGNED-VARIABLE(solution, domains);
   if Not empty(variable)
            valuesList ← ORDER-DOMAIN-VALUES(variable, domains);
            nValues  ←  length(valuesList);
            while Not Complete(solution) AND Not empty(values_list)
                     value←next(values_list)
                     if Consistent(solution, variable, value)
                              [solution, domains] ← AssignValue(solution, domains, variable, value);
                              [solution, domains] ← backtracking(solution, domains);
                              if Not Complete(solution)
                                       [solution, domains]←Undo(solution, domains, variable, value);
                              end
                     end
            end
   end
end %bactracking
```

3.3 General purpose heuristics

Variable selection

Values Selection

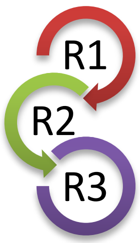
Degree Heuristic : most constraint variable

Minimum Remaining Values

Least-Constraining-Value

Information propagation through constraints

Backtracking



Fordward Checking

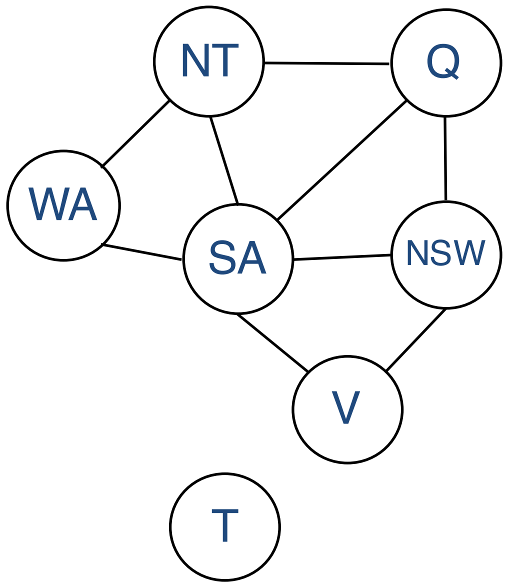
Arc-Consistency AC3

Intelligent backtrack in case of failure

Variable Selection: Example of Degree Heuristic

Wich variable would go first?

Number of Constraints:
WA=2
NT=3
Q=3
SA=5
NSW=3
V=2
T=0

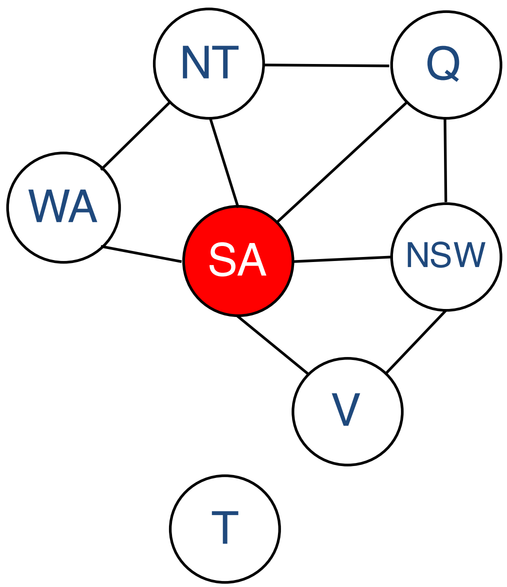


Variable Selection: Example of Degree Heuristic

SA=red

Degree Heuristic selects SA

Number of Constraints:
WA=2
NT=3
Q=3
SA=5
NSW=3
V=2
T=0



Variable Selection: Example of MRV

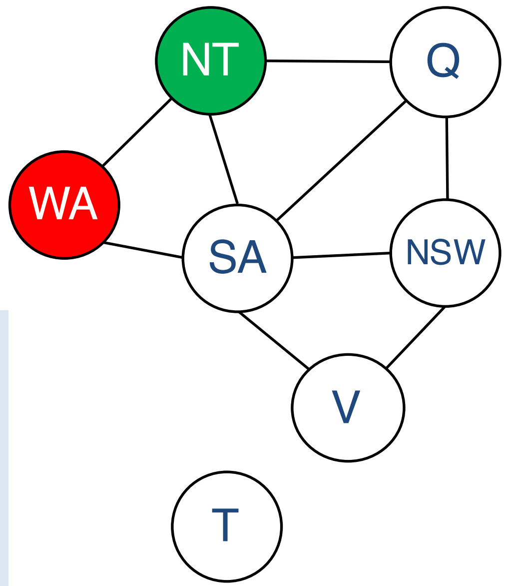
WA=red

WA=red
NT=green

Which variable would go next?

Possible values:

SA={blue}
Q={red, blue}
NSW={red,green, blue)
V={red,green, blue)
T={red,green, blue)



Variable Selection: Example of MRV

WA=red

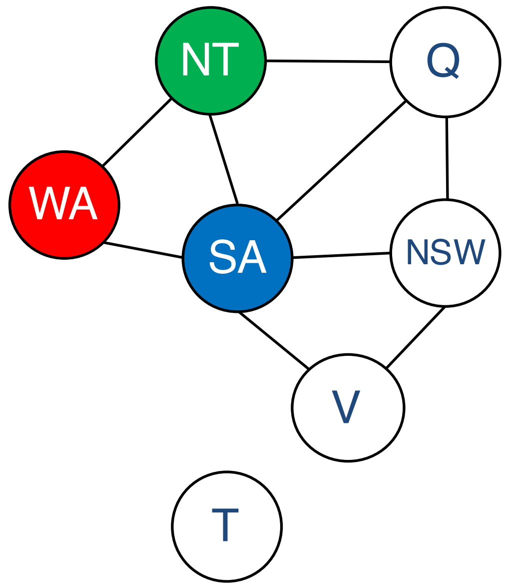
WA=red
NT=green

WA=red
NT=green
SA=blue

Which variable would go next?

Possible values:
SA={blue}
Q={red, blue}

MRV selects SA



A. Variable selection

Variable Selection

var←SELECT-UNASSIGNED-VARIABLE(solution, domains)

Degree Heuristic:

selects the variable that is involved in the largest number of constraints of other unassigned variables

- useful as a tie-breaker or at the beginning of the search process

Minimum Remaining Values (MRV):

Selects the variable with less legal values

- To increase the probability of pruning

Order of Values: Example of LCV

Assuming that Q is the next variable …

WA=red

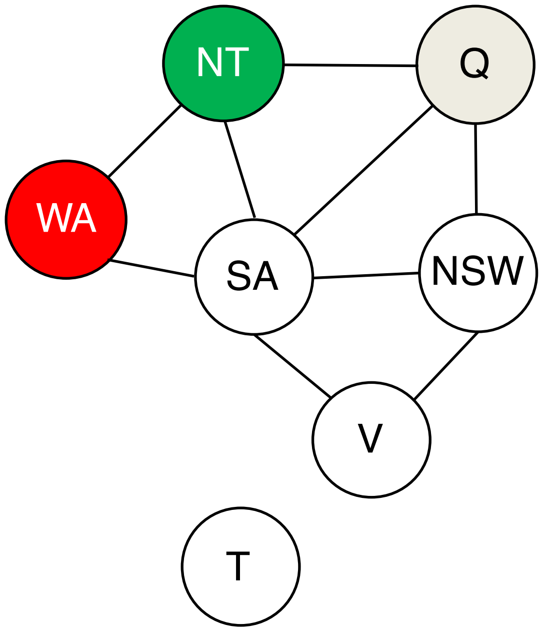
WA=red
NT=green

Which value of Q should I select?

WA=red
NT=green
Q= ?

Possible values:

SA={blue}
Q={red, blue}
NSW={red,green, blue)
V={red,green, blue)
T={red,green, blue)



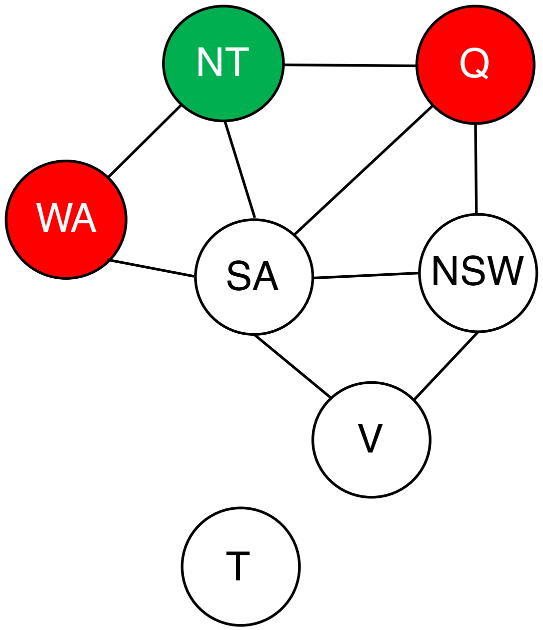
Order of Values: Example of LCV

Assuming that Q is the next variable …

WA=red

WA=red
NT=green

Which value of Q should I select?



WA=red
NT=green
Q= red

Possible values:

SA={blue}
Q={red, blue}
NSW={red,green, blue)
V={red,green, blue)
T={red,green, blue)

red appears less times than blue

B. Order of Values

Once a variable has been selected, the algorithm must decide the order in which it will examine its values.

Order of values

values_list←ORDER-DOMAIN-VALUES(solution, domains)

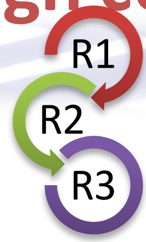
Least-Constraining-Value (LCV)	**Selects the value that appears in fewer constraints** e.g., the most free value • Tries to leave as many options for the rest of the variables to be assigned

3.2 Backtracking Algorithm

```
function [solution, domains] = backtracking(solution, domains)
   variable = SELECT-UNASSIGNED-VARIABLE(solution, domains);
   if Not empty(variable)
         valuesList ← ORDER-DOMAIN-VALUES(variable, domains);
         nValues ← length(valuesList);
         while Not Complete(solution) AND Not empty(values_list)
               value←next(values_list)
               if Consistent(solution, variable, value)
                     [solution, domains] ← AssignValue(solution, domains, variable, value);
                     [solution, domains] ← backtracking(solution, domains);
                     if Not Complete(solution)
                           [solution, domains]←Undo(solution, domains, variable, value);
                     end
               end
         end
   end
end %bactracking
```

C. Propagating information through constraints

Forward Checking



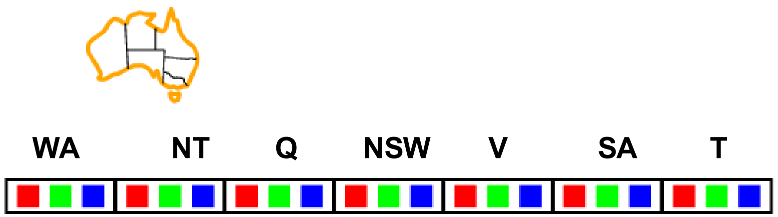
- When X is assigned a value …

  FC looks at each unassigned variable Y that is connected to X by a constraint and deletes from Y ’s domain any value that is inconsistent with the value chosen for X

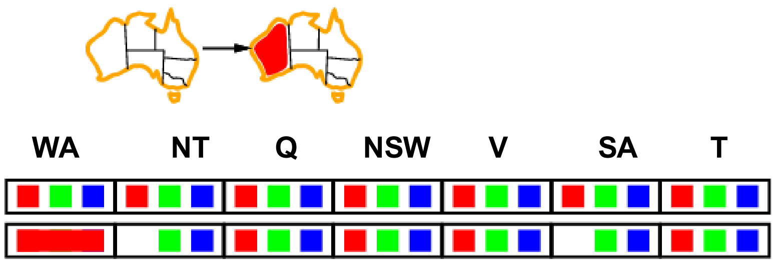
  - Each node in the search tree must contain the state and the list of possible values
  - MRV is an obvious partner of FC

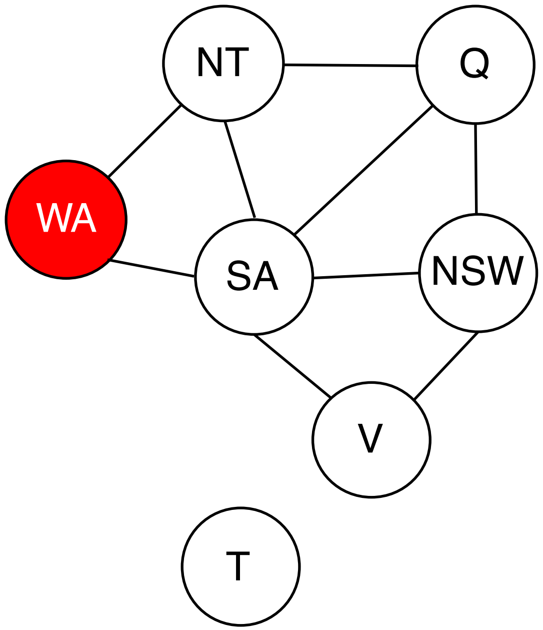
Example of FC

- Initially



- WA=red

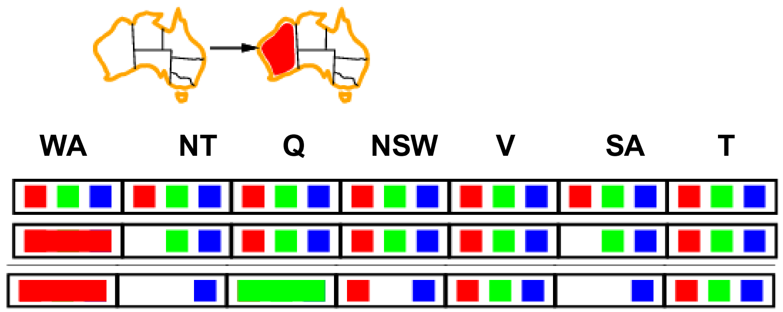


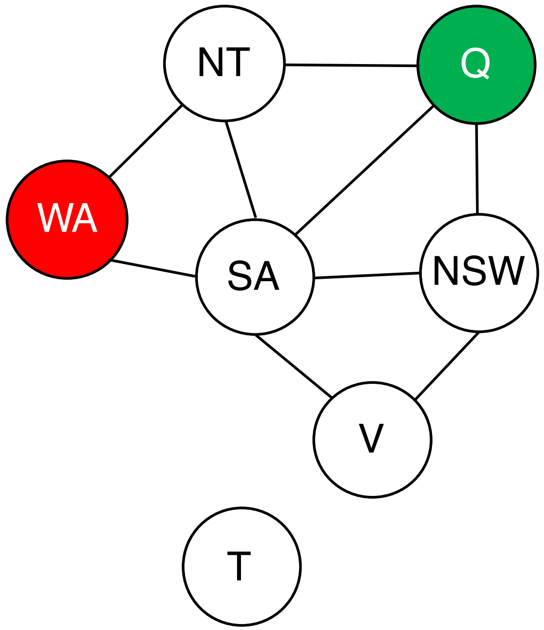


Red value is removed from NT and SA domains

Example of FC

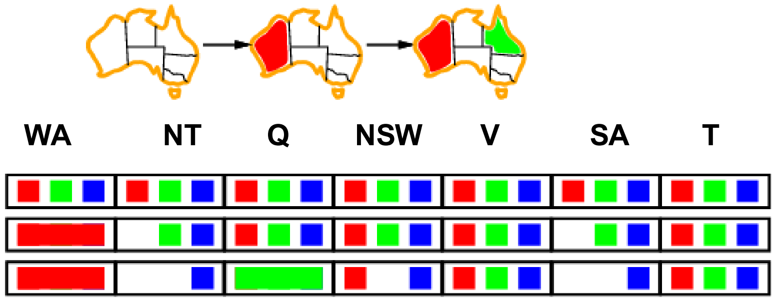
- WA=red Q=green

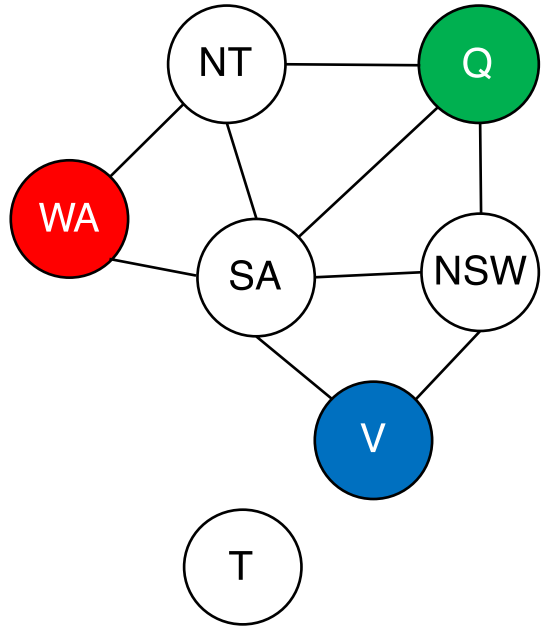




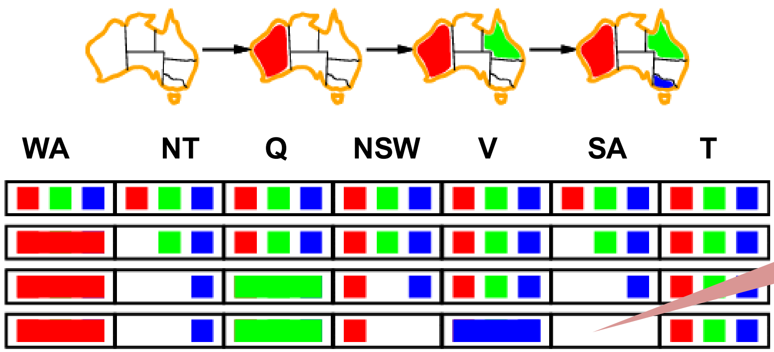
Example of FC

- WA=red   Q=green





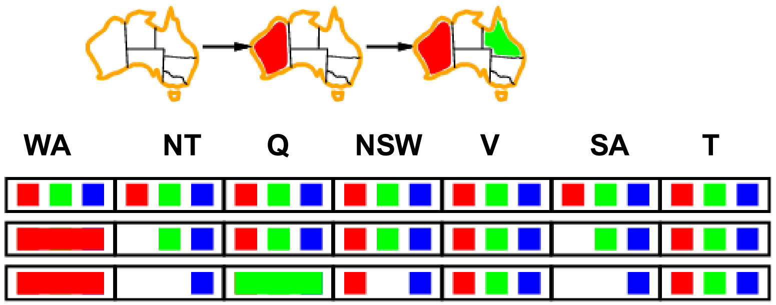
- WA=red   Q=green   V=blue

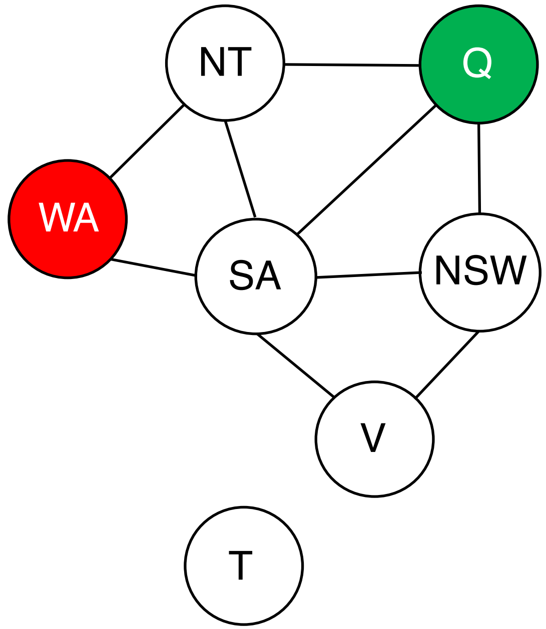


SA domain is empty !!

Example of FC

- LIMITATION: Although forward checking detects many inconsistencies, it does not detect all of them





- NT and SA were both blue !!

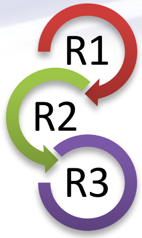
Forward checking algorithm

- Forward checking can be seen as the application of a simple step of arc-consistency between the variable that has been assigned a value and each of the variables that remain to be instantiated:
  1. Select xi.
  2. Assign $xi \leftarrow aj : aj \in Di$.
  3. REPEAT:
     1. forward-check:

        Remove from the domains of the variables (xi+1.. xn), those values that are inconsistent with respect to the assignment (xi, aj), according to the set of constraints.

        Increment i
  4. UNTIL i>n
  5. If there exists a unassigned variable, and its domain is empty then retract assignment $xi \leftarrow aj$. Do:
     - Try with other values of *Di*, go to step(2).
     - If Di is empty:
       - If $i > 1$, decrement *i (try with previous variable)* and go back to step (2).
       - If $i = 1$, exit (No Solution).

Limitations of Forward Checking



Forward Checking

Propagation of constraints

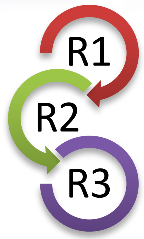
NO COMPLETE	Speed is required to be efficient

ALTERNATIVE:

Arc-consistent VARIABLES

4. Constraint Propagation

arc-consistency AC3



- **Arc-Consistent:** when for each pair of variables (X, Y) and for each value x_i of D_x there exists a value y_j of D_y such as Constraints are satisfied
  - *Current domains must be consistent with all the constraints*

4.1 Arc-consistency

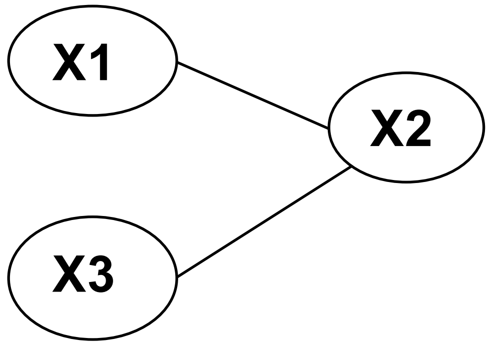
D1={1..10}

D2={5..15}

D3={8..15}

Constraints:

- X1>X2
- X2>X3

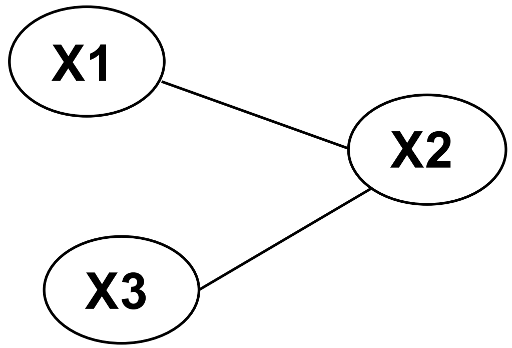


4.1 Arc-consistency

D1={1..10}

D2={5..15}

D3={8..15}



Constraint1: X1>X2

Constraint2: X2>X3

Before any assignment:

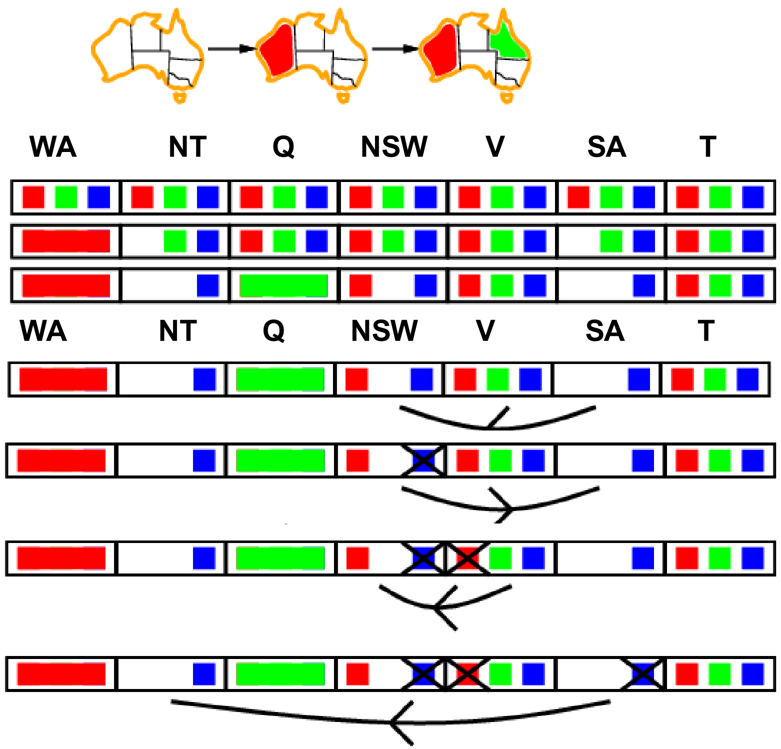
1. In order to satisfy Constraint1 and make Arc(1,2) a consistent arc,
   - D1 = {6..10} and D2 = {5..9}
2. To make Arc(2,3) consistent
   - D2={9} and D3={8}
3. Next iteration,
   - D1={10}, D2={9} and D3={8}
4. Each domain is now unique, solution has been reached

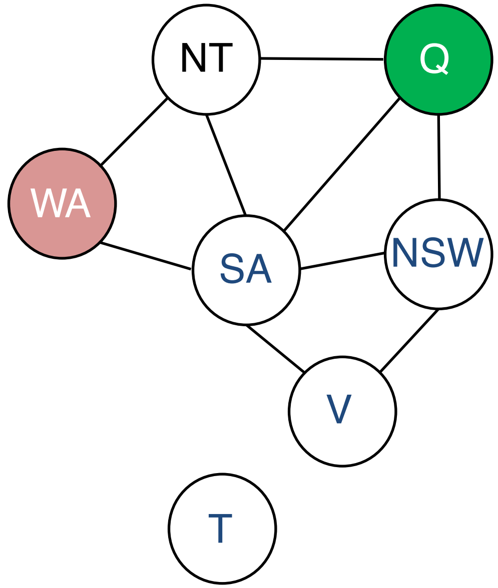
4.1 ARC-CONSISTENCY

(X Y) Is consistent if and only if:

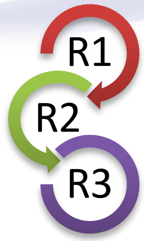
- For each value x_i of X there exists some allowed value y_j

- WA=red Q=green





4.2 AC3 Algorithm



Procedure (intuitive idea):

From initial domains:

- Update domain in each step
- Return a set of updated domains where all arcs are consistent

Domains update:

- If an arc is inconsistent, try to remove from the distinguished variable those values that do not satisfy any constraint

Stop criteria

- All arcs are consistent
- Inconsistency: A domain is empty

4.2 AC3 Algorithm

function AC-3(csp) returns 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 all the arcs in csp
while queue is not empty **do**
$(X_i, X_j) \leftarrow$ 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 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 constraint between X_i and X_j
    then delete x from DOMAIN[X_i];
    removed ← true
return removed

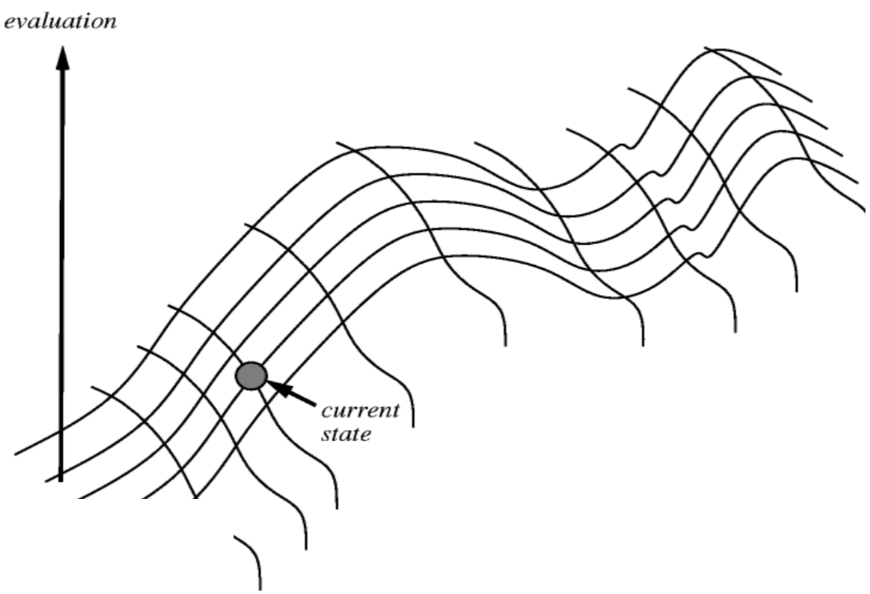
4.2 AC3 Algorithm

- From the application of AC3 we can end with:
  - An empty domain: no solution
  - Unique domain: a unique solution
  - At least a domain is not unique: more than one solution might exist

- AC3 can be used in combination with any other search strategy:
  - Backtracking and backjumping
  - Local search and heuristcs of minimum conflicts

5. Local or Hill Climbing Search

- Loop that continuously moves forward:
  - Increasing values (if the goal is to maximize the evaluation function)
  - Decreasing values (if the goal is to minimize the evaluation function)



5.1 Local Search

For combinatorial optimization problems

1. Start with initial configuration
2. Repeatedly search neighborhood (Successors) and select the best neighbor as candidate
3. Apply a cost function (or fitness function) and accept candidate if it is better than current
4. Stop if quality is sufficiently high, if no improvement can be found or after some fixed time

- Candidate is always and only accepted if cost is lower than current configuration
- Stop when no neighbor with lower cost (higher fitness) can be found

5.1 Local or Hill-climbing Search

```
neighbor = successor of actual with min(feval)
if feval(neighbor) < feval(current)
           current = neighbor
```

Successors(current)

- *m=state with min(feval(Successors))*

IF *feval(m) < feval(current)*

- current=m

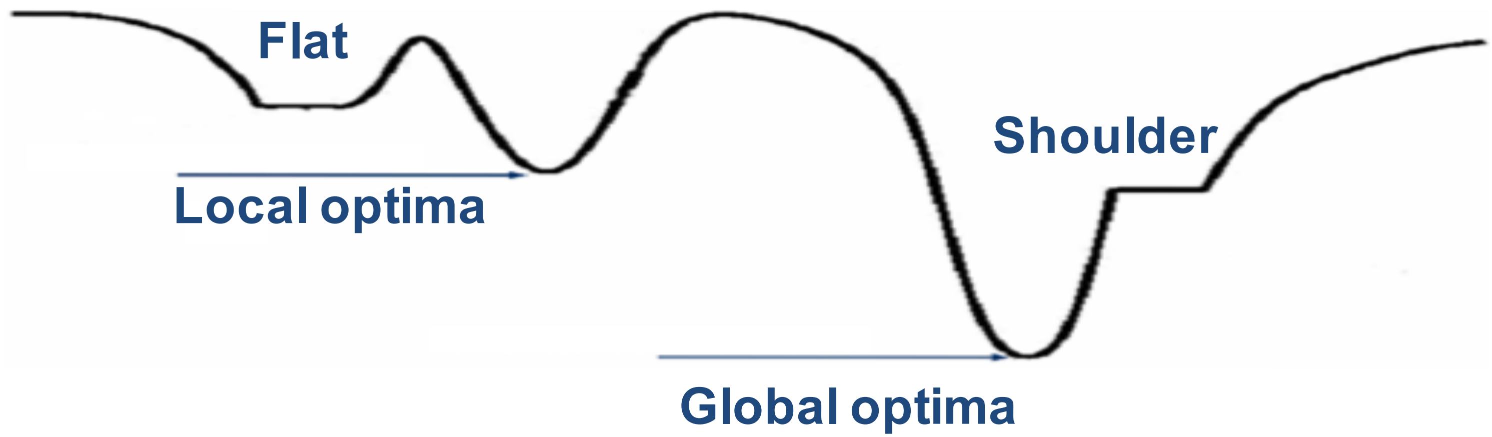
Continue the trace through states that decrease the evaluation function

It ends when a peak is reached: where none neighbor has a lower value of feval

5.2 Problems with Local Search

It can get stuck in local optima

…



5.3 Implementation

1. A method to generate initial configuration
2. A Successor function to generate new states
3. A Cost function
4. A Decision Criterion to select next candidates from the list of successors
5. A Stop Criterion

5.3 Alternative: Beam Local Search

- Begin with k random generated states
- Loop until the solution state is found
  - Generate the list of all the successors of the k States.
  - Select the k best states from this list

5.4 Local Search for CSP

- **States:** They use a complete-state formulation (consistent or inconsistent)
- **Initial State:** random generated
- **Final State:** Solution to CSP
- **Successors:** usually works by changing the value of one variable at a time

Minimum Conflict Heuristics

- Select variable other than the last modified one that participates in more unsatisfied constraints in the state
- Select value that causes the least number of conflicts with other variables (Least Restricted Value)

5.5 Local Search Main Features

When the path to the Solution is irrelevant:

- They keep only one State in memory: the current State
- They move only to the neighbouring nodes of the current node
- They are not systematic in the search
- They use little memory
- They can find reasonable solutions in large or infinite spaces of States
- They can get stuck in local maxima/minima

References

- Russell, S. y Norvig, P. *Artificial Intelligence (a modern approach).* Ch. 5: "Constraint Satisfaction Problems"
- Schalkoff, R.J. *Intelligent Systems: Principles, Paradigms and Pragmatics. Ch. 4*
- Poole, D.; Mackworth, A. y Goebel, R. *Computational Intelligence (A Logical Approach) (Oxford University Press,* 1998) "Constraint Satisfaction Problems"