Constraint Satisfaction Problem

Instructor

LE Thanh Sach, Ph.D.

Instructor's Information

LE Thanh Sach, Ph.D.

Office:

Department of Computer Science,
Faculty of Computer Science and Engineering,
HoChiMinh City University of Technology.

Office Address:

268 LyThuongKiet Str., Dist. 10, HoChiMinh City, Vietnam.

E-mail: LTSACH@hcmut.edu.vn

E-home: http://cse.hcmut.edu.vn/~ltsach/

Tel: (+84) 83-864-7256 (Ext: 5839)

Acknowledgment

The slides in this PPT file are composed using the materials supplied by

- **Prof. Stuart Russell and Peter Norvig**: They are currently from University of California, Berkeley. They are also the author of the book "Artificial Intelligence: A Modern Approach", which is used as the textbook for the course

- **Prof. Tom Lenaerts**, from Université Libre de Bruxelles

Outline

- CSP - Introduction
- Backtracking for CSP
- Local search for CSPs
- Problem structure and decomposition

Constraint Satisfaction Problem: Introduction

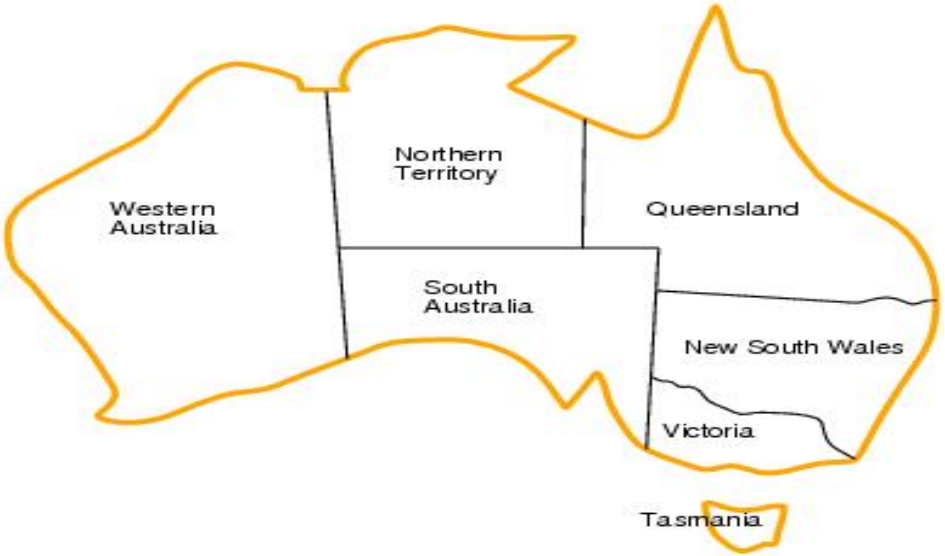
Constraint satisfaction problems

- What is a CSP?
  - Finite set of variables $V_1, V_2, \ldots, V_n$
  - Finite set of constraints $C_1, C_2, \ldots, C_m$
  - Nonemtpy domain of possible values for each variable $D_{V1}, D_{V2}, \ldots D_{Vn}$
  - Each constraint C_i limits the values that variables can take, e.g., $V_1 \neq V_2$
- A *state* is defined as an *assignment* of values to some or all variables. Trạng thái là một phép gán giá trị cho biến
- *Consistent assignment*: assignment does not not violate the constraints.

Constraint satisfaction problems

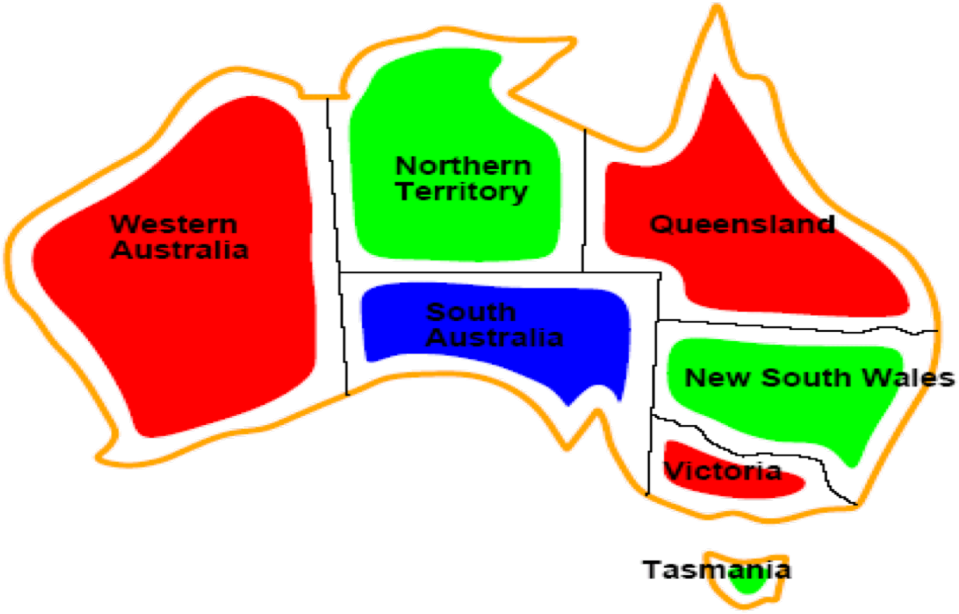
- An assignment is *complete:* every variable is assigned.
- A *solution* to a CSP is a complete assignment that satisfies all constraints.
- Some CSPs require a solution that maximizes an *objective function*.
- Applications: Scheduling the time of observations on the Hubble Space Telescope, Floor planning, Map coloring, Cryptography, Scheduling, etc

CSP 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* (if the language allows this)
  - ✓ E.g. *(WA,NT)* ≠ *{(red,green),(red,blue),(green,red),…}*

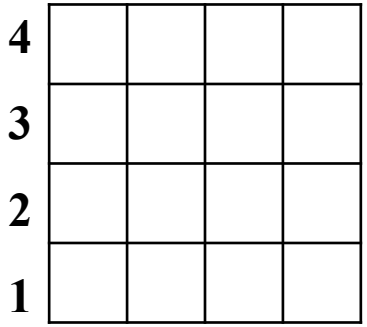
CSP example: map coloring



- Solutions are assignments satisfying all constraints, e.g.

  $\{WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green\}$

CSP example: n-quees

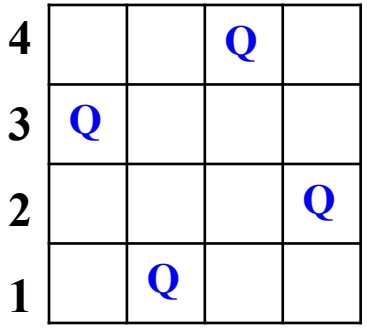


n-queens (n=4)

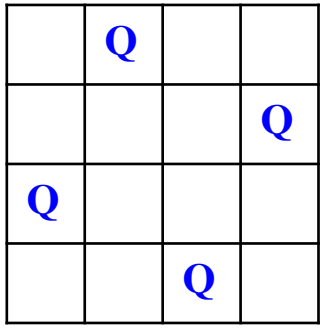
Placing four queens on the board
so that they are not attacked each other

- Variables: four vars, each for a queen; Q_1, Q_2, Q_3, and Q_4.
- Domains: each queen can move in column, domain: {1,2,3,4}
- Constraints: non-attacking
  - Non-attacking(Q_i, Q_j); $\forall i \neq j$ in {1,2,3,4}

CSP example: n-quees



A solution

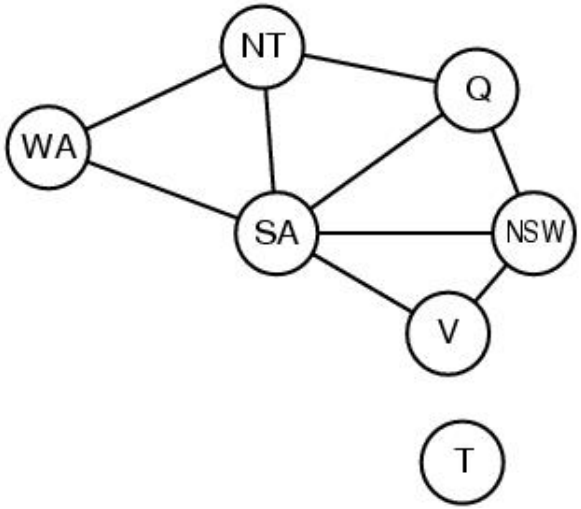


A solution

- May have more than one solution, i.e., complete and consistent assignment
  - $\{Q_1=3, Q_2=1, Q_3=4, Q_4=2\}$
  - $\{Q_1=2, Q_2=4, Q_3=1, Q_4=3\}$

Constraint graph

- CSP benefits
  - Standard representation pattern
  - Generic goal and successor functions
  - Generic heuristics (no domain specific expertise).

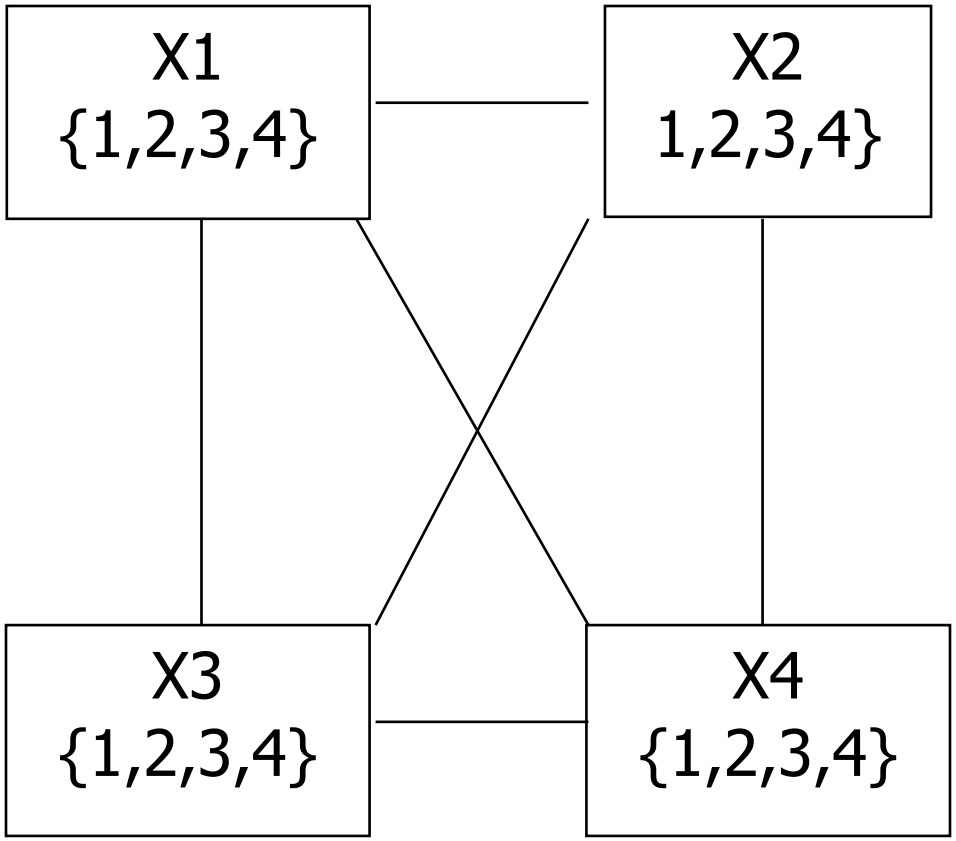


- Constraint graph = nodes are variables, edges show constraints.
  - Graph can be used to simplify search.
    - ✓ e.g. Tasmania is an independent subproblem.

Nodes: variables, edges: constraints

- Question: how can we generate the constraint graph automatically (for map coloring)?

Constraint graph



4-queens constraint graph

Varieties of CSPs

- Discrete variables
  - Finite domains; size $d \Rightarrow O(d^n)$ complete assignments.
    - ✓ E.g. Boolean CSPs, include. 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 $StartJob_1 + 5 \leq StartJob_3$.
    - ✓ Linear constraints solvable, nonlinear undecidable.
- Continuous variables
  - e.g. start/end times for Hubble Telescope observations.
  - Linear constraints solvable in poly time by LP methods.

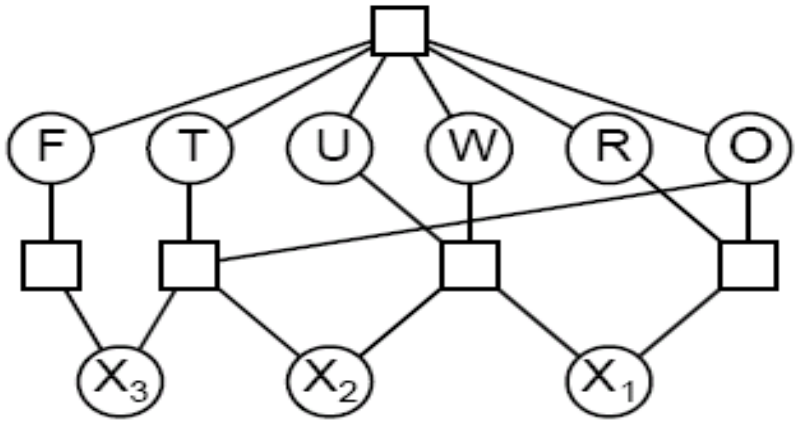
Varieties of constraints

+ Unary Constraint
+ Binary Constraint
+ Higher-order Constraint
+ Soft Constraint

- 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. cryptharithmetic column constraints.
- Preference (soft constraints) e.g. *red* is better than *green* often representable by a cost for each variable assignment → constrained optimization problems.

Example; cryptharithmetic

$$\begin{array}{cccc} & T & W & O \\ + & T & W & O \\ \hline F & O & U & R \end{array}$$



Variables: $F\ T\ U\ W\ R\ O\ X_1\ X_2\ X_3$
Domains: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
Constraints
alldiff(F, T, U, W, R, O)
$O + O = R + 10 \cdot X_1$, etc.

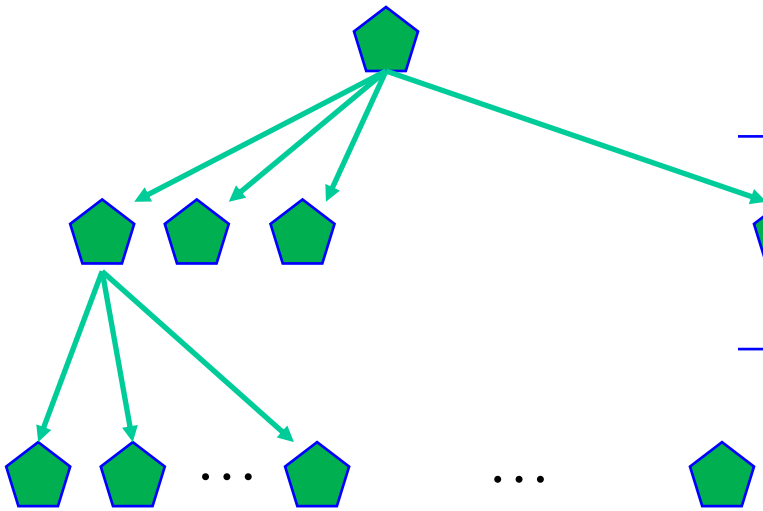
CSP as a standard search problem

- A CSP can easily expressed as a standard search problem.
- Incremental formulation
  - *Initial State*: the empty assignment {}.
  - *Successor function*: Assign value to unassigned variable provided that there is not conflict.
  - *Goal test*: the current assignment is complete.
  - *Path cost*: as constant cost for every step.

CSP as a standard search problem

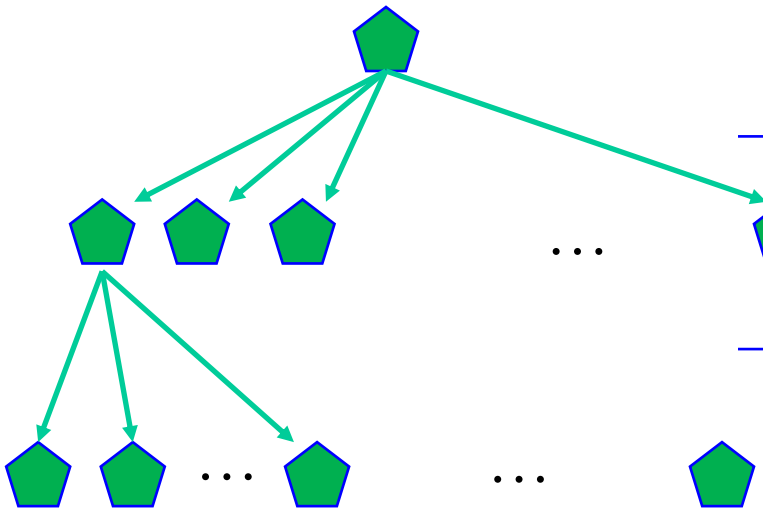
- This is the same for all CSP's !!!
- Solution is found at depth *n* (if there are *n* variables).
  - Hence depth first search can be used.
- Path is irrelevant, so complete state representation can also be used (local search).

CSP as a standard search problem



#vars not assigned	#values possible/var	#moves possible
n	d	nd
(n-1)	d	$n(n-1)d^2$
(n-2)	d	$n(n-1)(n-2)d^3$
⋮	⋮	⋮
1	d	$n!d^n$

CSP as a standard search problem



#vars not assigned	#values possible/var	#moves possible
n	d	nd
(n-1)	d	$n(n-1)d^2$
(n-2)	d	$n(n-1)(n-2)d^3$
⋮	⋮	⋮
1	d	**Impractical** $n!d^n$

Commutativity

- CSPs are commutative.
  - The order of any given set of actions has no effect on the outcome.
  - Example: choose colors for Australian territories one at a time
    - ✓ [WA=red then NT=green] same as [NT=green then WA=red]
    - ✓ All CSP search algorithms consider a single variable assignment at a time $\Rightarrow$ there are d^n leaves.

Backtracking search

Backtracking search

- Cfr. Depth-first search
- Chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.
- Uninformed algorithm
  - No good general performance (see table p. 143)

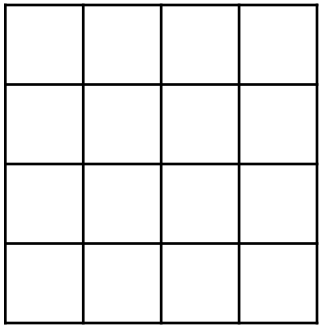
Backtracking search

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

function RECURSIVE-BACKTRACKING(*assignment, csp*) **return** a solution or failure
    if *assignment* is complete **then return** *assignment* check if the problem is complete after assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*],*assignment*,*csp*)
    for each *value* **in** ORDER-DOMAIN-VALUES(*var, assignment, csp*) **do**
Assign all possible values to var **if** *value* is consistent with *assignment* according to CONSTRAINTS[*csp*] **then**
            add *{var=value}* to assignment assignment keeps assign actions for next state
            result ← RECURSIVE-BACTRACKING(*assignment, csp*)
            if *result* ≠ *failure* **then return** *result*
            remove *{var=value}* from *assignment*
    return *failure*

consistent: satisfying

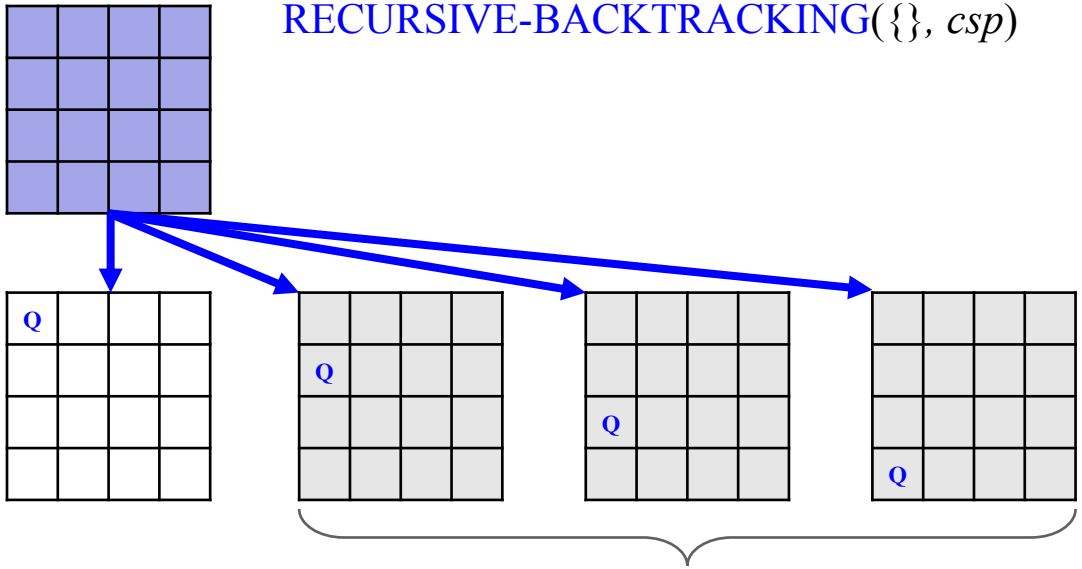
Backtracking example



4-queens

Placing four queens on the board
so that they are not attacked each other

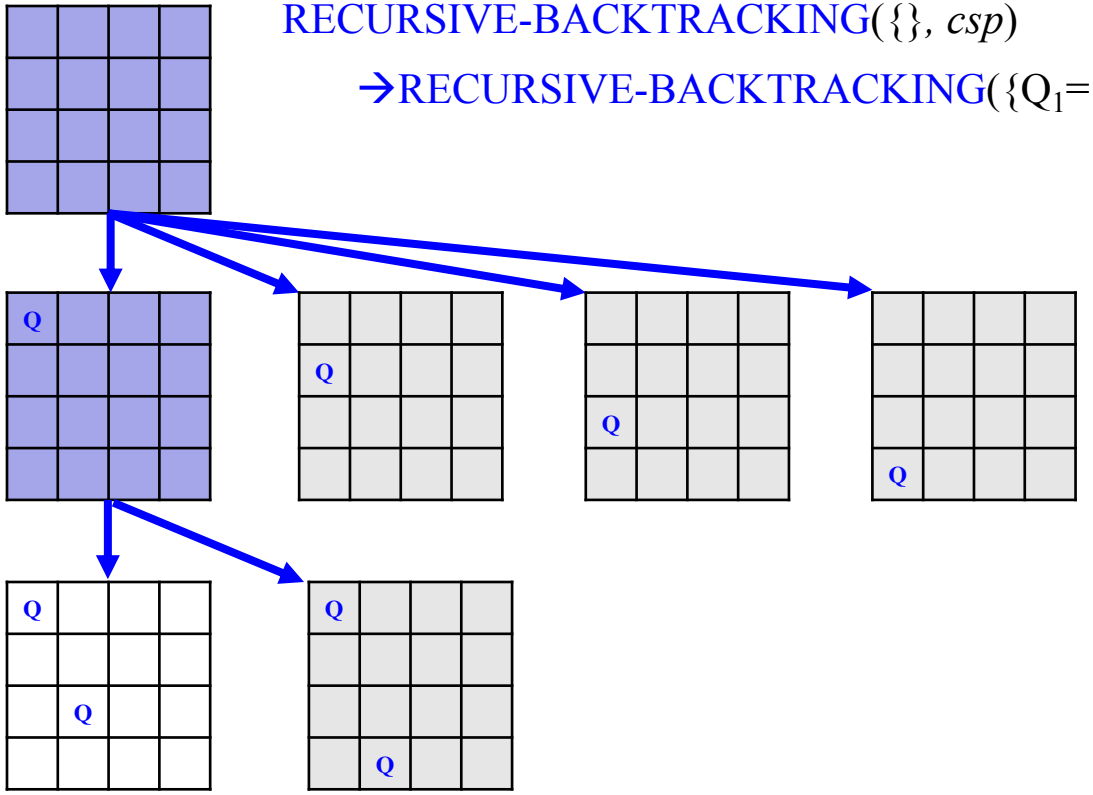
RECURSIVE-BACKTRACKING({}, *csp*)



not generated yet! ➔ efficient using of memory

RECURSIVE-BACKTRACKING({}, *csp*)

→RECURSIVE-BACKTRACKING({Q_1=1}, *csp*)



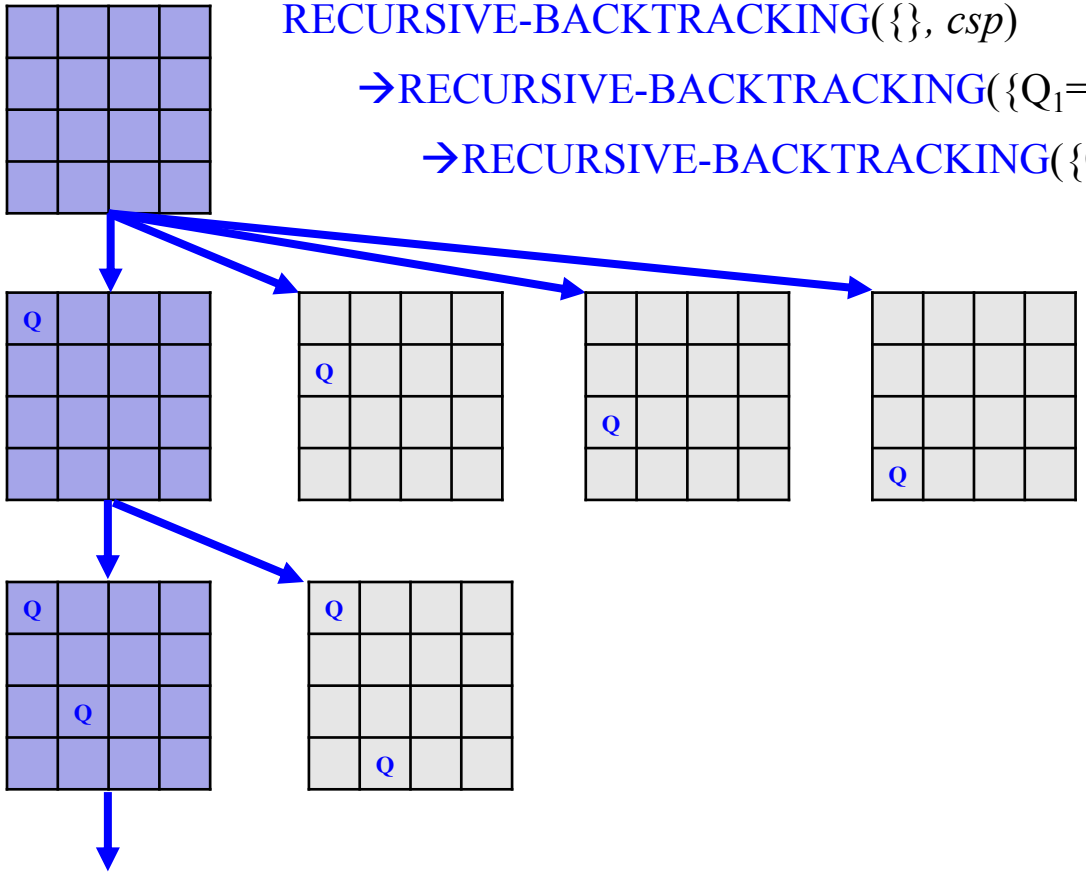
Node: state

Edge: function (assignment)

RECURSIVE-BACKTRACKING({}, *csp*)

→RECURSIVE-BACKTRACKING({Q_1=1}, *csp*)

→RECURSIVE-BACKTRACKING({Q_1=1, Q_2=3}, *csp*)



no valid move

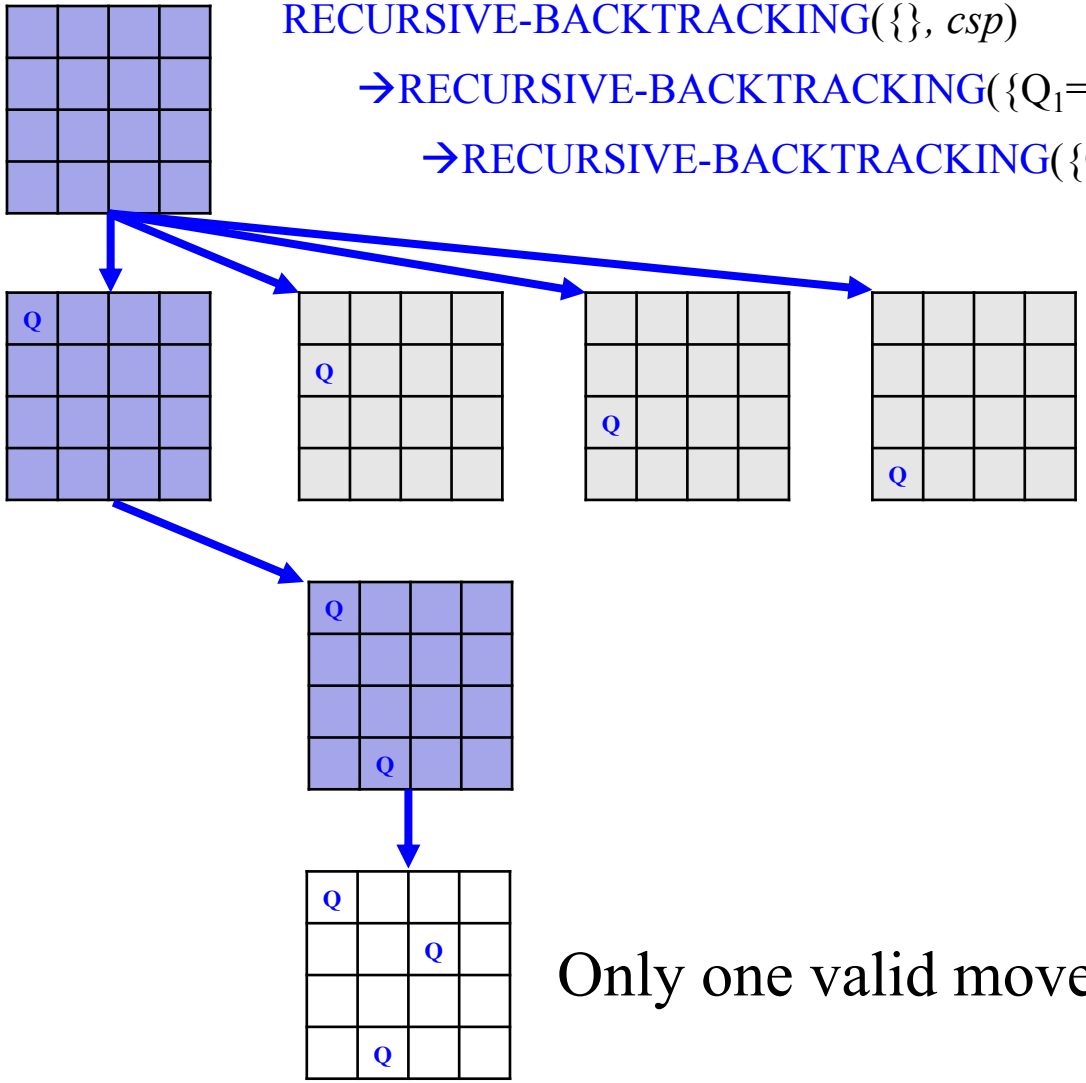
RECURSIVE-BACKTRACKING({Q_1=1, Q_2=3}, *csp*) return *failure*

i.e, do *backtracking*

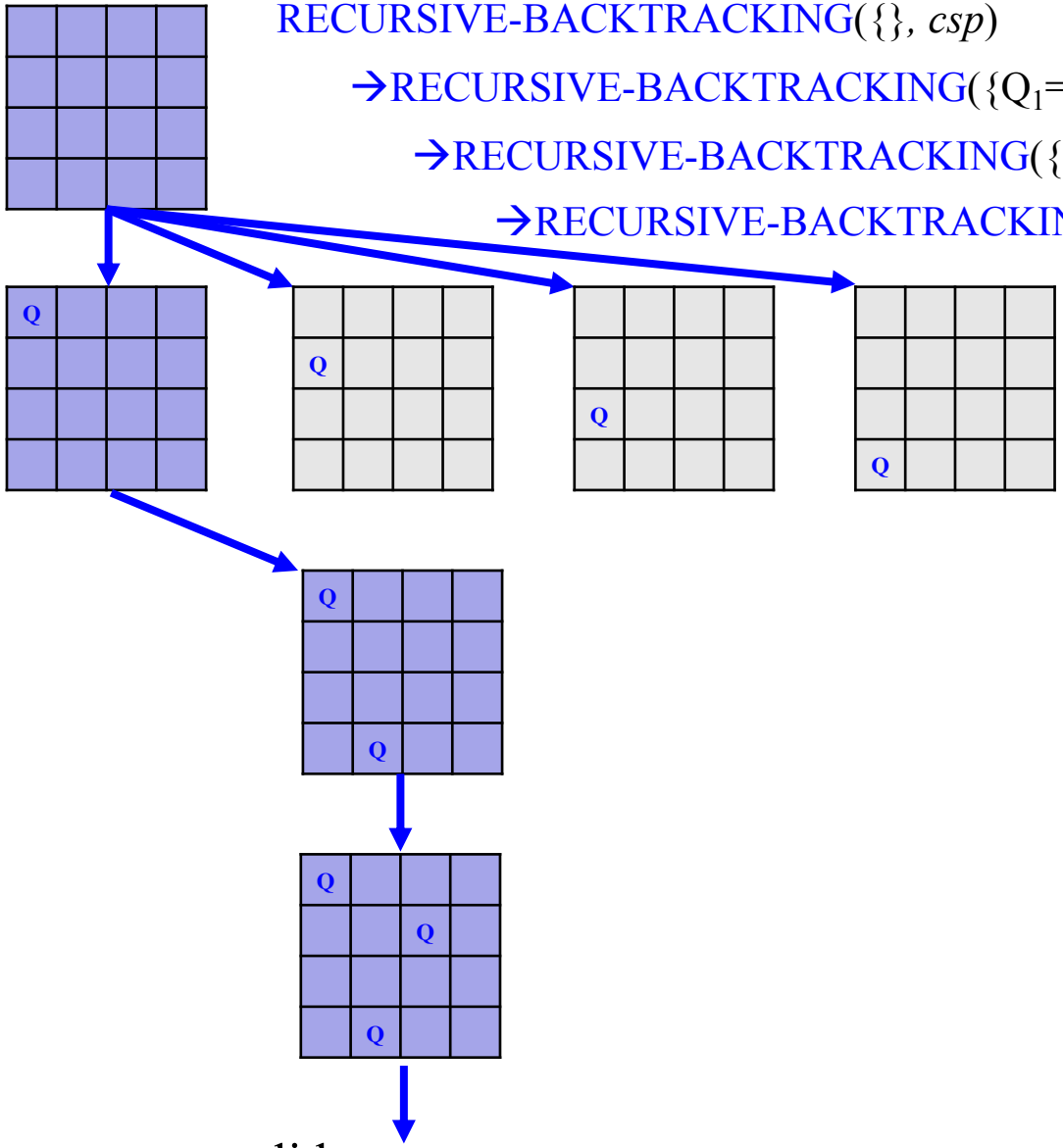
RECURSIVE-BACKTRACKING({}, *csp*)

→RECURSIVE-BACKTRACKING({Q_1=1}, *csp*)

→RECURSIVE-BACKTRACKING({Q_1=1, Q_2=4}, *csp*)



Only one valid move



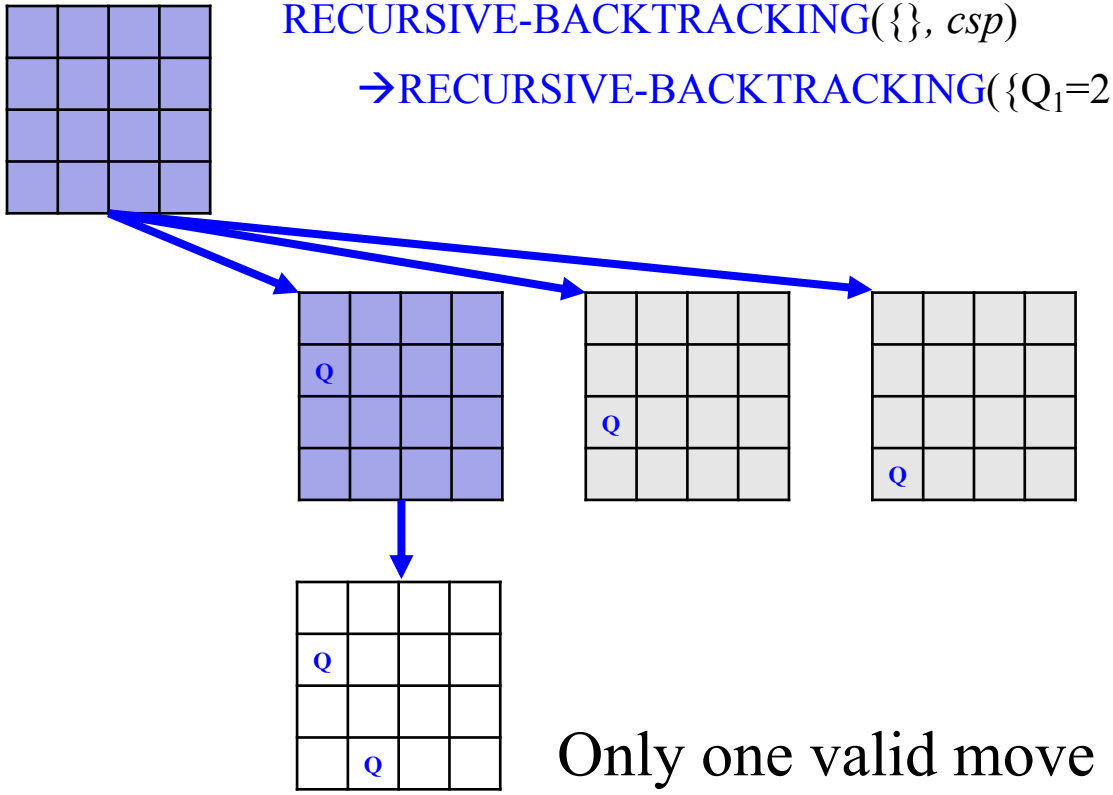
no valid move

RECURSIVE-BACKTRACKING({Q_1=1, Q_2=4, Q_3=2}, *csp*) return *failure*

i.e, do *backtracking*

RECURSIVE-BACKTRACKING({}, *csp*)

→RECURSIVE-BACKTRACKING({Q_1=2}, *csp*)

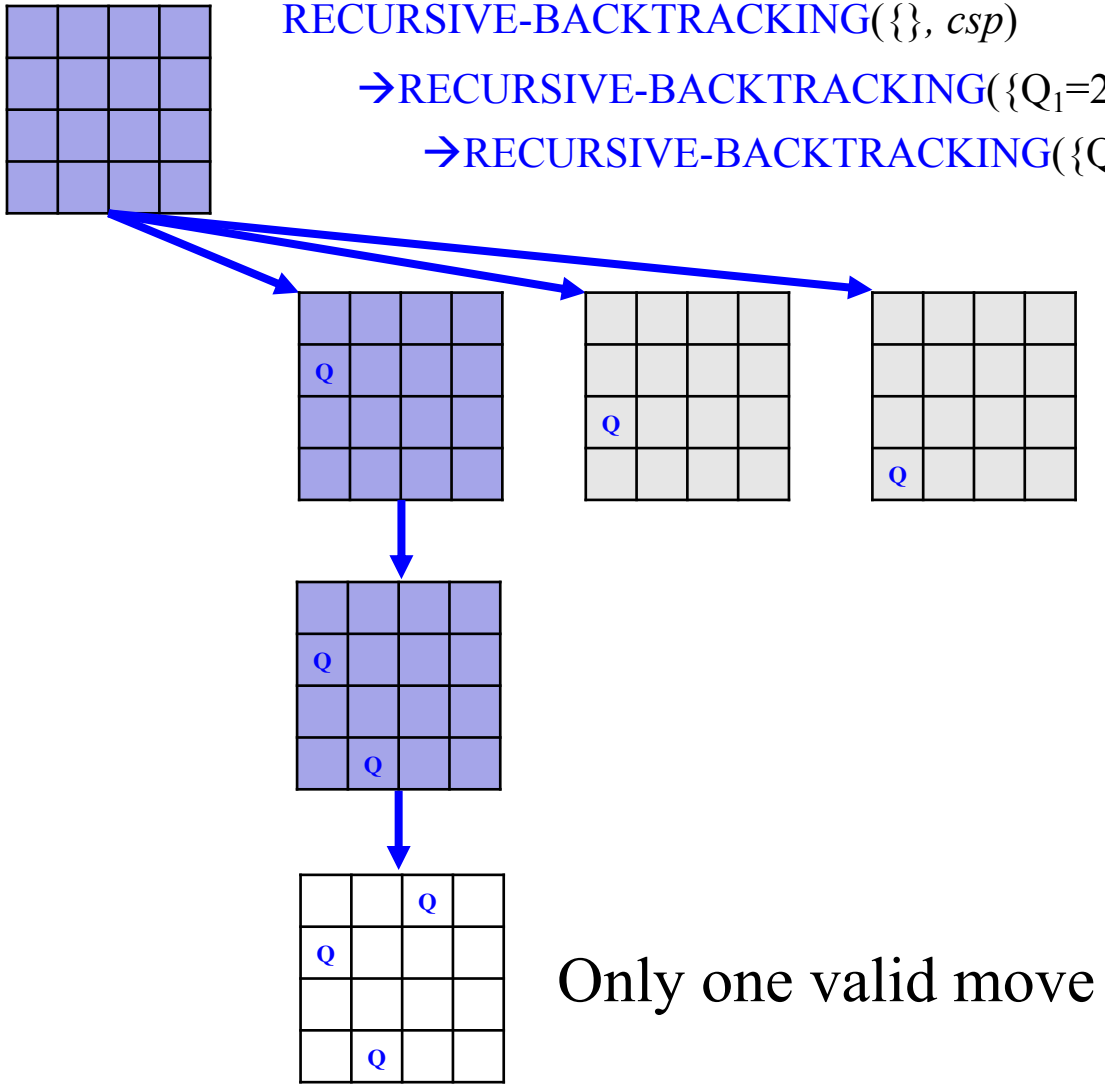


Only one valid move

RECURSIVE-BACKTRACKING({}, *csp*)

→RECURSIVE-BACKTRACKING({Q_1=2}, *csp*)

→RECURSIVE-BACKTRACKING({Q_1=2, Q_2=4}, *csp*)



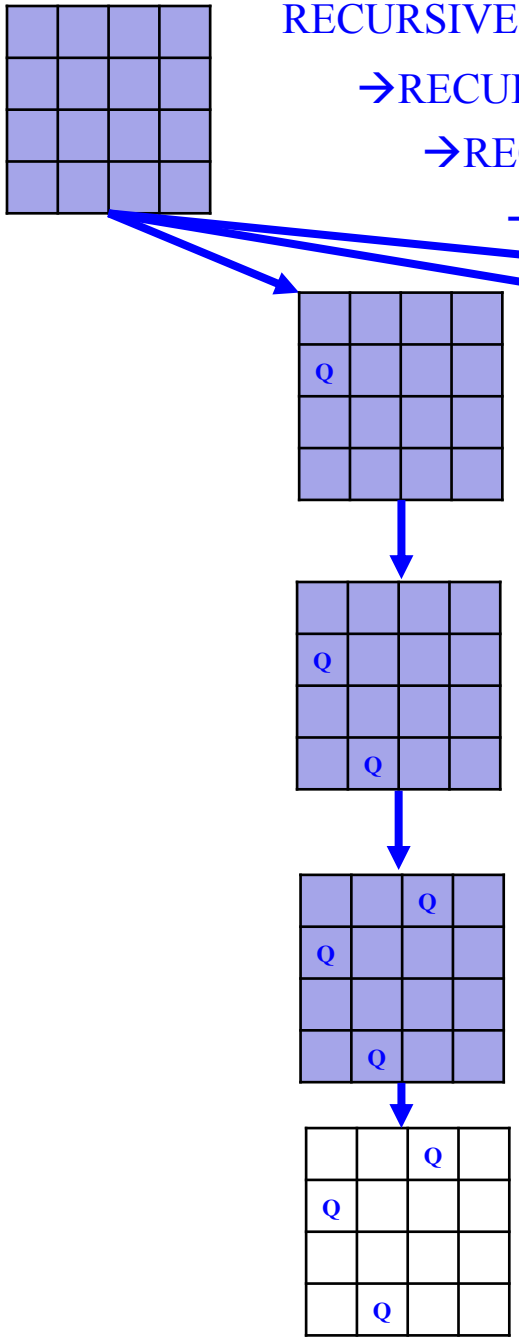
Only one valid move

RECURSIVE-BACKTRACKING({}, *csp*)

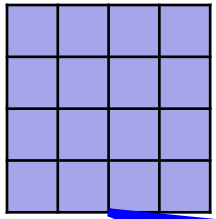
→RECURSIVE-BACKTRACKING({Q_1=2}, *csp*)

→RECURSIVE-BACKTRACKING({Q_1=2, Q_2=4}, *csp*)

→RECURSIVE-BACKTRACKING({Q_1=2, Q_2=4, Q_3=1}, *csp*)



Only one valid move



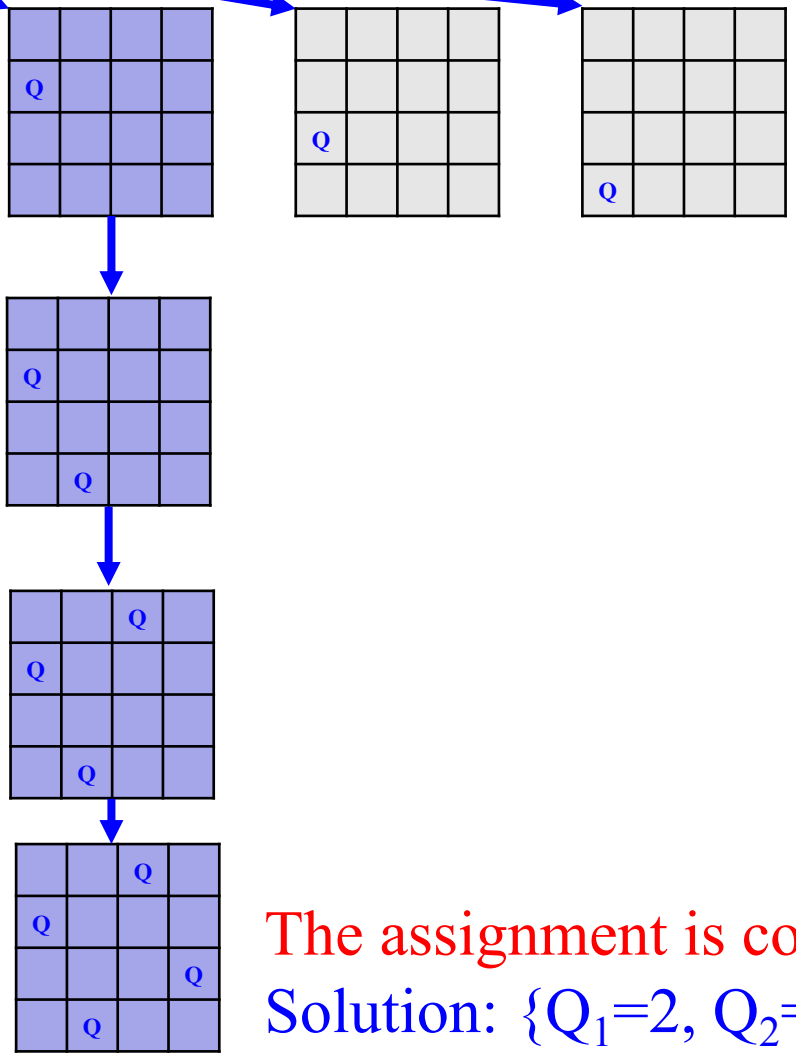
RECURSIVE-BACKTRACKING({}, *csp*)

→RECURSIVE-BACKTRACKING({Q_1=2}, *csp*)

→RECURSIVE-BACKTRACKING({Q_1=2, Q_2=4}, *csp*)

→RECURSIVE-BACKTRACKING({Q_1=2, Q_2=4, Q_3=1}, *csp*)

→RECURSIVE-BACKTRACKING({Q_1=2, Q_2=4, Q_3=1, Q_4=3}, *csp*)



The assignment is complete => stop successfully

Solution: {Q_1=2, Q_2=4, Q_3=1, Q_4=3}

Heuristics for CSP

Improving backtracking efficiency

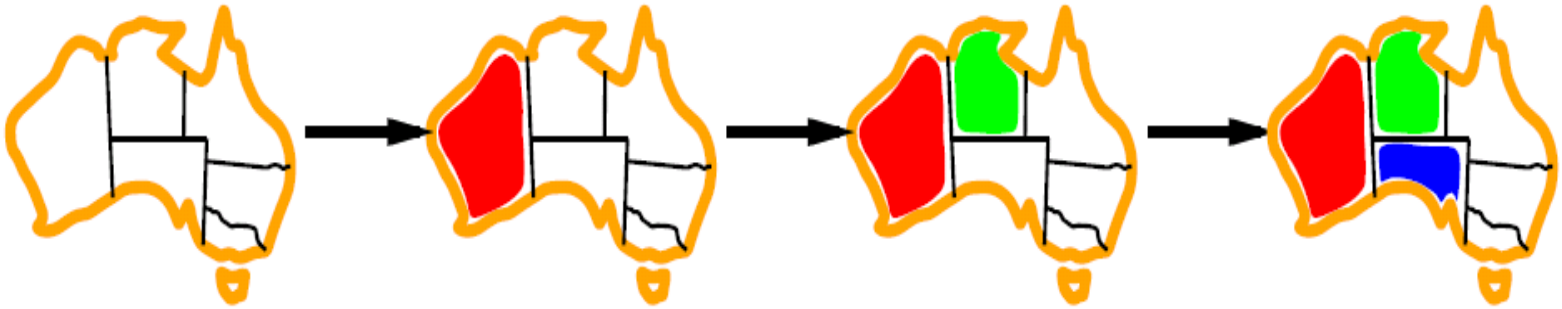
- General-purpose methods can give huge gains in speed:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?
  - Can we take advantage of problem structure?

Minimum remaining values (MRV)

Choose variable with the fewest legal moves ==> Decrease number of backtracking

Decrease legal value rendered by ORDER-DOMAIN-VALUES

Algorithm for: SELECT-UNASSIGNED-VARIABLE



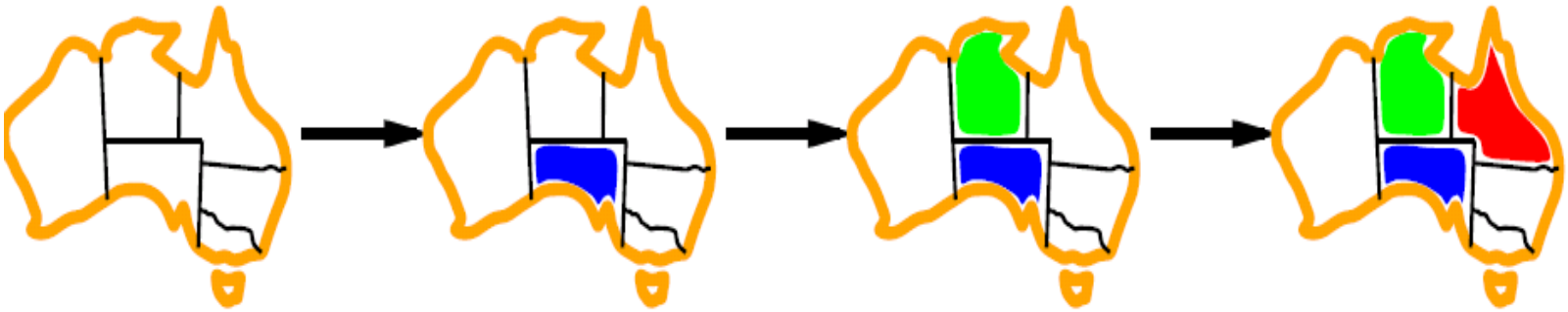
var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*],*assignment*,*csp*)

- A.k.a. most constrained variable heuristic
- *Rule*: choose variable with the fewest legal moves
- *Which variable shall we try first?*
- *MRV means "if we need to backtrack, should do backtracking on shallow levels"*

Degree heuristic

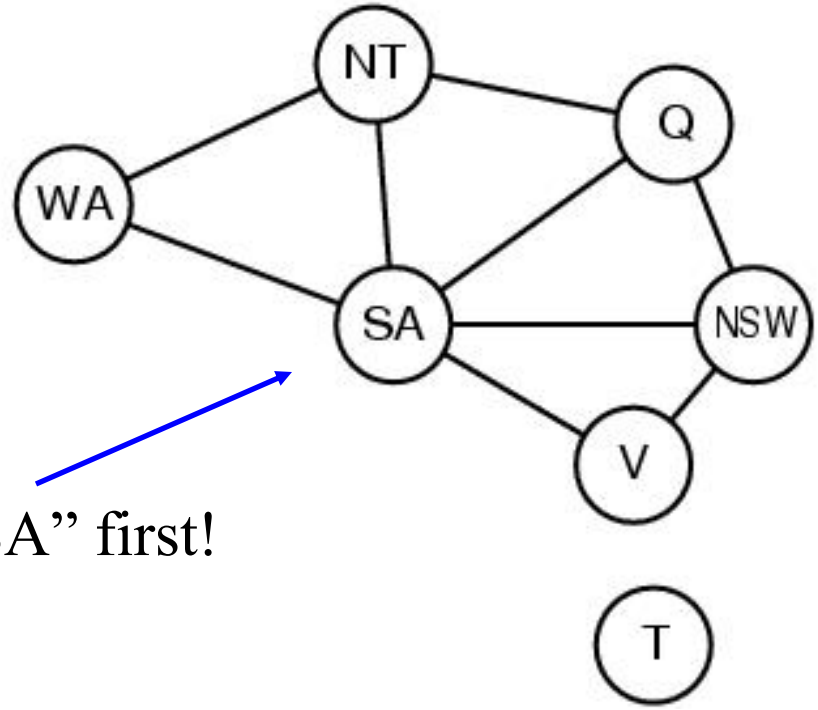
Use

Choose variable having the most constraints with other UNASSIGNED variables



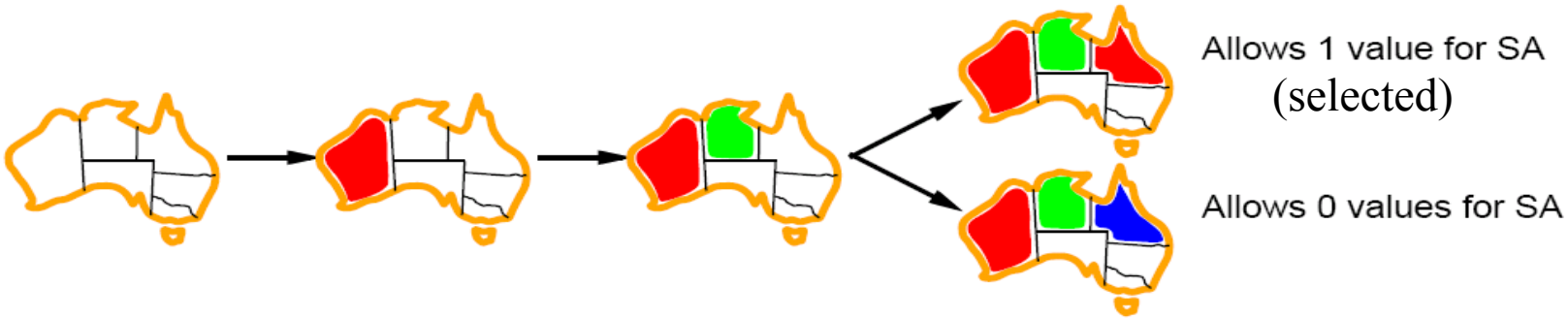
- Use degree heuristic
- *Rule*: select variable that is involved in the largest number of constraints on other unassigned variables.
- Degree heuristic is very useful as a tie breaker.
- *In what order should its values be tried?*
- *Degree heuristic: "involved in many constraints => easier to violate constraints => do backtracking on shallow levels"*

Degree heuristic



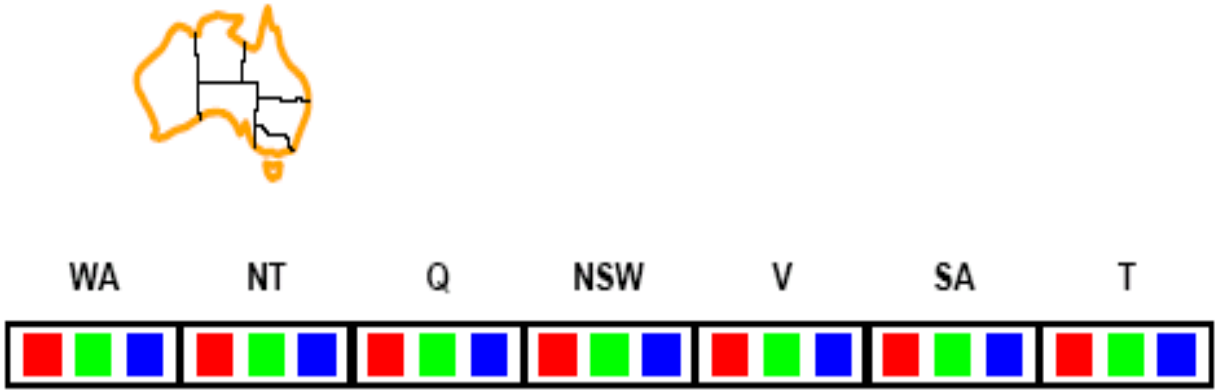
Should try with “SA” first!

Least constraining value



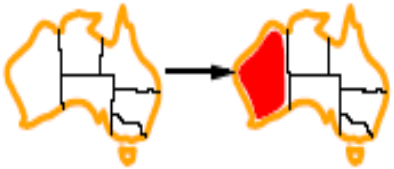
- Least constraining value heuristic
- Rule: given a variable choose the least constraing value i.e. the one that leaves the maximum flexibility for subsequent variable assignments.
  - leaves the maximum flexibility => hope to have a solution

Forward checking



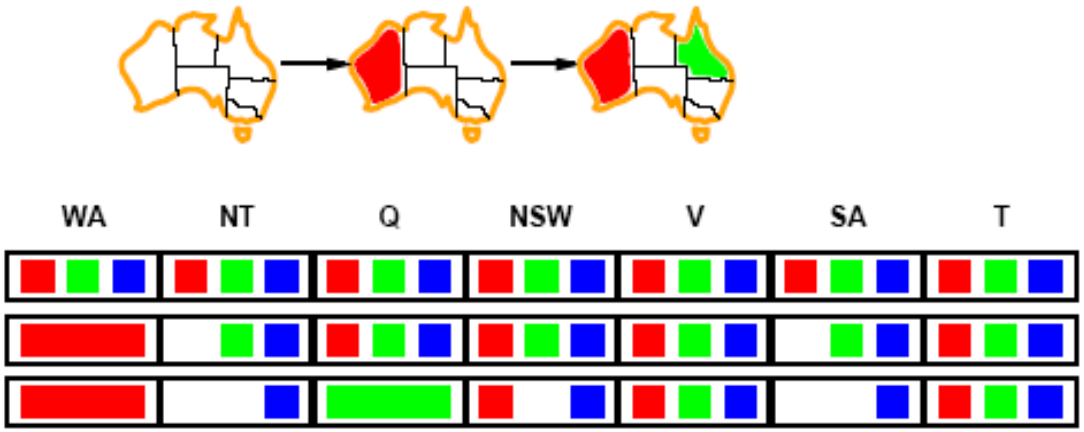
- Can we detect inevitable failure early?
  - *And avoid it later?*
- *Forward checking idea:* keep track of remaining legal values for unassigned variables.
- Terminate search when any variable has no legal values.

Forward checking



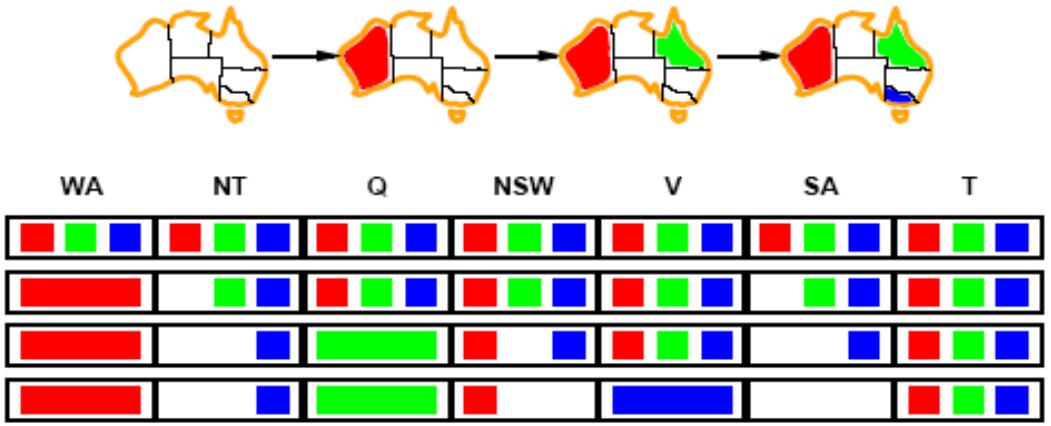
- Assign *{WA=red}*
- Effects on other variables connected by constraints with WA
  - *NT can no longer be red*
  - *SA can no longer be red*

Forward checking



- Assign *{Q=green}*
- Effects on other variables connected by constraints with WA
  - *NT can no longer be green*
  - *NSW can no longer be green*
  - *SA can no longer be green*
- *MRV heuristic* will automatically select NT and SA next, why?

Forward checking

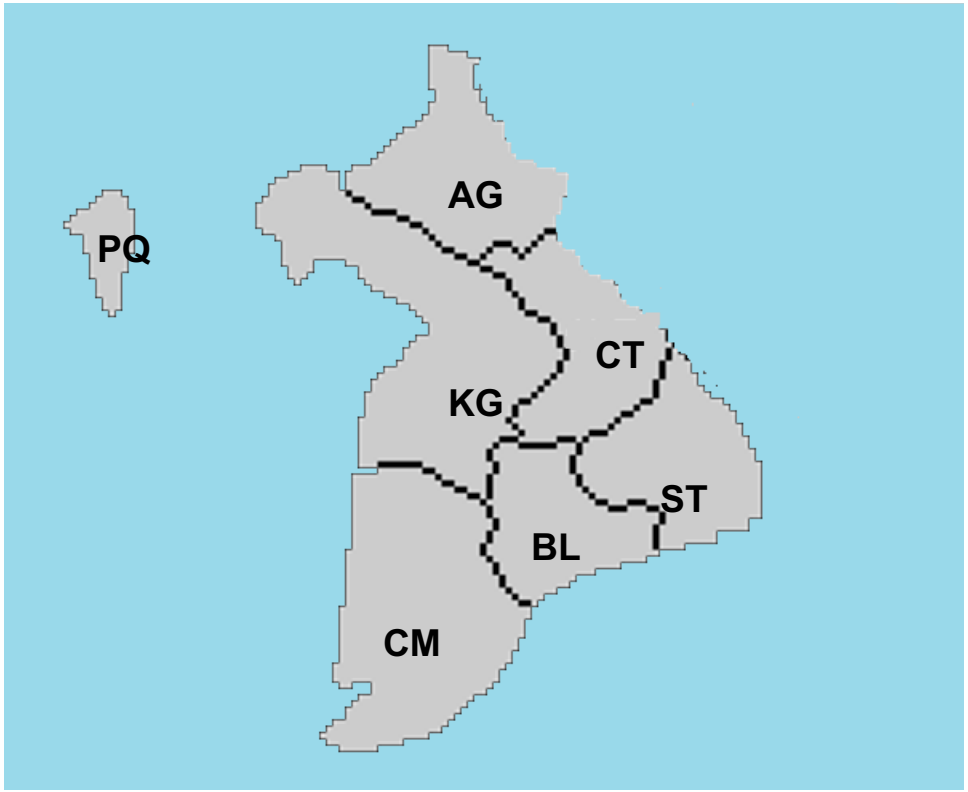


- If *V* is assigned *blue*
- Effects on other variables connected by constraints with WA
  - *SA is empty*
  - *NSW can no longer be blue*
- FC has detected that partial assignment is *inconsistent* with the constraints and backtracking can occur.

CSP's Examples

Map coloring

1. PQ: Phú Quốc
2. KG: Kiên Giang
3. CM: Cà Mâu
4. BL: Bạc Liêu
5. ST: Sóc Trăng
6. CT: Cần Thơ
7. AG: An Giang



Map coloring

Step		PQ	KG	CM	BL	ST	CT	AG
0		RGB	RGB	RGB	RGB	RGB	RGB	RGB
1	MRV	= R						
2								
3								
4								
5								
6								
7								
8								
9								

Map coloring

Step		PQ	KG	CM	BL	ST	CT	AG
0		RGB	RGB	RGB	RGB	RGB	RGB	RGB
1	MRV	= R						
2	FC	= R	RGB	RGB	RGB	RGB	RGB	RGB
3								
4								
5								
6								
7								
8								
9								

Map coloring

Step		PQ	KG	CM	BL	ST	CT	AG
0		RGB	RGB	RGB	RGB	RGB	RGB	RGB
1	MRV	= R						
2	FC	= R	RGB	RGB	RGB	RGB	RGB	RGB
3	MRV	= R	= R					
4								
5								
6								
7								
8								
9								

Map coloring

Step		PQ	KG	CM	BL	ST	CT	AG
0		RGB	RGB	RGB	RGB	RGB	RGB	RGB
1	MRV	= R						
2	FC	= R	RGB	RGB	RGB	RGB	RGB	RGB
3	MRV	= R	= R					
4	FC	= R	= R	GB	GB	RGB	GB	GB
5								
6								
7								
8								
9								

Map coloring

Step		PQ	KG	CM	BL	ST	CT	AG
0		RGB	RGB	RGB	RGB	RGB	RGB	RGB
1	MRV	= R						
2	FC	= R	RGB	RGB	RGB	RGB	RGB	RGB
3	MRV	= R	= R					
4	FC	= R	= R	GB	GB	RGB	GB	GB
5	MRV	= R	= R	=G				
6								
7								
8								
9								

Map coloring

Step		PQ	KG	CM	BL	ST	CT	AG
0		RGB	RGB	RGB	RGB	RGB	RGB	RGB
1	MRV	= R						
2	FC	= R	RGB	RGB	RGB	RGB	RGB	RGB
3	MRV	= R	= R					
4	FC	= R	= R	GB	GB	RGB	GB	GB
5	MRV	= R	= R	=G				
6	FC	= R	= R	=G	B	RGB	GB	GB
7								
8								
9								

Map coloring

Step		PQ	KG	CM	BL	ST	CT	AG
0		RGB	RGB	RGB	RGB	RGB	RGB	RGB
1	MRV	= R						
2	FC	= R	RGB	RGB	RGB	RGB	RGB	RGB
3	MRV	= R	= R					
4	FC	= R	= R	GB	GB	RGB	GB	GB
5	MRV	= R	= R	=G				
6	FC	= R	= R	=G	B	RGB	GB	GB
7	MRV	= R	= R	=G	= B			
8								
9								

Map coloring

Step		PQ	KG	CM	BL	ST	CT	AG
0		RGB	RGB	RGB	RGB	RGB	RGB	RGB
1	MRV	= R						
2	FC	= R	RGB	RGB	RGB	RGB	RGB	RGB
3	MRV	= R	= R					
4	FC	= R	= R	GB	GB	RGB	GB	GB
5	MRV	= R	= R	=G				
6	FC	= R	= R	=G	B	RGB	GB	GB
7	MRV	= R	= R	=G	= B			
8	FC	= R	= R	=G	= B	RG	G	GB
9								

Map coloring

Step		PQ	KG	CM	BL	ST	CT	AG
0		RGB	RGB	RGB	RGB	RGB	RGB	RGB
1	MRV	= R						
2	FC	= R	RGB	RGB	RGB	RGB	RGB	RGB
3	MRV	= R	= R					
4	FC	= R	= R	GB	GB	RGB	GB	GB
5	MRV	= R	= R	=G				
6	FC	= R	= R	=G	B	RGB	GB	GB
7	MRV	= R	= R	=G	= B			
8	FC	= R	= R	=G	= B	RG	G	GB
9	MRV	= R	= R	=G	= B		= G	

Map coloring

Step		PQ	KG	CM	BL	ST	CT	AG
8	FC	= R	= R	=G	= B	RG	G	GB
9	MRV	= R	= R	=G	= B		= G	
10	FC	= R	= R	=G	= B	R	= G	B
11								
12								
13								

Map coloring

Step		PQ	KG	CM	BL	ST	CT	AG
8	FC	= R	= R	=G	= B	RG	G	GB
9	MRV	= R	= R	=G	= B		= G	
10	FC	= R	= R	=G	= B	R	= G	B
11	MRV	= R	= R	=G	= B	=R	= G	
12								
13								

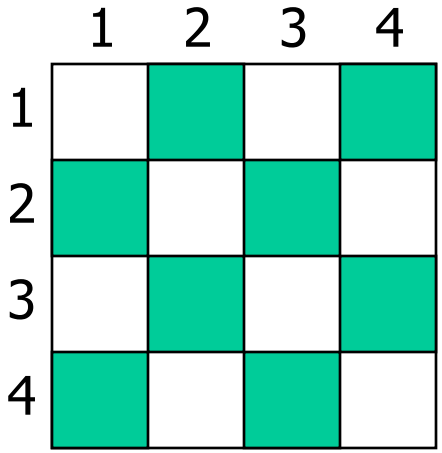
Map coloring

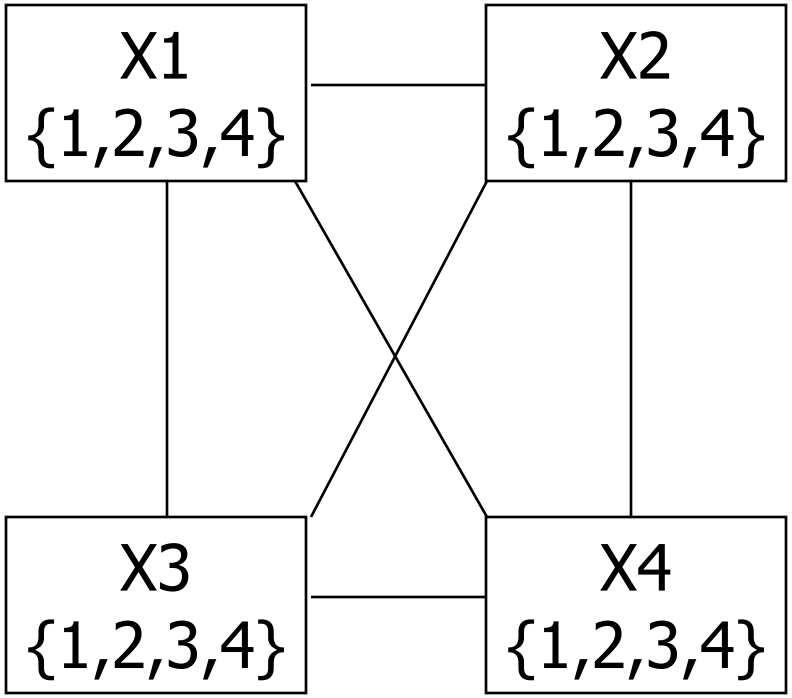
Step		PQ	KG	CM	BL	ST	CT	AG
8	FC	= R	= R	=G	= B	RG	G	GB
9	MRV	= R	= R	=G	= B		= G	
10	FC	= R	= R	=G	= B	R	= G	B
11	MRV	= R	= R	=G	= B	=R	= G	
12	FC	= R	= R	=G	= B	=R	= G	B
13								

Map coloring

Step		PQ	KG	CM	BL	ST	CT	AG
8	FC	= R	= R	=G	= B	RG	G	GB
9	MRV	= R	= R	=G	= B		= G	
10	FC	= R	= R	=G	= B	R	= G	B
11	MRV	= R	= R	=G	= B	=R	= G	
12	FC	= R	= R	=G	= B	=R	= G	B
13	MRV	= R	= R	=G	= B	=R	= G	=B

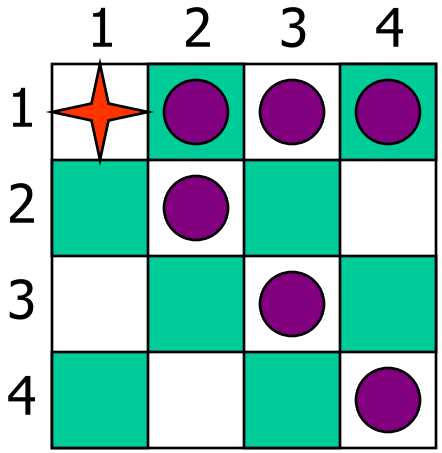
Example: 4-Queens Problem

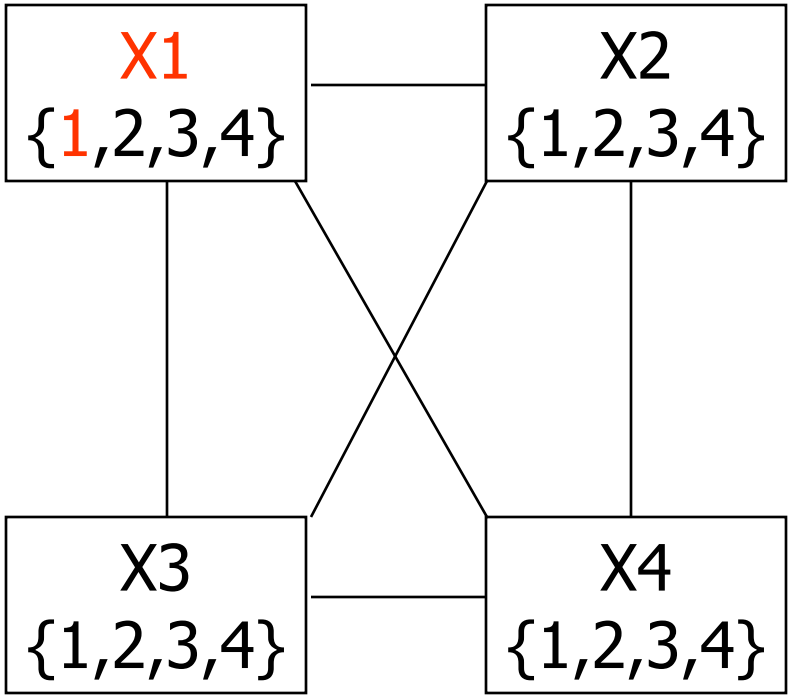




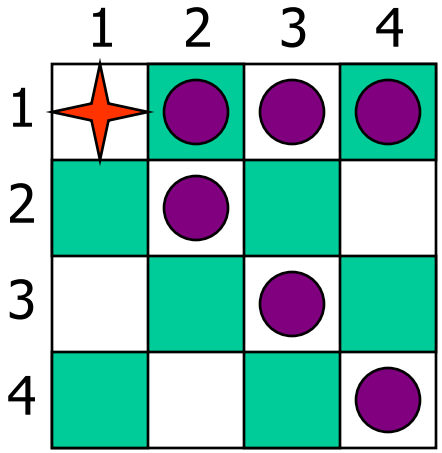
[4-Queens slides copied from B.J. Dorr CMSC 421 course on AI]

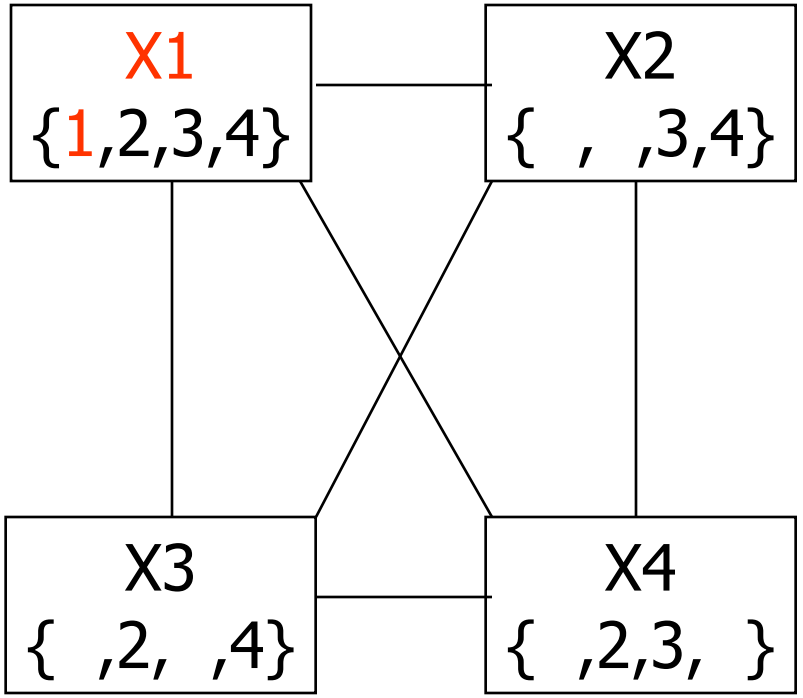
Example: 4-Queens Problem





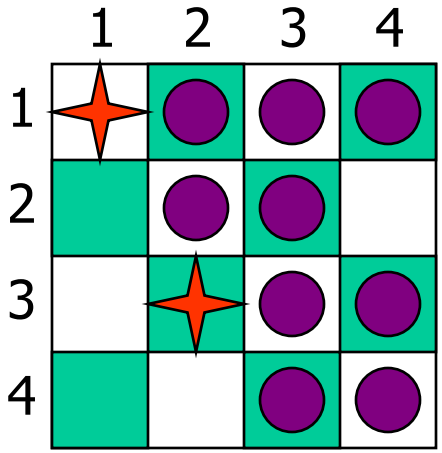
Example: 4-Queens Problem

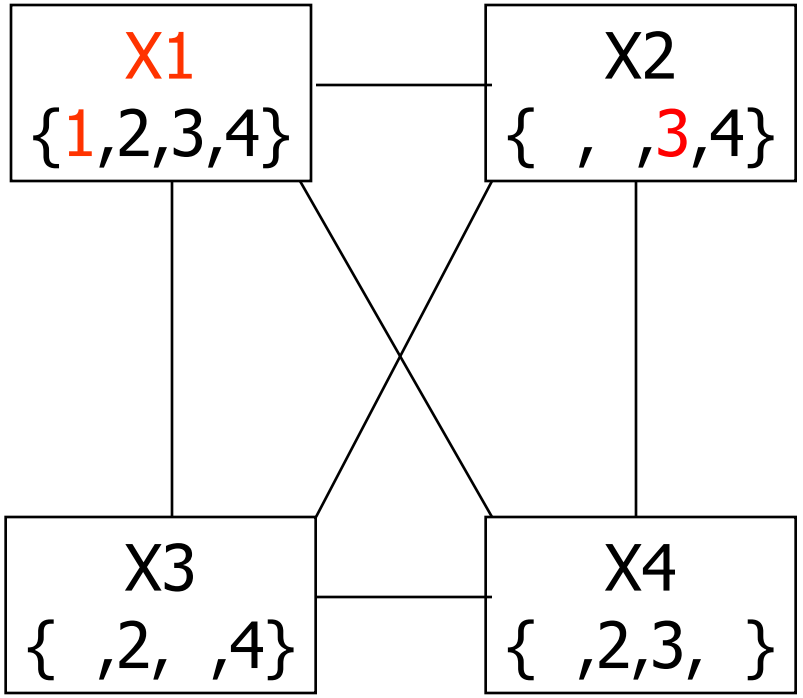




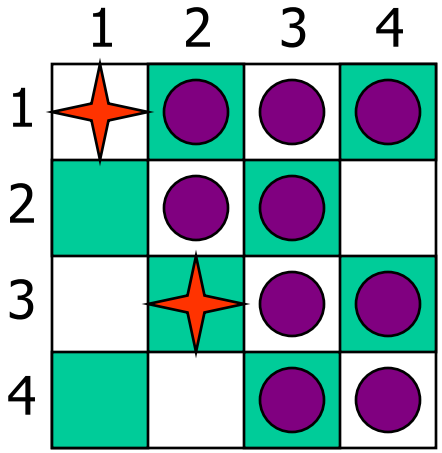
FW Checking → Remove

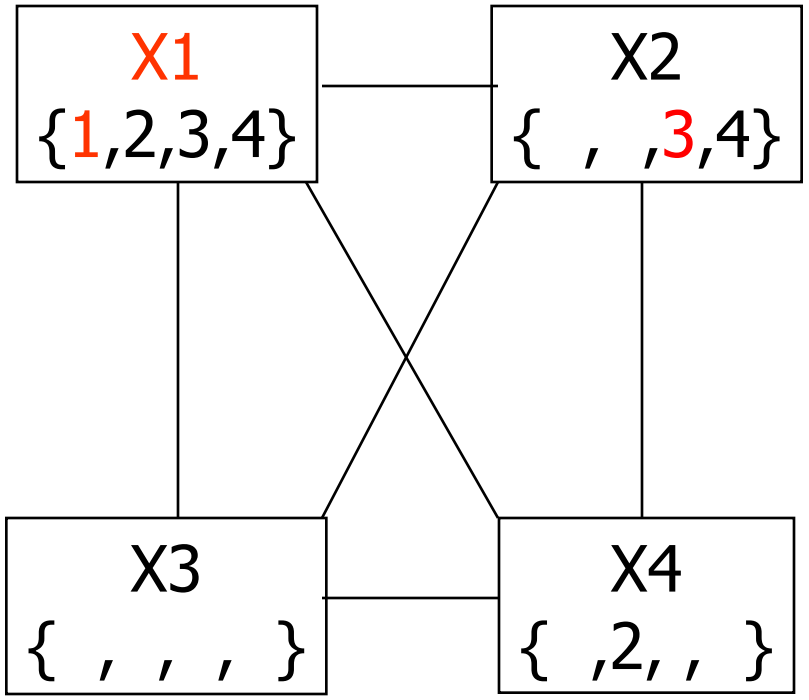
Example: 4-Queens Problem





Example: 4-Queens Problem

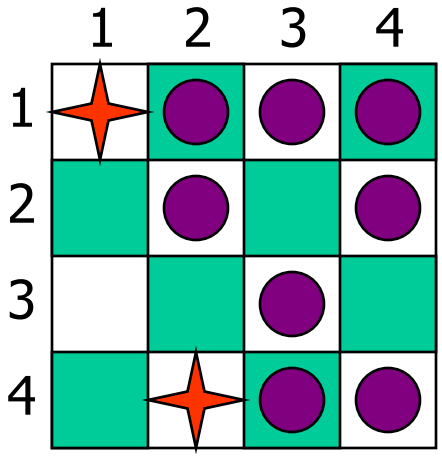


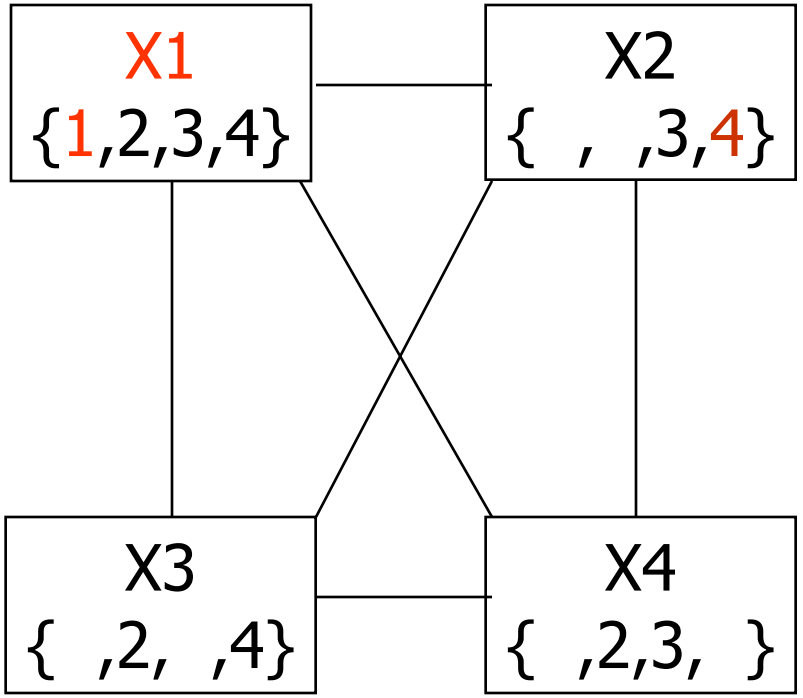


FW Checking → Remove

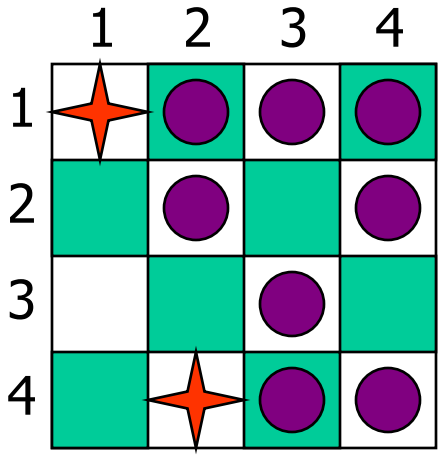
Backtrack

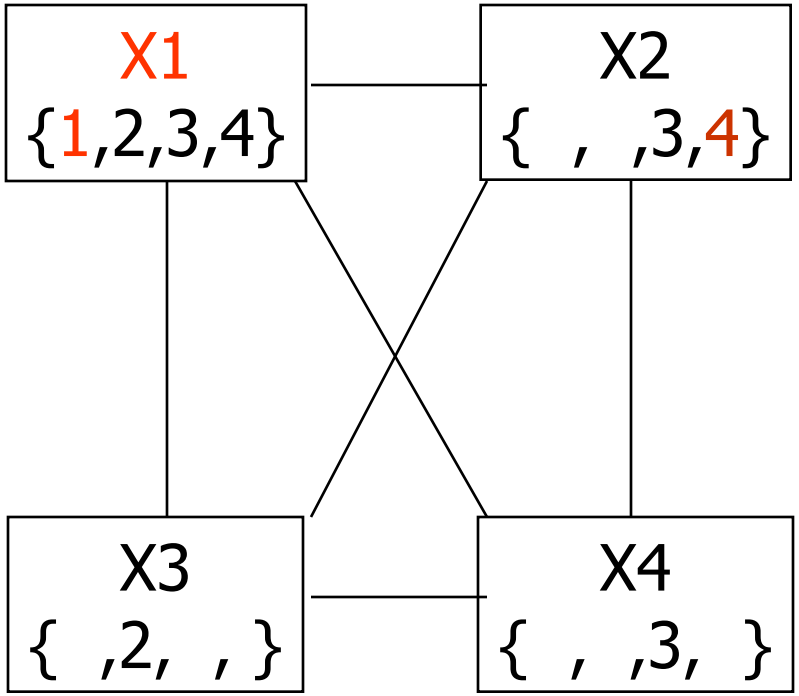
Example: 4-Queens Problem





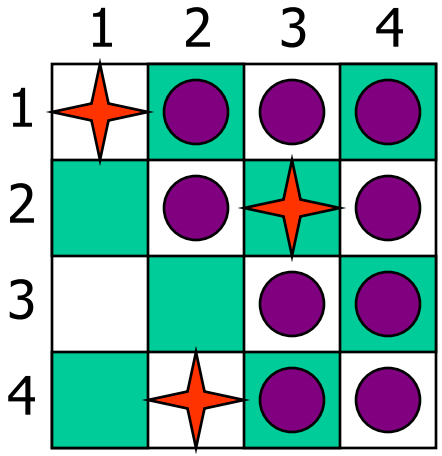
Example: 4-Queens Problem

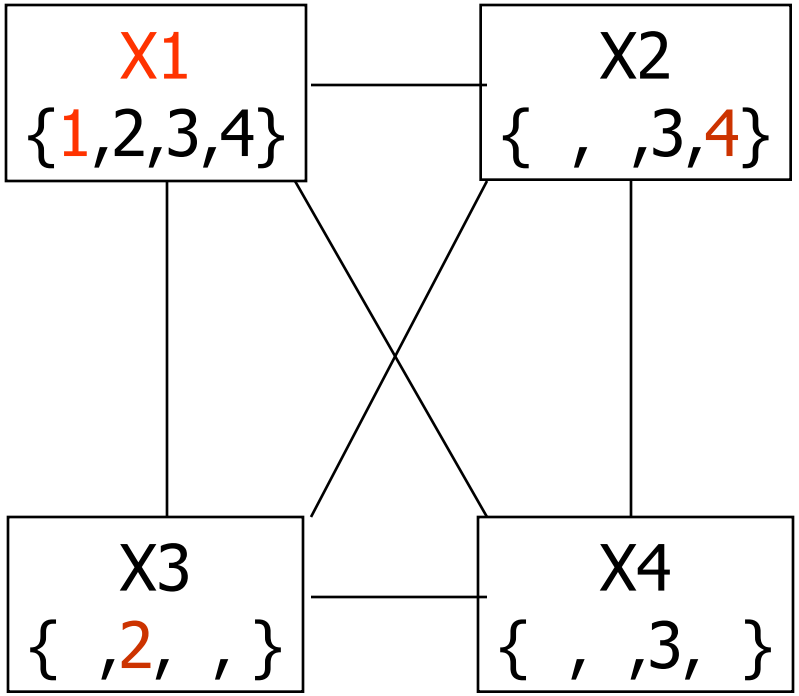




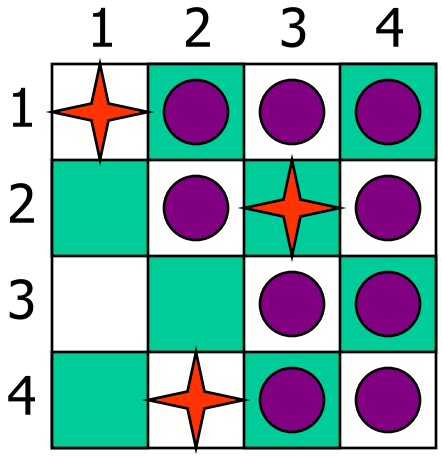
FW Checking → Remove

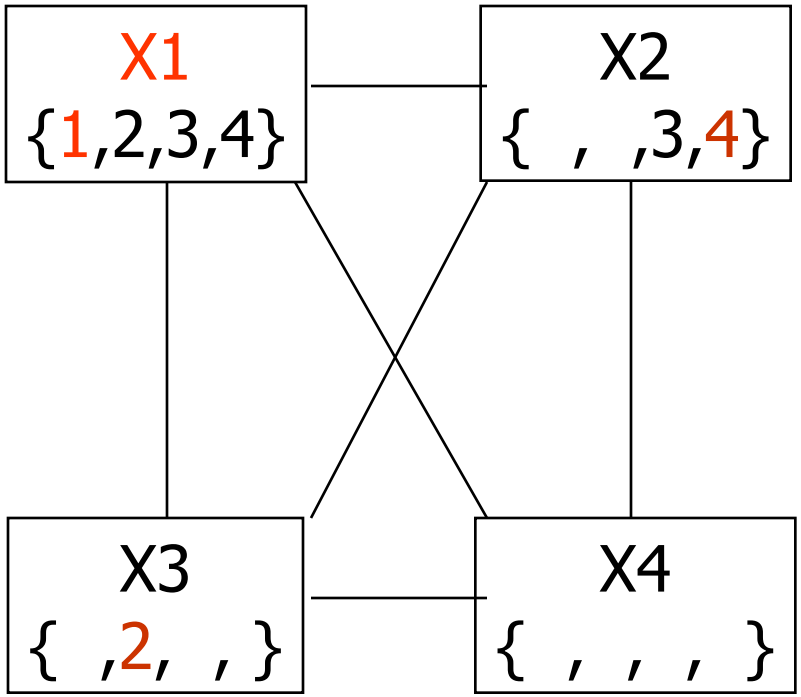
Example: 4-Queens Problem





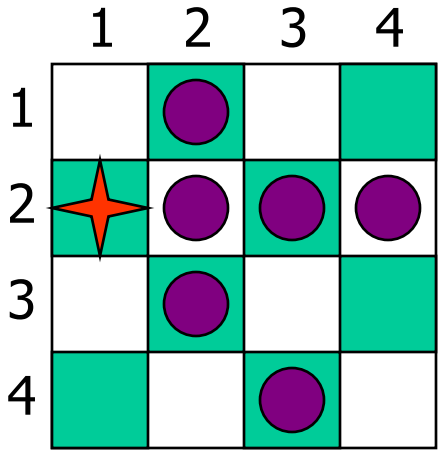
Example: 4-Queens Problem

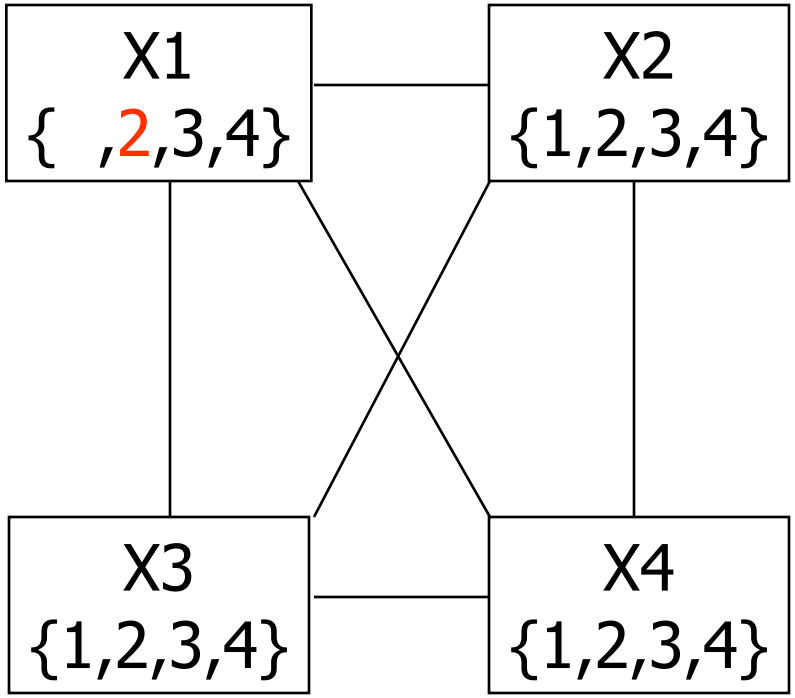




FW Checking → Remove

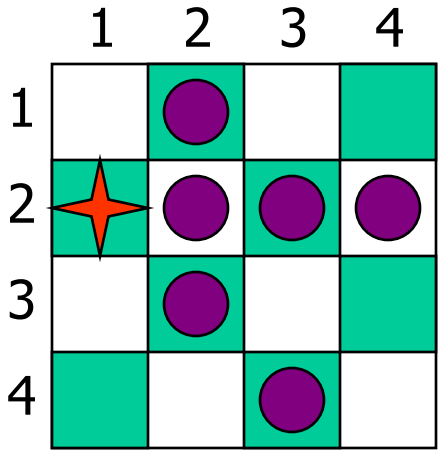
Example: 4-Queens Problem

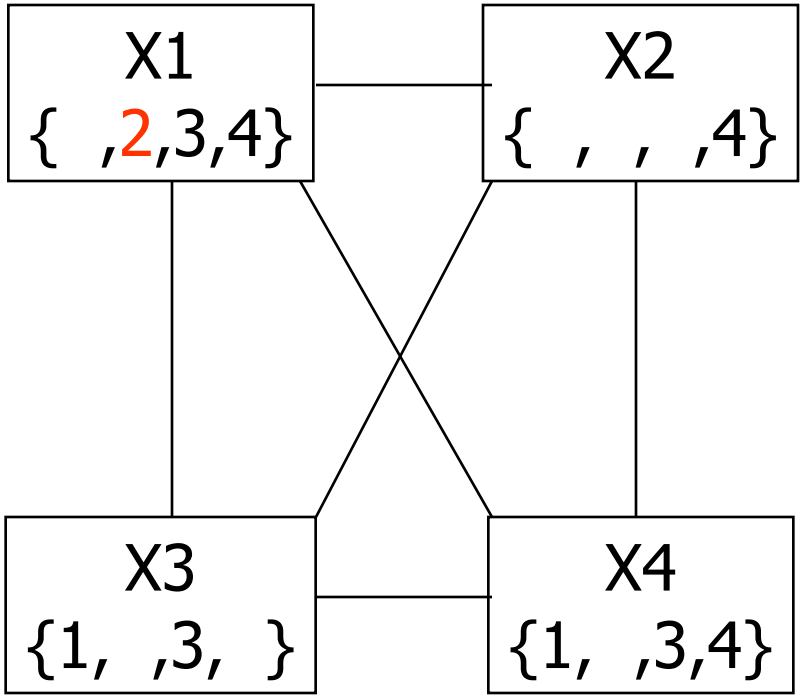




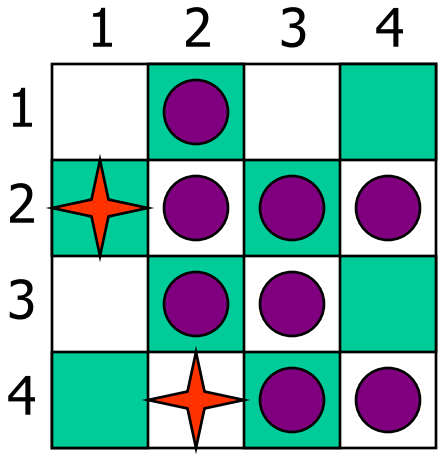
Do backtracking + Try another assignment for X1

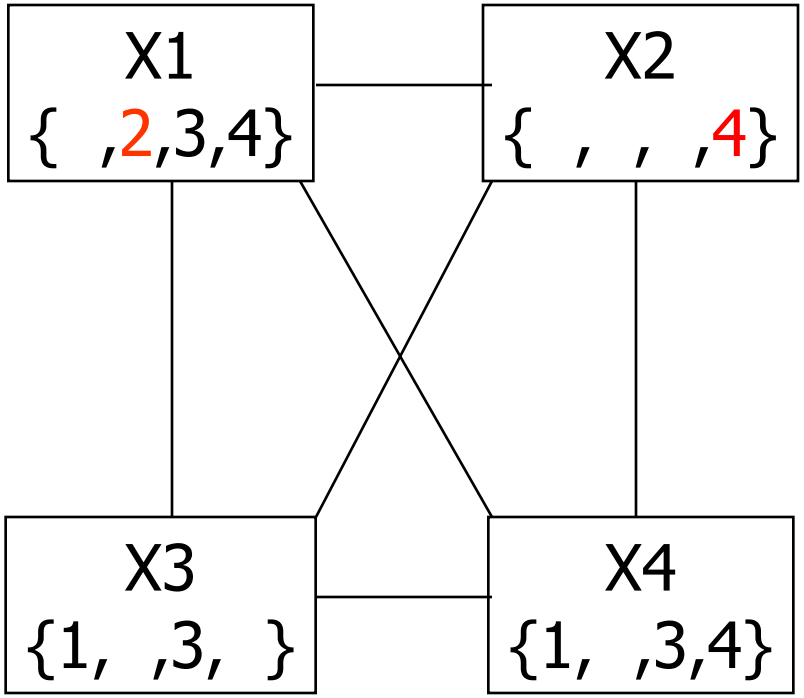
Example: 4-Queens Problem



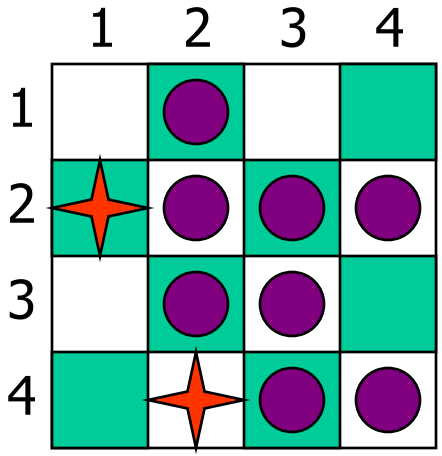


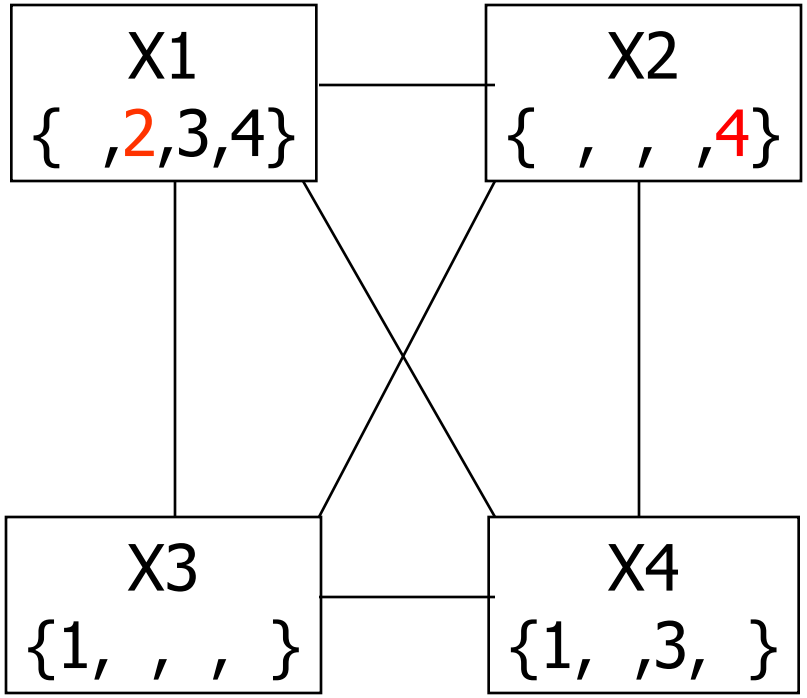
Example: 4-Queens Problem



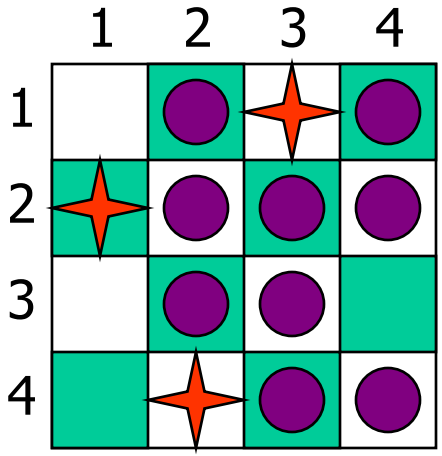


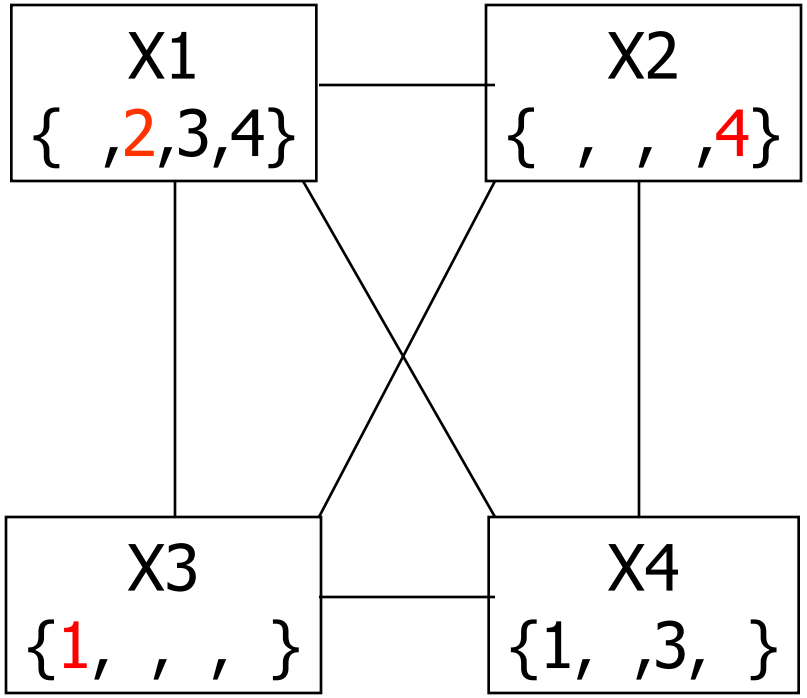
Example: 4-Queens Problem



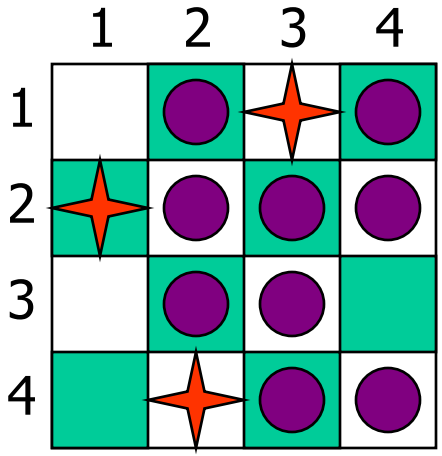


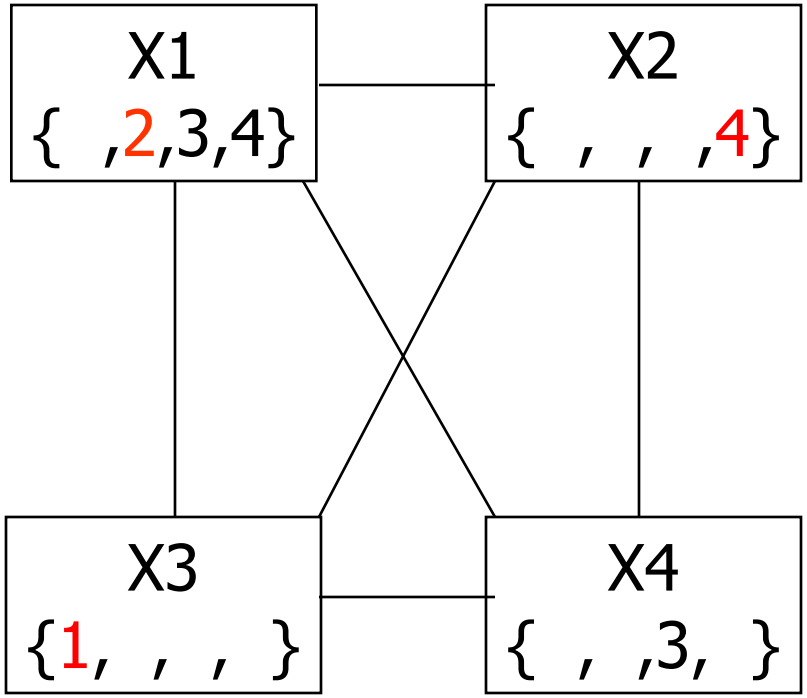
Example: 4-Queens Problem



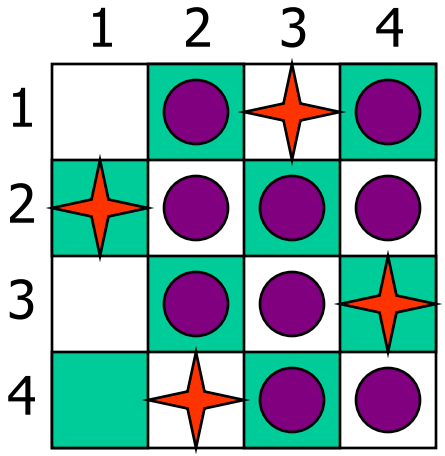


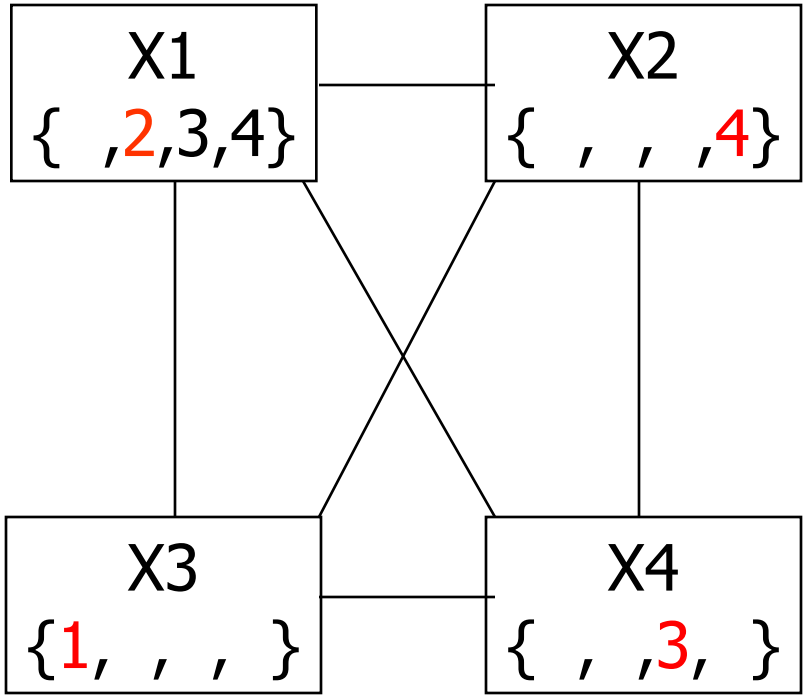
Example: 4-Queens Problem





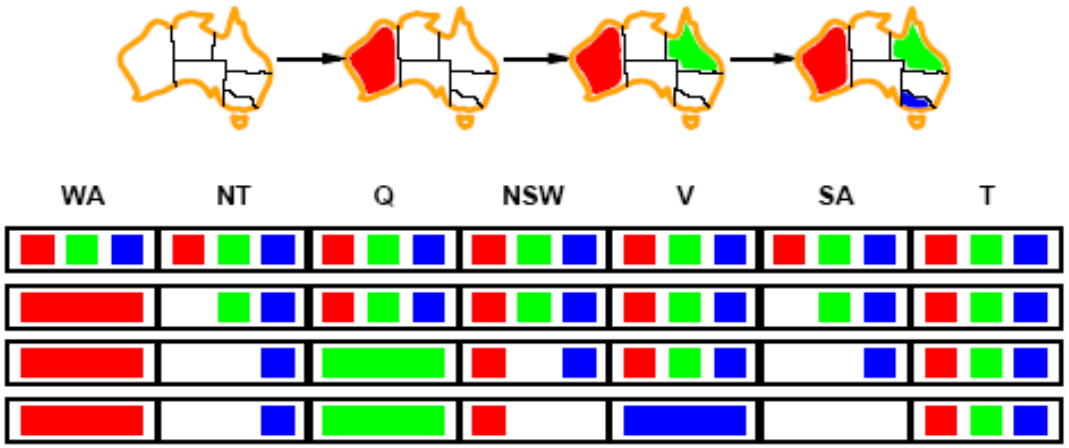
Example: 4-Queens Problem





Constraint propagation

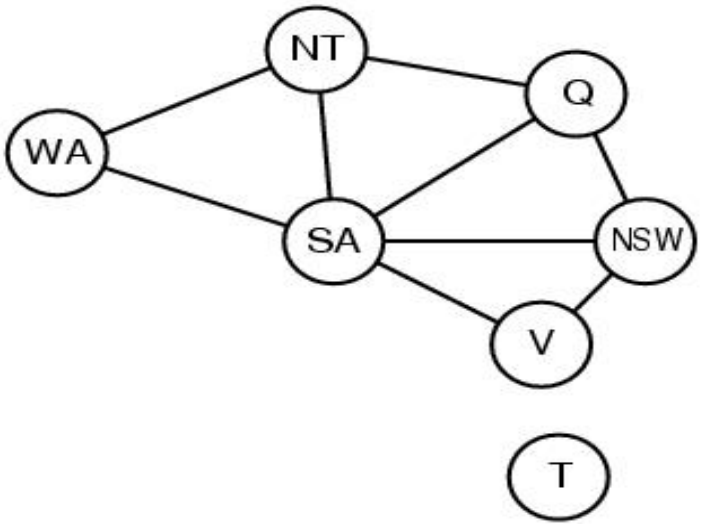
Constraint propagation



- Solving CSPs with combination of heuristics plus forward checking is more efficient than either approach alone.
- FC checking propagates information from assigned to unassigned variables but does not provide detection for all failures.
  - NT and SA cannot be blue!
- Constraint propagation repeatedly enforces constraints locally

Node consistency

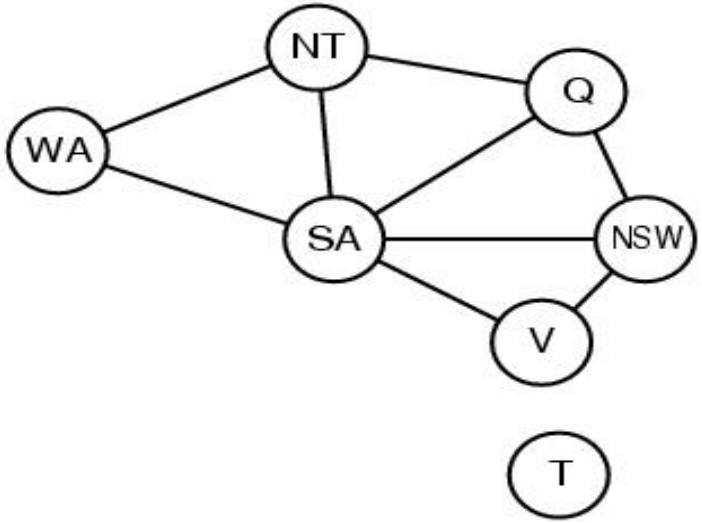
- A variable is node-consistent iif
  - All values in its domain satisfy its unary constraints



- Assume that there a unary constraint on SA is $SA \neq red$
- But, SA's domain: [red, green, blue]
  - $\Rightarrow$ SA is node-inconsistent
  - $\Rightarrow$ Remove red from its domain to make it node-consistent
  - $\Rightarrow$ After removing: its domain is [green, blue]

Arc consistency

- $X \rightarrow Y$ is arc-consistent iff
  - for *every* value *x* of *X*
    there is some allowed *y* of Y
- Example
  - Assume that,
    - ✓ SA's domain: [red, blue]
    - ✓ NT's domain: [red]
  - Consider an arc: SA→NT
    - ✓ red (SA)→ red (NT): inconsistent
    - ✓ blue (SA) → red (NT): consistent
    - ⇒ SA→NT is arc-inconsistent
    - ⇒ remove red from SA to force SA→NT is arc-consistent



Arc consistency

- Question:
  - arc-consistency enforcement: how can we ensure arc-consistent for a set initialized with n arcs (binary constraints)?
  - arc-consistency integration: how can we integrate arc consistency to backtracking?

Arc consistency

- Question:
  - arc-consistency enforcement: how can we ensure arc-consistent for a set initialized with n arcs (binary constraints)?
    => ARC-3 algorithm
  - arc-consistency integration: how can we integrate arc consistency to backtracking?
    => Backtracking revisivion

Arc consistency algorithm

function AC-3(*csp*) **return** 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 the arcs in *csp*

  initialize queue with csp's arcs
  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] − {X_j} **do**
      add (X_k, X_i) to queue

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) **return** *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 constraints between X_i and X_j
    then delete x from DOMAIN[X_i]; *removed* ← *true*
  return *removed*

Arc consistency algorithm

function AC-3(*csp*) **return** 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 the arcs in *csp*

initialize queue with csp's arcs

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$[X_i] - \{X_j\}$ **do**

add (X_k, X_i) to queue

Initialization with all or some arcs in the constraint graph

function **REMOVE-INCONSISTENT-VALUES**(X_i, X_j) **return** *true* iff we remove a value

removed $\leftarrow$ *false*

for each x **in** DOMAIN$[X_i]$ **do**

if no value y in DOMAIN$[X_j]$ allows (x,y) to satisfy the constraints between X_i and X_j

then delete x from DOMAIN$[X_i]$; *removed* $\leftarrow$ *true*

return *removed*

Arc consistency algorithm

function AC-3(*csp*) **return** 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 the arcs in *csp*

initialize queue with csp's arcs

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] – {X_j} **do**

add (X_k, X_i) to queue

Initialization with all or some arcs in the constraint graph

Recheck neigbors of a node if its domain modifed

function **REMOVE-INCONSISTENT-VALUES**(X_i, X_j) **return** *true* iff we remove a value

removed ← *false*

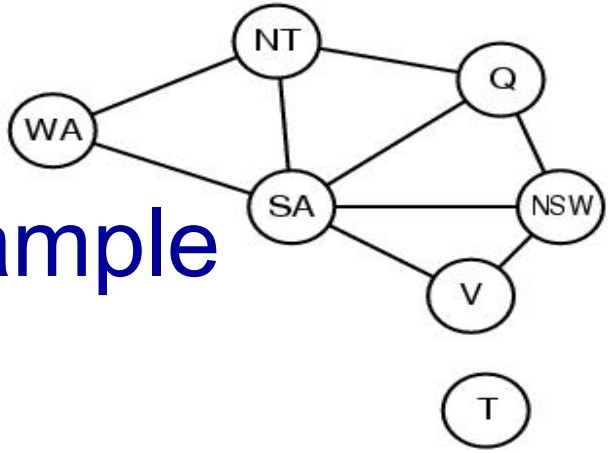
for each *x* **in** DOMAIN[X_i] **do**

if no value *y* in DOMAIN[X_j] allows (x,y) to satisfy the constraints between X_i and X_j

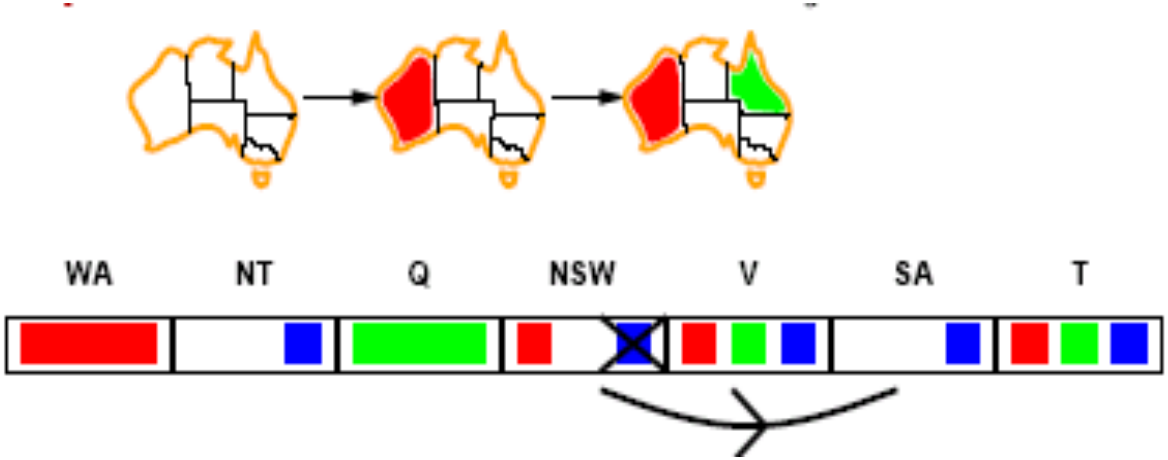
then delete x from DOMAIN[X_i]; *removed* ← *true*

return *removed*

Removing a value to enforce the consistency



Arc consistency, example



- NSW → SA is arc-inconsistent, because

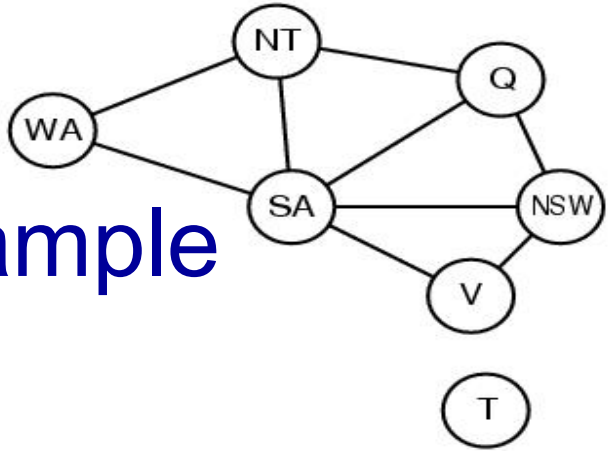
  blue (NSW) → blue (SA) : inconsistent

  => remove blue from NSW to enforce the consistency

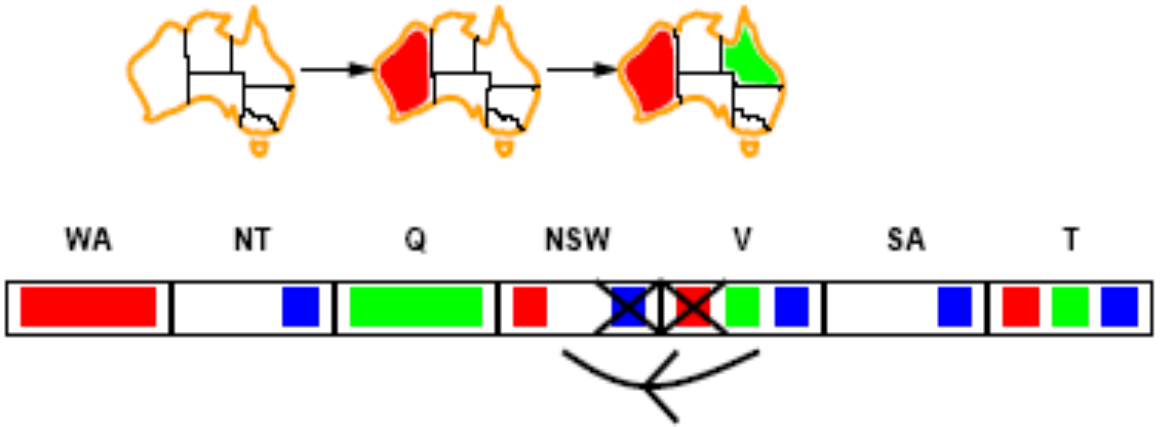
  and,

  => recheck NSW's neighbours, only V, because Q has been assigned and SA involved in the current arc NSW→ SA

  => enforce V→ NSW



Arc consistency, example



- V→NSW is arc-inconsistent, because

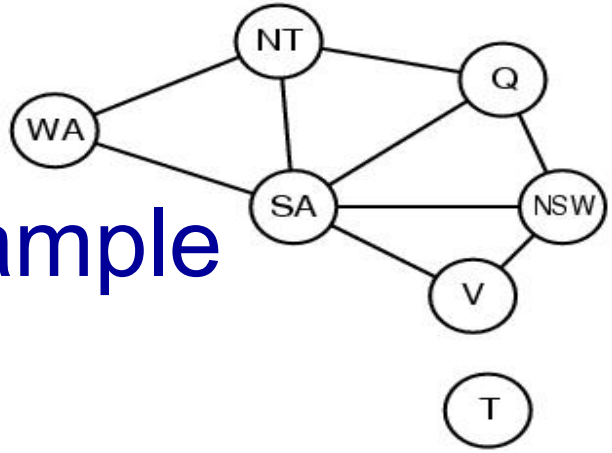
  red (V) → red (NSW) : inconsistent

  => remove red from V to enforce the consistency

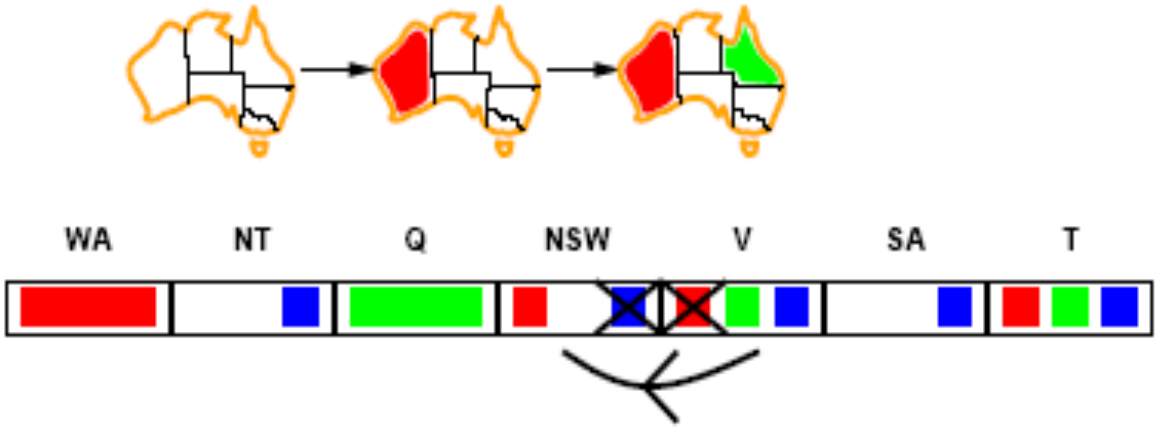
  and,

  => recheck NSW's neighbours, only SA, because NSW is involved in the current arc V→NSW

  => enforce SA→ NSW



Arc consistency, example



- SA→V is arc-consistent, because all values in SA's domain are consistent
  blue (SA) → green (V) : consistent
  => SA's domain is not changed
  => no rechecking necessary!

Arc consistency algorithm's complexity

- Any arc: X→Y
  - Need $O(d^2)$ for generating pairs of values and checking the consistency.
  - If we remove a value from X's domain to enforce the consistency then we need to recheck it neighbors
  - X has d values in its domain
    - $\Rightarrow$ X_k → X: added with d times maximun for a variable X_k
    - $\Rightarrow$ need $O(d^3)$ to enforce an arc in the constraint graph
- A constraint graph has c arcs
  - Complexity of Arc consistency algorithm: $O(cd^3) \propto O(n^2d^3)$
    - ✓Because $c = C_2^n \propto n^2$

Arc consistency

- Question:
  - arc-consistency enforcement: how can we ensure arc-consistent for a set initialized with n arcs (binary constraints)?
    => ARC-3 algorithm
  - arc-consistency integration: how can we integrate arc consistency to backtracking?
    => Backtracking revisivion

Backtracking Revision

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

function BACKTRACKING(assignment, csp) return a solution or 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 according to CONSTRAINTS[csp] 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
```

Peform arc consistency enforcement here for preprocessing

arc consistency enforcement

K-consistency

- Arc consistency does not detect all inconsistencies:
  - Partial assignment *{WA=red, NSW=red}* is inconsistent.
- Stronger forms of propagation can be defined using the notion of 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.
  - E.g. 1-consistency or node-consistency
  - E.g. 2-consistency or arc-consistency
  - E.g. 3-consistency or path-consistency

K-consistency

- A CSP with k-consistent
  - Time complexity for searching a solution: $O(n^2 d^3)$
    - ✓ See Arc-consistency algorithm
- A graph is strongly k-consistent if
  - It is k-consistent and
  - Is also (k-1) consistent, (k-2) consistent, … all the way down to 1-consistent.
- A CSP with strongly k-consistent
  - Time complexity for searching a solution: $O(n^2 d)$
    - ✓ For each variable => need to check the consistency with previous variables only!
  - YET *no free lunch*: any algorithm for establishing n-consistency must take time exponential in n, in the worst case.

Further improvements

- Checking special constraints
  - Checking Alldif(…) constraint
    - ✓ *E.g. {WA=red, NSW=red}*
  - Checking Atmost(…) constraint
    - ✓ *Bounds propagation for larger value domains*
- Intelligent backtracking
  - Standard form is chronological backtracking i.e. try different value for preceding variable.
  - More intelligent, backtrack to conflict set.
    - ✓ Set of variables that caused the failure or set of previously assigned variables that are connected to X by constraints.
    - ✓ Backjumping moves back to most recent element of the conflict set.
    - ✓ Forward checking can be used to determine conflict set.

Local search for CSP

Local search for CSP

- Use complete-state representation
- For CSPs
  - allow states with unsatisfied constraints
  - operators **reassign** variable values
- Variable selection: randomly select any conflicted variable
- Value selection: *min-conflicts heuristic*
  - Select new value that results in a minimum number of conflicts with the other variables

Local search for CSP

function MIN-CONFLICTS(*csp, max_steps*) **return** 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*

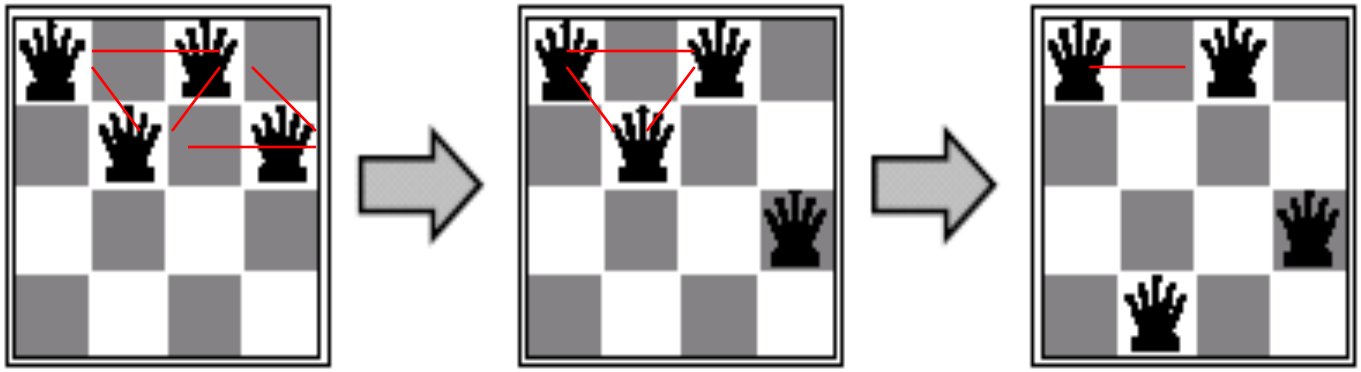
var ← a randomly chosen, conflicted variable from VARIABLES[*csp*]

value ← the value *v* for *var* that minimizes CONFLICTS(*var,v,current,csp*)

set *var* = *value* in *current*

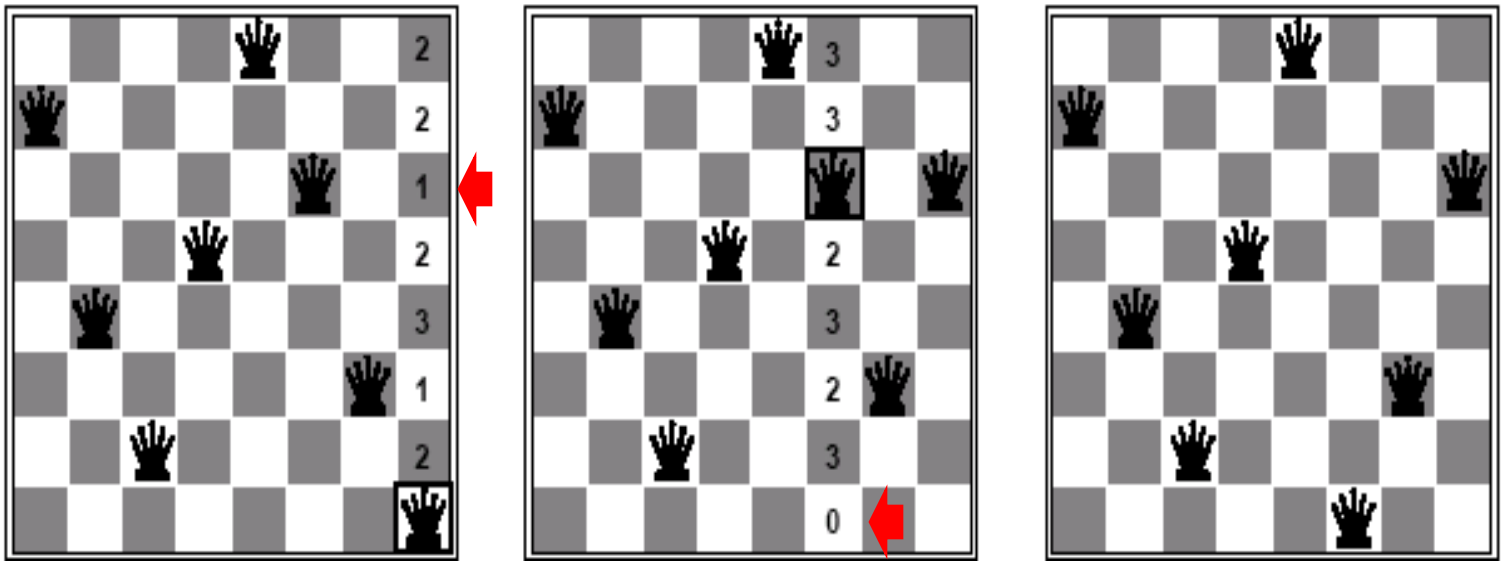
return *failure*

Min-conflicts example 1



❖ Use of min-conflicts heuristic in hill-climbing.

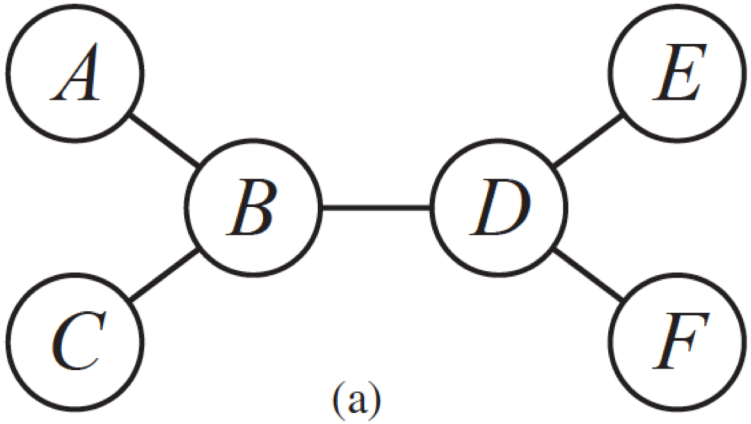
Min-conflicts example 2



- A two-step solution for an 8-queens problem using min-conflicts heuristic.
- At each stage a queen is chosen for reassignment in its column.
- The algorithm moves the queen to the min-conflict square breaking ties randomly.

Problem structure

Constraint graph is a tree
what will happen?



(a)

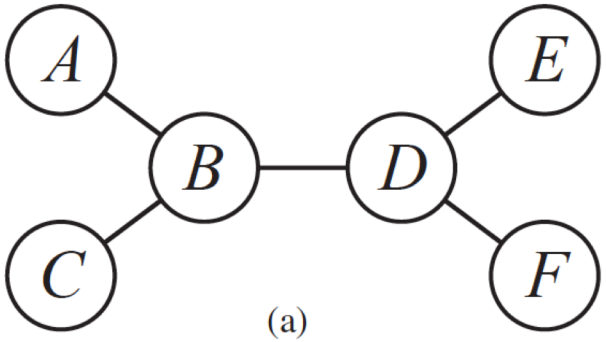
- The solution can be found with cost $O(nd^2)$ instead of $O(n^2d^3)$ as with the standard backtracking + k-consistency
- How?

Constraint graph is a tree
what will happen?

function TREE-CSP-SOLVER(csp) **returns** a solution, or failure
inputs: csp, a CSP with components X, D, C

$n \leftarrow$ number of variables in X
$assignment \leftarrow$ an empty assignment
$root \leftarrow$ any variable in X
$X \leftarrow$ TOPOLOGICALSORT($X, root$)
for $j = n$ **down to** 2 **do**
  MAKE-ARC-CONSISTENT(PARENT(X_j), X_j)
  if it cannot be made consistent **then return** $failure$
for $i = 1$ **to** n **do**
  $assignment[X_i] \leftarrow$ any consistent value from D_i
  if there is no consistent value **then return** $failure$
return $assignment$

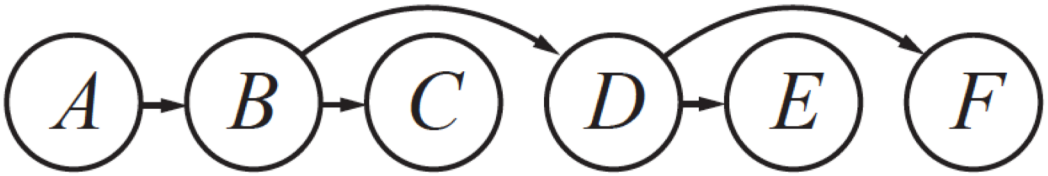
Constraint graph is a tree
what will happen?



(a)

#edges = n -1

$O(n + e) = O(n)$



(b)

Topological sort for the tree in (a)

(see Course "Data Structures and Algorithms")

Constraint graph is a tree
what will happen?

function TREE-CSP-SOLVER(csp) **returns** a solution, or failure
  inputs: csp, a CSP with components X, D, C

  $n \leftarrow$ number of variables in X
  $assignment \leftarrow$ an empty assignment
  $root \leftarrow$ any variable in X
  $X \leftarrow$ TOPOLOGICALSORT(X, $root$) ← $O(n)$
  for $j = n$ **down to** 2 **do**
    MAKE-ARC-CONSISTENT(PARENT(X_j), X_j) ← $O(d^2)$
    if it cannot be made consistent **then return** $failure$
  for $i = 1$ **to** n **do**
    $assignment[X_i] \leftarrow$ any consistent value from D_i
    if there is no consistent value **then return** $failure$
  return $assignment$

$O(nd^2)$ (inner loop)

$O(n)$ (outer loop)

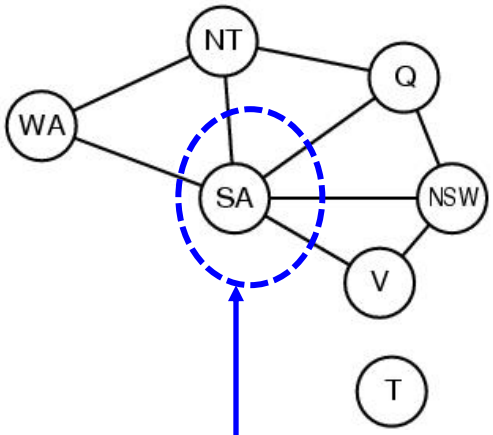
$O(nd^2)$ (total)

How can we solve a general CSP with tree constraint graph?

- Approach 1: remove some nodes (variables)
  - By fixing their values

- Approach 2: collapse some nodes
  - By graph decomposition

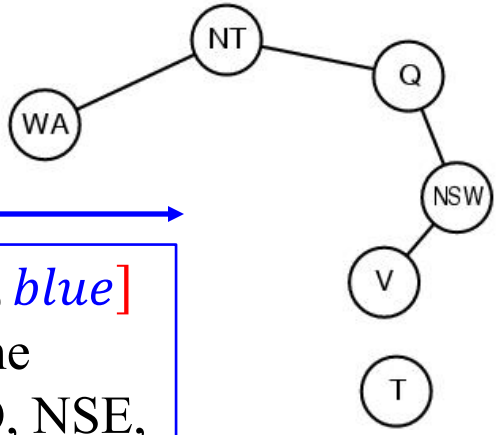
Approach 1: removing some nodes



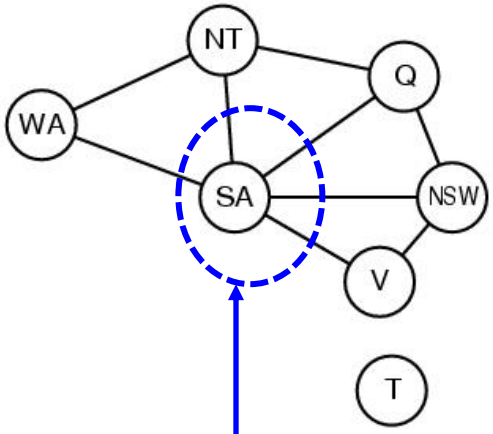
cycle cutset
{SA}

(1) Assign $SA = x \in [red, green, blue]$
(2) Remove x from the domain of the connected nodes, ie., WA, NT, Q, NSE, and V

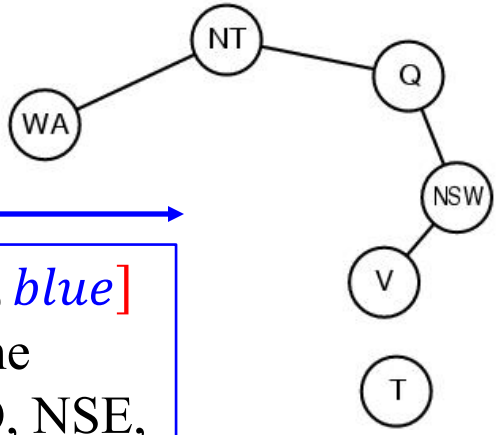
Try with different $x \in$ SA' domain



Approach 1: removing some nodes



(1) Assign $SA = x \in [red, green, blue]$
(2) Remove x from the domain of the connected nodes, ie., WA, NT, Q, NSE, and V



cycle cutset
{SA}

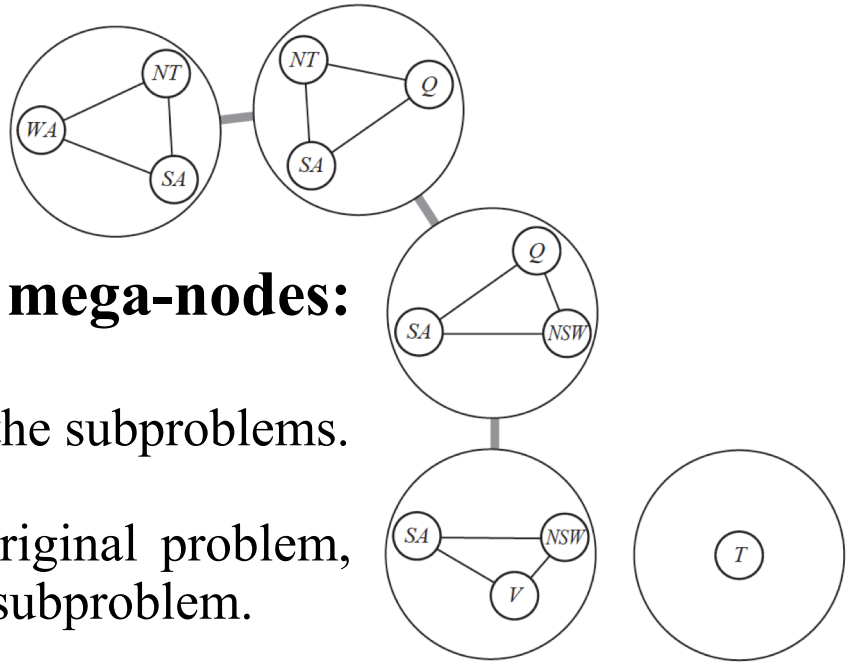
c: #vars in the cutset

Try with different $x \in$ SA' domain

Complexity: $O(d^c.(n-c)d^2)$

instead of $O(n^2d^3)$ in the standard backtracking:

Approach 2: collapsing some nodes



From a general graph to tree with mega-nodes:

Necessary requirements

1. Every variable appears in ar least one of the subproblems.

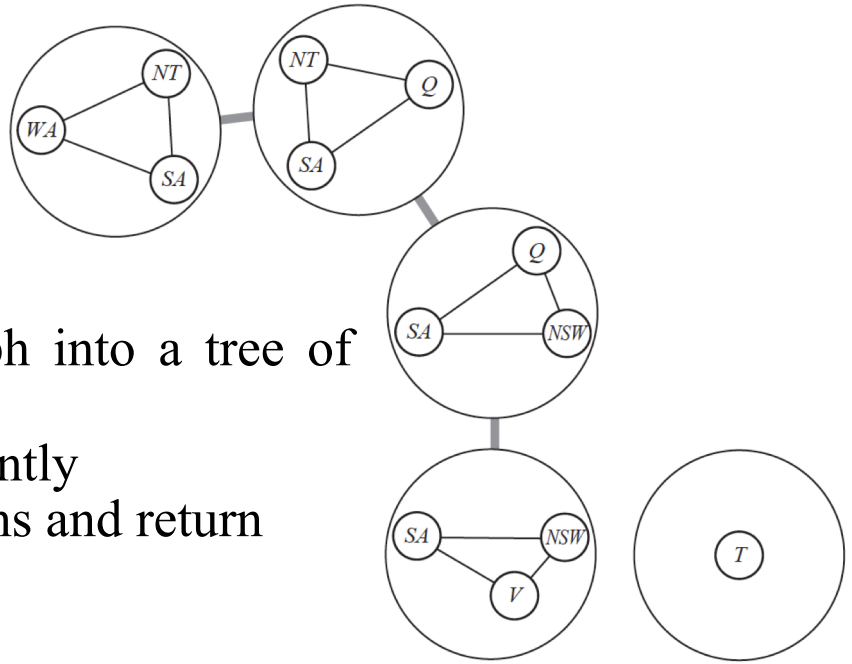
2. If two variables are connected in the original problem, they must appear together in at least one subproblem.

3. If a variable appears in two subproblems, it must appear in each node on the path.

↓

Link between subproblems (mega-nodes)

Approach 2: collapsing some nodes



Steps:

1. Decompose the original constraint graph into a tree of mega-nodes
2. Solve subprogram in mega-nodes indepently
3. Combine the results from the subproblems and return

Summary

- CSPs are a special kind of problem: states defined by values of a fixed set of variables, goal test defined by constraints on variable values
- Backtracking=depth-first search with one variable assigned per node
- Variable ordering and value selection heuristics help significantly
- Forward checking prevents assignments that lead to failure.
- Constraint propagation does additional work to constrain values and detect inconsistencies.
- The CSP representation allows analysis of problem structure.
- Tree structured CSPs can be solved in linear time.
- Iterative min-conflicts is usually effective in practice.