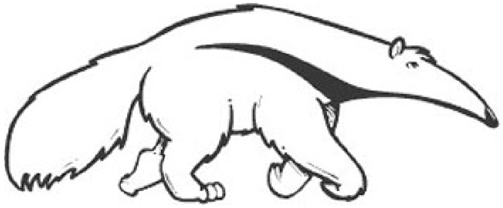
Constraint Satisfaction Problems B: Constraint Propagation, Structure

CS171, Summer Session I, 2018
Introduction to Artificial Intelligence
Prof. Richard Lathrop



Read Beforehand: R&N 6.1-6.4, except 6.3.3

BREN:ICS
INFORMATION AND COMPUTER SCIENCES

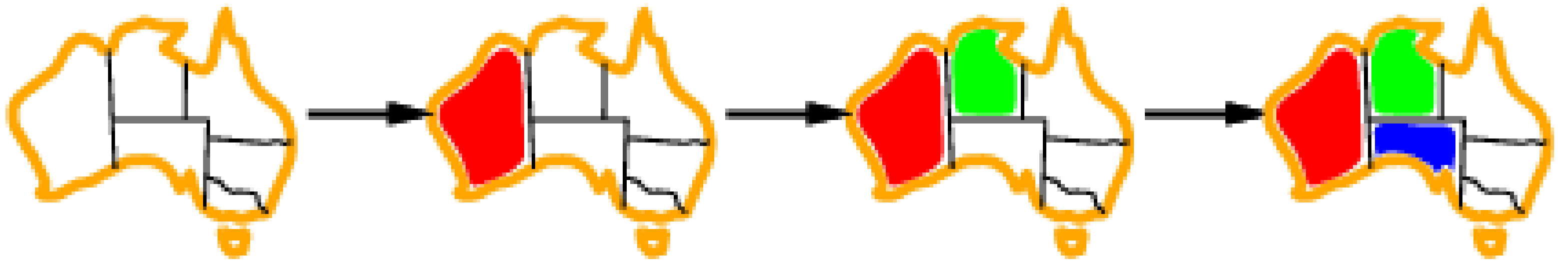
UNIVERSITY of CALIFORNIA IRVINE

You Should Know

- Node consistency, arc consistency, path consistency, K-consistency (6.2)
- Forward checking (6.3.2)
- Local search for CSPs
  - Min-Conflict Heuristic (6.4)
- The structure of problems (6.5)

Minimum remaining values (MRV)

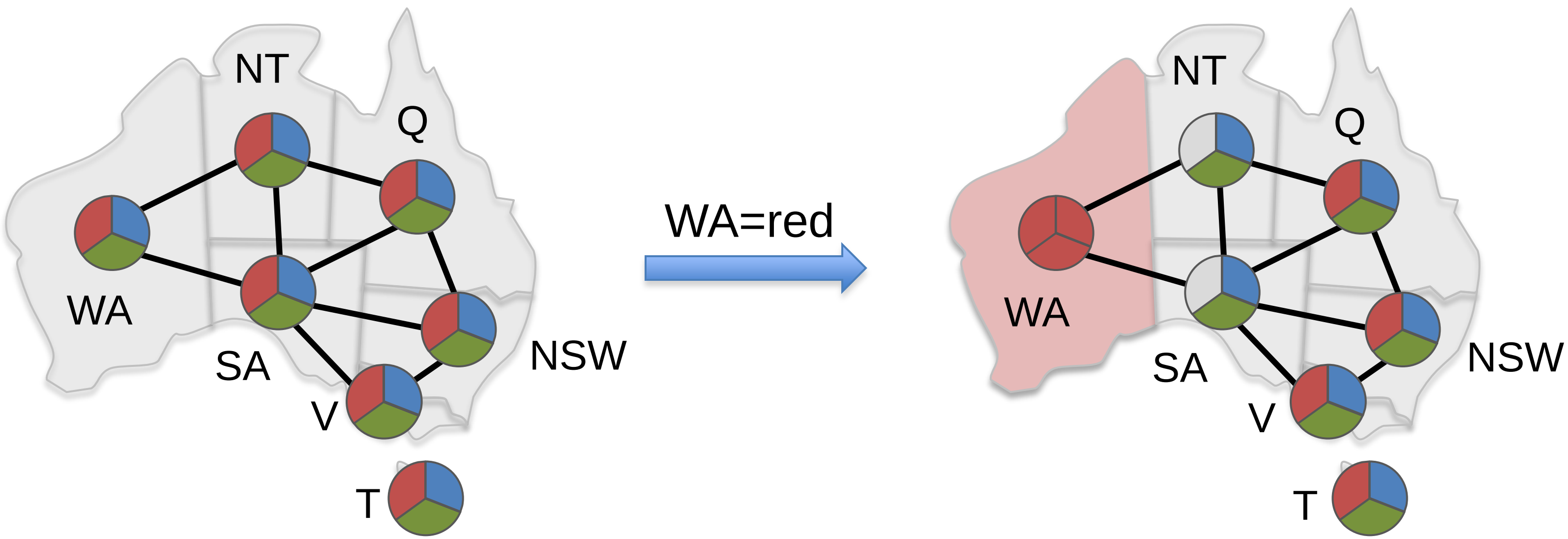
- A heuristic for selecting the next variable
  - also called most constrained variable (MCV) heuristic



  - choose the variable X with the fewest legal values
  - will immediately detect failure if X has no legal values
  - (Related to forward checking, later)

Idea: reduce the branching factor now
Smallest domain size = fewest # of children = least branching

Detailed MRV example



Initially, all regions have $|D_i|=3$
Choose one randomly, e.g.*, WA
& pick value, e.g., red

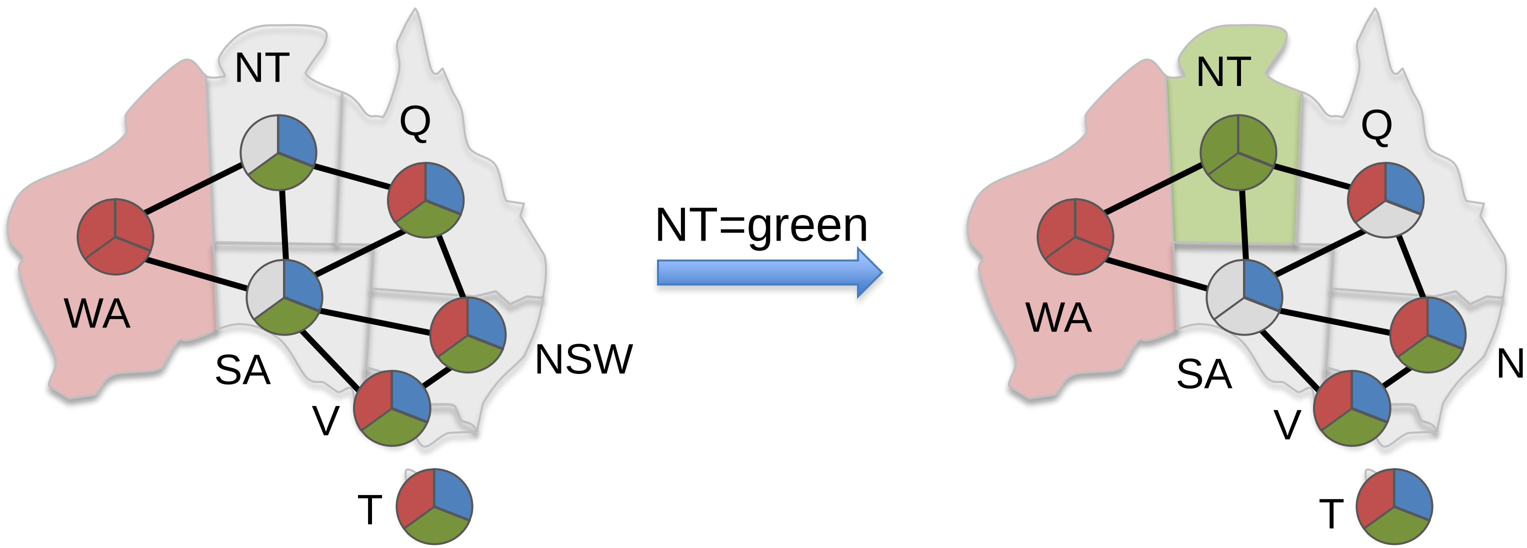
(Better: tie-break with degree…)

Do forward checking (next topic)
NT & SA cannot be red

Now NT & SA have 2 possible values
– pick one randomly

* e.g. = *exempli gratia* = for example

Detailed MRV example



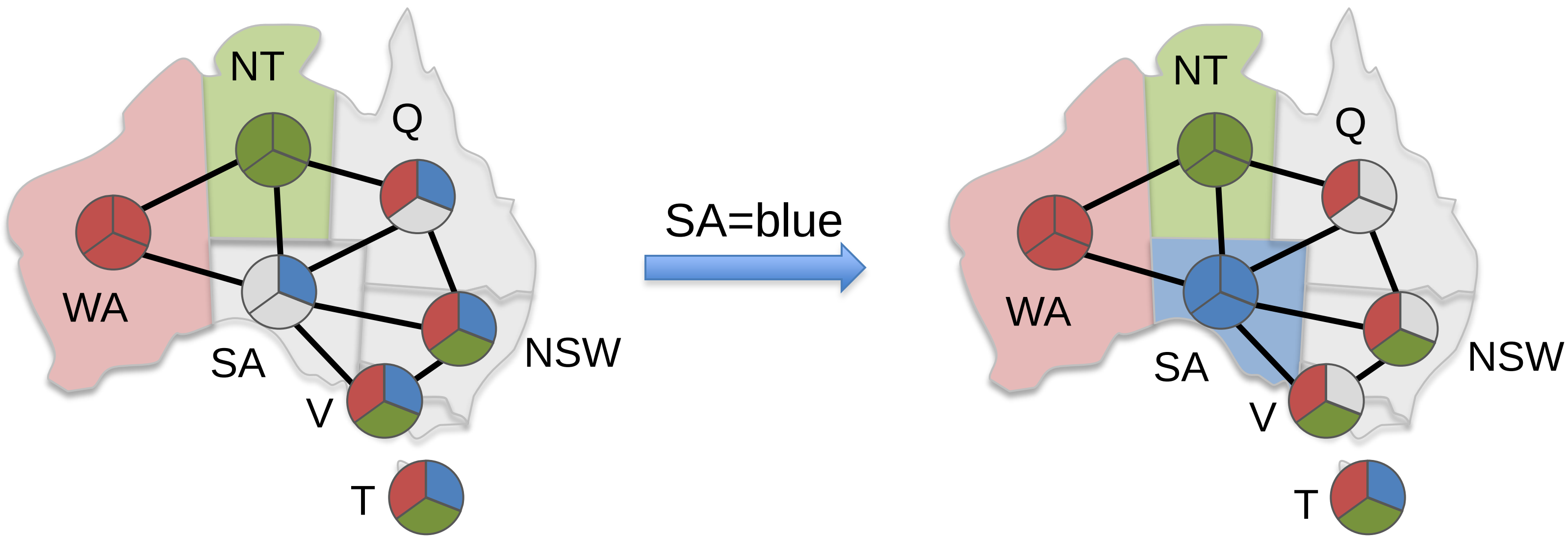
NT & SA have two possible values
Choose one randomly, e.g., NT,
& pick value, e.g., green

(Better: tie-break with degree;
select value by least constraining)

Do forward checking (next topic)
SA & Q cannot be green

Now SA has only 1 possible value;
Q has 2 values.

Detailed MRV example



SA has only one possible value
Assign it

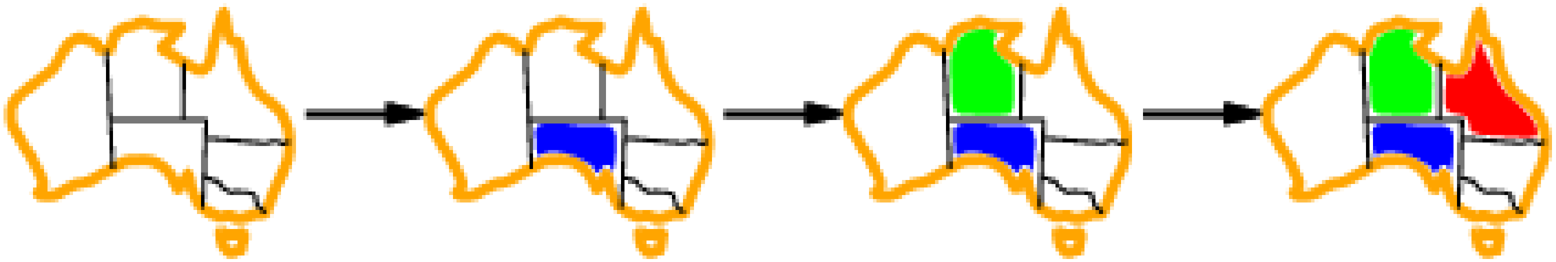
Do forward checking (next topic)
Now Q, NSW, V cannot be blue

Now Q has only 1 possible value;
NSW, V have 2 values.

We will assign Q its only value and
solve the remainder with no search

Degree heuristic

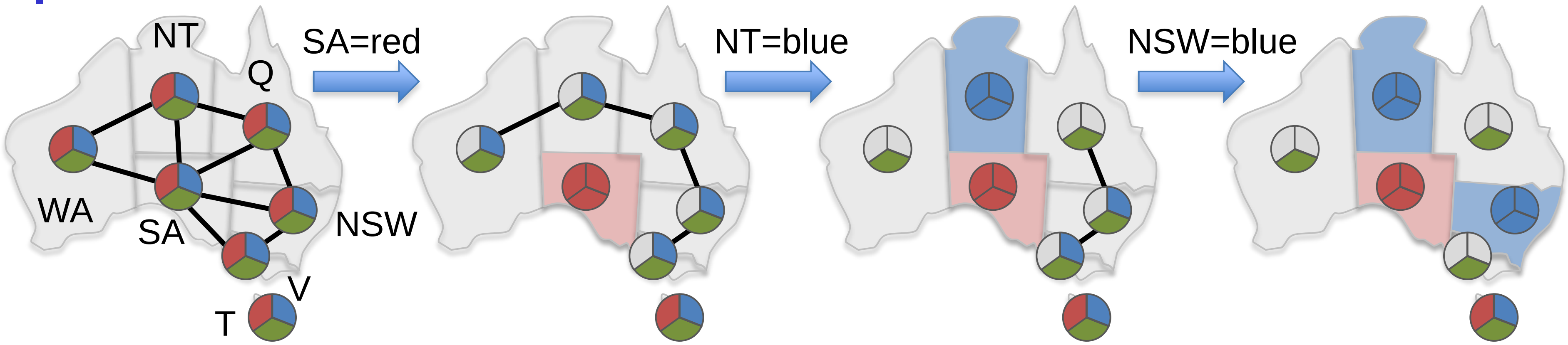
- Another heuristic for selecting the next variable
  - also called most constraining variable heuristic



  - Select variable involved in the most constraints on other **unassigned** variables
  - Useful as a tie-breaker among most constrained variables

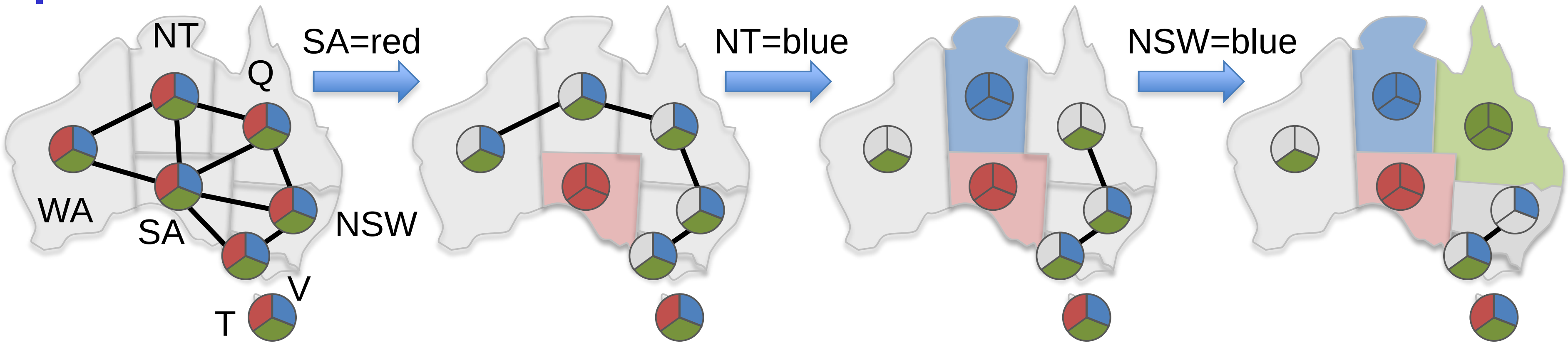
Note: usually (and in picture above) we use the degree heuristic as a tie-breaker for MRV; however, in homework & exams we may use it without MRV to show how it works. Let's see an example.

Ex: Degree heuristic (only)



- Select variable involved in largest # of constraints with other **unassigned** vars
- Initially: degree(SA) = 5 > degree(other vars); assign it a value, e.g., red
  - No neighbor can be red; **we remove the edges to assist in counting degree**
- Now, degree(NT) = degree(Q) = degree(NSW) = 2; degree(WA) = degree(V) = 1
  - Select one at random, e.g., NT; assign it a value, e.g., blue
- Now, degree(NSW)=2; degree(Q) = degree(V) = 1; degree(WA) = 0
  - Select NSW; assign it a value, e.g., blue; solve remaining problem with no search
- Idea: reduce branching in the future
  - The variable with the largest # of constraints will likely knock out the most values from other variables, reducing the branching factor in the future

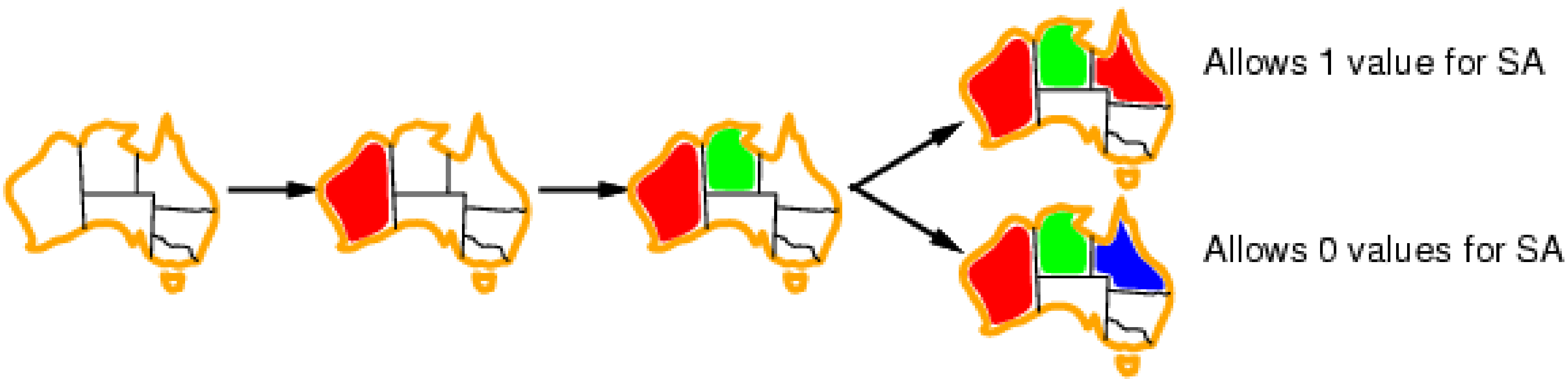
Ex: MRV + degree



- Initially, all variables have 3 values; tie-breaker degree => SA, e.g., assign red
  - No neighbor can be red; **we remove the edges to assist in counting degree**
- Now, WA, NT, Q, NSW, V have 2 values each
  - WA,V have degree 1; NT,Q,NSW all have degree 2
  - Select one at random, e.g. NT; assign it a value, e.g., blue
- Now, WA and Q have only one possible value; degree(Q)=1 > degree(WA)=0
  - We will solve the remaining problem with no search
- Idea: reduce branching in the future
  - The variable with the largest # of constraints will likely knock out the most values from other variables, reducing the branching factor in the future

Least Constraining Value

- Heuristic for selecting what value to try next
- Given a variable, choose the least constraining value:
  - the one that rules out the fewest values in the remaining variables



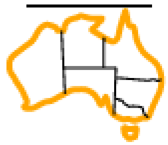
  - Makes it more likely to find a solution early

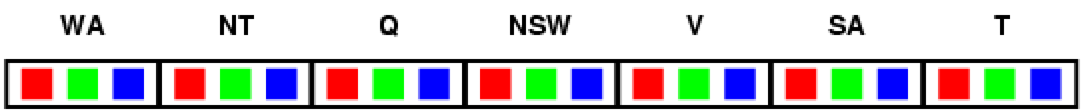
Look-ahead: Constraint propagation

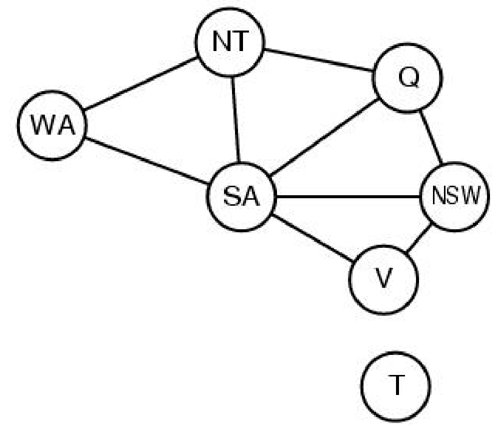
- Intuition:
  - Some domains have values that are inconsistent with the values in some other domains
  - Propagate constraints to remove inconsistent values
  - Thereby reduce future branching factors
- Forward checking
  - Check each unassigned neighbor in constraint graph
- Arc consistency (AC-3 in R&N)
  - Full arc-consistency everywhere until quiescence
  - Can run as a preprocessor
    - Remove obvious inconsistencies
  - Can run after each step of backtracking search
    - Maintaining Arc Consistency (MAC)

Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Backtrack when any variable has no legal values
  - ONLY check neighbors of most recently assigned variable

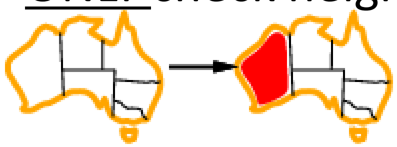


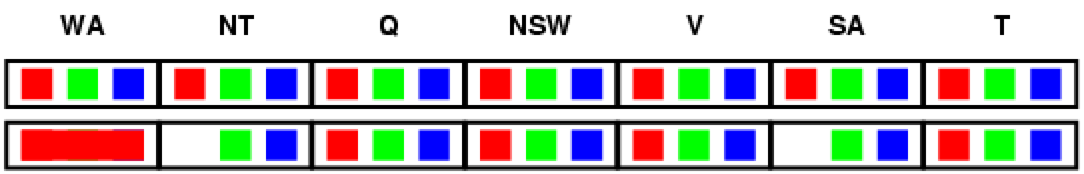


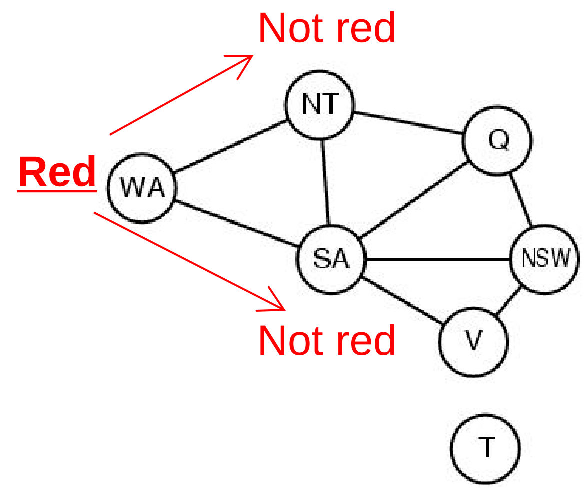


Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Backtrack when any variable has no legal values
  - ONLY check neighbors of most recently assigned variable





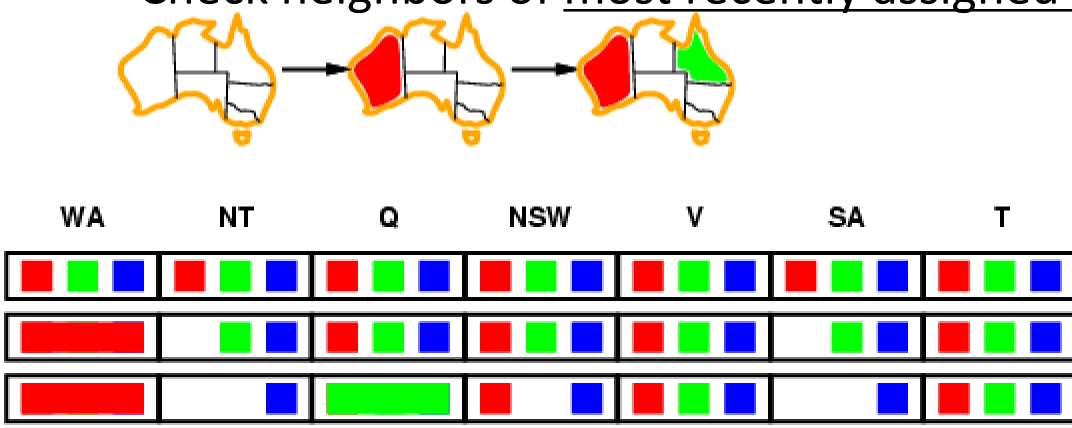


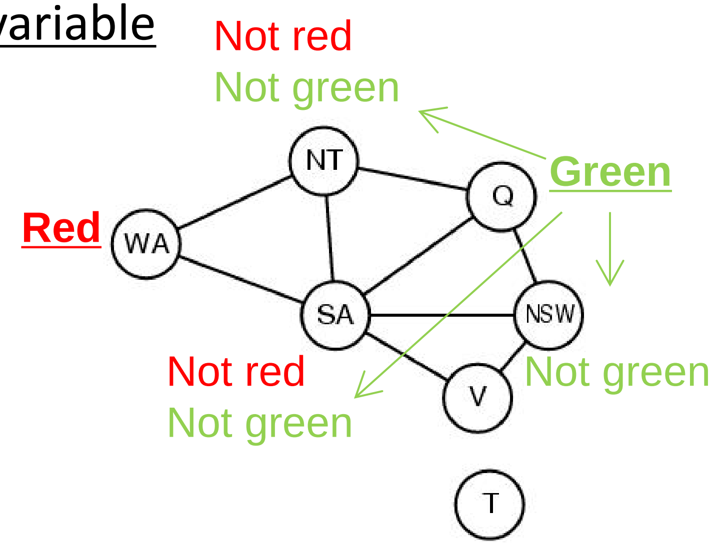
Assign {WA = red}
Effect on other variables (neighbors of WA):

- NT can no longer be red
- SA can no longer be red

Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Backtrack when any variable has no legal values
  - Check neighbors of <u>most recently assigned variable</u>





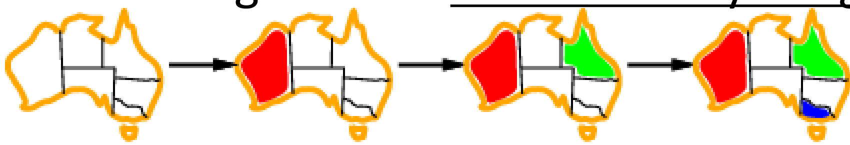
Assign {Q = green}
Effect on other variables (neighbors of Q):

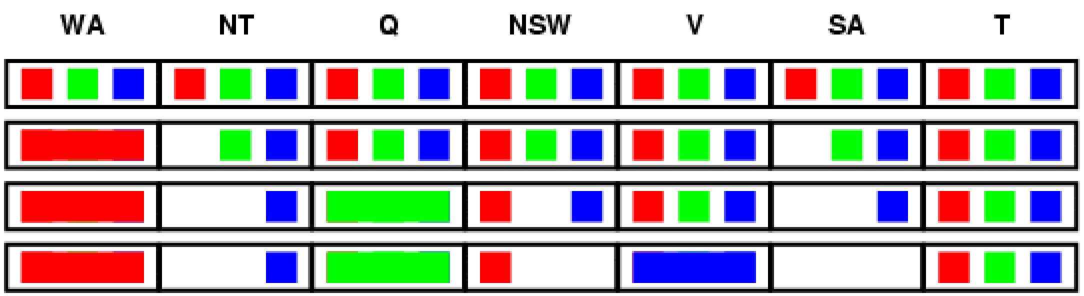
- NT can no longer be green
- SA can no longer be green
- NSW can no longer be green

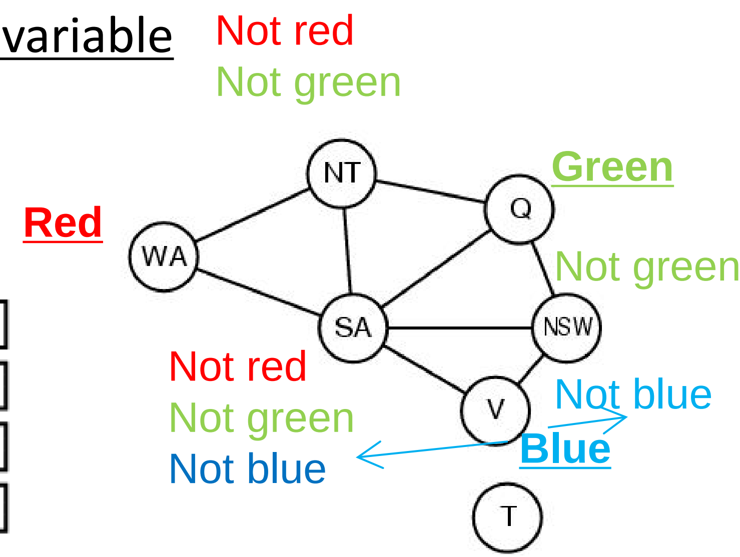
<u>(We already have failure, but FC is too simple to detect it now)</u>

Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Backtrack when any variable has no legal values
  - Check neighbors of <u>most recently assigned variable</u>







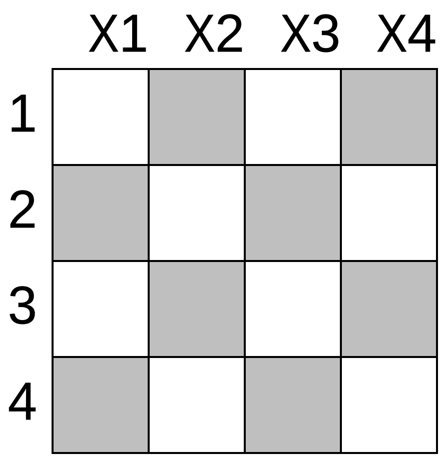
Assign {V = blue}
Effect on other variables (neighbors of V):
- NSW can no longer be blue
- SA can no longer be blue **(no values possible!)**

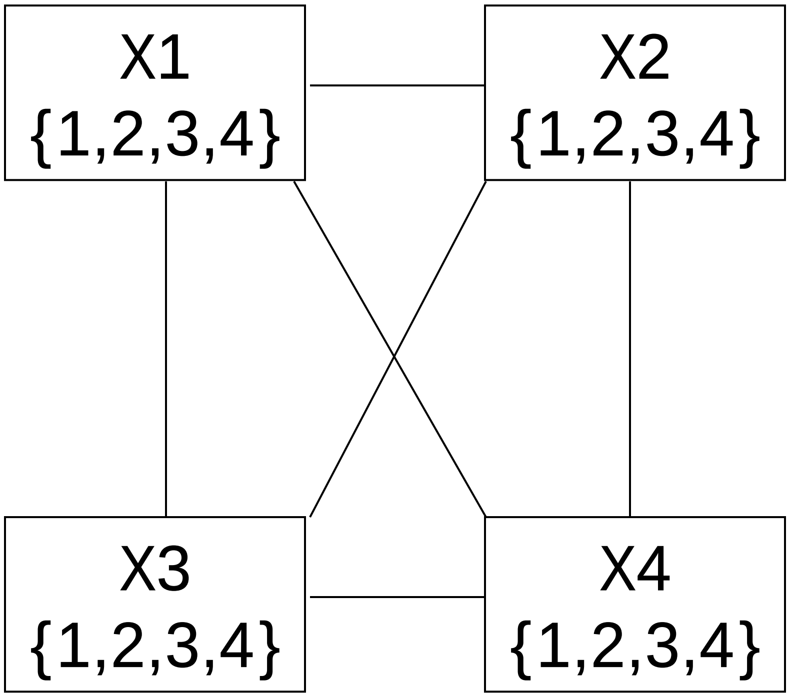
Forward checking has detected that this partial assignment is inconsistent with any complete assignment

Ex: 4-Queens Problem

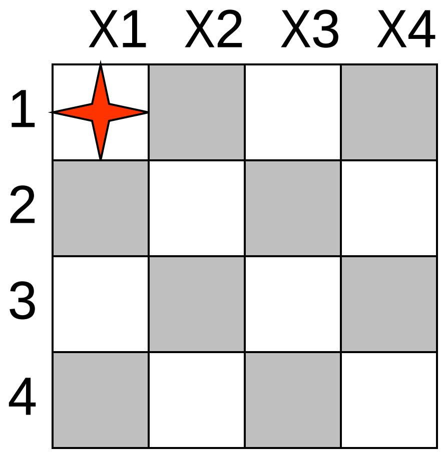
Backtracking search with forward checking

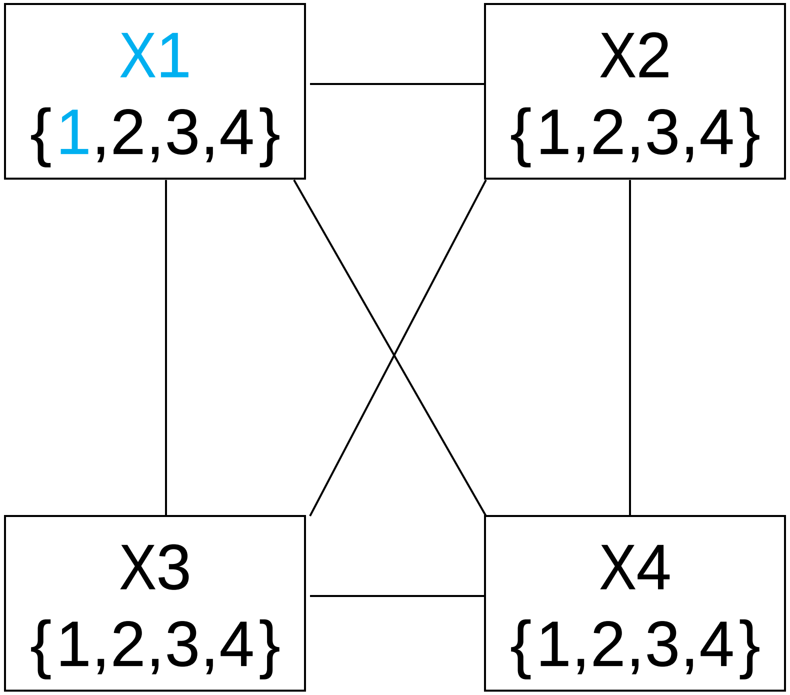
Bookkeeping is tricky & complicated





Ex: 4-Queens Problem

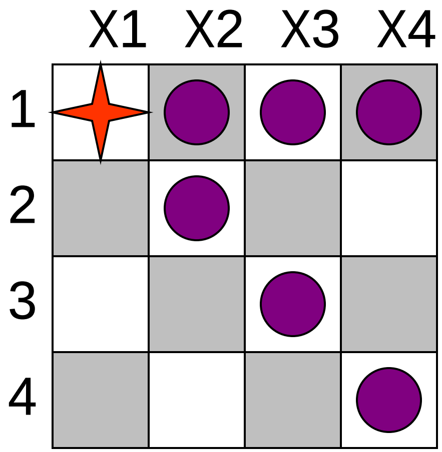


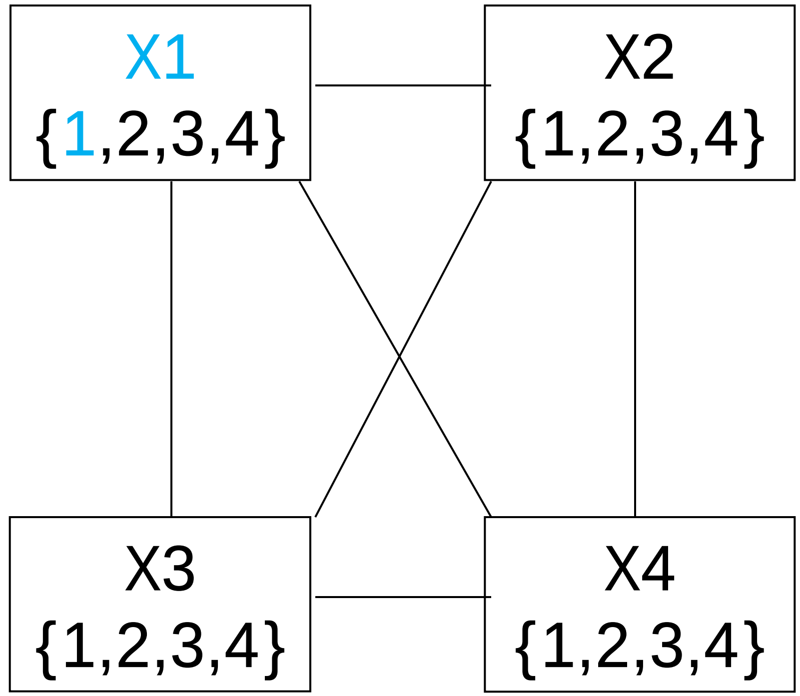


Red = value is assigned to variable

Blue = most recent variable/value pair

Ex: 4-Queens Problem





Red = value is assigned to variable

Blue = most recent variable/value pair

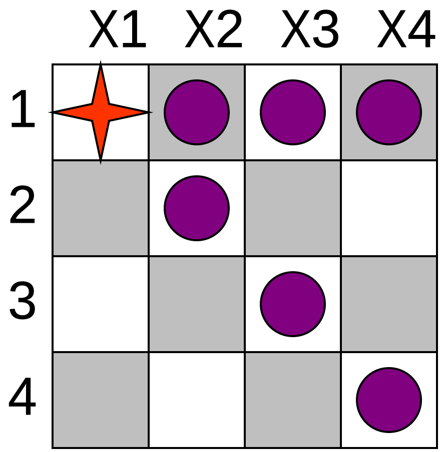
Ex: 4-Queens Problem

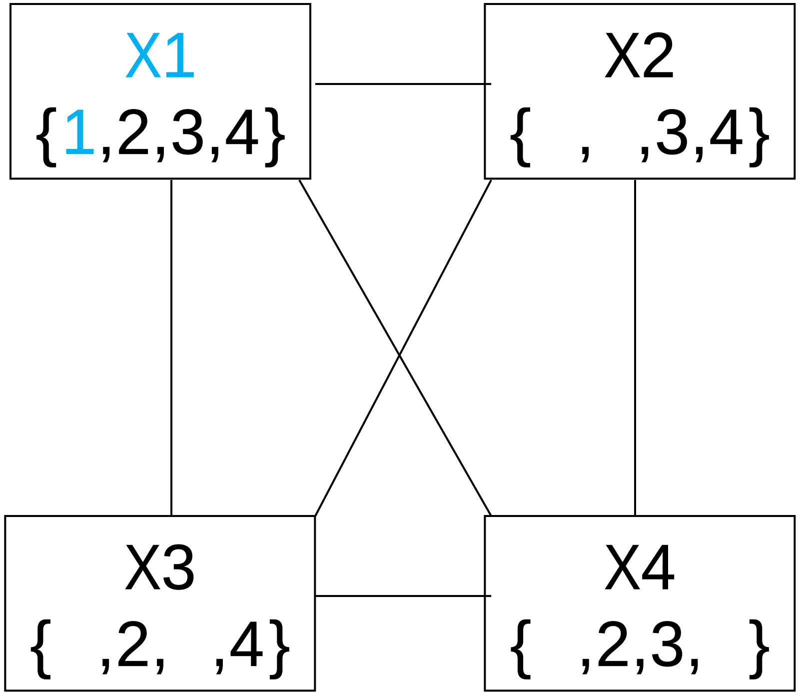
- X1 Level:
  - **Deleted:**
    - { (X2,1) (X2,2) (X3,1) (X3,3) (X4,1) (X4,4) }

- (**Please note:** As always in computer science, there are many different ways to implement anything. The book-keeping method shown here was chosen because it is easy to present and understand visually. It is not necessarily the most efficient way to implement the book-keeping in a computer. Your job as an algorithm designer is to think long and hard about your problem, then devise an efficient implementation.)

- One possibly more efficient equivalent alternative (of many):
  - **Deleted:**
    - { (X2:1,2) (X3:1,3) (X4:1,4) }

Ex: 4-Queens Problem

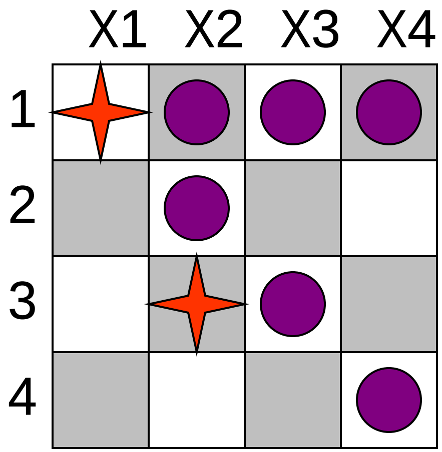


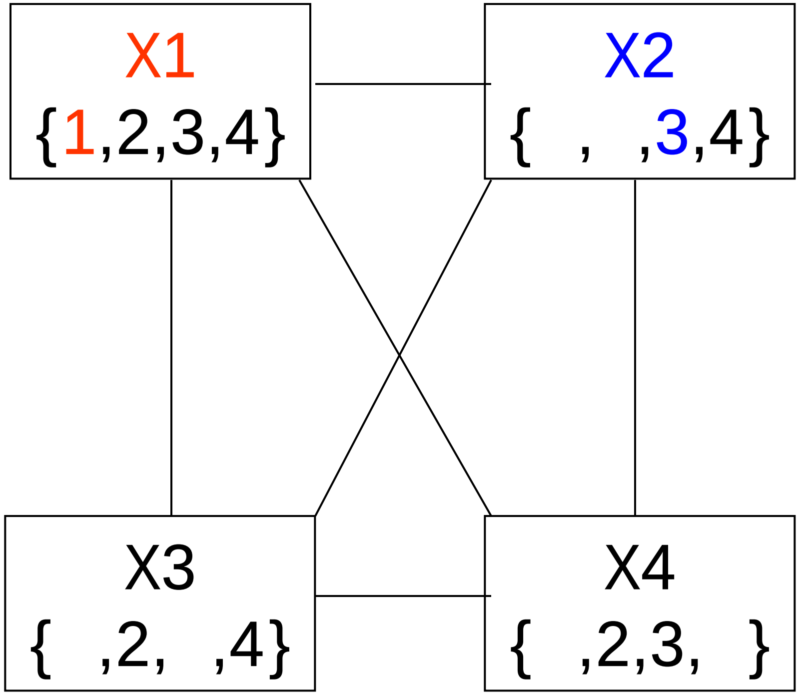


Red = value is assigned to variable

Blue = most recent variable/value pair

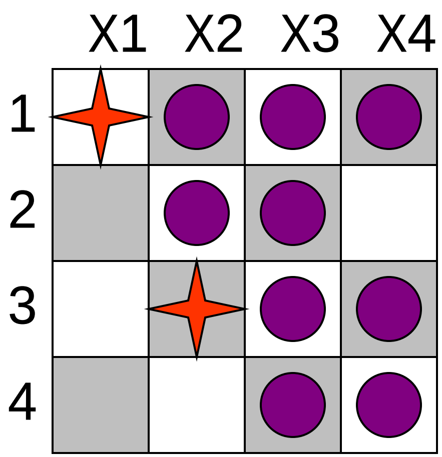
Ex: 4-Queens Problem

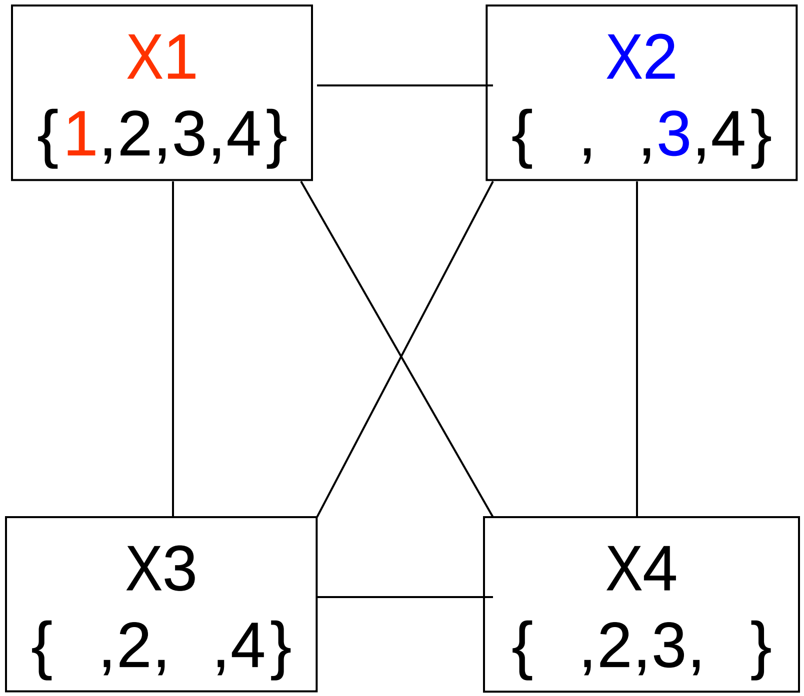




Red = value is assigned to variable
Blue = most recent variable/value pair

Ex: 4-Queens Problem





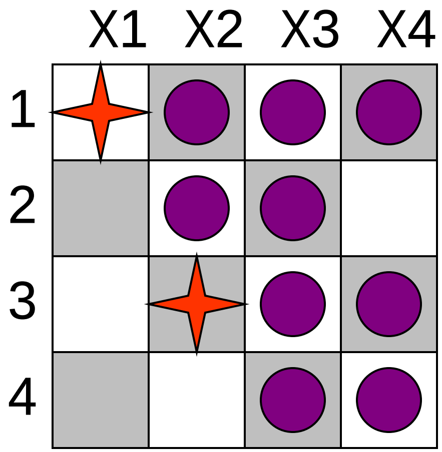
Red = value is assigned to variable

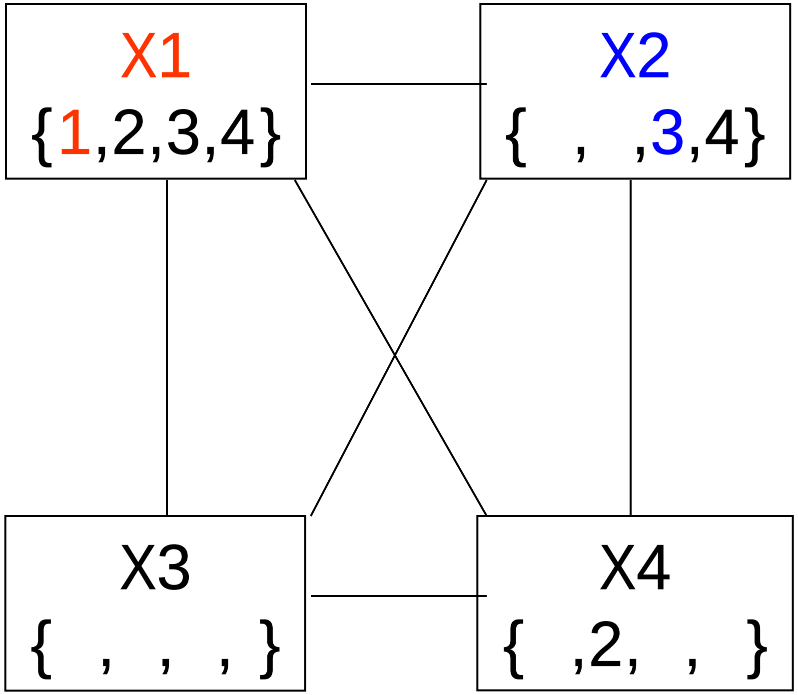
Blue = most recent variable/value pair

Ex: 4-Queens Problem

- X1 Level:
  - Deleted:
    - { (X2,1) (X2,2) (X3,1) (X3,3) (X4,1) (X4,4) }
- X2 Level:
  - **Deleted:**
    - { (X3,2) (X3,4) (X4,3) }
    - (**Please note:** Of course, we could have failed as soon as we deleted { (X3,2) (X3,4) }. There was no need to continue to delete (X4,3), because we already had established that the domain of X3 was null, and so we already knew that this branch was futile and we were going to fail anyway. The book-keeping method shown here was chosen because it is easy to present and understand visually. It is not necessarily the most efficient way to implement the book-keeping in a computer. Your job as an algorithm designer is to think long and hard about your problem, then devise an efficient implementation.)

Ex: 4-Queens Problem





Red = value is assigned to variable

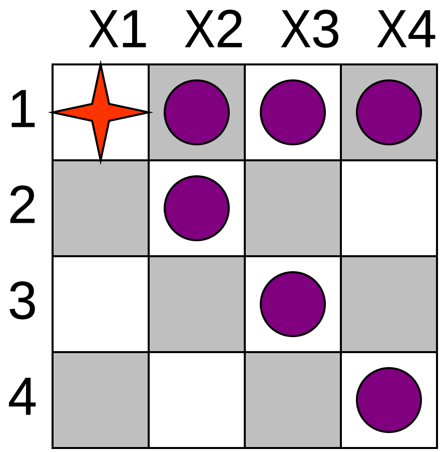
Blue = most recent variable/value pair

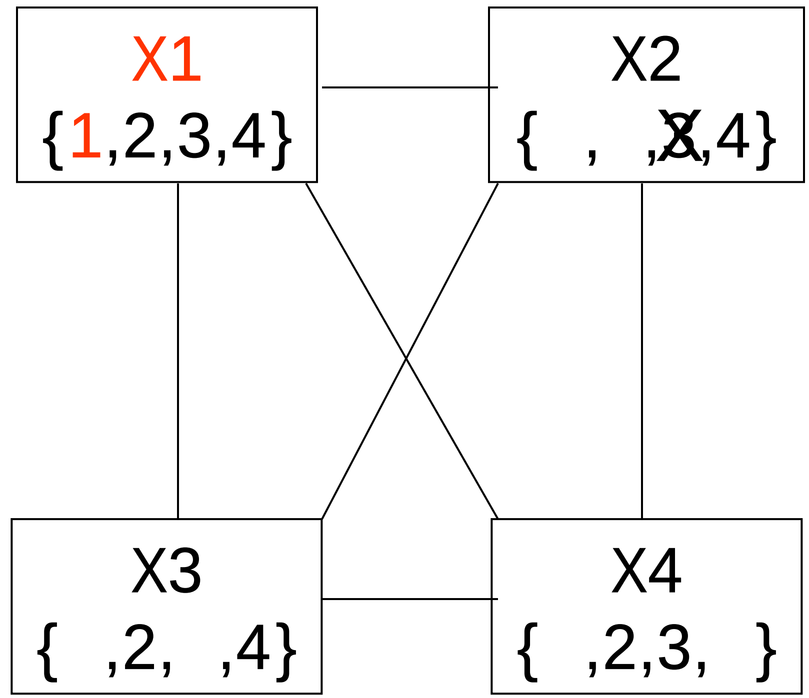
Ex: 4-Queens Problem

- X1 Level:
  - Deleted:
    - { (X2,1) (X2,2) (X3,1) (X3,3) (X4,1) (X4,4) }

- X2 Level:
  - **FAIL at X2=3.**
  - **Restore:**
    - { (X3,2) (X3,4) (X4,3) }

Ex: 4-Queens Problem

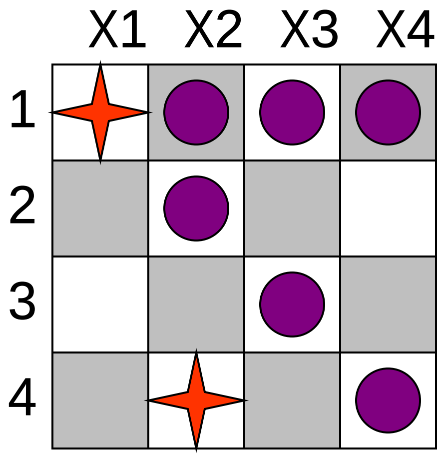


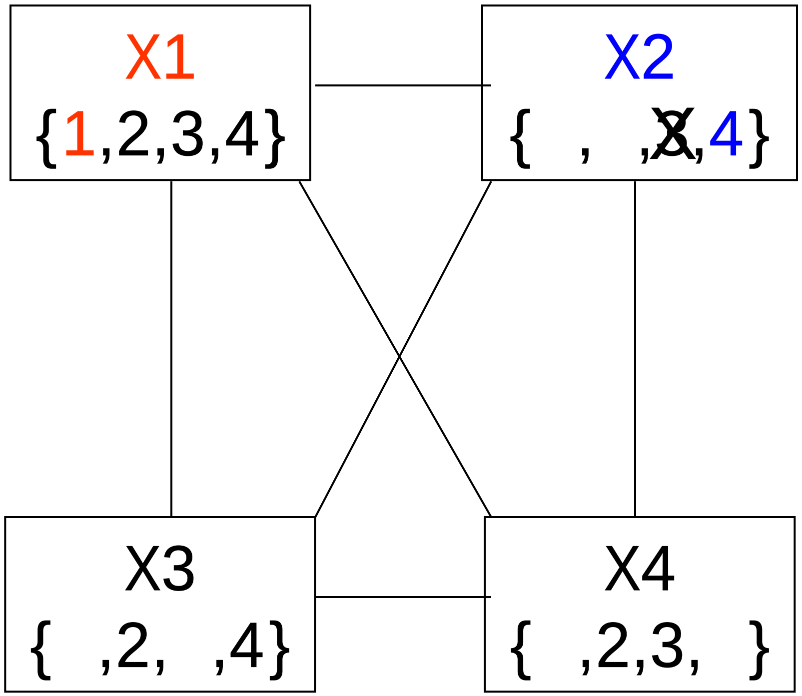


Red = value is assigned to variable

X = value led to failure

Ex: 4-Queens Problem



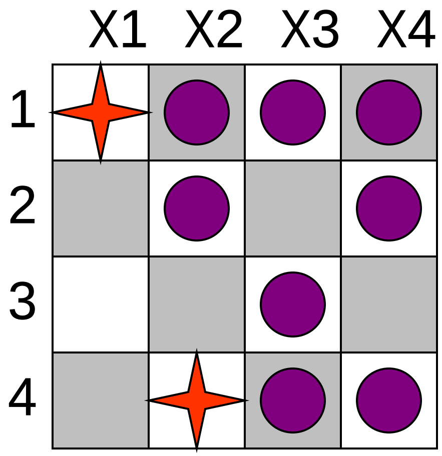


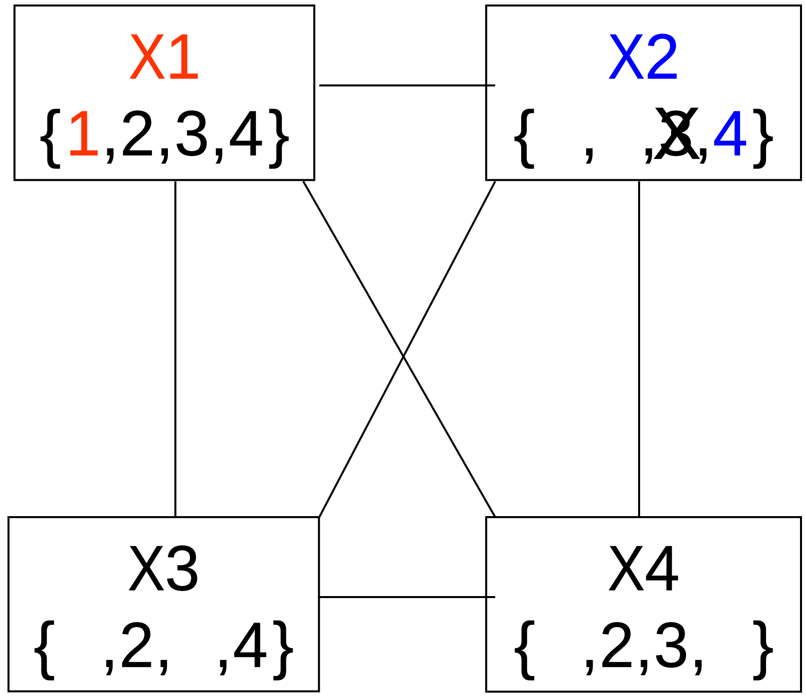
Red = value is assigned to variable

Blue = most recent variable/value pair

X = value led to failure

Ex: 4-Queens Problem



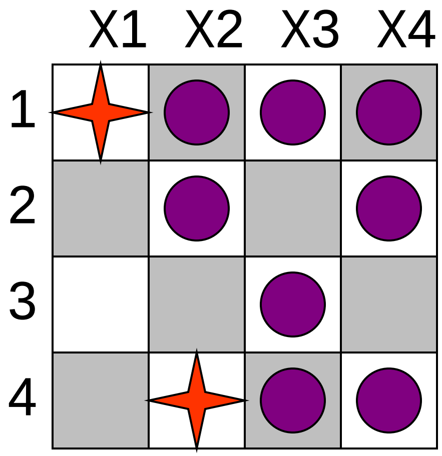


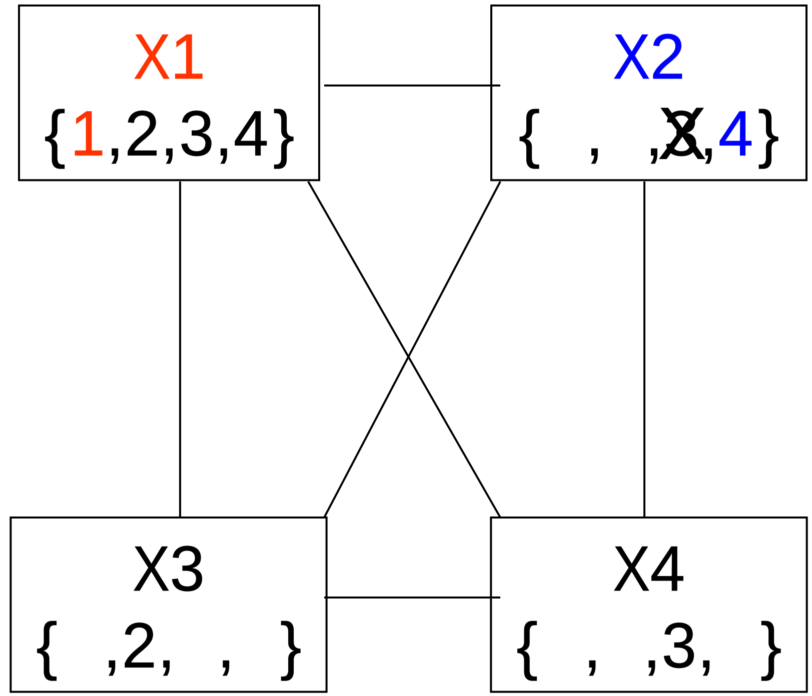
Red = value is assigned to variable
Blue = most recent variable/value pair
X = value led to failure

Ex: 4-Queens Problem

- X1 Level:
  - Deleted:
    - { (X2,1) (X2,2) (X3,1) (X3,3) (X4,1) (X4,4) }
- X2 Level:
  - **Deleted:**
    - { (X3,4) (X4,2) }

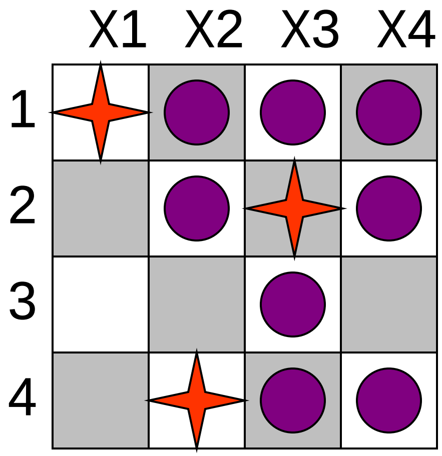
Ex: 4-Queens Problem

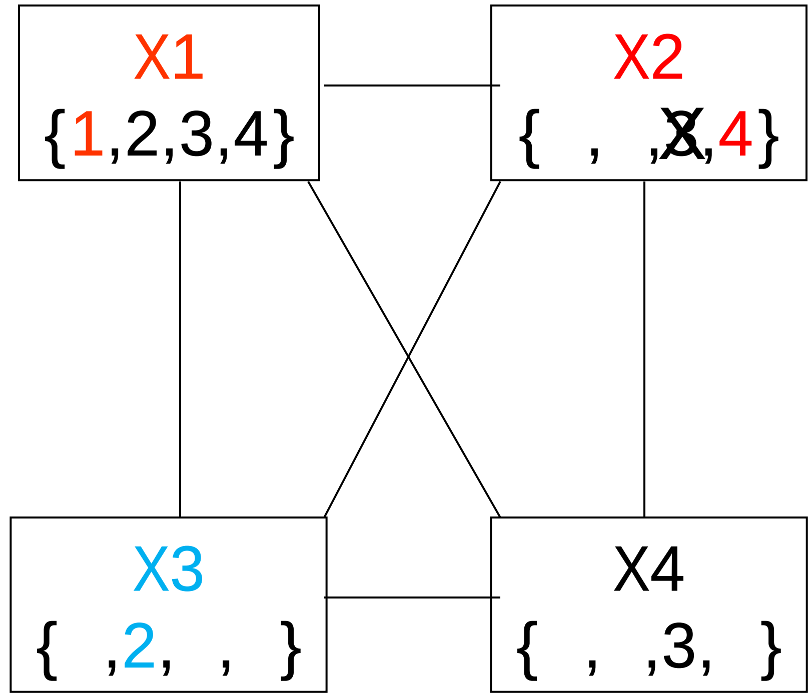




Red = value is assigned to variable
Blue = most recent variable/value pair
X = value led to failure

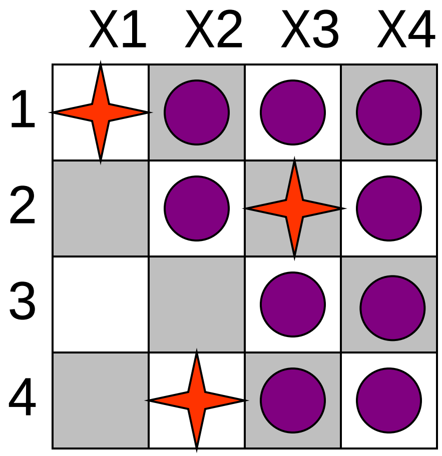
Ex: 4-Queens Problem

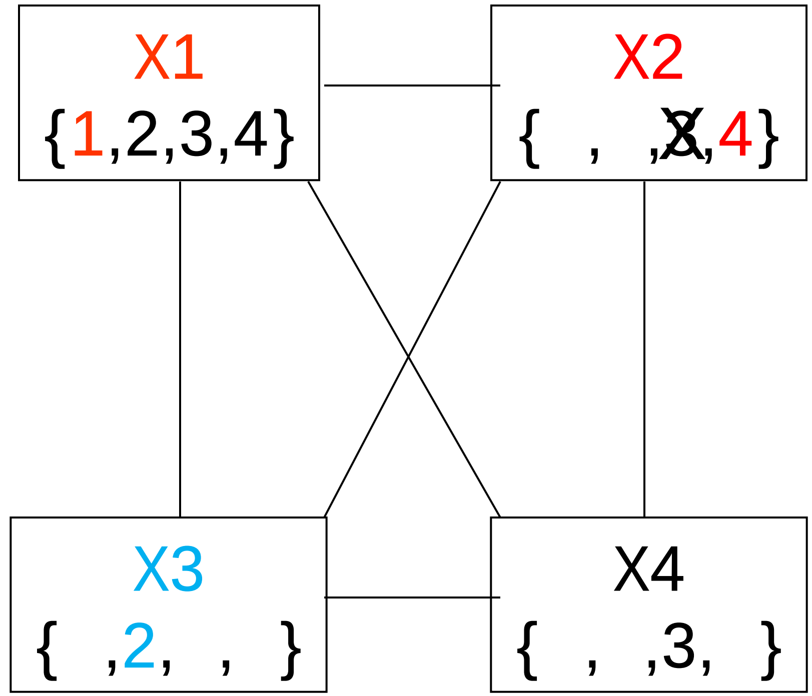




Red = value is assigned to variable
Blue = most recent variable/value pair
X = value led to failure

Ex: 4-Queens Problem





Red = value is assigned to variable

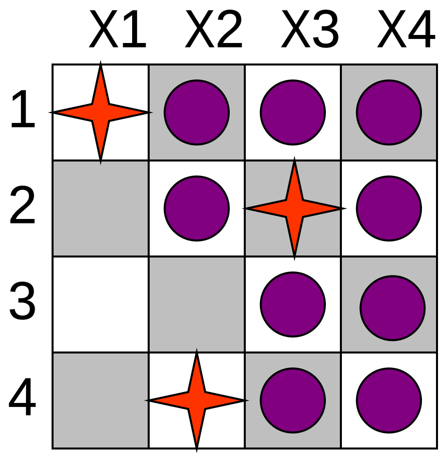
Blue = most recent variable/value pair

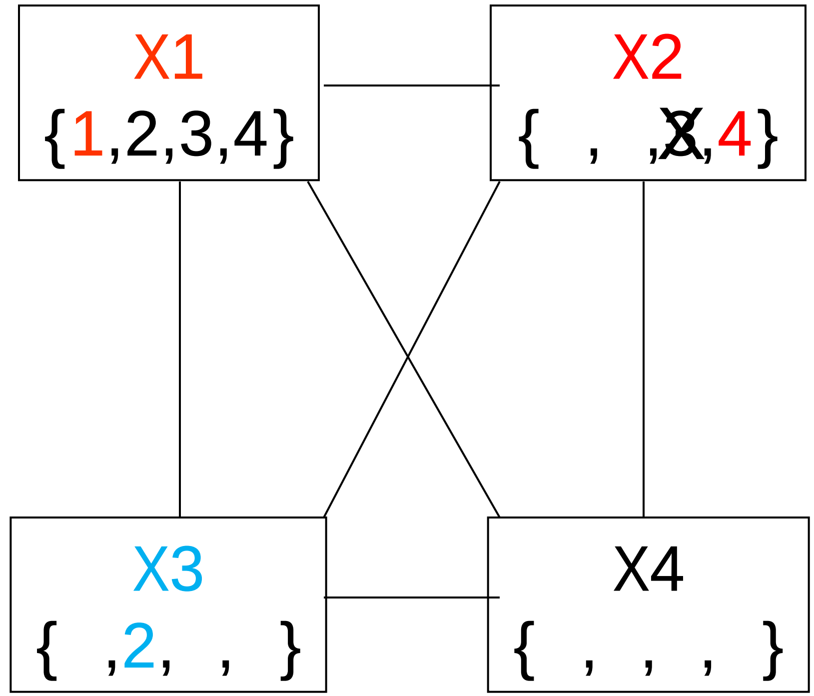
X = value led to failure

Ex: 4-Queens Problem

- X1 Level:
  - Deleted:
    - { (X2,1) (X2,2) (X3,1) (X3,3) (X4,1) (X4,4) }
- X2 Level:
  - Deleted:
    - { (X3,4) (X4,2) }
- X3 Level:
  - **Deleted:**
    - { (X4,3) }

Ex: 4-Queens Problem





Red = value is assigned to variable

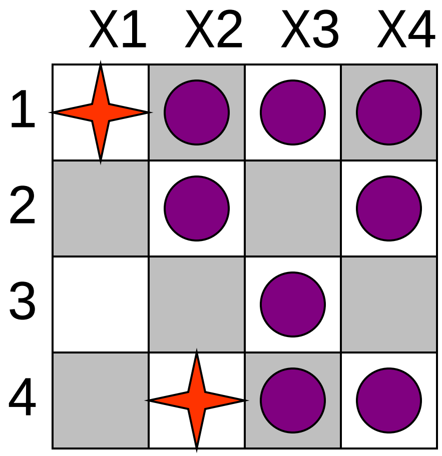
Blue = most recent variable/value pair

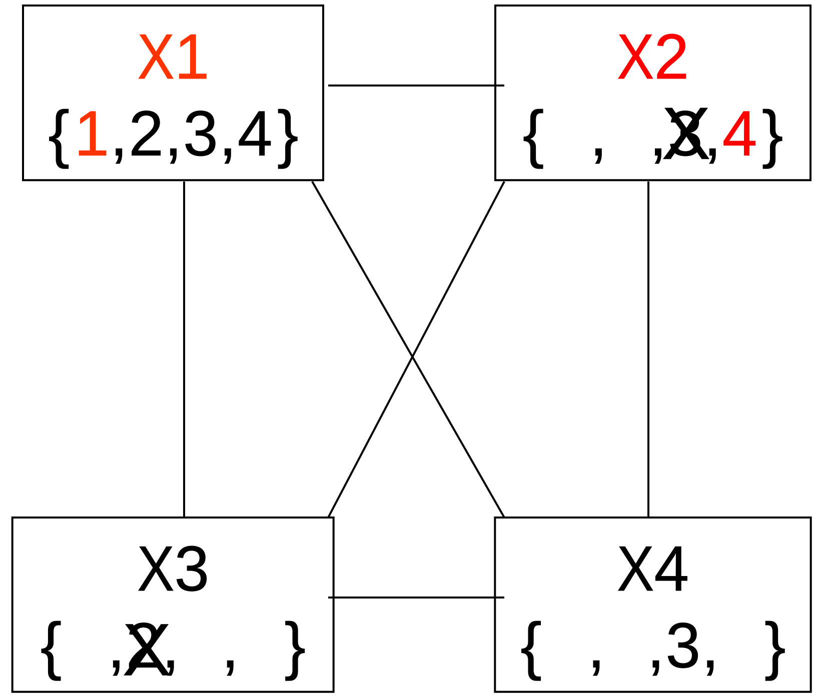
X = value led to failure

Ex: 4-Queens Problem

- X1 Level:
  - Deleted:
    - { (X2,1) (X2,2) (X3,1) (X3,3) (X4,1) (X4,4) }
- X2 Level:
  - Deleted:
    - { (X3,4) (X4,2) }
- X3 Level:
  - **Fail at X3=2.**
  - **Restore:**
    - { (X4,3) }

Ex: 4-Queens Problem





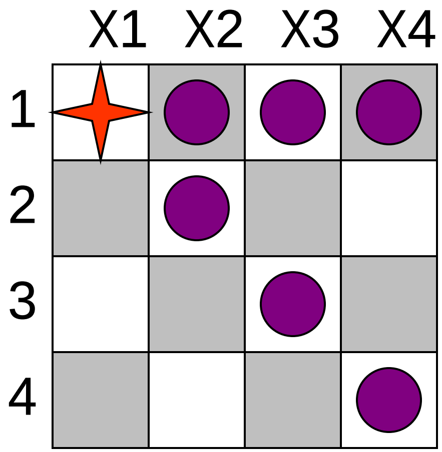
Red = value is assigned to variable

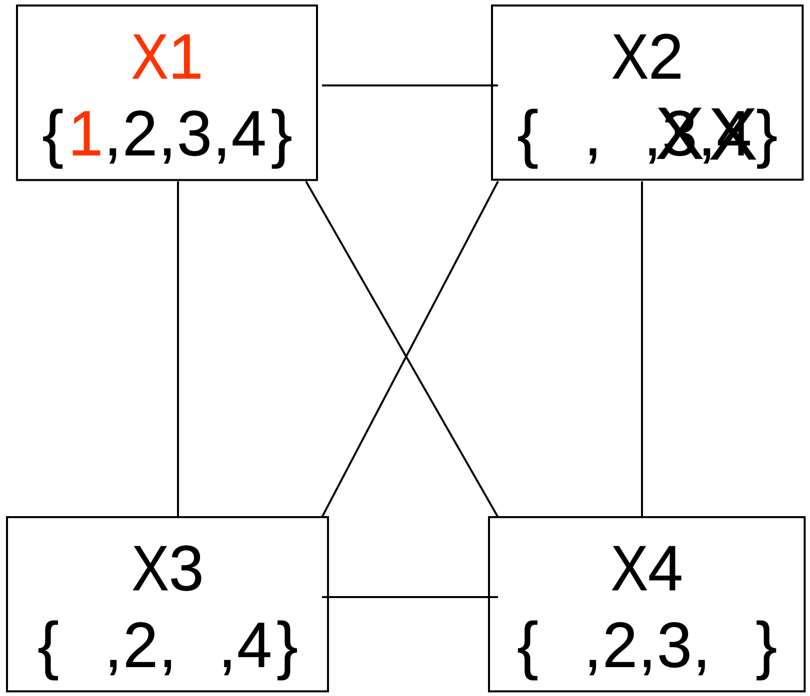
X = value led to failure

Ex: 4-Queens Problem

- X1 Level:
  - Deleted:
    - { (X2,1) (X2,2) (X3,1) (X3,3) (X4,1) (X4,4) }
- X2 Level:
  - **Fail at X2=4.**
  - **Restore:**
    - { (X3,4) (X4,2) }

Ex: 4-Queens Problem



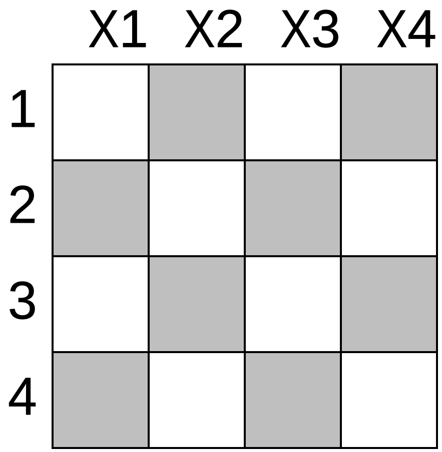


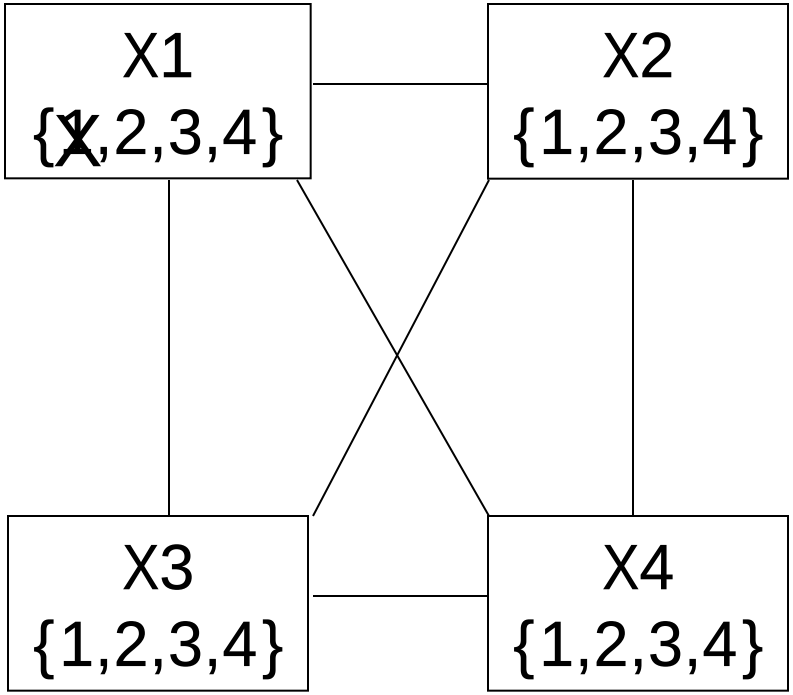
Red = value is assigned to variable
X = value led to failure

Ex: 4-Queens Problem

- X1 Level:
  - **Fail at X1=1.**
  - **Restore:**
    - { (X2,1) (X2,2) (X3,1) (X3,3) (X4,1) (X4,4) }

Ex: 4-Queens Problem

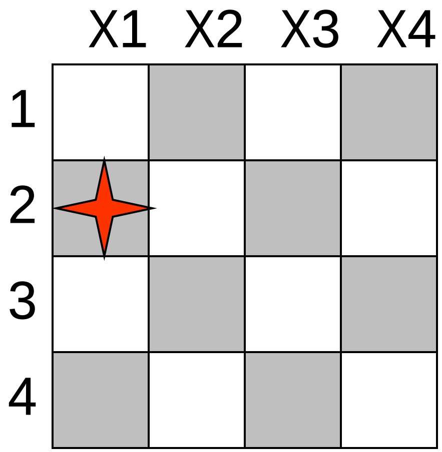


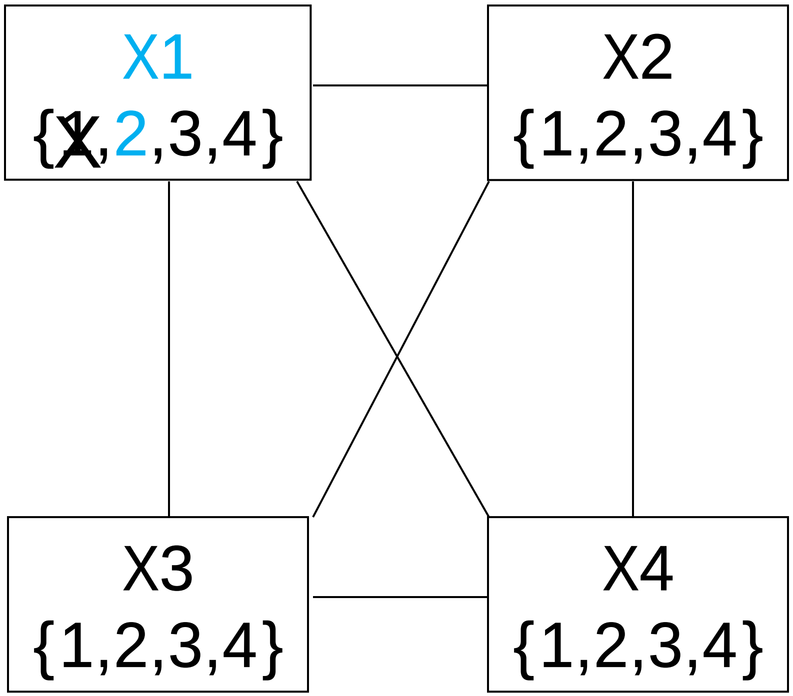


Red = value is assigned to variable

X = value led to failure

Ex: 4-Queens Problem



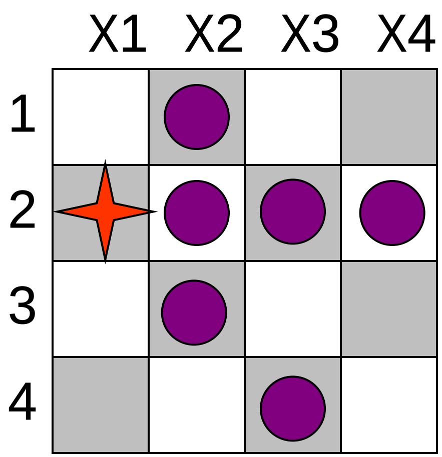


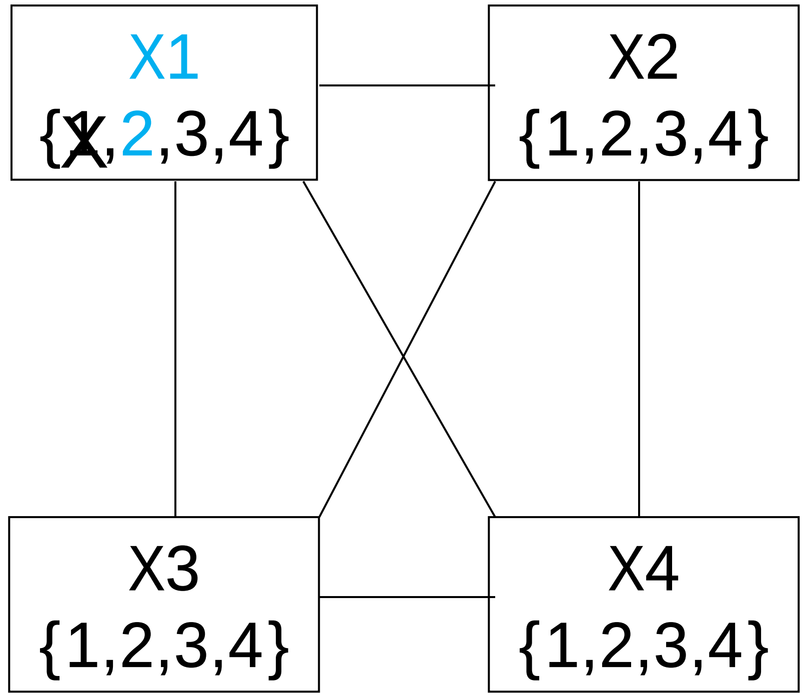
Red = value is assigned to variable

Blue = most recent variable/value pair

X = value led to failure

Ex: 4-Queens Problem





Red = value is assigned to variable

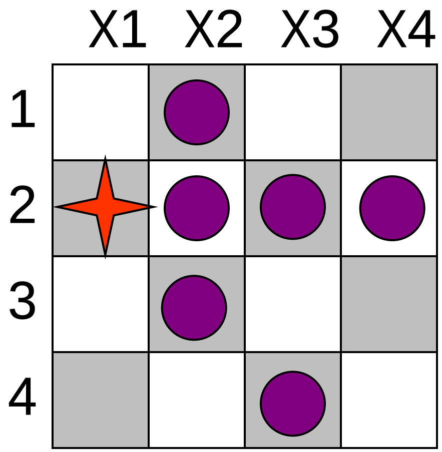
Blue = most recent variable/value pair

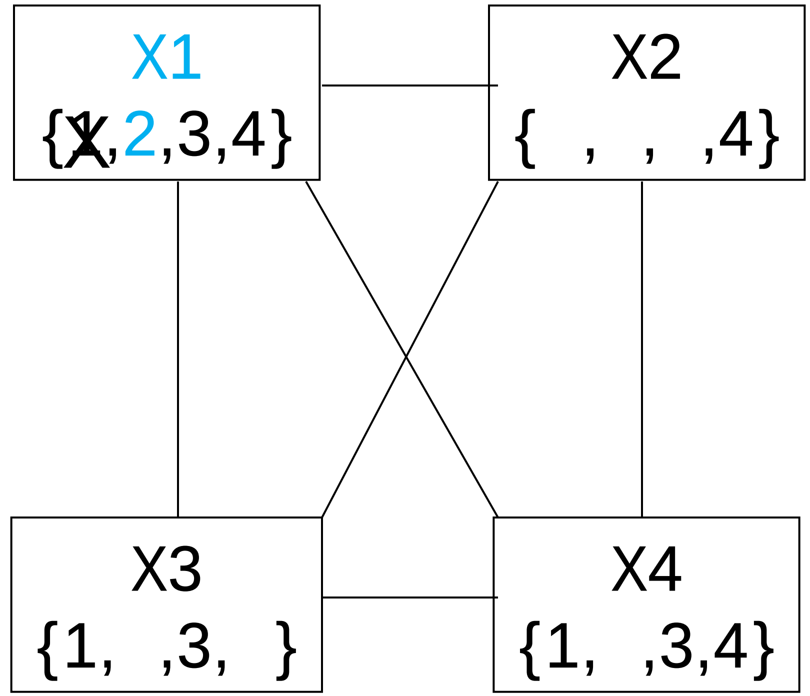
X = value led to failure

Ex: 4-Queens Problem

- X1 Level:
  - **Deleted:**
    - { (X2,1) (X2,2) (X2,3) (X3,2) (X3,4) (X4,2) }

Ex: 4-Queens Problem



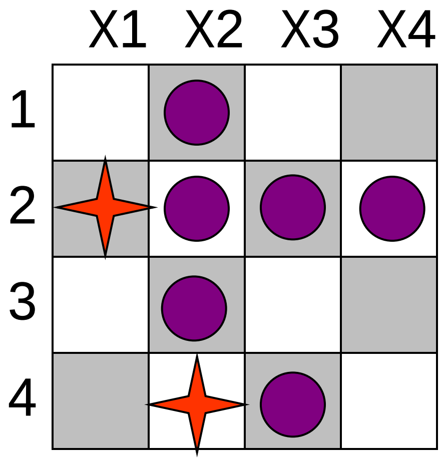


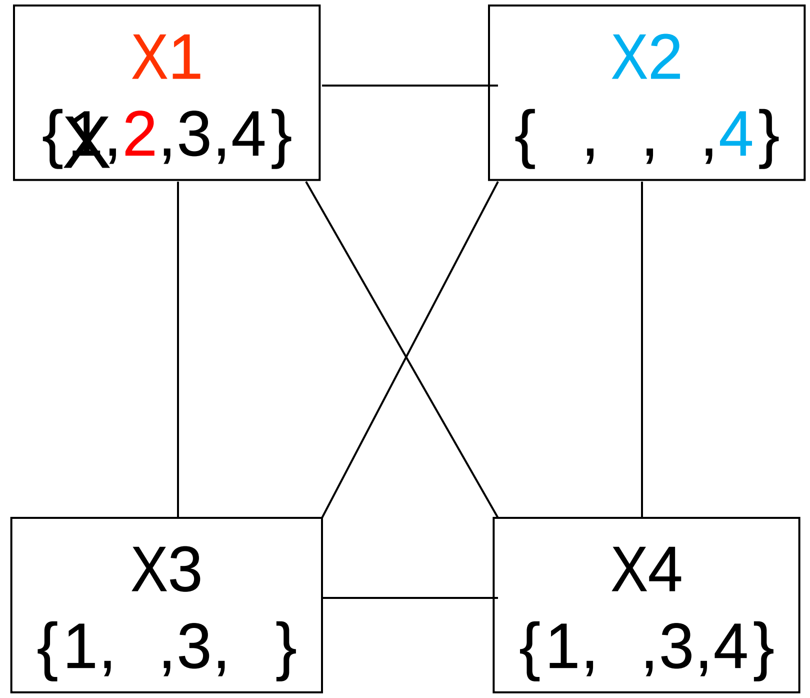
Red = value is assigned to variable

Blue = most recent variable/value pair

X = value led to failure

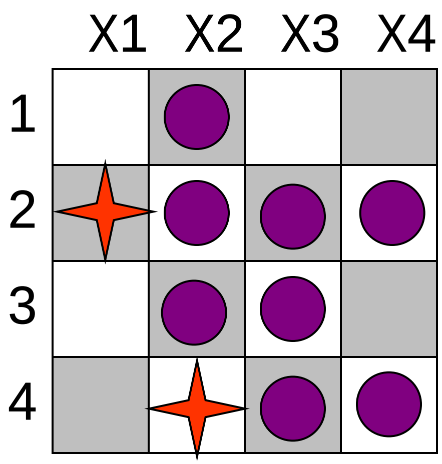
Ex: 4-Queens Problem

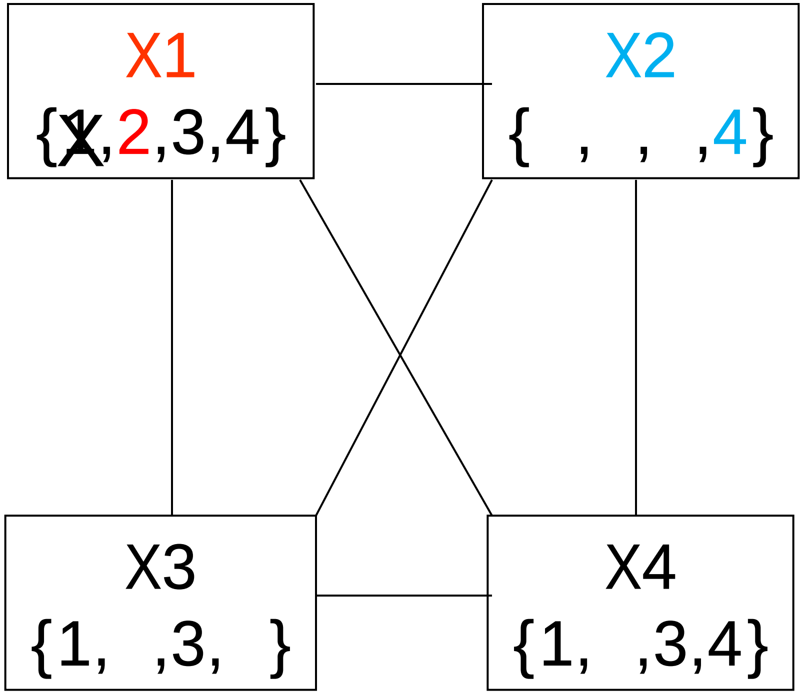




Red = value is assigned to variable
Blue = most recent variable/value pair
X = value led to failure

Ex: 4-Queens Problem





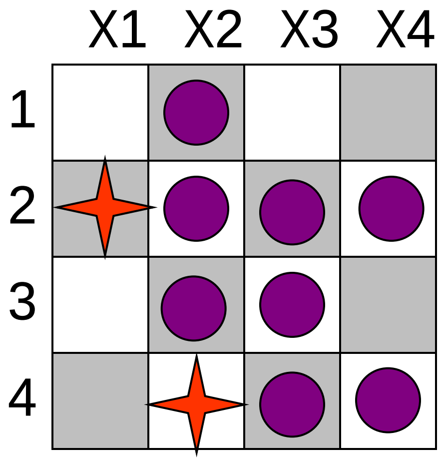
Red = value is assigned to variable
Blue = most recent variable/value pair
X = value led to failure

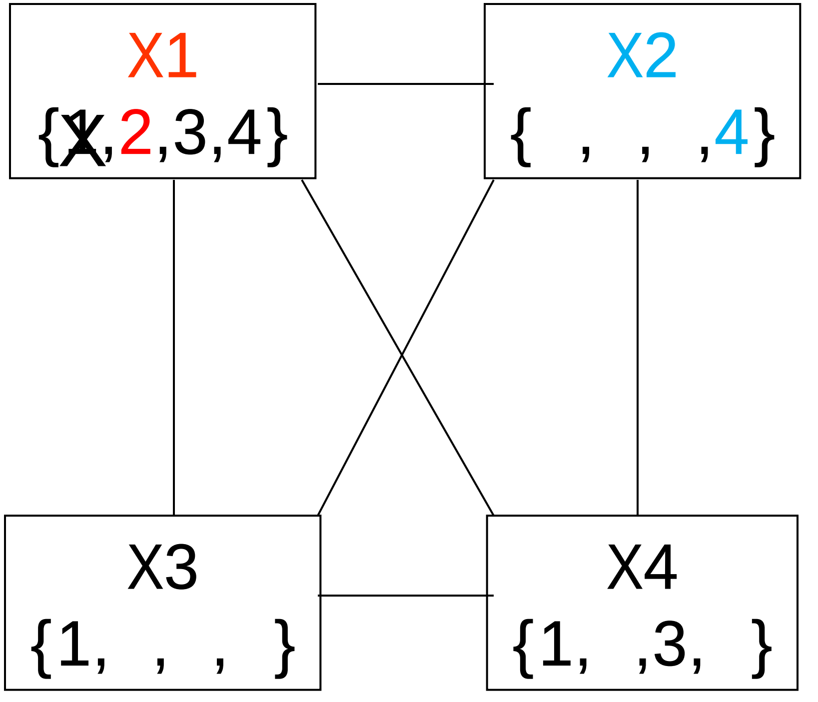
Ex: 4-Queens Problem

- X1 Level:
  - Deleted:
    - { (X2,1) (X2,2) (X2,3) (X3,2) (X3,4) (X4,2) }

- X2 Level:
  - **Deleted:**
    - { (X3,3) (X4,4) }

Ex: 4-Queens Problem



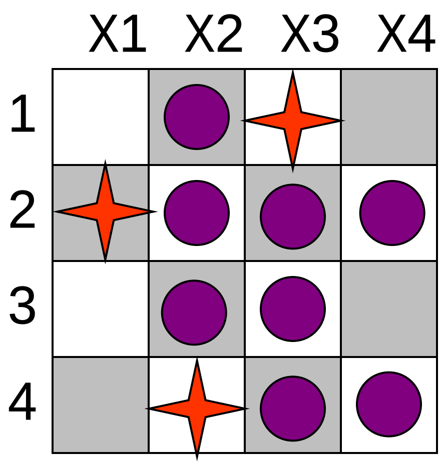


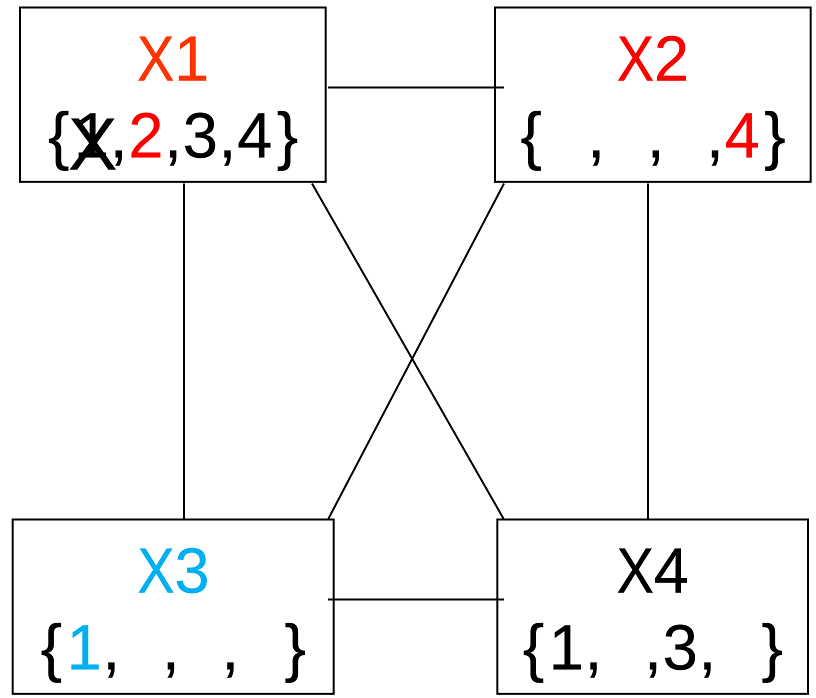
Red = value is assigned to variable

Blue = most recent variable/value pair

X = value led to failure

Ex: 4-Queens Problem



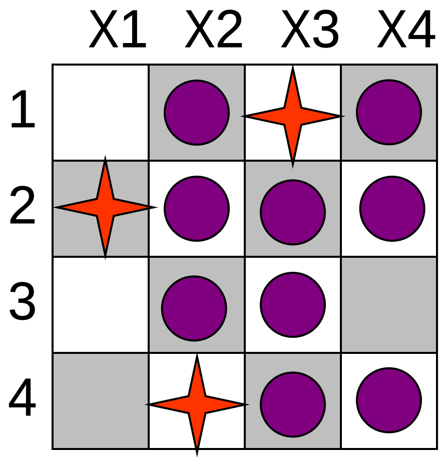


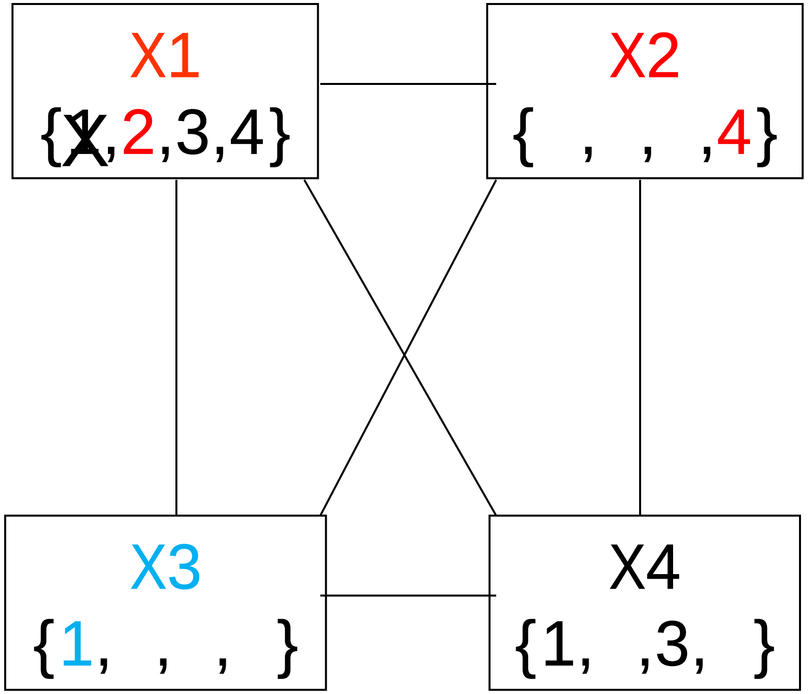
Red = value is assigned to variable

Blue = most recent variable/value pair

X = value led to failure

Ex: 4-Queens Problem





Red = value is assigned to variable

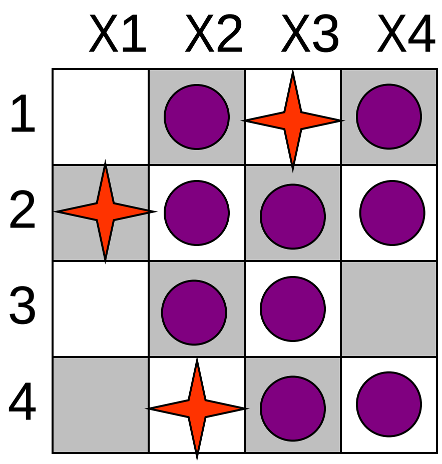
Blue = most recent variable/value pair

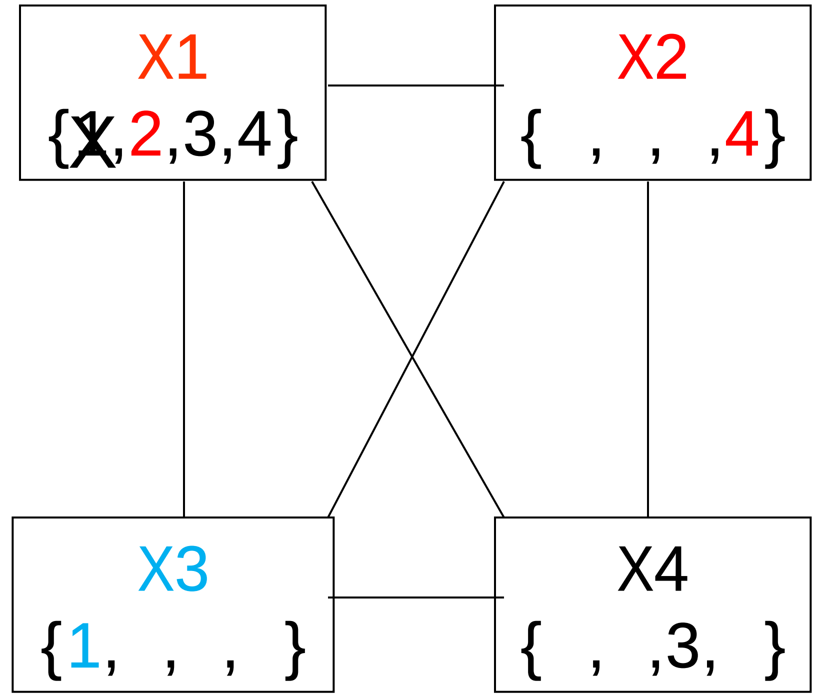
X = value led to failure

Ex: 4-Queens Problem

- X1 Level:
  - Deleted:
    - { (X2,1) (X2,2) (X2,3) (X3,2) (X3,4) (X4,2) }
- X2 Level:
  - Deleted:
    - { (X3,3) (X4,4) }
- X3 Level:
  - **Deleted:**
    - { (X4,1) }

Ex: 4-Queens Problem



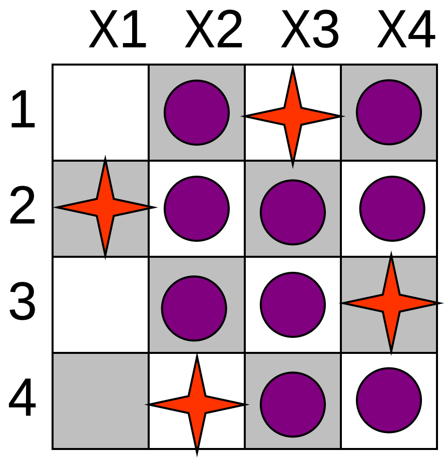


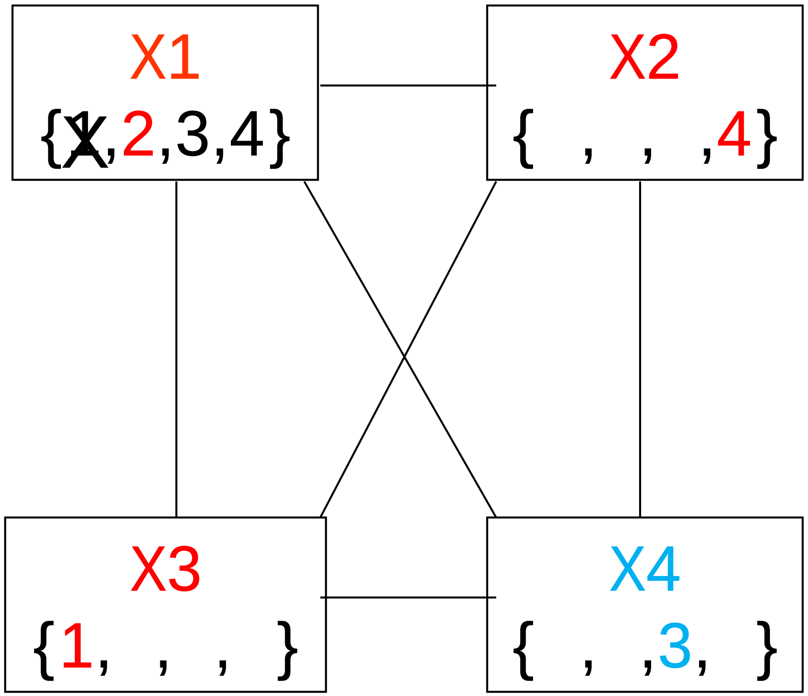
Red = value is assigned to variable

Blue = most recent variable/value pair

X = value led to failure

Ex: 4-Queens Problem





Red = value is assigned to variable
Blue = most recent variable/value pair
X = value led to failure

Norvig’s basic Sudoku strategies

- Norvig gives "two important strategies“ for Sudoku:
  - *(1) If a square has only one possible value, then eliminate that value from the square's peers.*
  - *(2) If a unit has only one possible place for a value, then put the value there.*
- Norvig’s first strategy is Forward Checking.

Norvig's second Sudoku strategy

After FC:

- D_{D1} = {1,2,3,4}
- D_{Others} = {1,2,3}

Yet D1 must be 4!

	1	2	3	4
A		4		
B			4	
C				4
D				

Must be 4!
Forward checking won't derive this.

Norvig's second Sudoku strategy

Allocate an array Counter[N]

For each Unit in {rows, cols, blocks*}
- Zero Counter
- For I from 1 to N
  - For each Value in $D_{Unit[I]}$
    - Increment Counter[Value]
- For I from 1 to N
  - If (Counter[I] = 1) then
    - Find the one domain in Unit that has I for a possible value, and set that cell to I

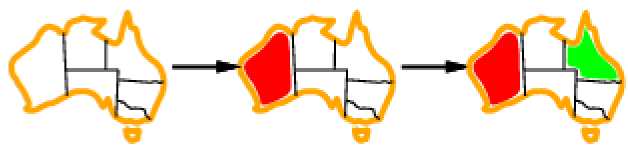
* Norvig calls these boxes

	1	2	3	4
A		4		
B			4	
C				4
D	**4**			

(Can also be done with FC, by using auxiliary variables for book-keeping)

Constraint propagation

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



- **Constraint propagation** repeatedly enforces constraints locally
  - Can detect failure earlier
  - But, takes more computation – **is it worth the extra effort?**

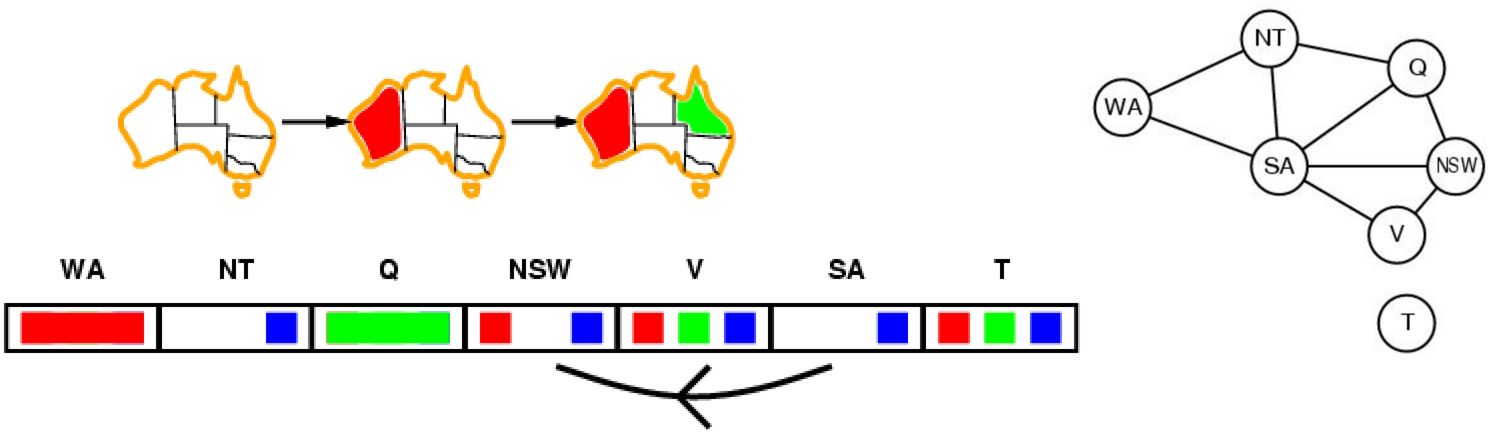
Arc consistency (AC-3) algorithm

- An Arc $X \rightarrow Y$ is consistent iff for *__every__* value x of X there is *__some__* value y of Y that is consistent with x
- Put all arcs $X \rightarrow Y$ on a queue
  - Each undirected constraint graph arc is two directed arcs
  - Undirected X—Y becomes directed $X \rightarrow Y$ and $Y \rightarrow X$
  - $X \rightarrow Y$ and $Y \rightarrow X$ both go on queue, separately
- Pop one arc $X \rightarrow Y$ and remove any inconsistent values from X
- If any change in X, put all arcs $Z \rightarrow X$ back on queue, where Z is any neighbor of X that is not equal to Y
- Continue until queue is empty

Arc consistency (AC-3)

- Simplest form of propagation makes each arc consistent
- $X \to Y$ is consistent iff (iff = if and only if)

  for every value *x* of *X* there is some allowed value *y for Y* *(note: directed!)*



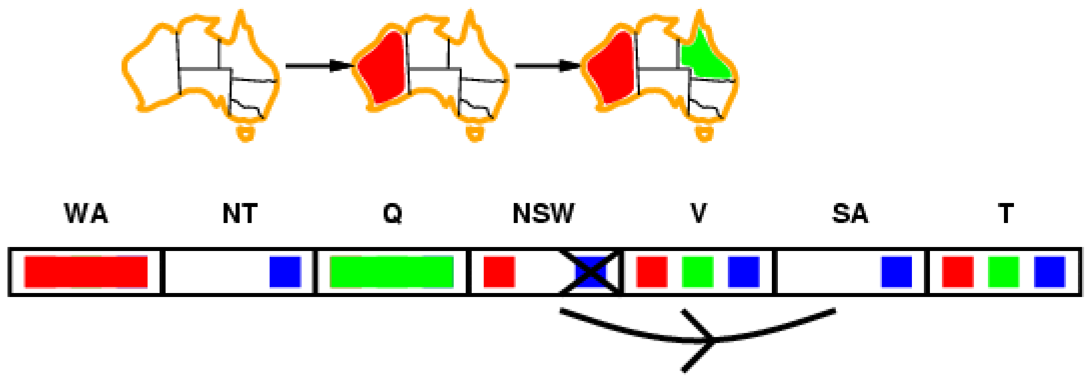
- Consider state after WA=red, Q=green
  - SA $\to$ NSW is consistent because

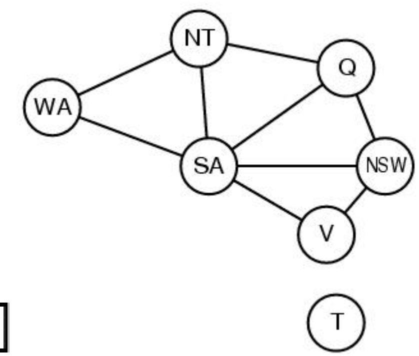
    SA = blue and NSW = red satisfies all constraints on SA and NSW

Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \to Y$ is consistent iff

  for every value x of X there is some allowed value y *for* Y *(note: directed!)*





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

- Consider state after WA=red, Q=green
  - NSW $\to$ SA consistent if

    NSW = red and SA = blue

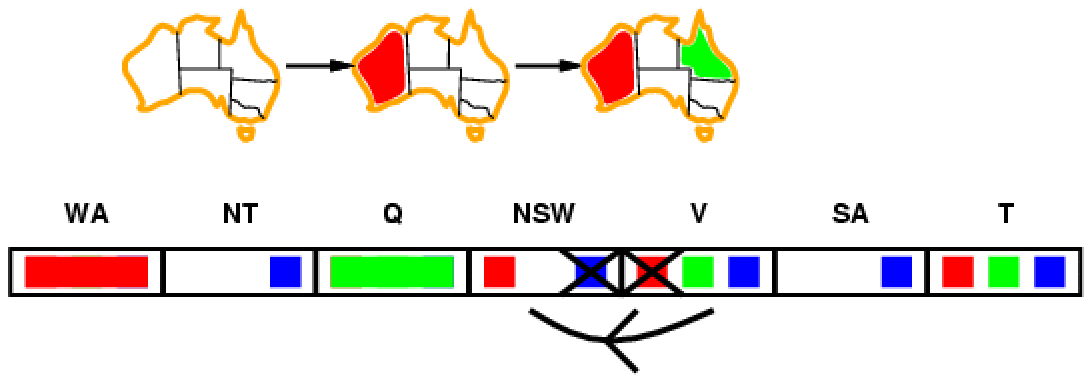
    NSW = blue and SA = ??? => NSW = blue can be pruned

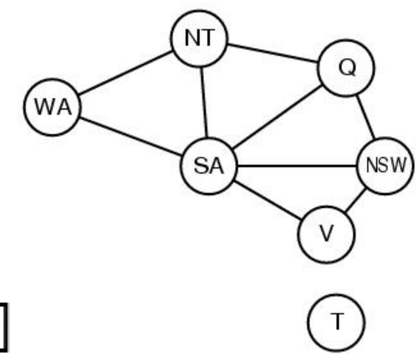
    No current domain value for SA is consistent

Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff

  for every value *x* of *X* there is some allowed value *y for Y* *(note: directed!)*



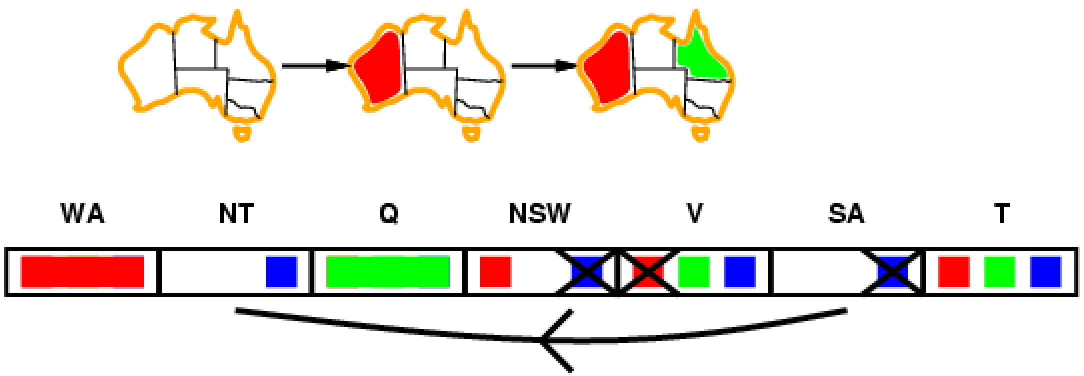


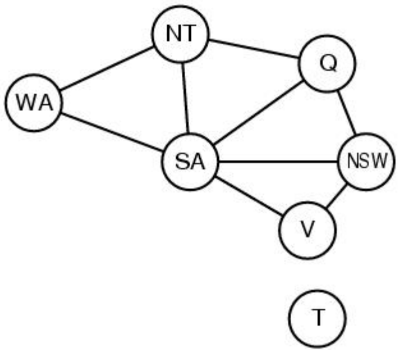
- **Enforce arc consistency:**
  - arc can be made consistent by removing blue from NSW
- **Continue to propagate constraints:**
  - Check V $\rightarrow$ NSW : not consistent for V = red; remove red from V

Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff

  for every value x of X there is some allowed value y *for* Y *(note: directed!)*





- Continue to propagate constraints
- SA $\rightarrow$ NT not consistent:
  - **And cannot be made consistent! Failure!**
- Arc consistency detects failure earlier than FC
  - But requires more computation: is it worth the effort?

Ex: Arc Consistency in Sudoku

		2	4		6			
8	6	5	1			2		
	1				8	6		9
9				4		8	6	
	4	7				1	9	
	5	8		6				3
4		6	9				7	
		9			4	5	8	1
			3		2	9		

- **Variables**: 81 cells
- **Domains** = {1,2,3,4,5,6,7,8,9}
- **Constraints**:
    - 27 all-diff

Each row, column and major block must be all different

"Well posed" if it has unique solution: 27 constraints

Ex: Arc Consistency in Sudoku

(Forward Checking yields the same result)

		2	4		6			
8	6	5	1			2		
	1				8	6		9
9				4		8	6	
	4	7				1	9	
	5	8		6				3
4		6	9				7	**2**
		9			4	5	8	1
			3		2	9		

- **Variables**: 81 cells
- **Domains** = {1,2,3,4,5,6,7,8,9}
- **Constraints**:
  - 27 all-diff

Each row, column and major block must be all different

"Well posed" if it has unique solution: 27 constraints

Arc consistency checking

- Can be run as a preprocessor, or after each assignment
  - As preprocessor before search: Removes obvious inconsistencies
  - After each assignment: Reduces search cost but increases step cost
- AC is run repeatedly until no inconsistency remains
  - Like Forward Checking, but exhaustive until quiescence
- Trade-off
  - Requires overhead to do; but usually better than direct search
  - In effect, it can successfully eliminate large (and inconsistent) parts of the state space more effectively than can direct search alone
- Need a systematic method for arc-checking
  - If *X* loses a value, neighbors of *X* need to be rechecked:

    i.e., incoming arcs can become inconsistent again (outgoing arcs stay consistent).

Arc consistency algorithm (AC-3)

function AC-3(*csp*) **returns** false if inconsistency found, else true, may reduce *csp* 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*
      / initial queue must contain both* (X_i, X_j) *and* (X_j, X_i) **/*
    while queue is not empty **do**
      $(X_i, X_j) \leftarrow$ REMOVE-FIRST(*queue*)
      if REMOVE-INCONSISTENT-VALUES(X_i, X_j) **then**
            if size of $D_i = 0$ **then return** false
            for each X_k **in** NEIGHBORS[X_i] − {X_j} **do**
                  add (X_k, X_i) to queue if not already there
    return true

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) **returns** *true* iff we delete a
      value from the domain of X_i
    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*

(from Mackworth, 1977)

Complexity of AC-3

- A binary CSP has at most n^2 arcs
- Each arc can be inserted in the queue d times (worst case)
  - (X, Y): only d values of X to delete
- Consistency of an arc can be checked in $O(d^2)$ time
- Complexity is $O(n^2 d^3)$
- Although substantially more expensive than Forward Checking, Arc Consistency is usually worthwhile.

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 = node-consistency
  - E.g. 2-consistency = arc-consistency
  - E.g. 3-consistency = path-consistency

- Strongly k-consistent:
  - k-consistent for all values {k, k-1, …2, 1}

Trade-offs

- Running stronger consistency checks…
  - Takes more time
  - But will reduce branching factor and detect more inconsistent partial assignments
  - No "free lunch"
    - In worst case n-consistency takes exponential time
- "Typically" in practice:
  - Often helpful to enforce 2-Consistency (Arc Consistency)
  - Sometimes helpful to enforce 3-Consistency
  - Higher levels may take more time to enforce than they save.

Improving backtracking

- Before search: (reducing the search space)
  - Arc-consistency, path-consistency, i-consistency
  - Variable ordering (fixed)
- During search:
  - **Look-ahead schemes**:
    - Value ordering/pruning *(choose a least restricting value*),
    - Variable ordering *(choose the most constraining variable)*
    - Constraint propagation *(take decision implications forward)*
  - **Look-back schemes**:
    - Backjumping
    - Constraint recording
    - Dependency-directed backtracking

Further improvements

- Checking special constraints
  - Checking Alldiff(…) 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: min-conflicts heuristic

- Use complete-state representation
  - Initial state = all variables assigned values
  - Successor states = change 1 (or more) values

- For CSPs
  - allow states with unsatisfied constraints (unlike backtracking)
  - operators **reassign** variable values
  - hill-climbing with n-queens is an example

- **Variable selection:** randomly select any conflicted variable
- **Value selection:** <u>*min-conflicts heuristic*</u>
  - Select new value that results in a minimum number of conflicts with the other variables

Local search: min-conflicts heuristic

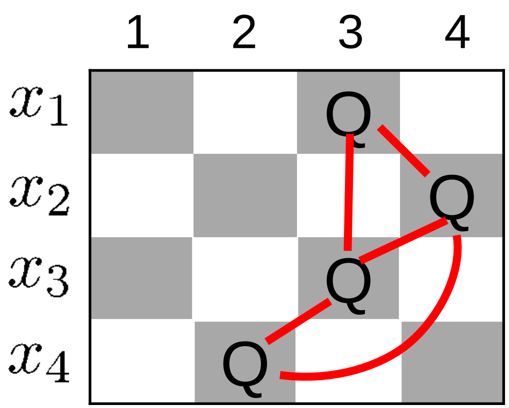
function MIN-CONFLICTS(*csp, max_steps*) **return** solution or failure
inputs: *csp*, a constraint satisfaction problem
max_steps, the number of steps allowed before giving up

current $\leftarrow$ a (random) initial complete assignment for *csp*
for *i =* 1 to *max_steps* **do**
if *current* is a solution for *csp* then return *current*
var $\leftarrow$ a randomly chosen, conflicted variable from VARIABLES[*csp*]
value $\leftarrow$ the value *v* for *var* that minimize CONFLICTS(*var,v,current,csp*)
set *var = value* in *current*
return *failure*

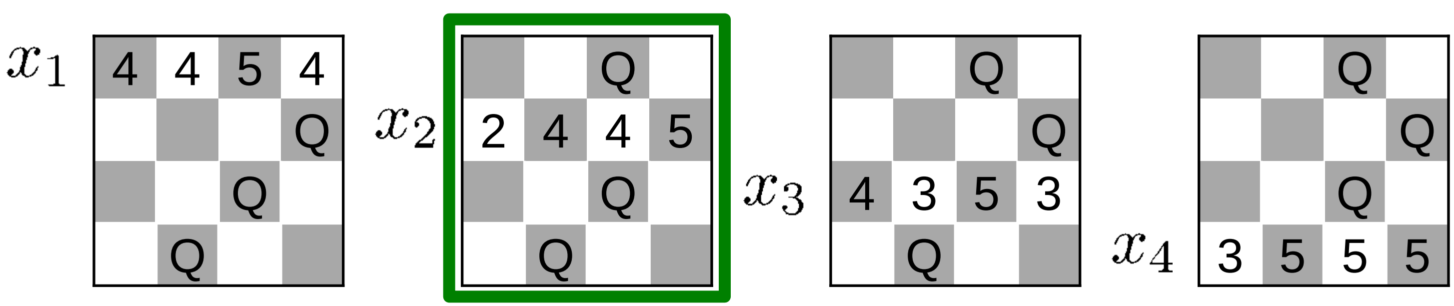
Number of conflicts

Note: here I check **all** neighbors & pick the best; typically in practice pick one at random

- Solving 4-queens with local search



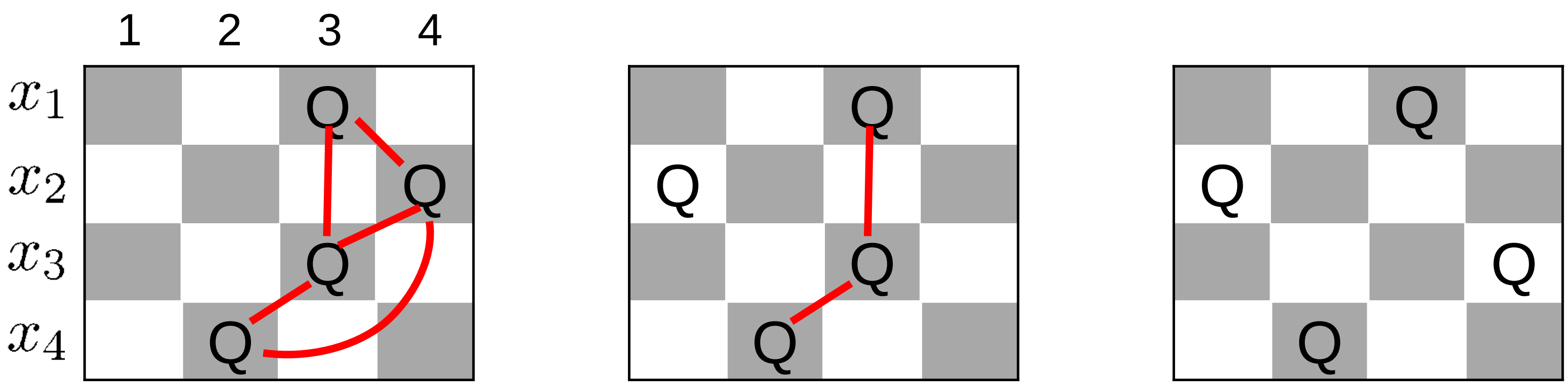
(5 conflicts)



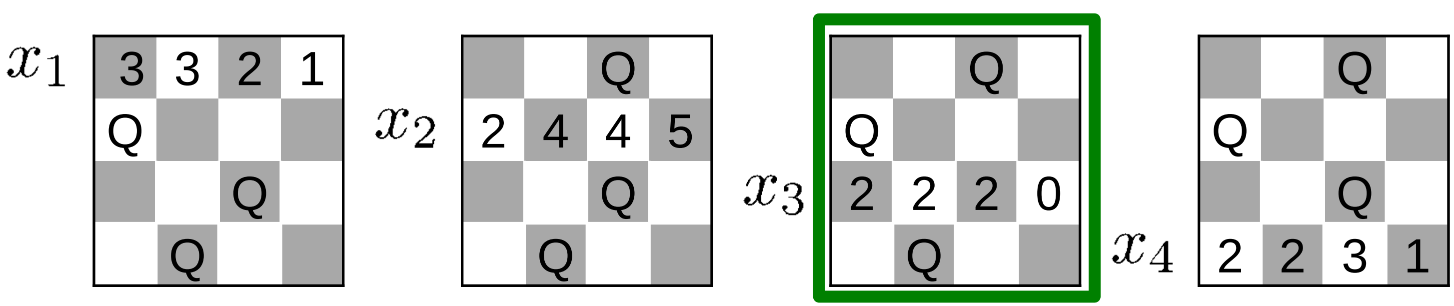
Number of conflicts

Note: here I check **all** neighbors & pick the best; typically in practice pick one at random

- Solving 4-queens with local search



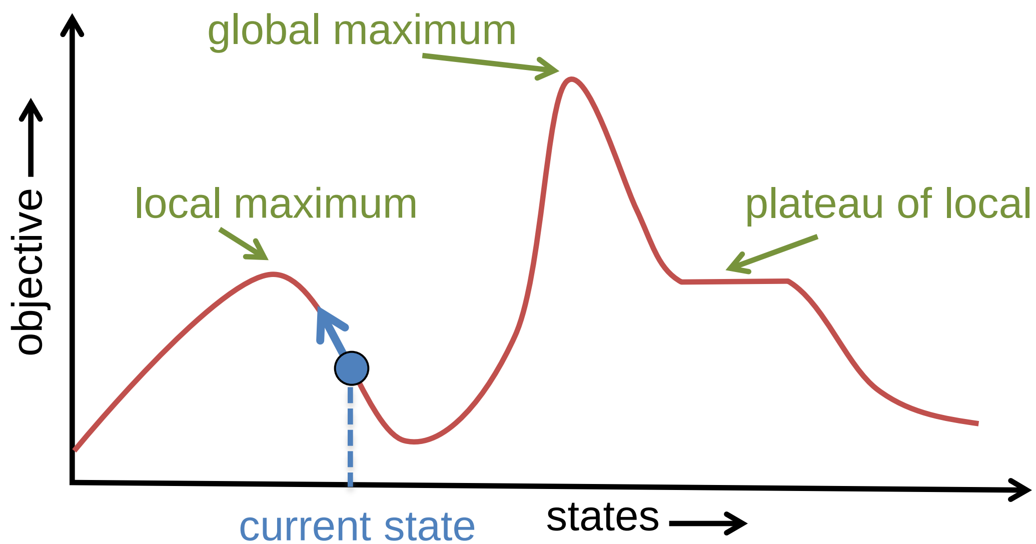
(5 conflicts) (2 conflicts) (0 conflicts)



Local optima

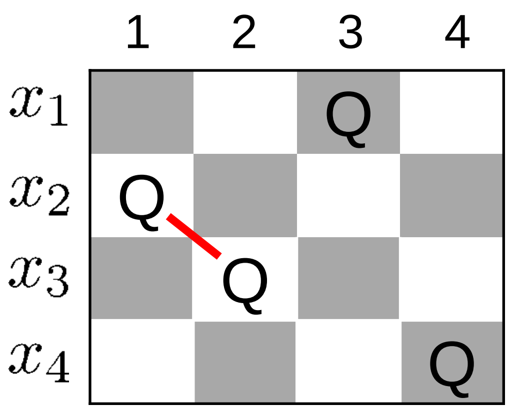
- Local search may get stuck at local optima
  - Locations where no neighboring value is better
  - Success depends on initialization quality & basins of attraction
- Can use multiple initializations to improve:
  - Re-initialize randomly (“repeated” local search)
  - Re-initialize by perturbing last optimum (“iterated” local search)
- Can also add sideways & random moves (e.g., WalkSAT)

(R&N Fig 7.18)



Local optimum example

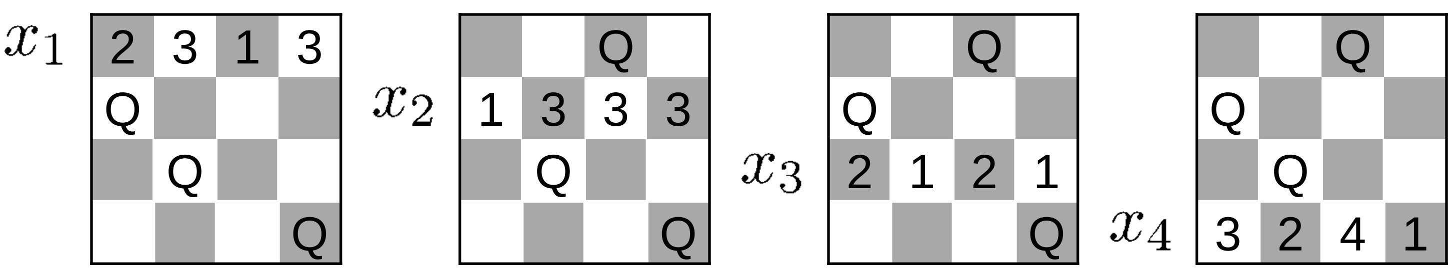
- Solving 4-queens with local search



(1 conflict)

"Plateau" example:

no single move can decrease # of conflicts



Comparison of CSP algorithms

Evaluate methods on a number of problems

Problem	Backtracking	BT+MRV	Forward Checking	FC+MRV	Min-Conflicts
USA	(> 1,000K)	(> 1,000K)	2K	60	64
n-Queens	(> 40,000K)	13,500K	(> 40,000K)	817K	4K
Zebra	3,859K	1K	35K	0.5K	2K
Random 1	415K	3K	26K	2K	
Random 2	942K	27K	77K	15K	

Median number of consistency checks over 5 runs to solve problem

Parentheses -> no solution found

USA: 4 coloring
n-queens: n = 2 to 50
Zebra: see exercise 6.7 (3rd ed.); exercise 5.13 (2nd ed.)

Advantages of local search

- Local search can be particularly useful in an online setting
  - Airline schedule example
    - E.g., mechanical problems require than 1 plane is taken out of service
    - Can locally search for another “close” solution in state-space
    - Much better (and faster) in practice than finding an entirely new schedule

- Runtime of min-conflicts is roughly independent of problem size.
  - Can solve the millions-queen problem in roughly 50 steps.
  - Why?
    - n-queens is easy for local search because of the relatively high density of solutions in state-space

Hardness of CSPs

- $x_1 \ldots x_n$ discrete, domain size d: $O(d^n)$ configurations
- "SAT": Boolean satisfiability: d=2
  - One of the first known NP-complete problems
- "3-SAT"
  - Conjunctive normal form (CNF)
  - At most 3 variables in each clause:

    $$(x_1 \vee \neg x_7 \vee x_{12}) \wedge (\neg x_3 \vee x_2 \vee x_7) \wedge \ldots$$

  - Still NP-complete

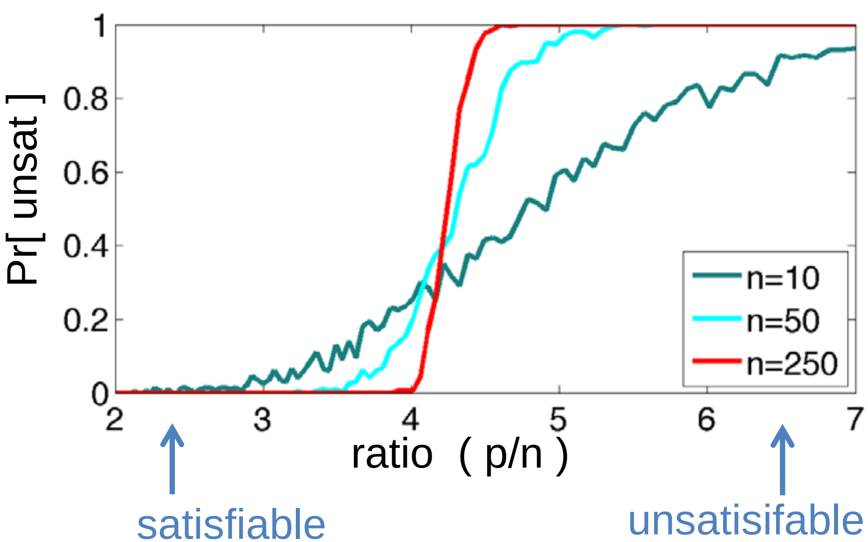
CNF clause: rule out one configuration

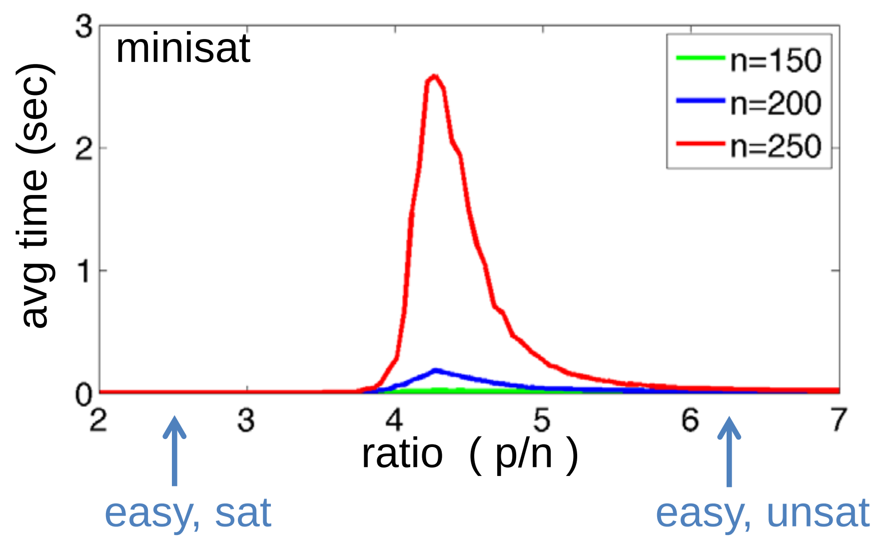
- How hard are "typical" problems?

Hardness of random CSPs

- Random 3-SAT problems:
  - n variables, p clauses in CNF: $(x_1 \vee \neg x_7 \vee x_{12}) \wedge (\neg x_3 \vee x_2 \vee x_7) \wedge \ldots$
  - Choose any 3 variables, signs uniformly at random
  - What's the probability there is **no** solution to the CSP?

  - Phase transition at (p/n) ¼ 4.25
  - "Hard" instances fall in a very narrow regime around this point!

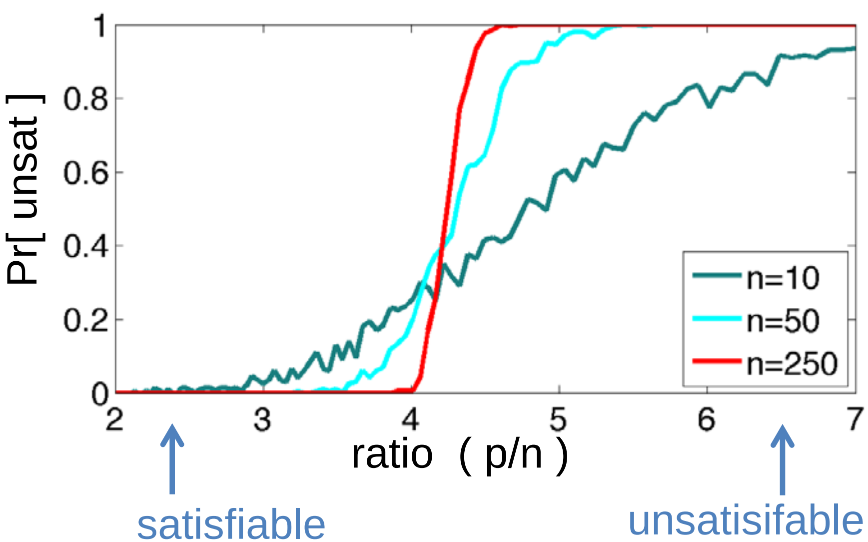


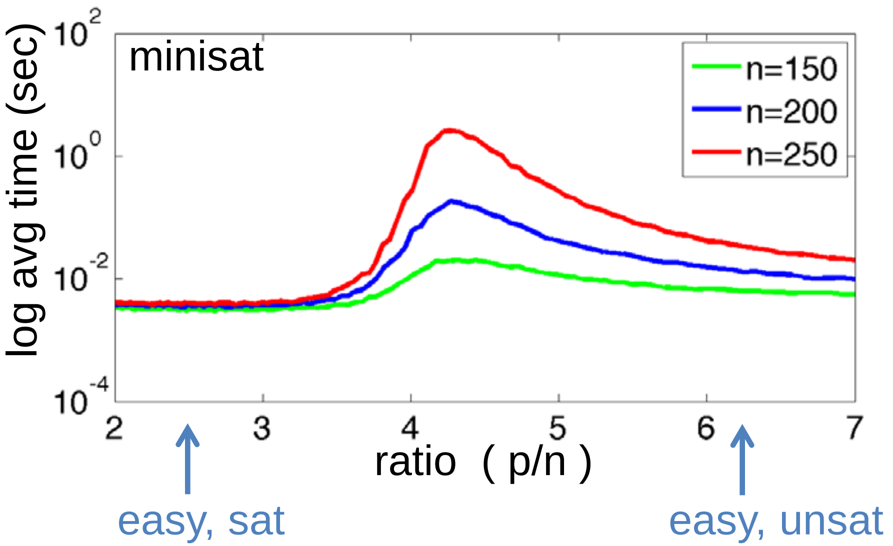


Hardness of random CSPs

- Random 3-SAT problems:
  - n variables, p clauses in CNF: $(x_1 \vee \neg x_7 \vee x_{12}) \wedge (\neg x_3 \vee x_2 \vee x_7) \wedge \ldots$
  - Choose any 3 variables, signs uniformly at random
  - What's the probability there is **no** solution to the CSP?

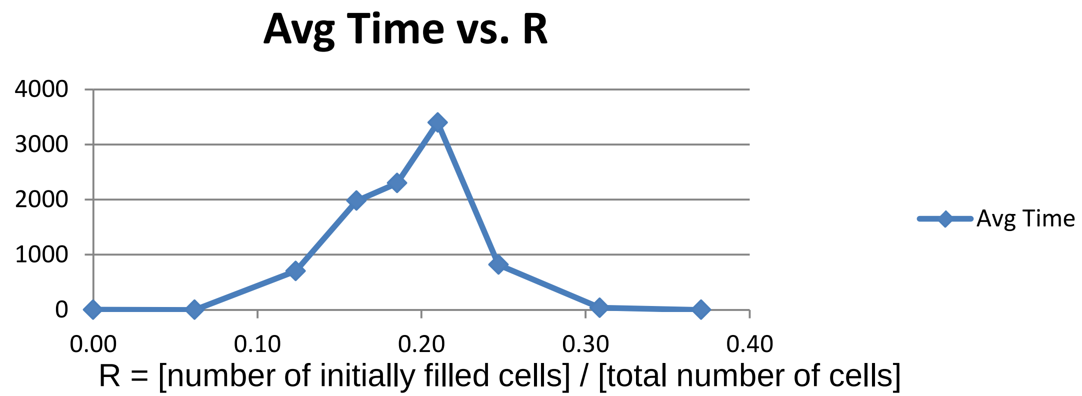
  - Phase transition at (p/n) ¼ 4.25
  - "Hard" instances fall in a very narrow regime around this point!

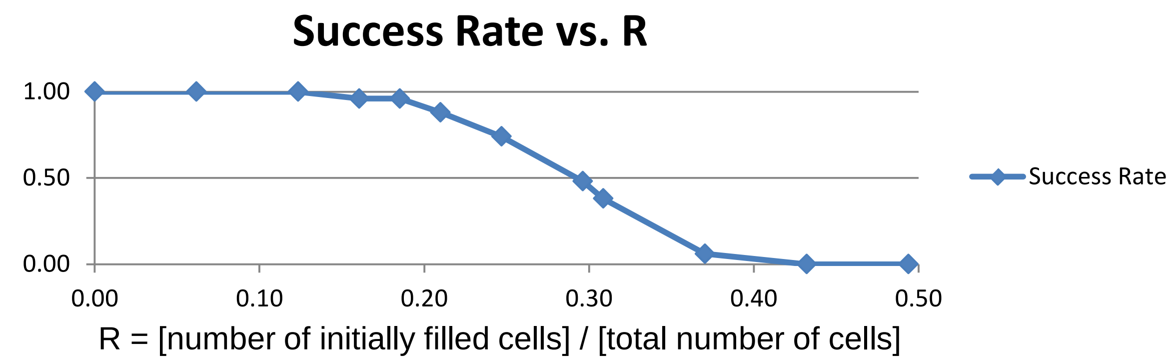




Ex: Sudoku

Backtracking search
+ forward checking

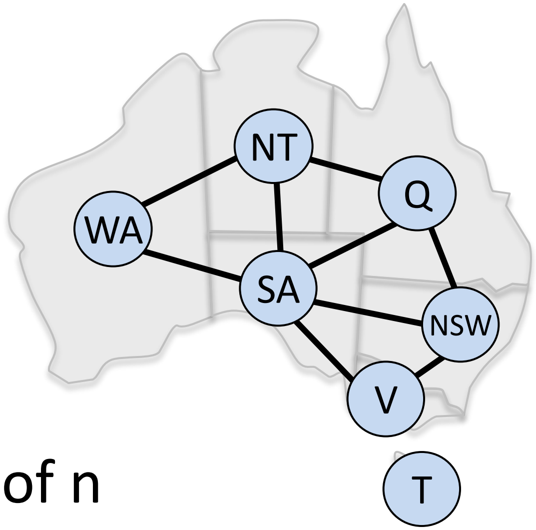




- R = [number of initially filled cells] / [total number of cells]
- Success Rate = P(random puzzle is solvable)
- [total number of cells] = 9x9 = 81
- [number of initially filled cells] = variable

Graph structure and complexity

- Disconnected subproblems
  - Configuration of one subproblem cannot affect the other: independent!
  - Exploit: solve independently

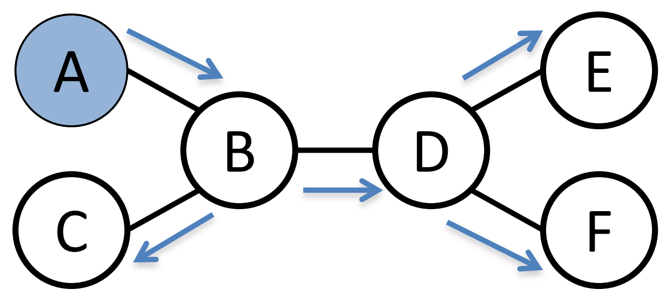


- Suppose each subproblem has c variables out of n
  - Worse case cost: $O(n/c\ d^c)$
  - Compare to $O(d^n)$, exponential in n
  - Ex: n=80, c=20, d=2)
    - 2^{80} = 4 billion years at 1 million nodes per second
    - $4 * 2^{20}$ = 0.4 seconds at 1 million nodes per second

Tree-structured CSPs

- Theorem: If a constraint graph has no cycles, then the CSP can be solved in O(n d^2) time.
  - Compare to general CSP: worst case O(d^n)

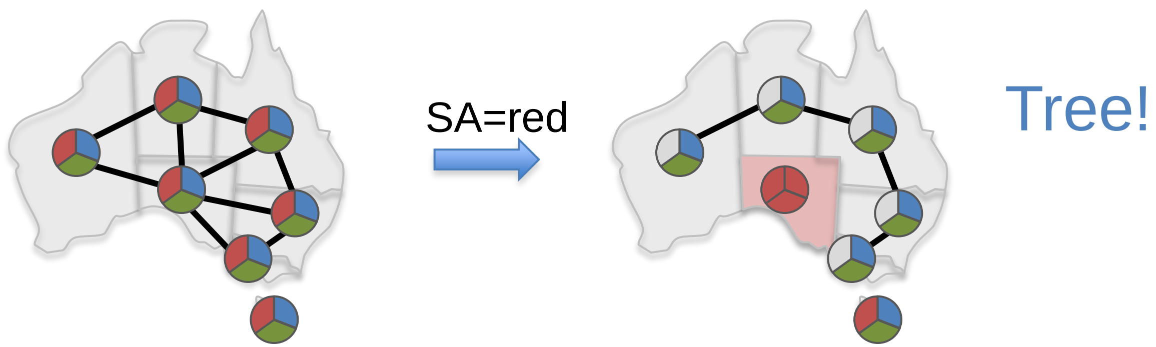
- Method: directed arc consistency (= dynamic programming)
  - Select a root (e.g., A) & do arc consistency from leaves to root:
  - D $\rightarrow$ F: remove values for D not consistent with any value for F, etc.)
  - D $\rightarrow$ E, B $\rightarrow$ D, … etc



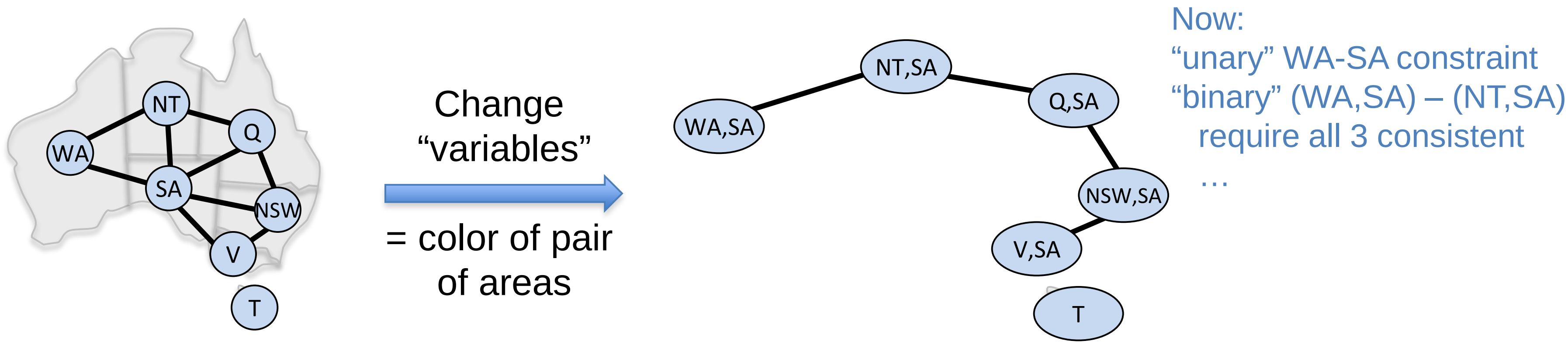
  - Select a value for A
  - There must be a value for B that is compatible; select it
  - There must be values for C, and for D, compatible with B's; select them
  - There must be values for E, F compatible with D's; select them.
  - You've found a consistent solution!

Exploiting structure

- How can we use efficiency of trees?
- Cutset conditioning
  - Exploit easy-to-solve problems during search



- Tree decomposition
  - Convert non-tree problems into (harder) trees



Summary

- CSPs
  - 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, one variable assigned per node
- Heuristics: variable order & value selection heuristics help a lot
- Constraint propagation
  - does additional work to constrain values and detect inconsistencies
  - Works effectively when combined with heuristics
- Iterative min-conflicts is often effective in practice.
- Graph structure of CSPs determines problem complexity
  - e.g., tree structured CSPs can be solved in linear time.