Constraint Propagation

(Where a better exploitation of the constraints further reduces the need to make decisions)

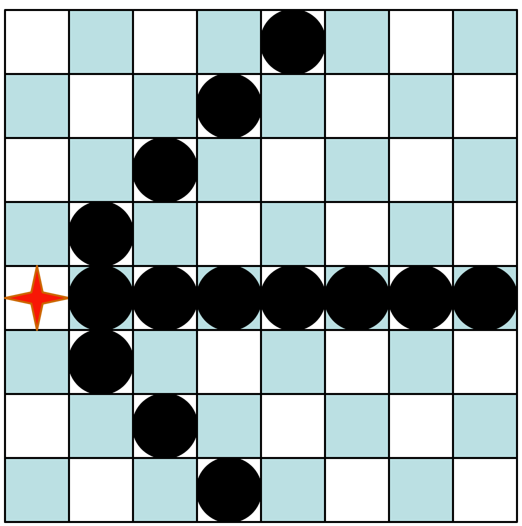
Constraint Propagation …

… is the process of determining how the constraints and the possible values of one variable affect the possible values of other variables

It is an important form of "least-commitment" reasoning

Forward checking is only on simple form of constraint propagation

When a pair (X←v) is added to assignment A do:
    For each variable Y not in A do:
        For every constraint C relating Y to variables in A do:
    Remove all values from Y's domain that do not satisfy C



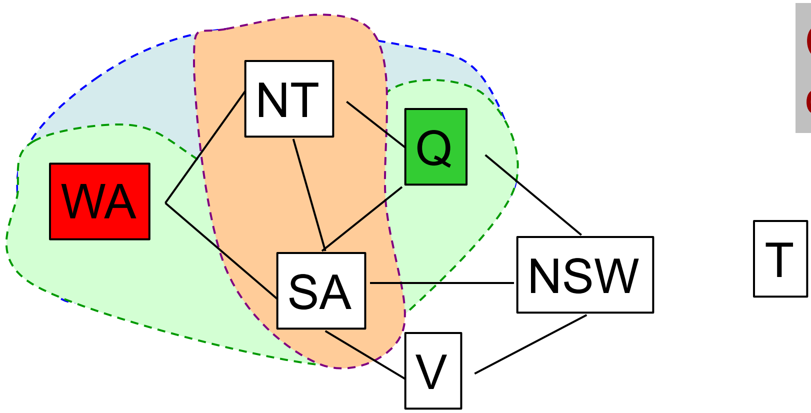
- n = number of variables
- d = size of initial domains
- s = maximum number of constraints involving a given variable (s ≤ n-1)
- Forward checking takes O(nsd) time

Forward Checking in Map Coloring

Empty set: the current assignment
{(WA $\leftarrow$ R), (Q $\leftarrow$ G), (V $\leftarrow$ B)}
does not lead to a solution

WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	~~R~~GB	RGB	RGB	RGB	~~R~~GB	RGB
R	~~G~~B	G	R~~G~~B	RGB	~~G~~B	RGB
R	B	G	R~~B~~	B	~~B~~	RGB

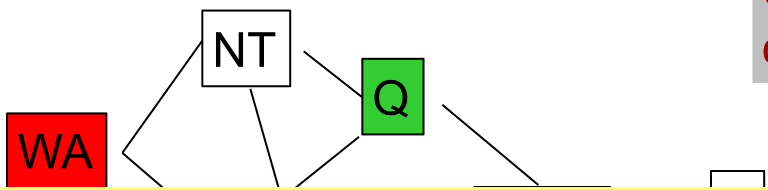
Forward Checking in Map Coloring



Contradiction that forward checking did not detect

WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	~~R~~GB	RGB	RGB	RGB	~~R~~GB	RGB
R	~~G~~B	G	R~~G~~B	RGB	~~G~~B	RGB
R	B	G	R~~B~~	B	~~B~~	RGB

Forward Checking in Map Coloring



Contradiction that forward checking did not detect

Detecting this contradiction requires a more powerful constraint propagation technique

WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	~~R~~GB	RGB	RGB	RGB	~~R~~GB	RGB
R	~~G~~B	G	R~~G~~B	RGB	~~G~~B	RGB
R	B	G	R~~B~~	B	~~B~~	RGB

Constraint Propagation for Binary Constraints

REMOVE-VALUES(X,Y) removes every value of Y that is incompatible with the values of X

REMOVE-VALUES(X,Y)

1. removed ← false
2. For every value v in the domain of Y do
   - If there is no value u in the domain of X such that the constraint on (X,Y) is satisfied then
     1. Remove v from Y's domain
     2. removed ← true

- Return removed

Constraint Propagation for Binary Constraints

AC3

1. Initialize queue Q with all variables (not yet instantiated)
2. While Q $\neq \emptyset$ do
   a. X $\leftarrow$ Remove(Q)
   - For every (not yet instantiated) variable Y related to X by a (binary) constraint do
     1. If REMOVE-VALUES(X,Y) then
        a. If Y's domain = $\emptyset$ then exit
        1. Insert(Y,Q)

Complexity Analysis of AC3

- n = number of variables
- d = size of initial domains
- s = maximum number of constraints involving a given variable ($s \le n-1$)
- Each variable is inserted in Q up to d times
- REMOVE-VALUES takes $O(d^2)$ time
- AC3 takes $O(n \times d \times s \times d^2) = O(n \times s \times d^3)$ time
- Usually more expensive than forward checking

AC3

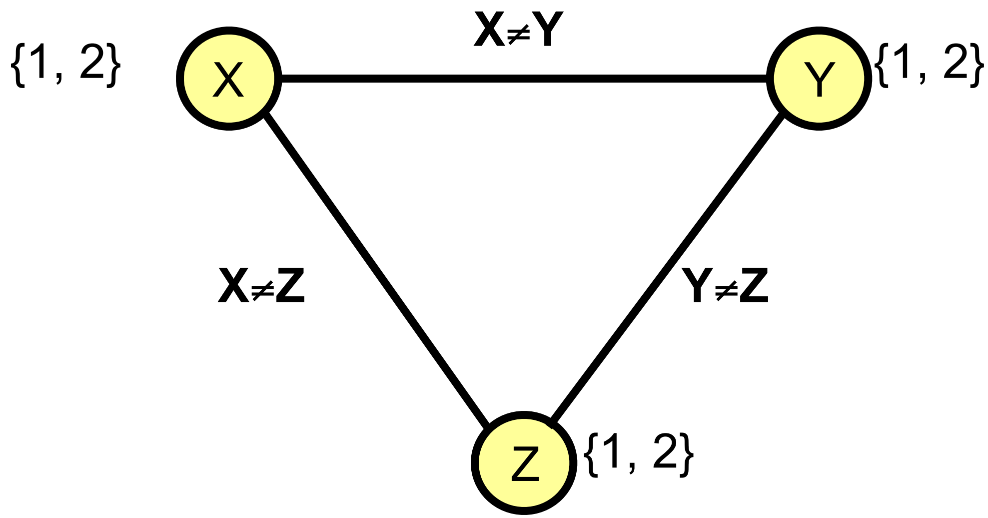
1. Initialize queue Q with all variables (not yet instantiated)
2. While $Q \neq \varnothing$ do
   a. X ← Remove(Q)
   - For every (not yet instantiated) variable Y related to X by a (binary) constraint do
     1. If REMOVE-VALUES(X,Y) then
        a. If Y's domain = ∅ then exit
        1. Insert(Y,Q)

REMOVE-VALUES(X,Y)

1. removed ← false
2. For every value v in the domain of Y do
   - If there is no value u in the domain of X such that the constraint on (x,y) is satisfied then
     a. Remove v from Y's domain
     b. removed ← true
3. Return removed

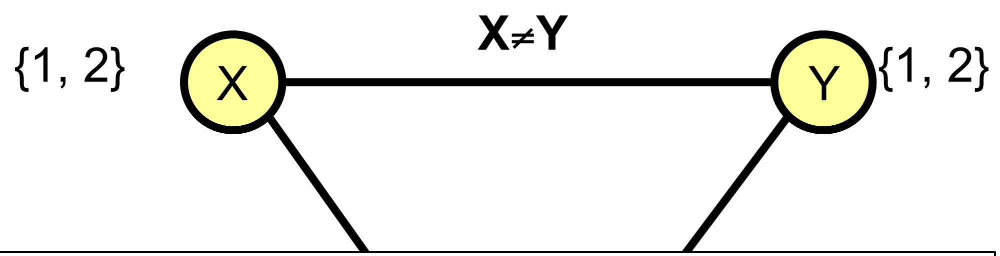
Is AC3 all that we need?

- No !!
- AC3 can't detect all contradictions among binary constraints



Is AC3 all that we need?

- No !!
- AC3 can't detect all contradictions among binary constraints



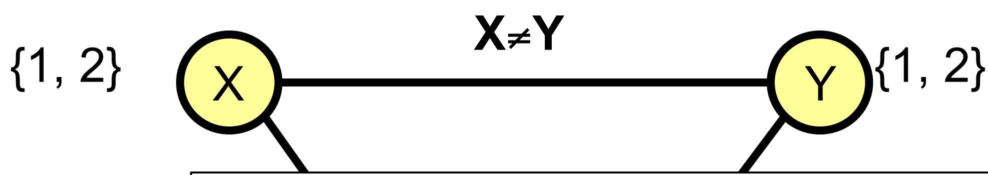
REMOVE-VALUES(X,Y)

1. removed ← false
2. For every value v in the domain of Y do
   - If there is no value u in the domain of X such that the constraint on (X,Y) is satisfied then
     1. Remove v from Y's domain
     2. removed ← true

- Return removed

Is AC3 all that we need?

- No !!
- AC3 can't detect all contradictions among binary constraints

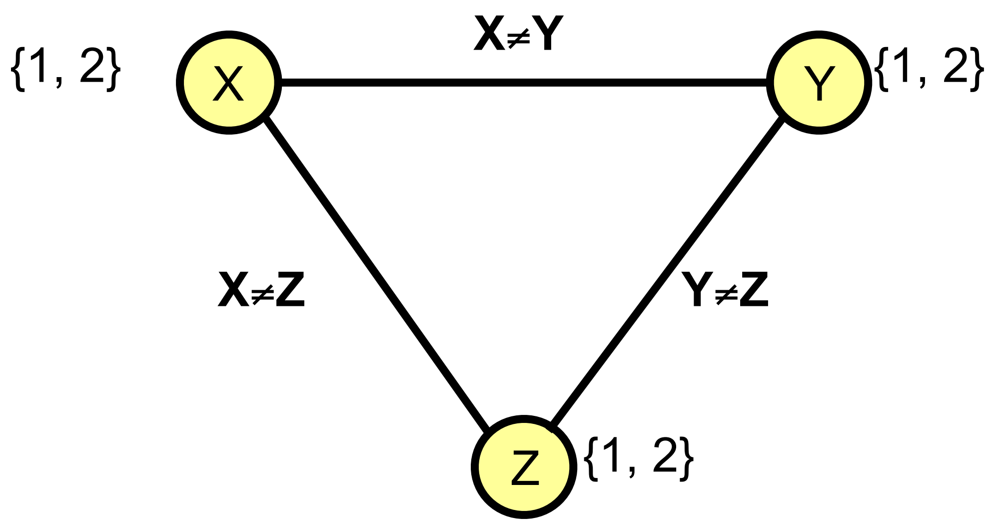


REMOVE-VALUES(X,Y
1. removed ← false
2. For every value v i
   - If there is no va
     the constraint c
     1. Remove v
     2. removed
- Return removed

REMOVE-VALUES(X,Y,Z)
1. removed ← false
2. For every value w in the domain of Z do
   - If there is no pair (u,v) of values in the domains of X and Y verifying the constraint on (X,Y) such that the constraints on (X,Z) and (Y,Z) are satisfied then
     - Remove w from Z's domain
     1. removed ← true
3. Return removed

Is AC3 all that we need?

- No !!
- AC3 can't detect all contradictions among binary constraints



- Not all constraints are binary

Tradeoff

Generalizing the constraint propagation algorithm increases its time complexity

→Tradeoff between time spent in backtracking search and time spent in constraint propagation

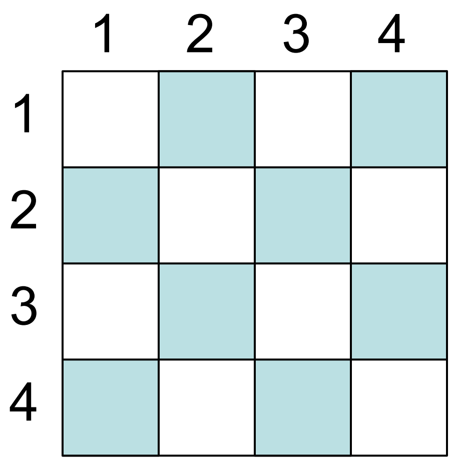
A good tradeoff when all or most constraints are binary is often to combine backtracking with forward checking and/or AC3 (with REMOVE-VALUES for two variables)

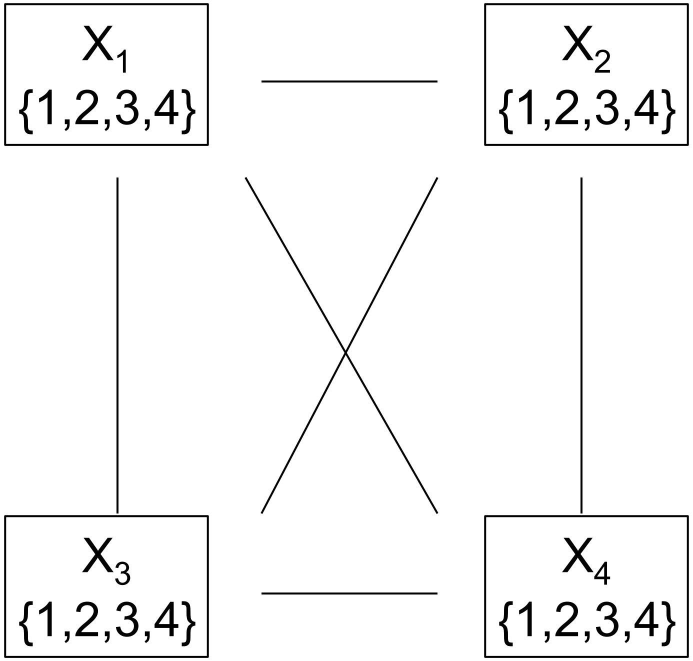
Modified Backtracking Algorithm with AC3

CSP-BACKTRACKING(A, var-domains)

1. If assignment A is complete then return A
2. Run AC3 and update var-domains accordingly
3. If a variable has an empty domain then return failure
4. X ← select a variable not in A
5. D ← select an ordering on the domain of X
6. For each value v in D do
    a. Add (X ← v) to A
    b. var-domains ← forward checking(var-domains, X, v, A)
    c. If no variable has an empty domain then
       (i) result ← CSP-BACKTRACKING(A, var-domains)
       (ii) If result ≠ failure then return result
    - Remove (X ← v) from A
7. Return failure

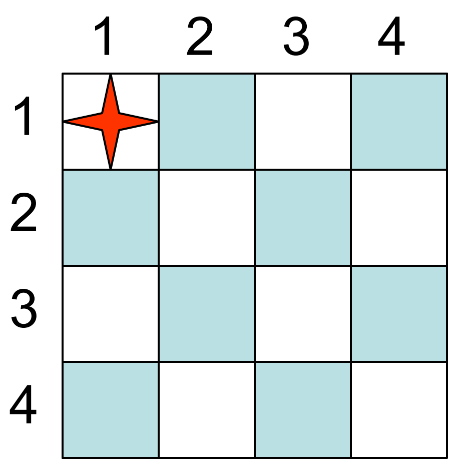
A Complete Example: 4-Queens Problem

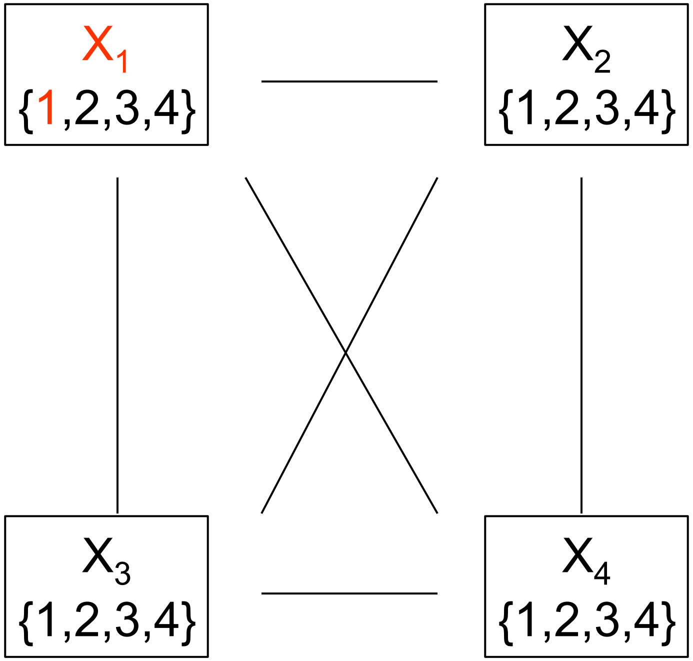




1) The modified backtracking algorithm starts by calling AC3, which removes no value

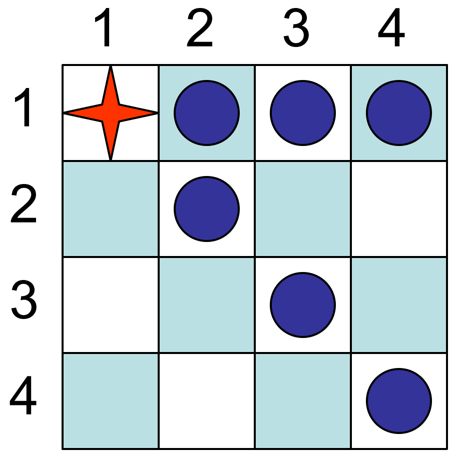
4-Queens Problem

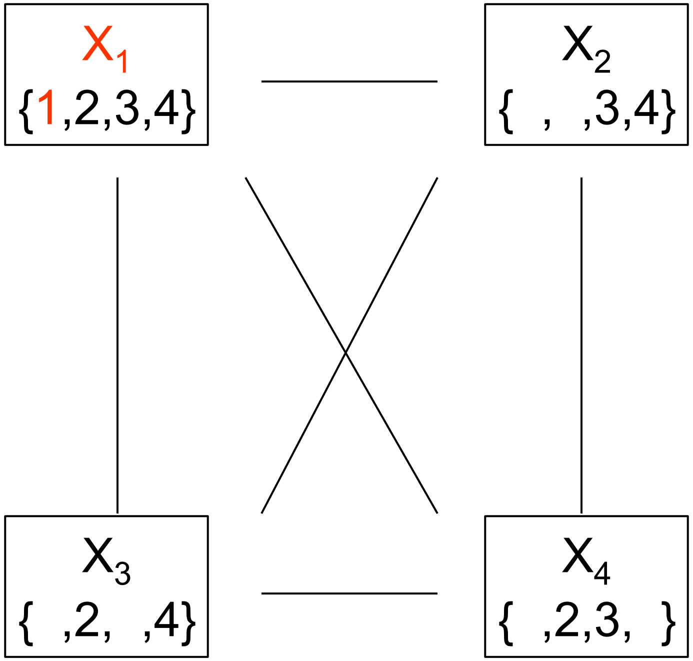




2) The backtracking algorithm then selects a variable and a value for this variable. No heuristic helps in this selection. X_1 and the value 1 are arbitrarily selected

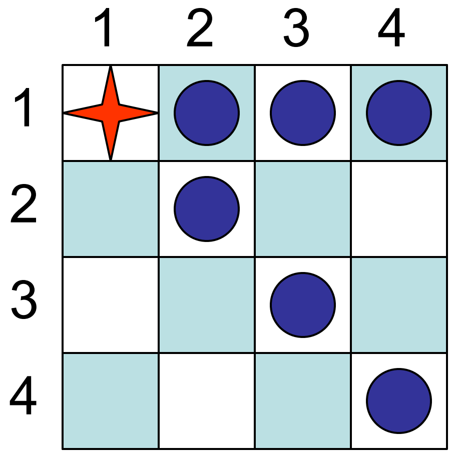
4-Queens Problem

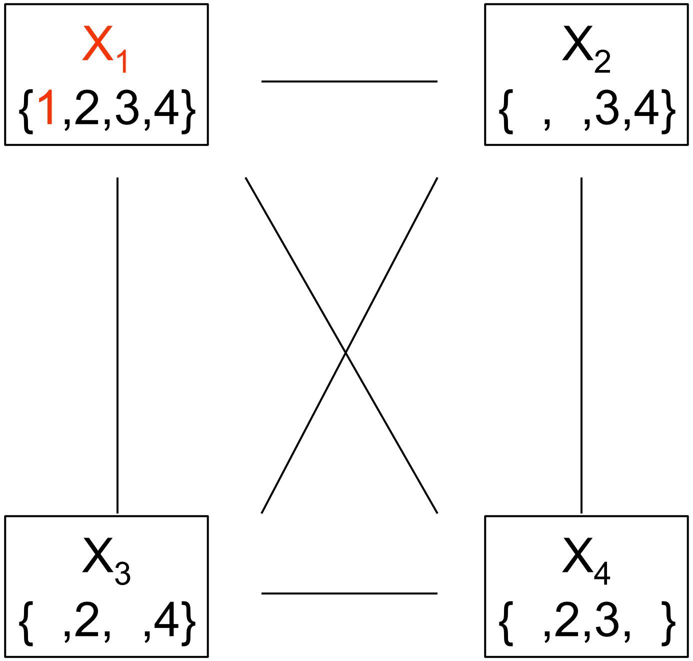




3) The algorithm performs forward checking, which eliminates 2 values in each other variable's domain

4-Queens Problem



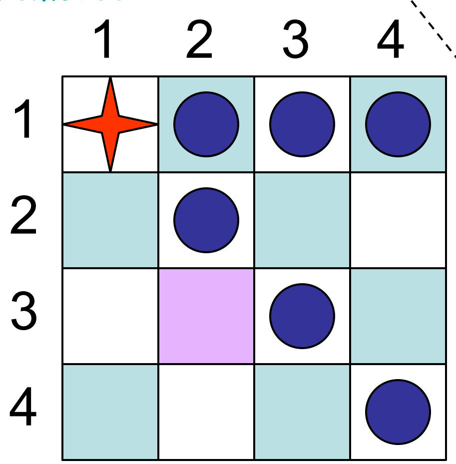


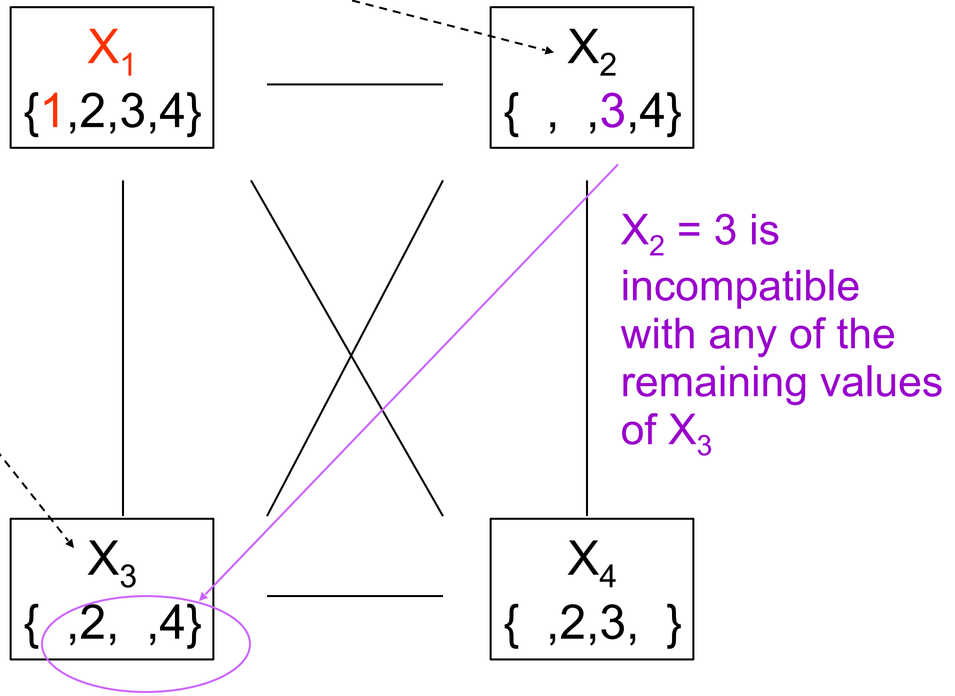
4) The algorithm calls AC3

4-Queens Problem

REMOVE-VALUES(X,Y)

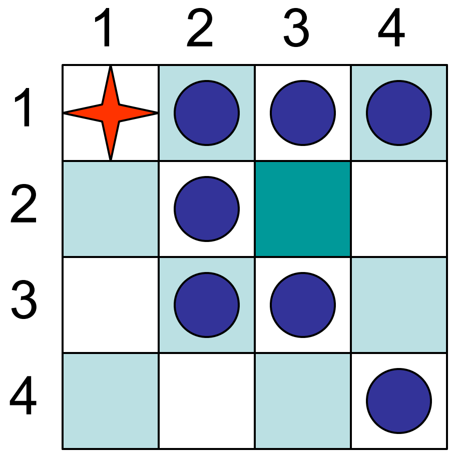
1. removed ← false
2. For every value v in the domain of Y do
   - If there is no value u in the domain of X such that the constraint on (x,y) is satisfied then
     a. Remove v from Y's domain
     b. removed ← true
3. Return removed

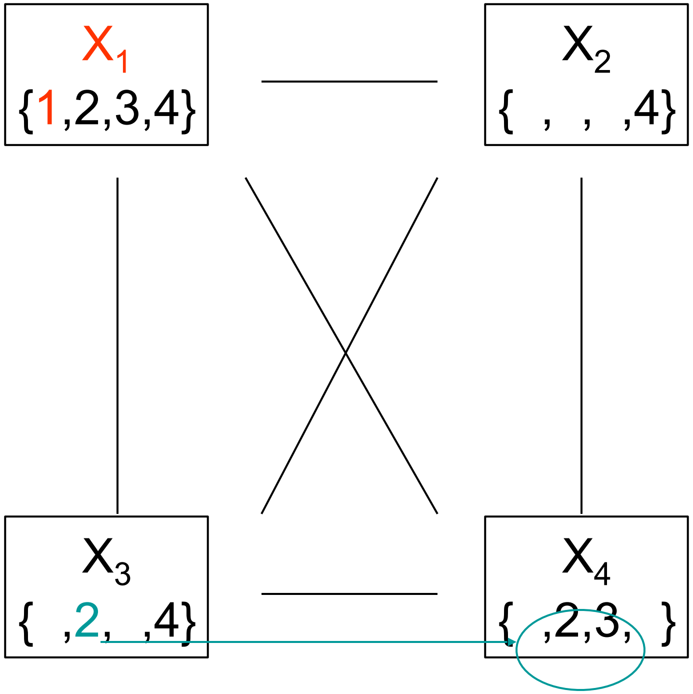




4) The algorithm calls AC3, which eliminates 3 from the domain of X_2

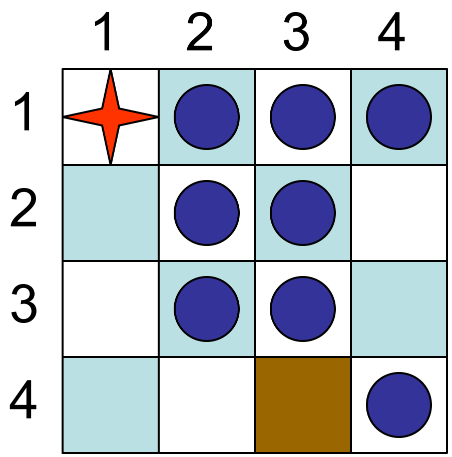
4-Queens Problem

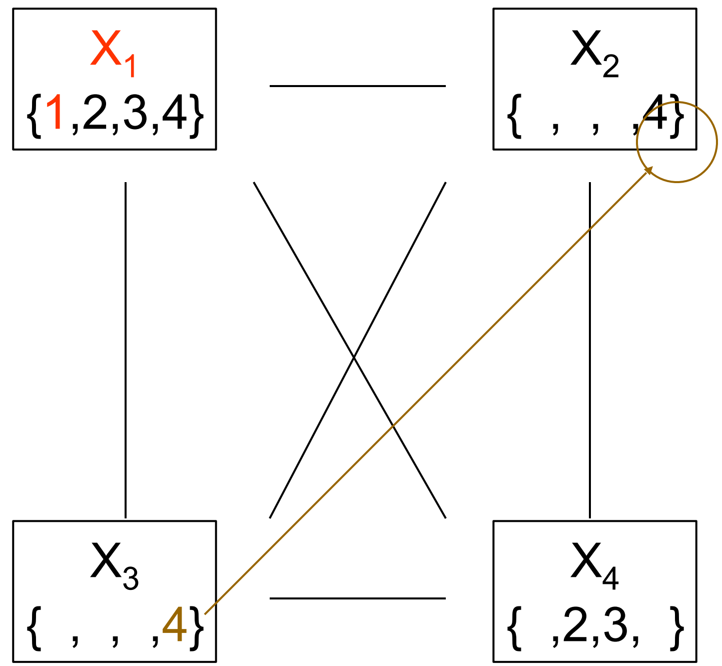




4) The algorithm calls AC3, which eliminates 3 from the domain of X_2, and 2 from the domain of X_3

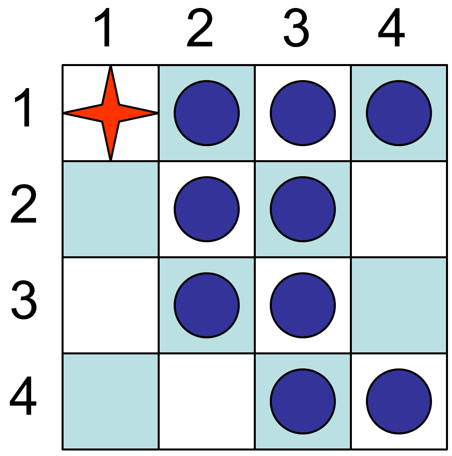
4-Queens Problem

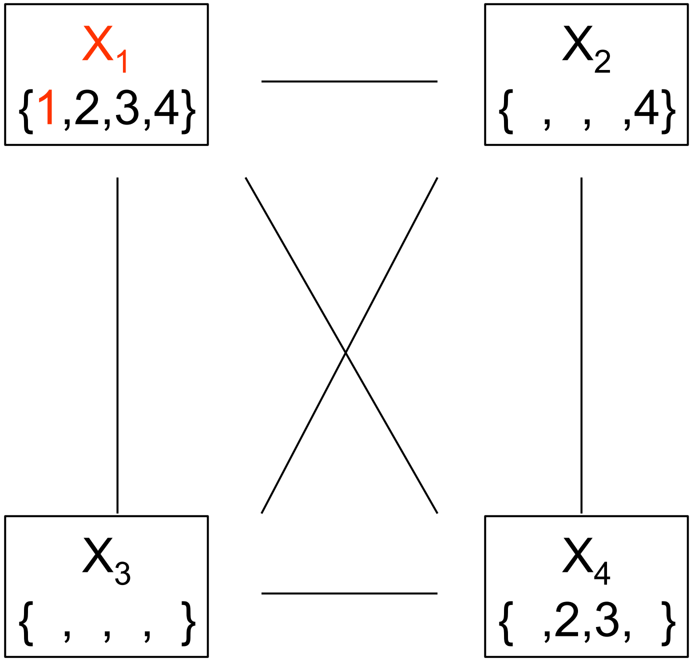




4) The algorithm calls AC3, which eliminates 3 from the domain of X_2, and 2 from the domain of X_3, and 4 from the domain of X_3

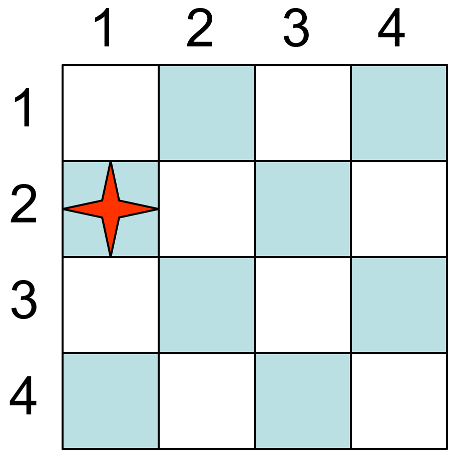
4-Queens Problem

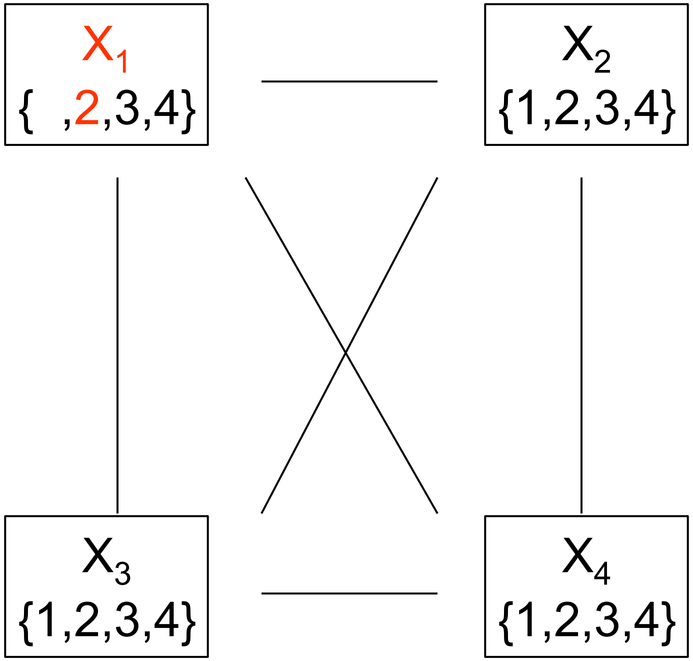




5) The domain of X_3 is **empty** → backtracking

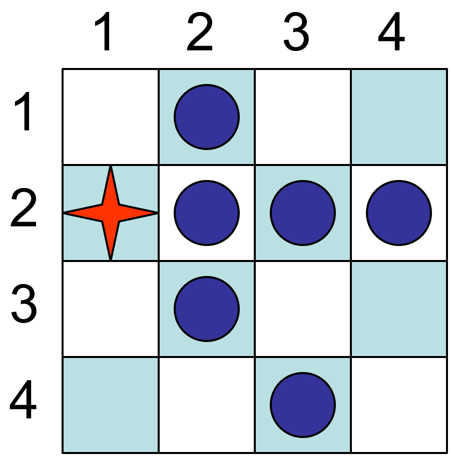
4-Queens Problem

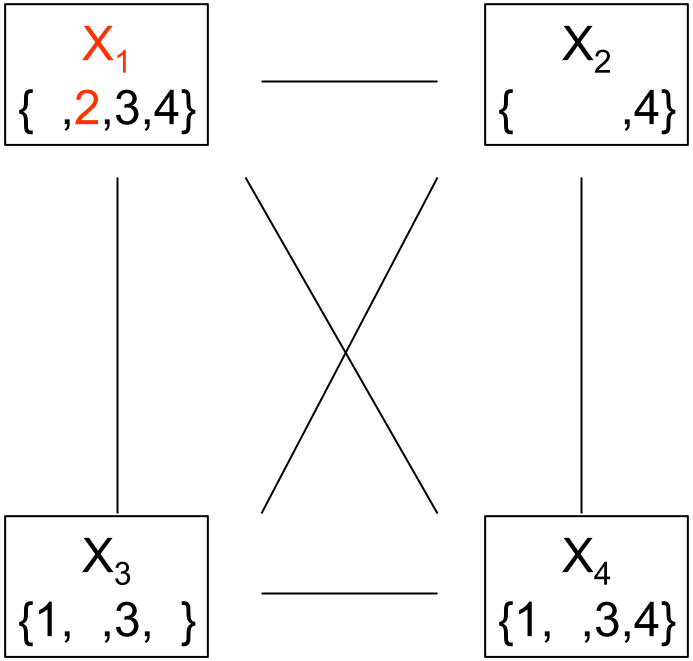




6) The algorithm removes 1 from X_1's domain and assign 2 to X_1

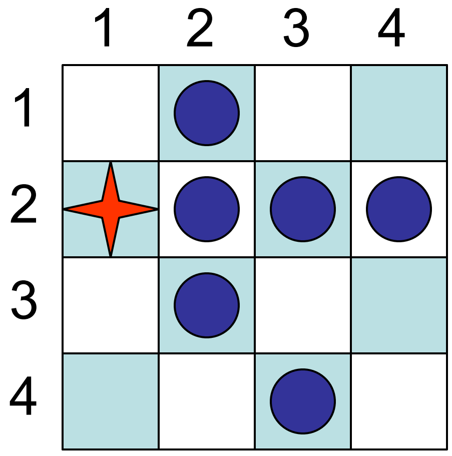
4-Queens Problem

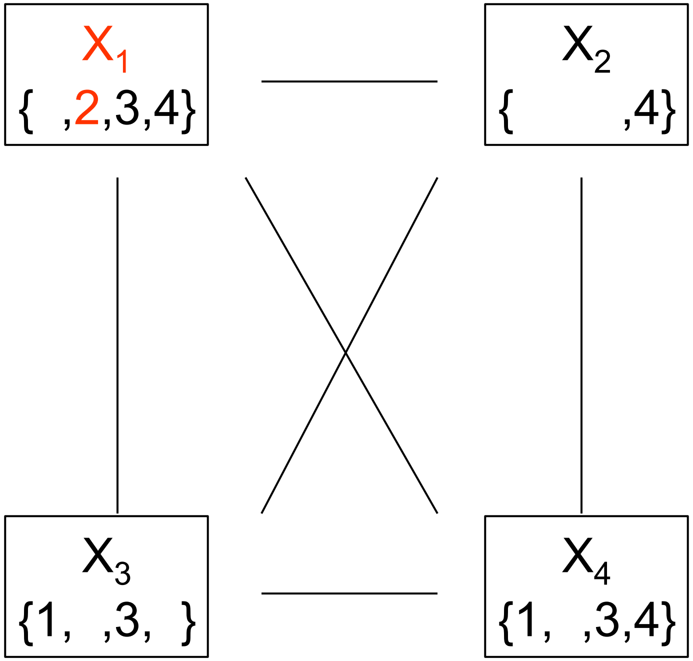




7) The algorithm performs forward checking

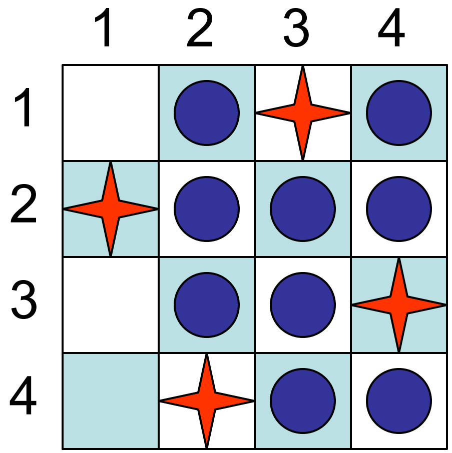
4-Queens Problem

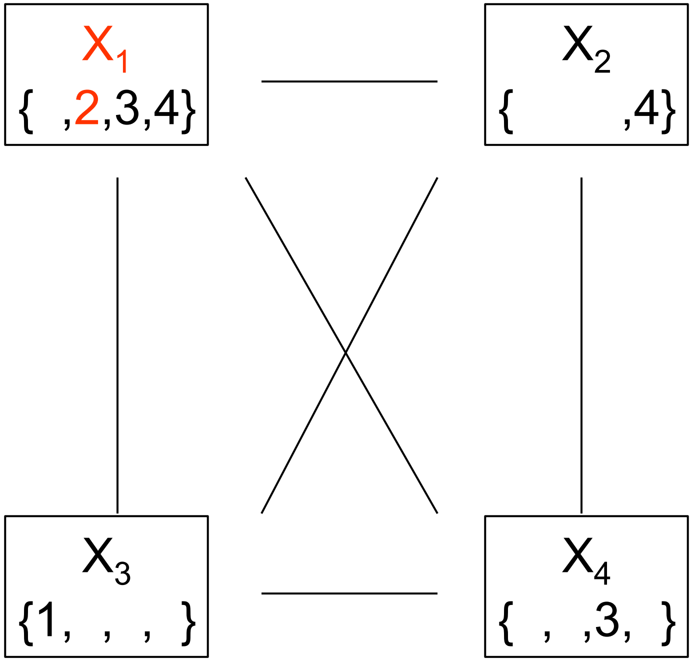




8) The algorithm calls AC3

4-Queens Problem





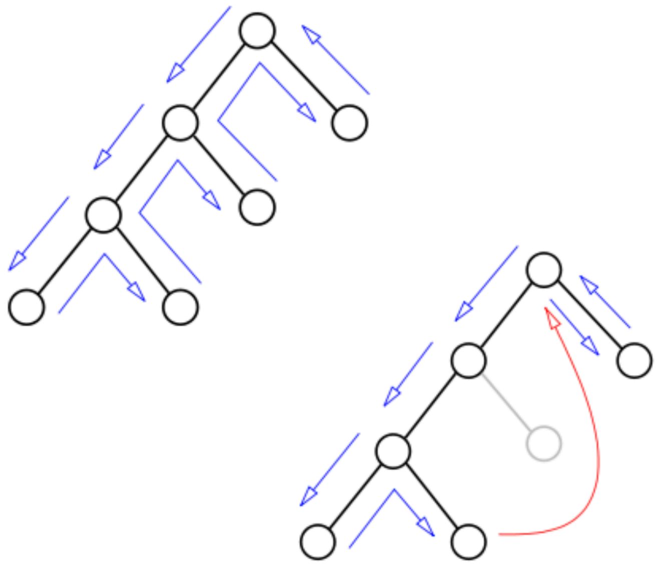
8) The algorithm calls AC3, which reduces the domains of X_3 and X_4 to a single value

Further Weaknesses of Backtracking

Trashing: Backtracking throws away the reason of the conflict

> Example: A,B,C,D,E::1..10, A>E
> BT tries all the assignments for B,C,D before finding that A≠1

Solution: **backjumping**
(= jump to the source of the failure)



Backjumping (BJ) explained

Let us find a safe jump:

algorithm could jump from x_{k+1} to whichever variable x_j is such that the current assignment to $x_1, \ldots, x_j$ cannot be extended to form a solution with any value of x_{k+1}.

Backjumping from **leaf nodes** (leaf dead-ends):

x_{k+1} are inconsistent with the current partial solution $x_1, \ldots, x_k = a_1, \ldots, a_k$

x_{k+1} is a leaf of the search tree

Safe jump:

the shortest prefix of $x_1, \ldots, x_k = a_1, \ldots, a_k$ inconsistent with $x_{k+1} = a_{k+1}$

$x_1 = a_1 \quad \dots \quad x_{k-1} = a_{k-1} \quad x_k = a_k \quad x_{k+1} = a_{k+1}$

$x_1 = a_1 \quad \dots \quad x_{k-1} = a_{k-1} \quad x_{k+1} = a_{k+1}$

...

$x_1 = a_1 \quad x_{k+1} = a_{k+1}$

Let us fin

algorithm cou … e current
assignment to … value of x_{k+1}.

Backjumping from **leaf nodes** (leaf dead-ends):

x_{k+1} are inconsistent with the current partial solution $x_1, \dots, x_k = a_1, \dots, a_k$

x_{k+1} is a leaf of the search tree

Safe jump:

the shortest prefix of $x_1, \dots, x_k = a_1, \dots, a_k$ inconsistent with $x_{k+1} = a_{k+1}$

Backjumping (BJ) explained

Let us find a safe jump:

algorithm could jump from x_{k+1} to whichever variable x_j is such that the current assignment to $x_1, \ldots, x_j$ cannot be extended to form a solution with any value of x_{k+1}.

Backjumping from **leaf nodes** (leaf dead-ends):

x_{k+1} are inconsistent with the current partial solution $x_1, \ldots, x_k = a_1, \ldots, a_k$

x_{k+1} is a leaf of the search tree

Safe jump:

the shortest prefix of $x_1, \ldots, x_k = a_1, \ldots, a_k$ inconsistent with $x_{k+1} = a_{k+1}$

Gashing backjumping: BJ from leaf nodes only

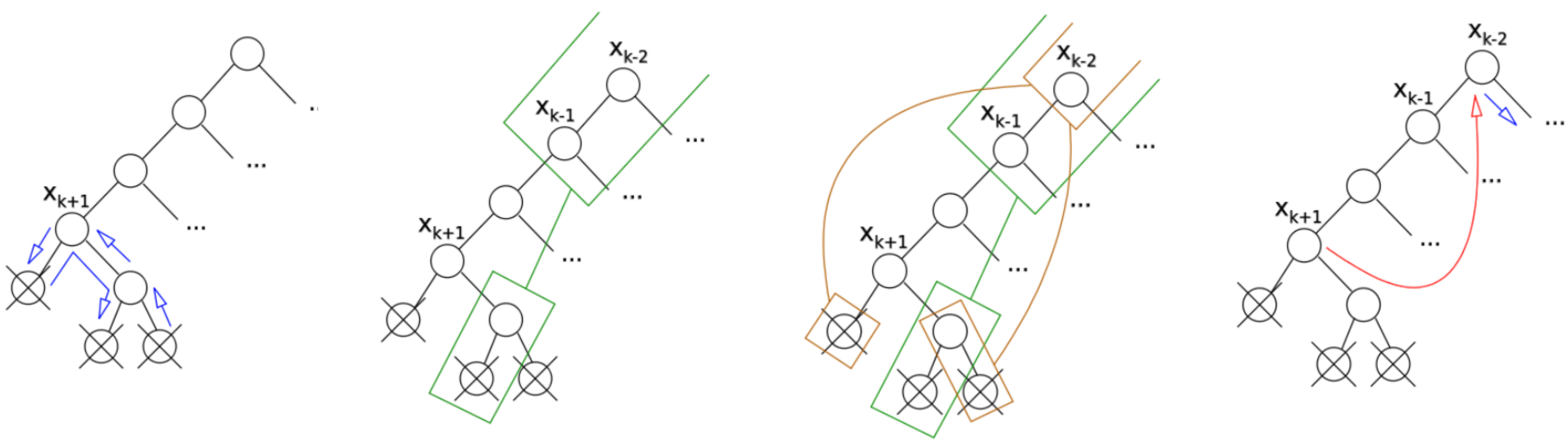
Backjumping (BJ) explained

Let us find a safe jump:

algorithm could jump from x_{k+1} to whichever variable x_j is such that the current assignment to $x_1, \ldots, x_j$ cannot be extended to form a solution with any value of x_{k+1}.

Backjumping from **internal nodes** (internal dead-ends):

the algorithm can backjump to a previous variable x_i provided that the current truth evaluation of $x_1, \ldots, x_i$ is inconsistent with all the truth evaluations of $x_{k+1}, x_{k+2}, \ldots$ in the leaf nodes that are descendants of the node x_{k+1}.

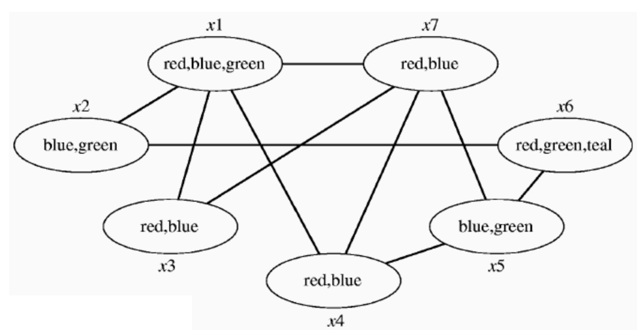


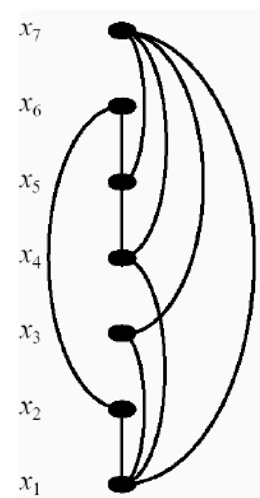
Graph Directed Backjumping

– driven by the structure of constraint network only (does not assume (dis)satisfaction of constraints)

– can do several jumps in a sequence

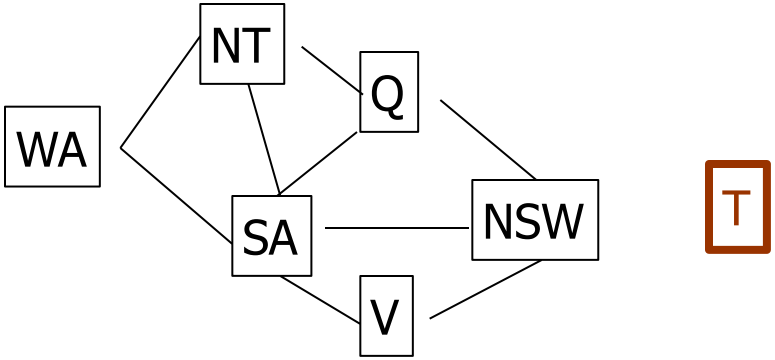
jump to a closest variable from the set of predecessors or the closest predecessor of all dead-ends visited on this backjump.





Exploiting the Structure of CSP

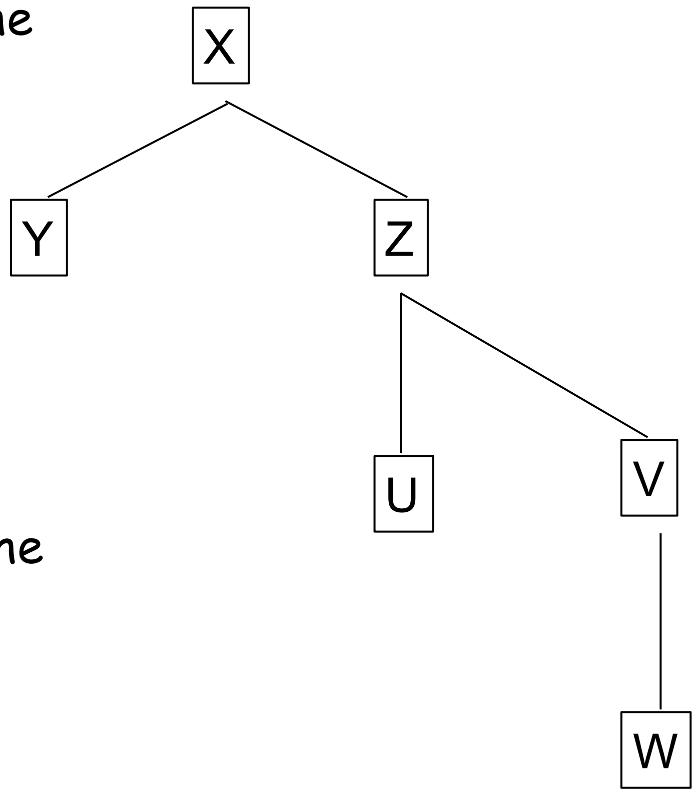
If the constraint graph contains several components, then solve one independent CSP per component



Exploiting the Structure of CSP

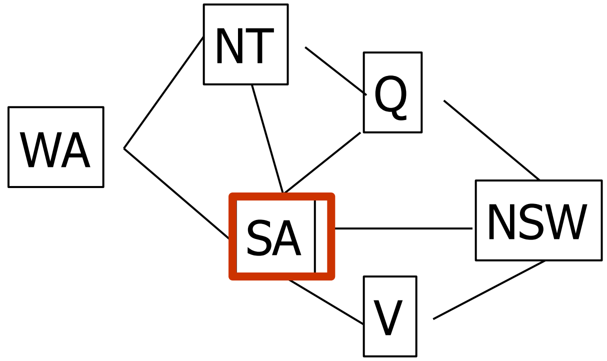
If the constraint graph is a tree, then :

1. Order the variables from the root to the leaves → $(X_1, X_2, \ldots, X_n)$
2. For $j = n, n-1, \ldots, 2$ call REMOVE-VALUES(X_j, X_i) where X_i is the parent of X_j
3. Assign any valid value to X_1
4. For $j = 2, \ldots, n$ do
   Assign any value to X_j consistent with the value assigned to its parent X_i



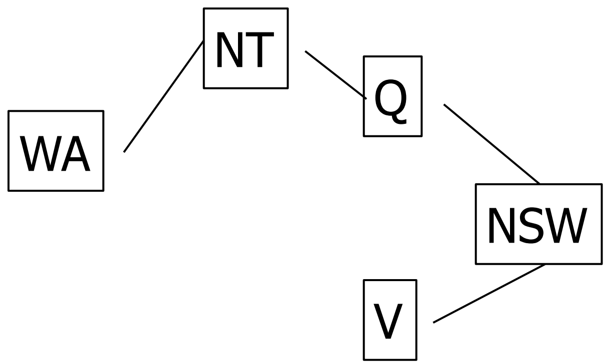
Exploiting the Structure of CSP

Whenever a variable is assigned a value by the backtracking algorithm, propagate this value and remove the variable from the constraint graph



Exploiting the Structure of CSP

Whenever a variable is assigned a value by the backtracking algorithm, propagate this value and remove the variable from the constraint graph



If the graph becomes a tree, then proceed as shown in previous slide