

Search for CSPs

Outline

I. Backtracking algorithm

II. Local search

III. Structure of CSP problems

I. Backtracking Search

- ♣ Constraint propagation often ends with partial solutions.
- ♣ Backtracking search can be employed to extend them to full solutions.

I. Backtracking Search

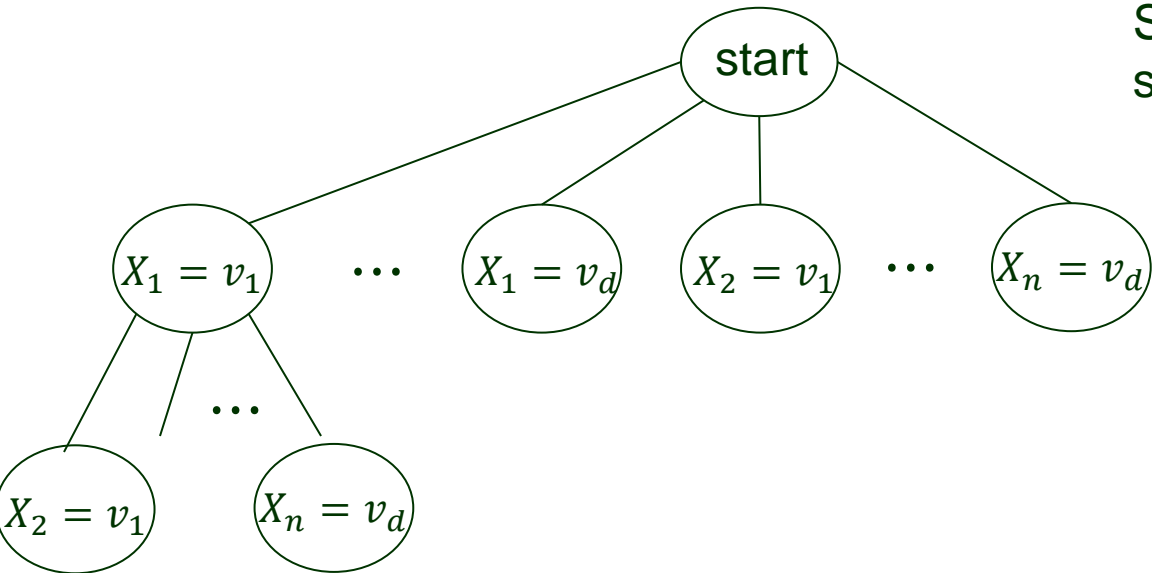
- ♣ Constraint propagation often ends with partial solutions.
- ♣ Backtracking search can be employed to extend them to full solutions.

Solve a CSP using depth-limited search.

n variables with
domain size d

I. Backtracking Search

- ♣ Constraint propagation often ends with partial solutions.
- ♣ Backtracking search can be employed to extend them to full solutions.

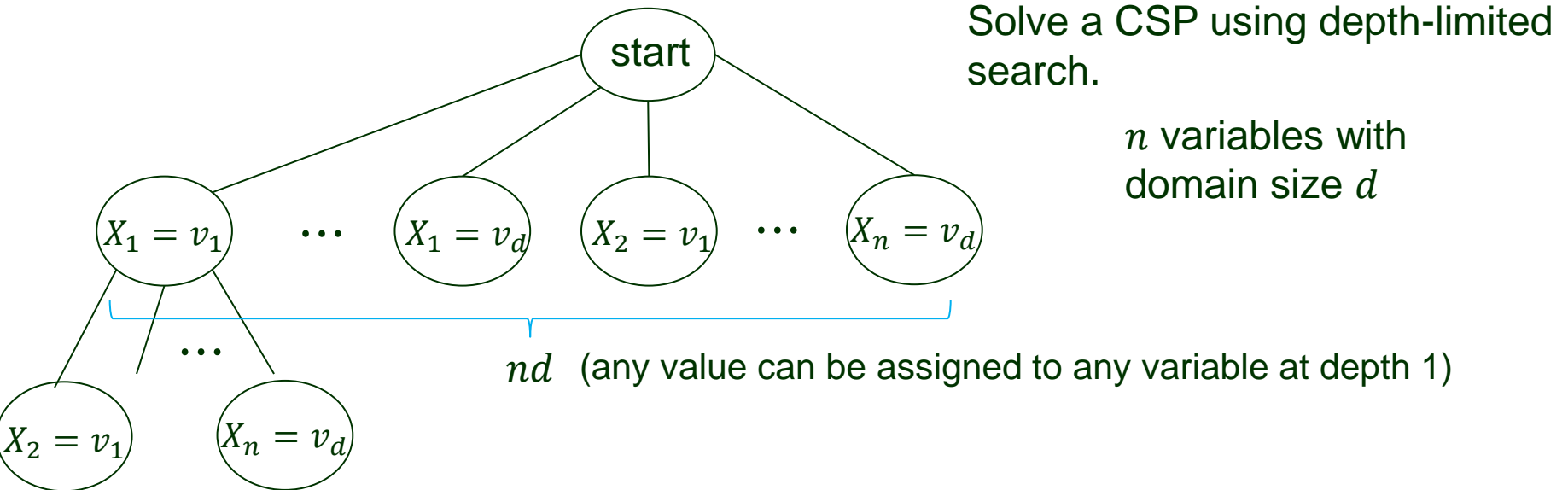


Solve a CSP using depth-limited search.

n variables with domain size d

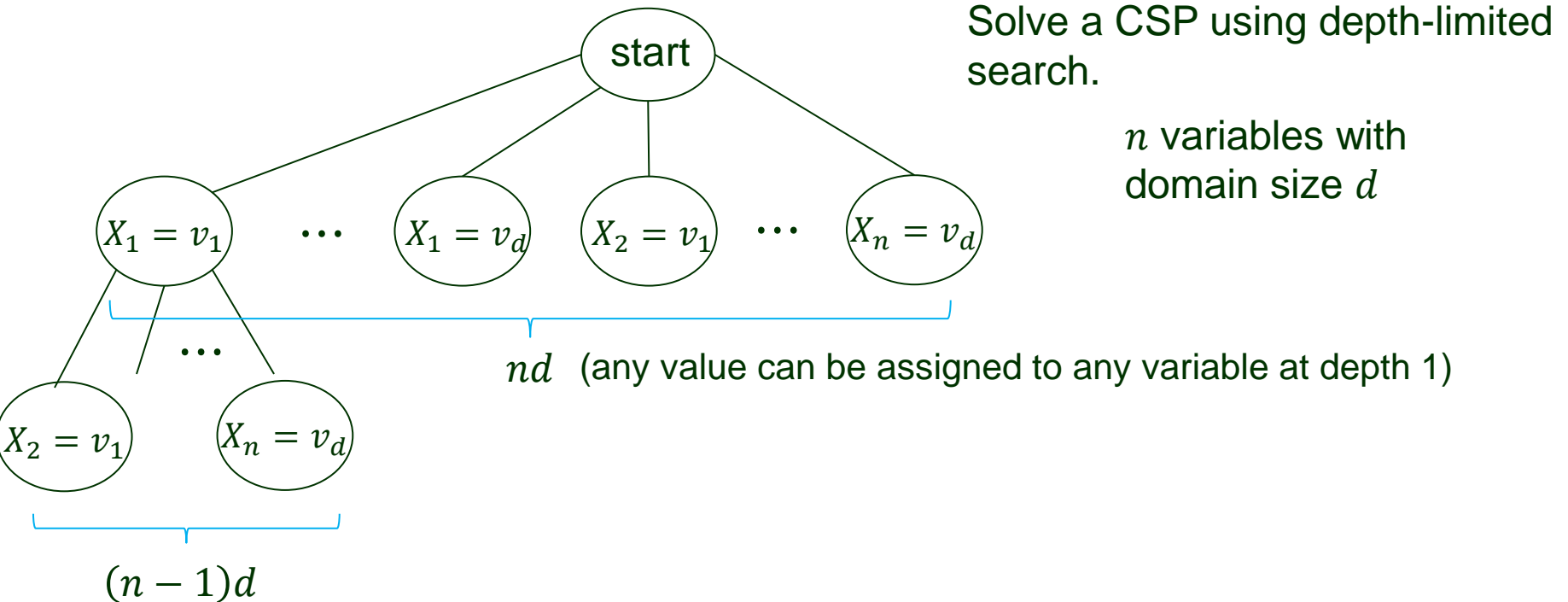
I. Backtracking Search

- ♣ Constraint propagation often ends with partial solutions.
- ♣ Backtracking search can be employed to extend them to full solutions.



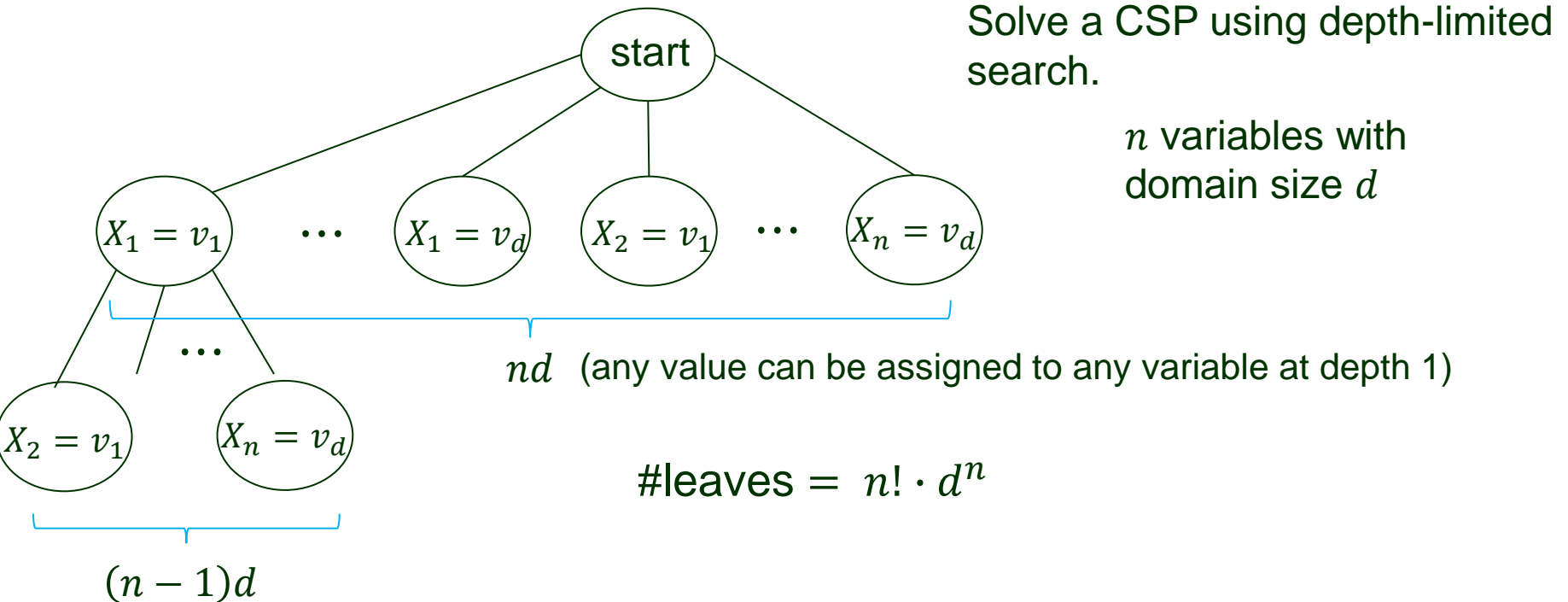
I. Backtracking Search

- ♣ Constraint propagation often ends with partial solutions.
- ♣ Backtracking search can be employed to extend them to full solutions.



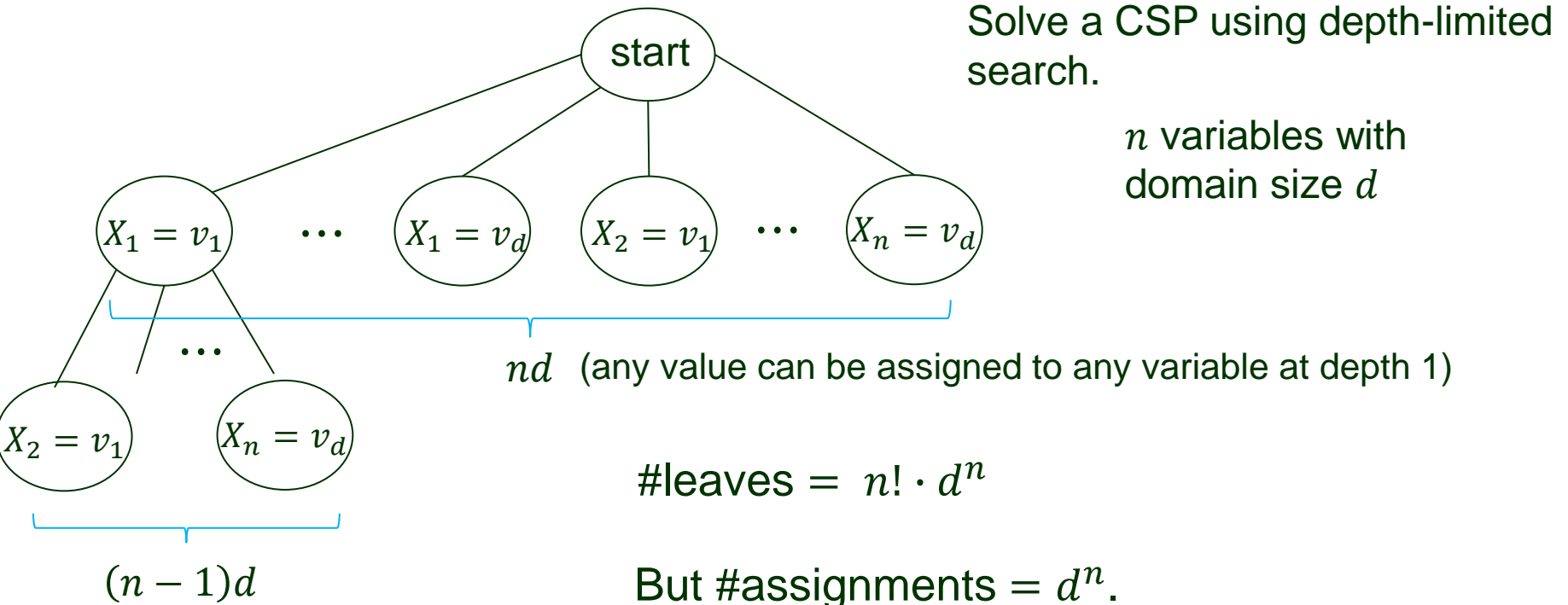
I. Backtracking Search

- ♣ Constraint propagation often ends with partial solutions.
- ♣ Backtracking search can be employed to extend them to full solutions.



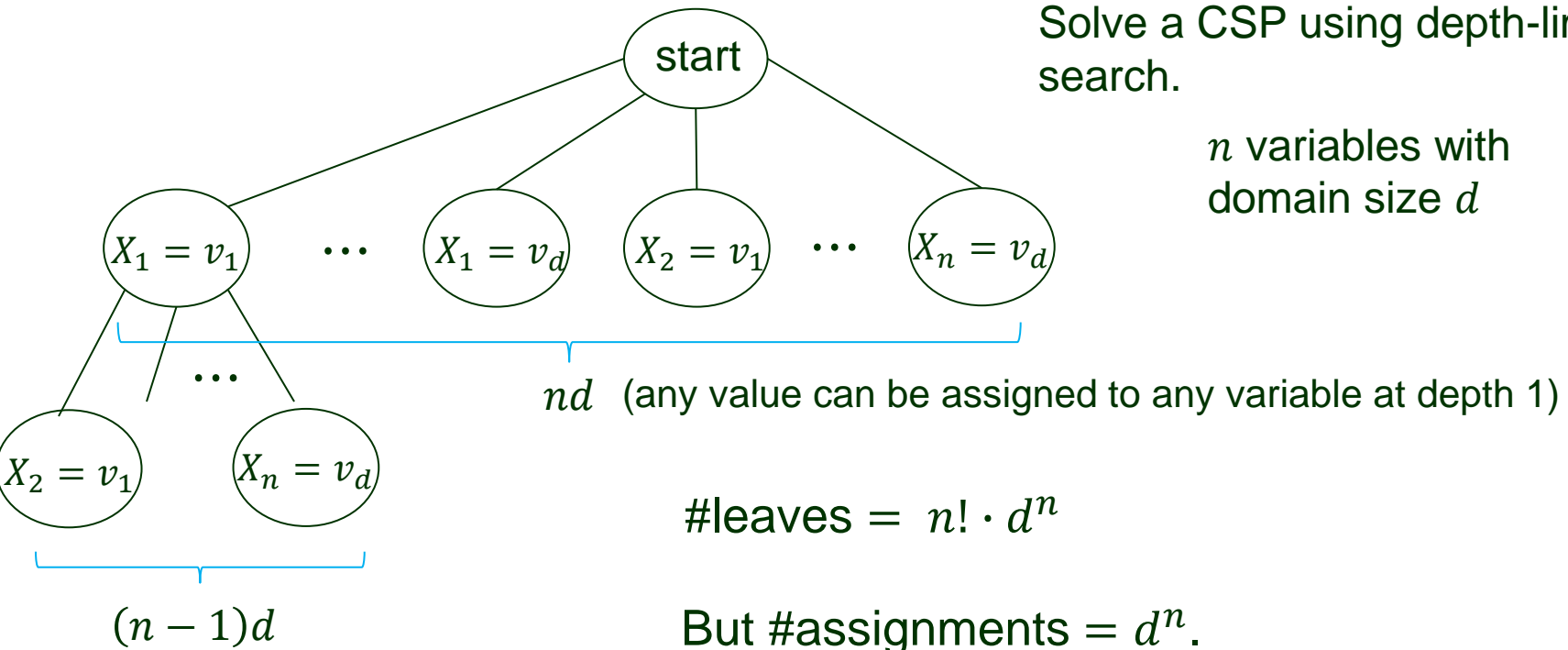
I. Backtracking Search

- ♣ Constraint propagation often ends with partial solutions.
- ♣ Backtracking search can be employed to extend them to full solutions.



I. Backtracking Search

- ♣ Constraint propagation often ends with partial solutions.
- ♣ Backtracking search can be employed to extend them to full solutions.



How to get back to d^n ?

Commutativity of CSP

A problem is *commutative* if the order of application of any given set of *actions* does not matter.

Commutativity of CSP

A problem is *commutative* if the order of application of any given set of actions does not matter.

↑
assignments in a CSP

Commutativity of CSP

A problem is *commutative* if the order of application of any given set of actions does not matter.

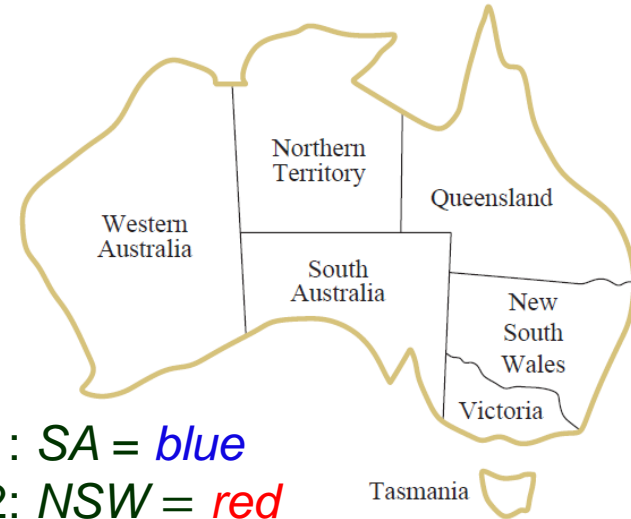
↑
assignments in a CSP

No difference between

Step 1: NSW = *red*
Step 2: SA = *blue*

and

Step 1: SA = *blue*
Step 2: NSW = *red*



Commutativity of CSP

A problem is *commutative* if the order of application of any given set of *actions* does not matter.

↑
assignments in a CSP

No difference between

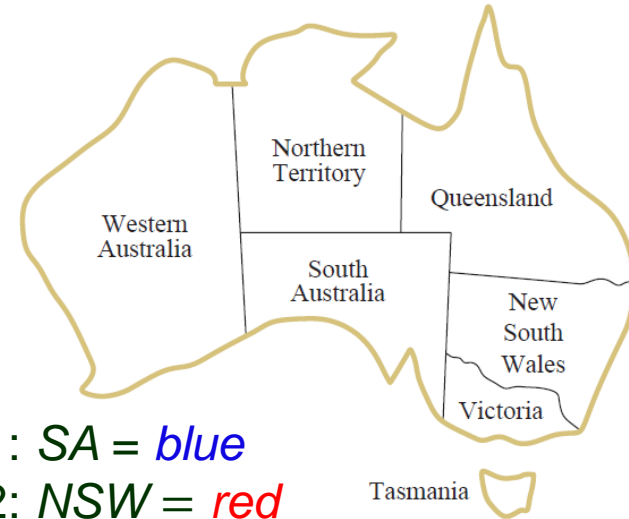
Step 1: *NSW = red*

Step 2: *SA = blue*

and

Step 1: *SA = blue*

Step 2: *NSW = red*



Need only consider a single variable at each node.

Commutativity of CSP

A problem is *commutative* if the order of application of any given set of *actions* does not matter.

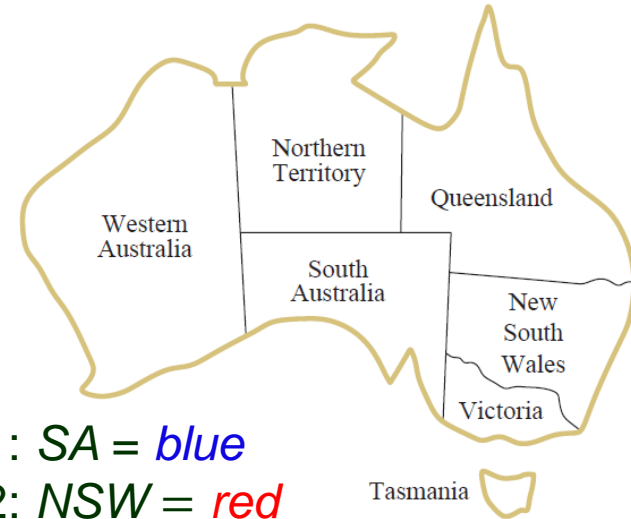
↑
assignments in a CSP

No difference between

Step 1: NSW = *red*
Step 2: SA = *blue*

and

Step 1: SA = *blue*
Step 2: NSW = *red*



Need only consider a single variable at each node.

At the root choose between

SA = *blue*, SA = *red*, and SA = *green*

Commutativity of CSP

A problem is *commutative* if the order of application of any given set of *actions* does not matter.

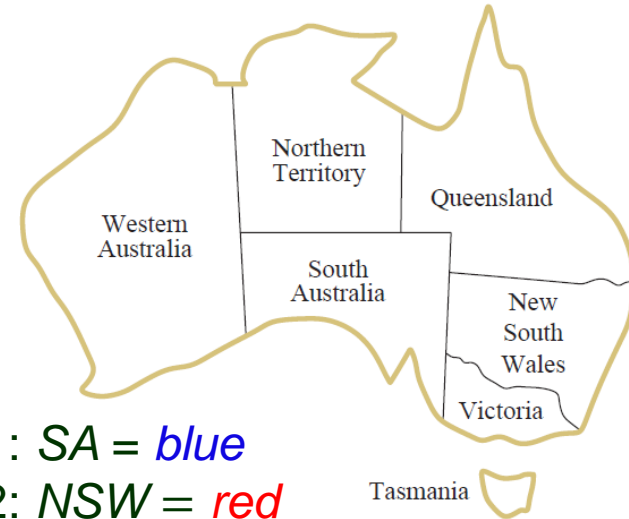
↑
assignments in a CSP

No difference between

Step 1: *NSW = red*
Step 2: *SA = blue*

and

Step 1: *SA = blue*
Step 2: *NSW = red*



Need only consider a single variable at each node.

At the root choose between

SA = blue, *SA = red*, and *SA = green*

but not between

SA = blue and *NSW = red*

Backtracking Algorithm

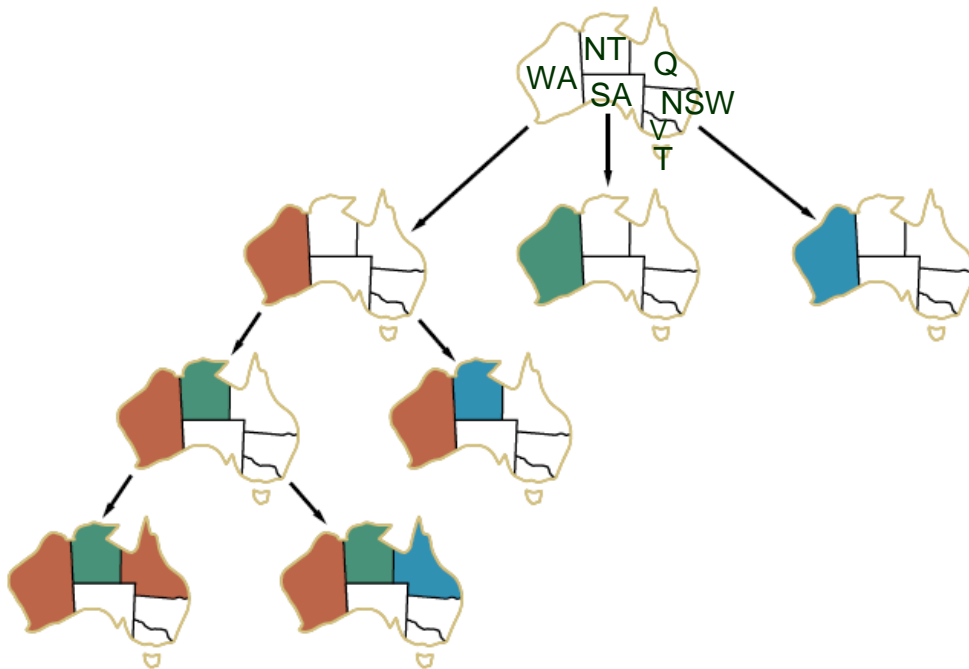
- Repeatedly chooses an unassigned variable X_i .
- Tries all values $v_j \in D_i$ (its domain).
 - ◆ Add $X_i = v_j$ to the partial solution (after consistency checking).
 - ◆ Try to extend it into a solution via a recursive call (in which another unassigned variable will be considered)

Backtracking Algorithm

- Repeatedly chooses an unassigned variable X_i .
- Tries all values $v_j \in D_i$ (its domain).

◆ Add $X_i = v_j$ to the partial solution (after consistency checking).

◆ Try to extend it into a solution via a recursive call (in which another unassigned variable will be considered)



Backtracking

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

function BACKTRACK(*csp*, *assignment*) **returns** a solution or *failure*

if *assignment* is complete **then return** *assignment*

var \leftarrow SELECT-UNASSIGNED-VARIABLE(*csp*, *assignment*)

for each *value* **in** ORDER-DOMAIN-VALUES(*csp*, *var*, *assignment*) **do**

if *value* is consistent with *assignment* **then**

add {*var* = *value*} to *assignment*

inferences \leftarrow INFERENCE(*csp*, *var*, *assignment*) // arc- or path-consistency, etc.

if *inferences* \neq *failure* **then**

// forward checking, etc. returns
// reduced domains.

add *inferences* to *csp*

result \leftarrow BACKTRACK(*csp*, *assignment*)

if *result* \neq *failure* **then return** *result*

remove *inferences* from *csp*

remove {*var* = *value*} from *assignment*

return *failure*

Order of Variables

$var \leftarrow \text{SELECT-UNASSIGNED-VARIABLE}(csp, assignment)$

In what order should we choose the variables?

Order of Variables

$var \leftarrow \text{SELECT-UNASSIGNED-VARIABLE}(csp, assignment)$

In what order should we choose the variables?

- Static order: $\{X_1, X_2, \dots\}$?

Order of Variables

$var \leftarrow \text{SELECT-UNASSIGNED-VARIABLE}(csp, assignment)$

In what order should we choose the variables?

- Static order: $\{X_1, X_2, \dots\}$?
- Random order: $\{X_1, X_2, \dots\}$?

Order of Variables

$var \leftarrow \text{SELECT-UNASSIGNED-VARIABLE}(csp, assignment)$

In what order should we choose the variables?

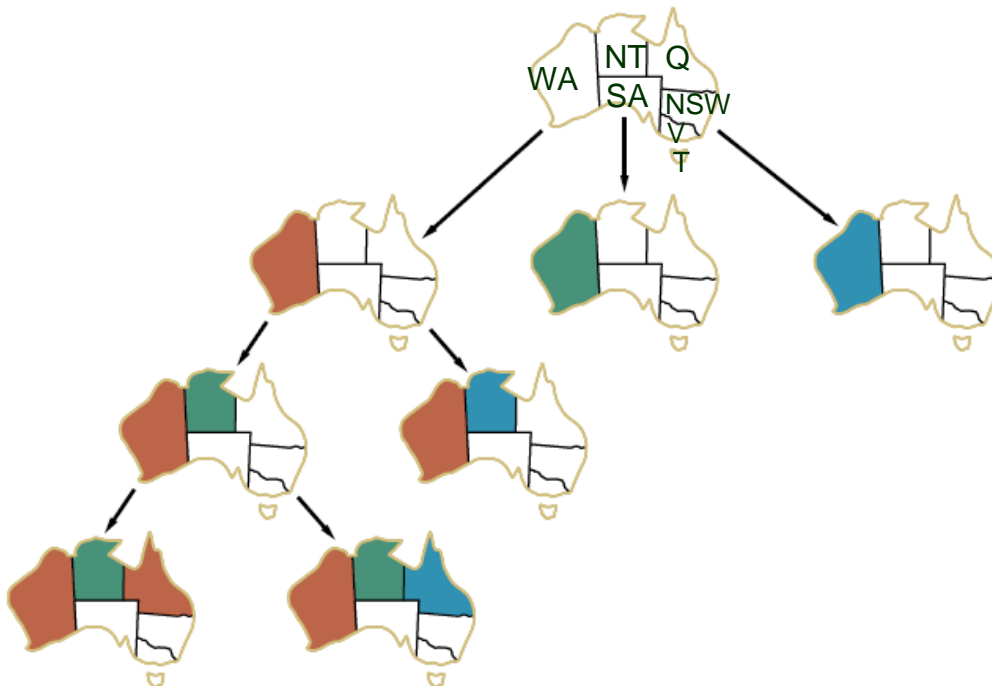
- Static order: $\{X_1, X_2, \dots\}$?
 - Random order: $\{X_1, X_2, \dots\}$?
- } Neither is optimal!

Order of Variables

$var \leftarrow \text{SELECT-UNASSIGNED-VARIABLE}(csp, assignment)$

In what order should we choose the variables?

- Static order: $\{X_1, X_2, \dots\}$?
 - Random order: $\{X_1, X_2, \dots\}$?
- } Neither is optimal!

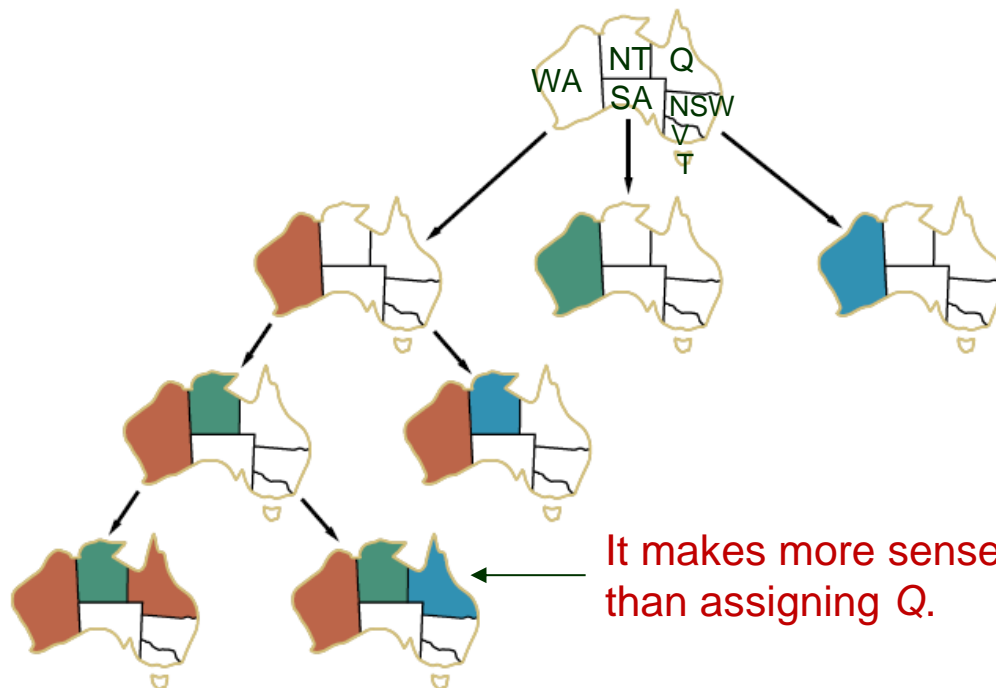


Order of Variables

$var \leftarrow \text{SELECT-UNASSIGNED-VARIABLE}(csp, assignment)$

In what order should we choose the variables?

- Static order: $\{X_1, X_2, \dots\}$?
 - Random order: $\{X_1, X_2, \dots\}$?
- } Neither is optimal!



It makes more sense to assign $SA = blue$ than assigning Q .

MRV and Degree Heuristics

1. *Minimum-remaining-values* (MRV): Choose the variable with the fewest “legal” values.

- ◆ A.k.a. “most constrained variable” or “fail first” heuristic

MRV and Degree Heuristics

1. *Minimum-remaining-values (MRV)*: Choose the variable with the fewest “legal” values.

- ◆ A.k.a. “most constrained variable” or “fail first” heuristic

If some variable has no legal values left, select it!

MRV and Degree Heuristics

1. *Minimum-remaining-values (MRV)*: Choose the variable with the fewest “legal” values.

- ◆ A.k.a. “most constrained variable” or “fail first” heuristic

If some variable has no legal values left, select it!

- ◆ Performs better than a random or static ordering.

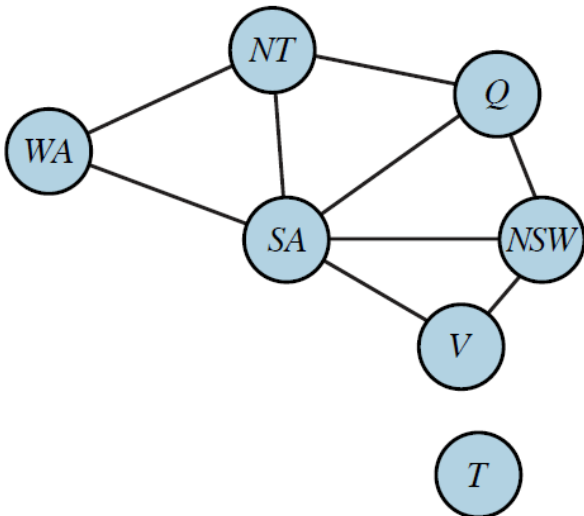
MRV and Degree Heuristics

1. *Minimum-remaining-values (MRV)*: Choose the variable with the fewest “legal” values.

- ◆ A.k.a. “most constrained variable” or “fail first” heuristic

If some variable has no legal values left, select it!

- ◆ Performs better than a random or static ordering.



2. Use the *degree heuristic* as a tie-breaker or at the start.

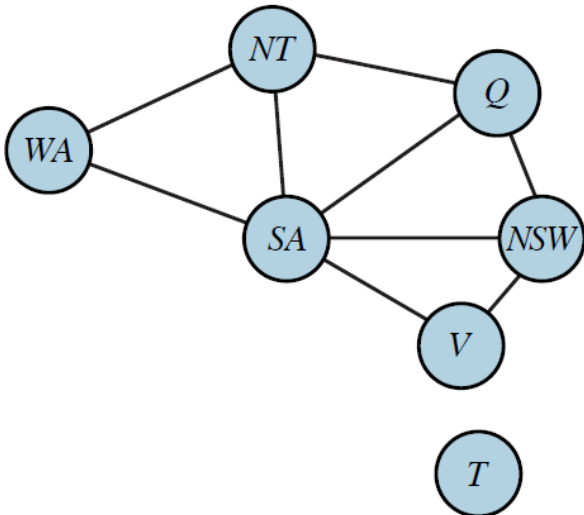
MRV and Degree Heuristics

1. *Minimum-remaining-values (MRV)*: Choose the variable with the fewest “legal” values.

- ◆ A.k.a. “most constrained variable” or “fail first” heuristic

If some variable has no legal values left, select it!

- ◆ Performs better than a random or static ordering.



2. Use the *degree heuristic* as a tie-breaker or at the start.

$$\text{deg}(SA) = 5$$

Others have degrees ≤ 3 .

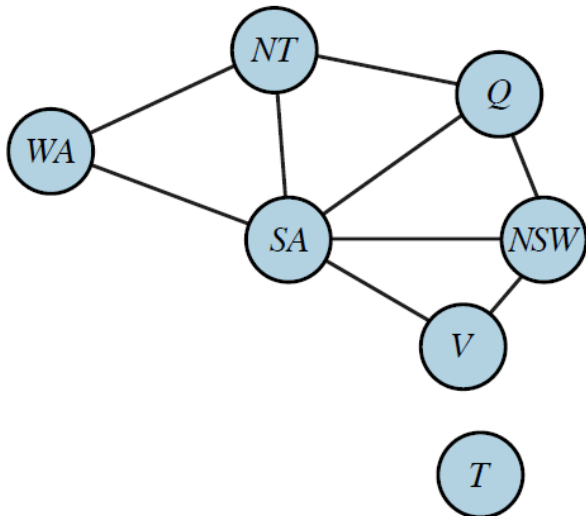
MRV and Degree Heuristics

1. *Minimum-remaining-values (MRV)*: Choose the variable with the fewest “legal” values.

- ◆ A.k.a. “most constrained variable” or “fail first” heuristic

If some variable has no legal values left, select it!

- ◆ Performs better than a random or static ordering.



2. Use the *degree heuristic* as a tie-breaker or at the start.

$$\text{deg}(SA) = 5$$

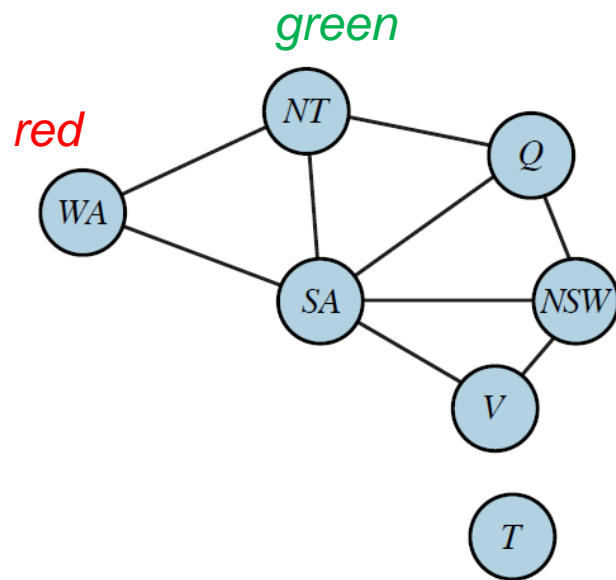
Others have degrees ≤ 3 .



Color SA first.

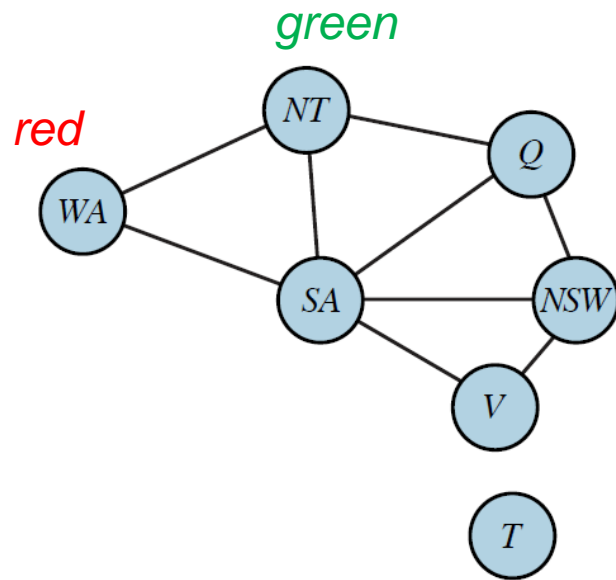
Least Constraining Value

For the selected variable, choose its value that *rules out the fewest choices* for the neighboring variables in the constraint graph.



Least Constraining Value

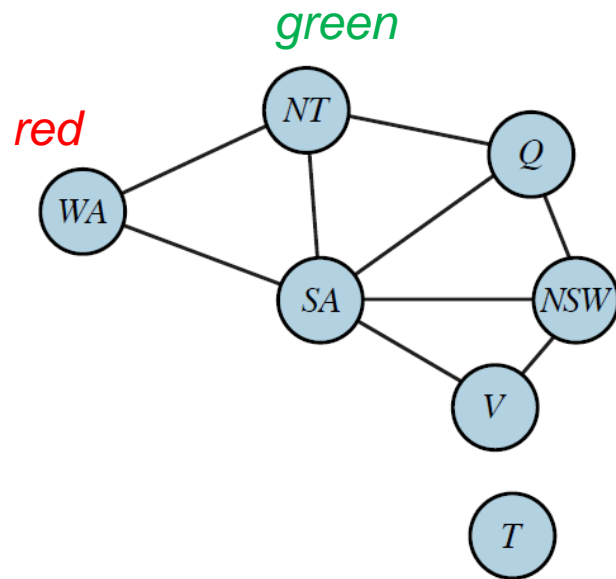
For the selected variable, choose its value that *rules out the fewest choices* for the neighboring variables in the constraint graph.



Which color to assign to Q next?

Least Constraining Value

For the selected variable, choose its value that *rules out the fewest choices* for the neighboring variables in the constraint graph.



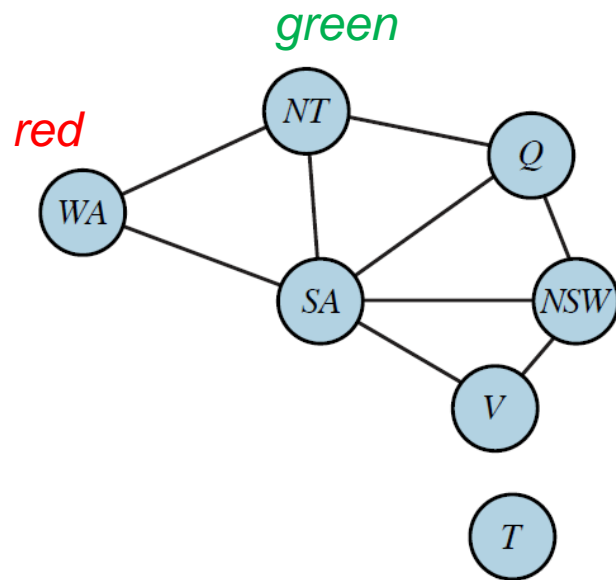
Which color to assign to Q next?

If *blue*, then SA would have no color left.

Choose *red*.

Least Constraining Value

For the selected variable, choose its value that *rules out the fewest choices* for the neighboring variables in the constraint graph.



Which color to assign to Q next?

If *blue*, then SA would have no color left.

Choose *red*.

The least-constraining-value heuristic tries to create the *maximum* room for subsequent variable assignments.

Variable vs. Value Selections

Variable order: *fail-first*.

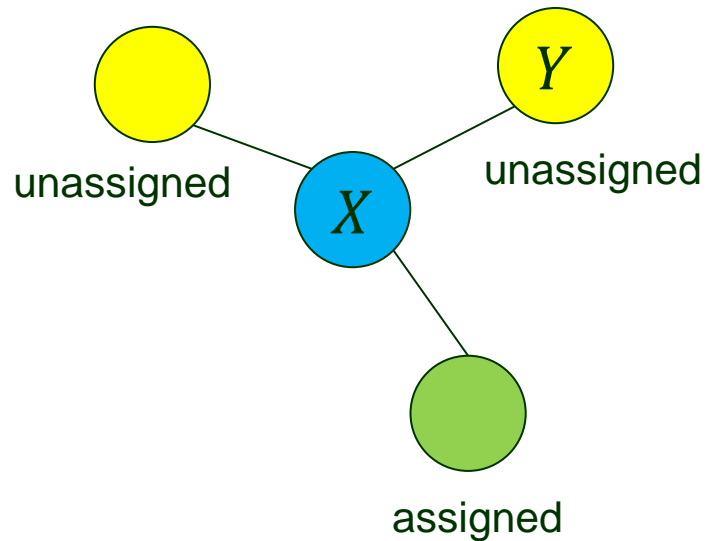
Fewer successful assignments to backtrack over.

Value order: *fail-last*.

- ◆ Only one solution needed.
- ◆ It makes sense to look for the most likely values first.

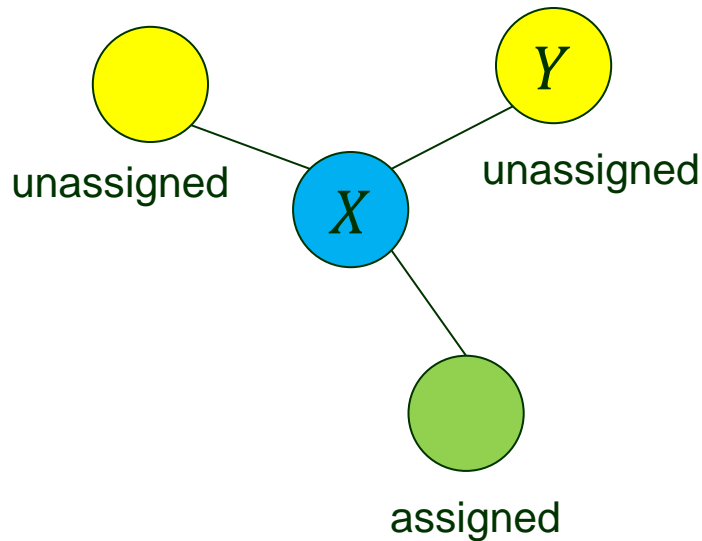
Forward Checking

Inference: Every new variable assignment opens the door for new domain reductions on neighboring variables.



Forward Checking

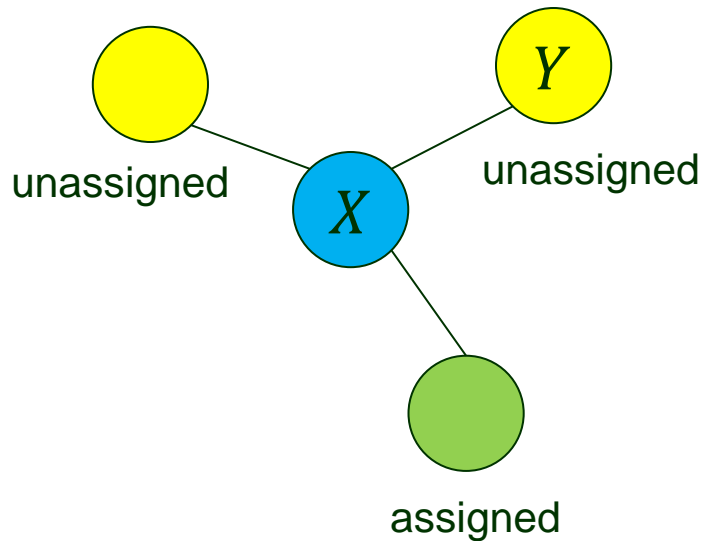
Inference: Every new variable assignment opens the door for new domain reductions on neighboring variables.



Assignment $X = v$

Forward Checking

Inference: Every new variable assignment opens the door for new domain reductions on neighboring variables.

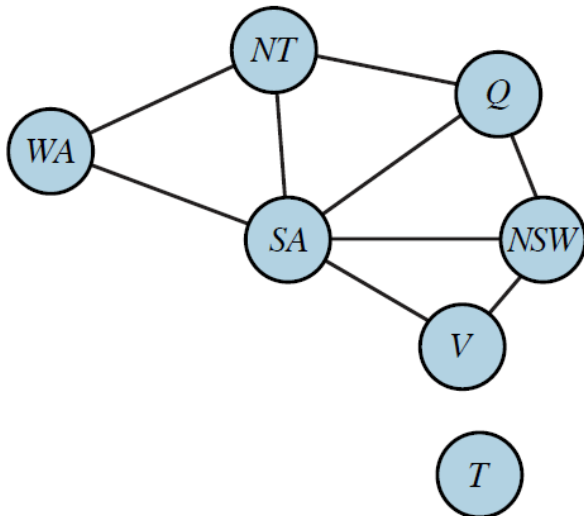


Assignment $X = v$

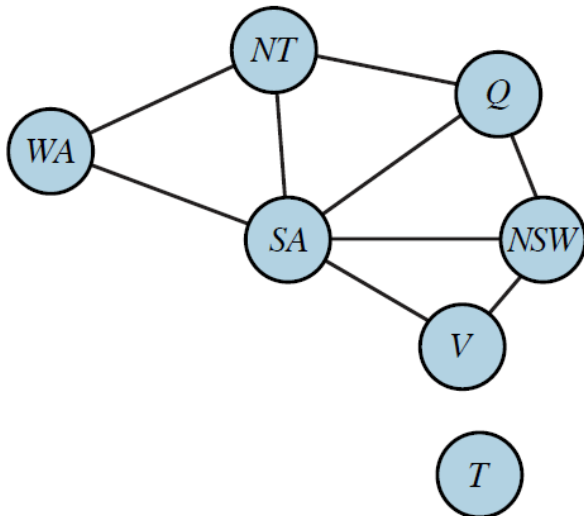


For every unassigned Y connected to X , delete any value from Y 's domain that is inconsistent with v .

Backtracking with Forward Checking

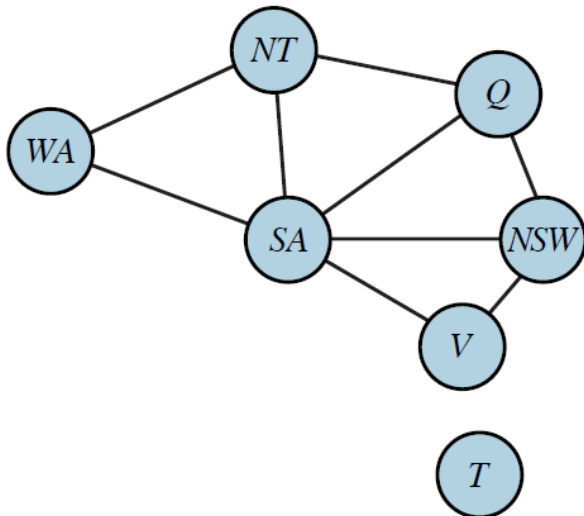


Backtracking with Forward Checking



- *WA = red*

Backtracking with Forward Checking

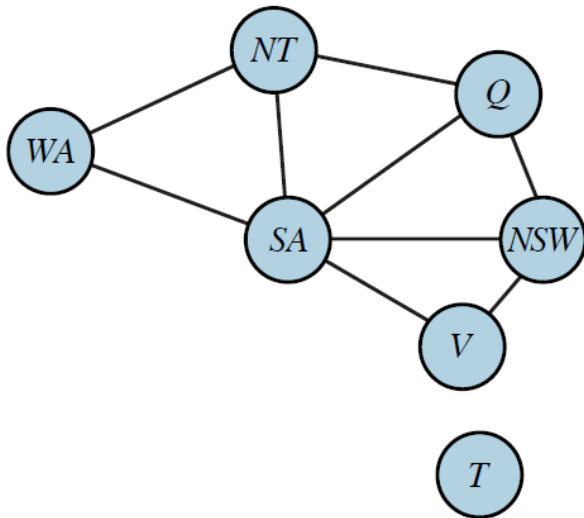


- *WA* = *red*

Deletes *red* for *NT* and *SA*.

Backtracking with Forward Checking















	<i>WA</i>	<i>NT</i>	<i>Q</i>	<i>NSW</i>	<i>V</i>	<i>SA</i>	<i>T</i>
Initial domains							
After <i>WA=red</i>							

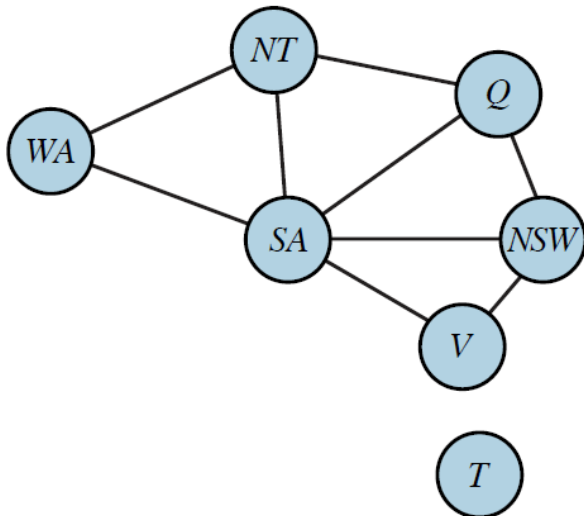


- *WA = red*

Deletes *red* for *NT* and *SA*.

Backtracking with Forward Checking

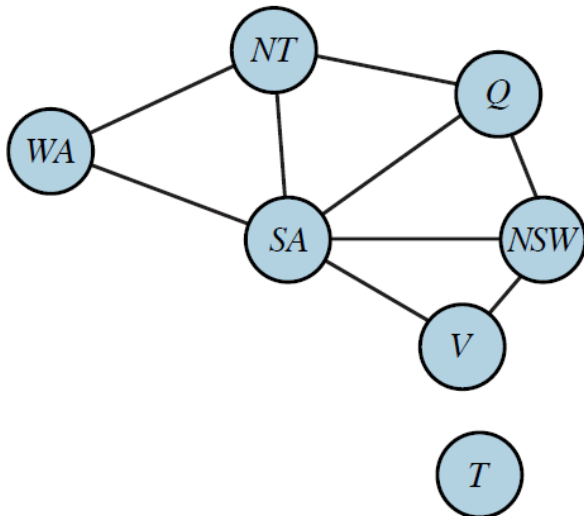
	<i>WA</i>	<i>NT</i>	<i>Q</i>	<i>NSW</i>	<i>V</i>	<i>SA</i>	<i>T</i>
Initial domains							
After <i>WA=red</i>							



- *WA = red*
Deletes *red* for *NT* and *SA*.
- *Q = green*

Backtracking with Forward Checking

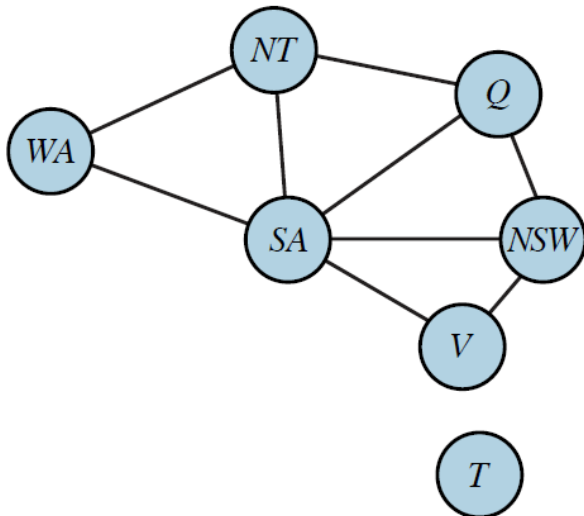
	<i>WA</i>	<i>NT</i>	<i>Q</i>	<i>NSW</i>	<i>V</i>	<i>SA</i>	<i>T</i>
Initial domains							
After <i>WA=red</i>							



- *WA = red*
Deletes *red* for *NT* and *SA*.
- *Q = green*
Deletes *green* for *NT*, *SA*, and *NSW*.

Backtracking with Forward Checking

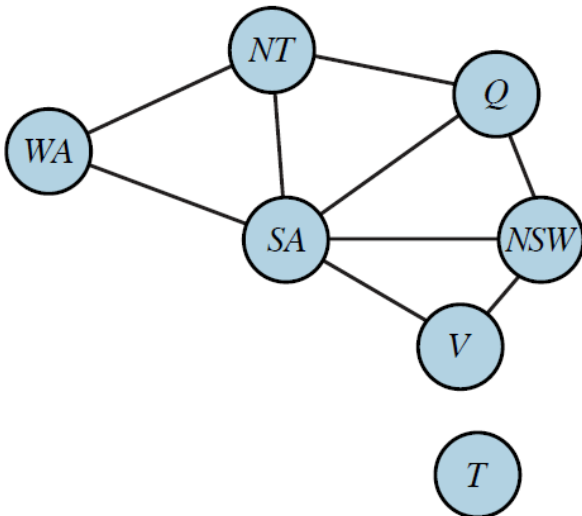
	<i>WA</i>	<i>NT</i>	<i>Q</i>	<i>NSW</i>	<i>V</i>	<i>SA</i>	<i>T</i>
Initial domains							
After <i>WA=red</i>							
After <i>Q=green</i>							



- *WA = red*
Deletes *red* for *NT* and *SA*.
- *Q = green*
Deletes *green* for *NT*, *SA*, and *NSW*.

Backtracking with Forward Checking

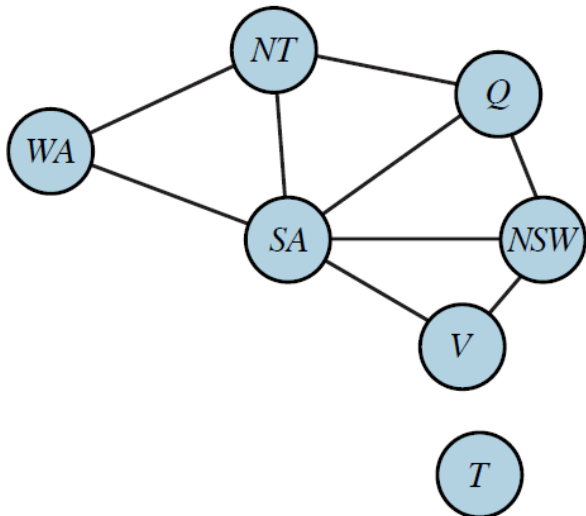
	<i>WA</i>	<i>NT</i>	<i>Q</i>	<i>NSW</i>	<i>V</i>	<i>SA</i>	<i>T</i>
Initial domains							
After <i>WA=red</i>							
After <i>Q=green</i>							



- *WA = red*
Deletes *red* for *NT* and *SA*.
- *Q = green*
Deletes *green* for *NT*, *SA*, and *NSW*.
NT & *SA* each has a single value.

Backtracking with Forward Checking

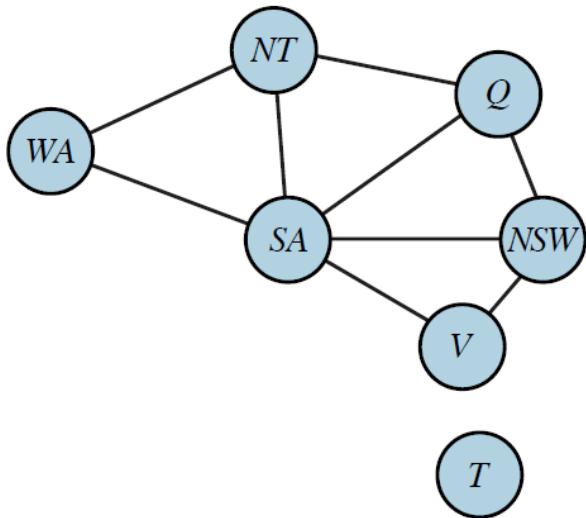
	<i>WA</i>	<i>NT</i>	<i>Q</i>	<i>NSW</i>	<i>V</i>	<i>SA</i>	<i>T</i>
Initial domains							
After <i>WA=red</i>							
After <i>Q=green</i>							



- *WA = red*
Deletes *red* for *NT* and *SA*.
- *Q = green*
Deletes *green* for *NT*, *SA*, and *NSW*.
NT & *SA* each has a single value.
- *V = blue*

Backtracking with Forward Checking

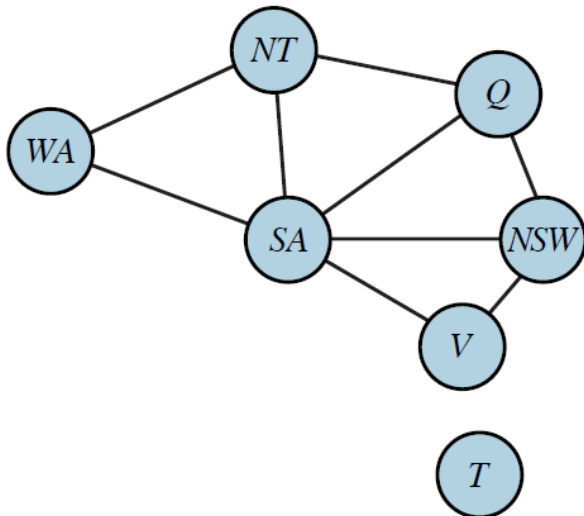
	<i>WA</i>	<i>NT</i>	<i>Q</i>	<i>NSW</i>	<i>V</i>	<i>SA</i>	<i>T</i>
Initial domains							
After <i>WA=red</i>							
After <i>Q=green</i>							
After <i>V=blue</i>							



- *WA = red*
Deletes *red* for *NT* and *SA*.
- *Q = green*
Deletes *green* for *NT*, *SA*, and *NSW*.
NT & *SA* each has a single value.
- *V = blue*

Backtracking with Forward Checking

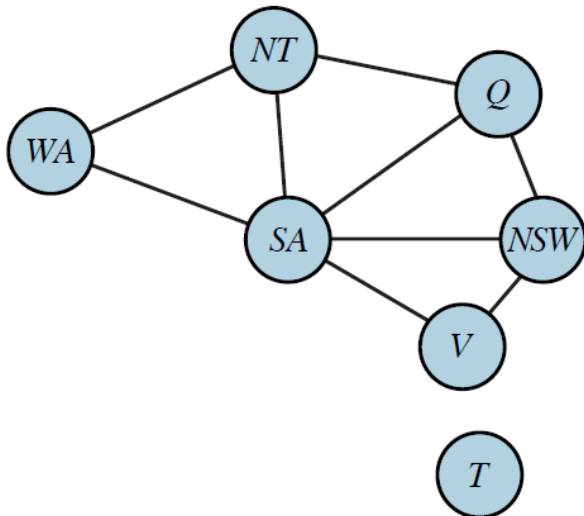
	<i>WA</i>	<i>NT</i>	<i>Q</i>	<i>NSW</i>	<i>V</i>	<i>SA</i>	<i>T</i>
Initial domains							
After <i>WA=red</i>							
After <i>Q=green</i>							
After <i>V=blue</i>							



- *WA = red*
Deletes *red* for *NT* and *SA*.
- *Q = green*
Deletes *green* for *NT*, *SA*, and *NSW*.
NT & *SA* each has a single value.
- *V = blue*
SA has no legal value.

Backtracking with Forward Checking

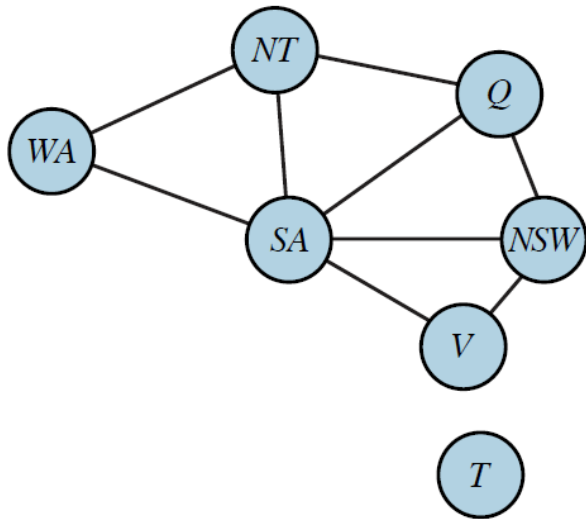
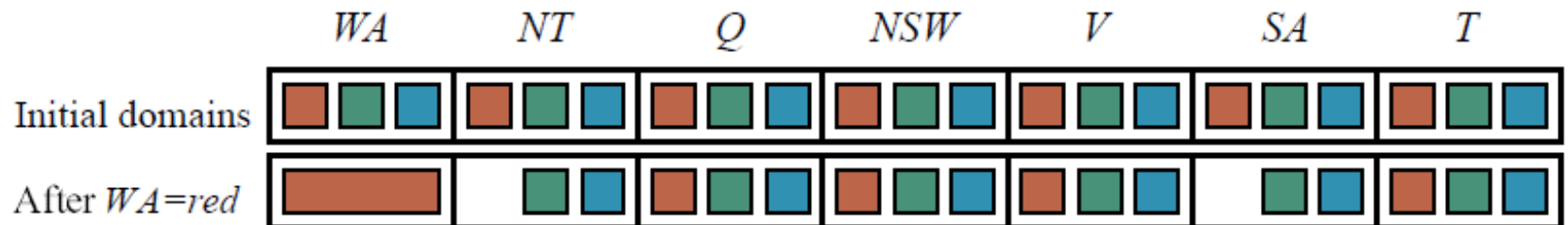
	<i>WA</i>	<i>NT</i>	<i>Q</i>	<i>NSW</i>	<i>V</i>	<i>SA</i>	<i>T</i>
Initial domains							
After <i>WA=red</i>							
After <i>Q=green</i>							
After <i>V=blue</i>							



- *WA = red*
Deletes *red* for *NT* and *SA*.
- *Q = green*
Deletes *green* for *NT*, *SA*, and *NSW*.
NT & *SA* each has a single value.
- *V = blue*
SA has no legal value.
Delete {*WA = red*, *Q = green*, *V = blue*}.
Start backtracking.

Combining MRV and FC Heuristics

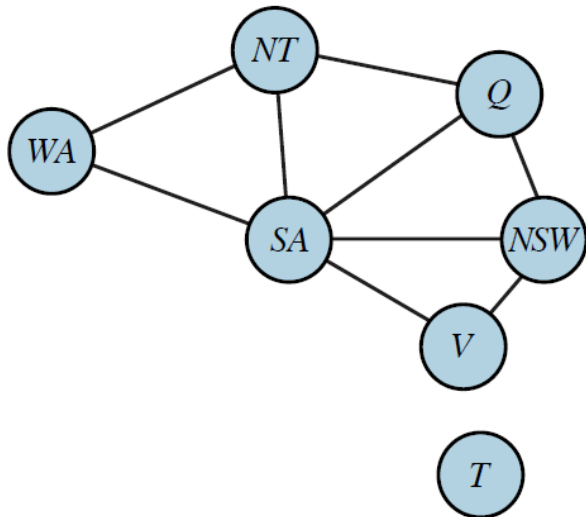
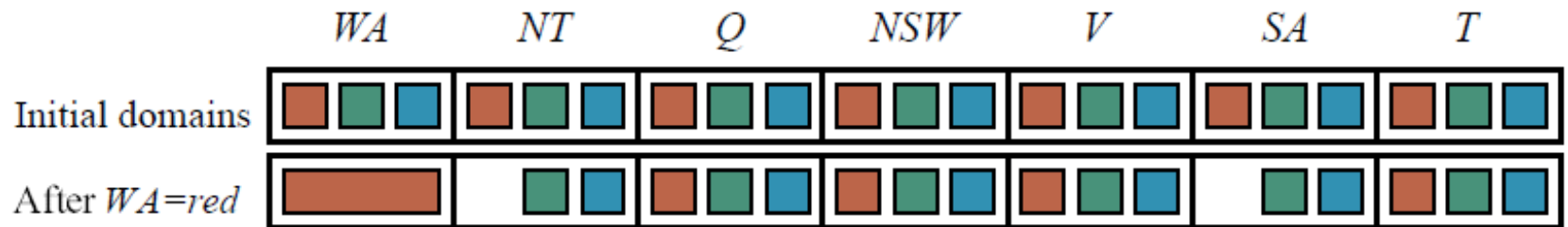
Search becomes more effective when they are combined.



- *NT* and *SA* are constrained by the assignment *WA=red*.
- Deal with them first according to MRV.

Combining MRV and FC Heuristics

Search becomes more effective when they are combined.



- *NT* and *SA* are constrained by the assignment *WA=red*.
- Deal with them first according to MRV.

Combination of MRV and FC can solve the 1000-queen problem.

II. Local Search

- ◆ Every state corresponds to a **complete assignment**.
- ◆ Search changes the value of one variable at a time.

II. Local Search

- ◆ Every state corresponds to a **complete assignment**.
- ◆ Search changes the value of one variable at a time.

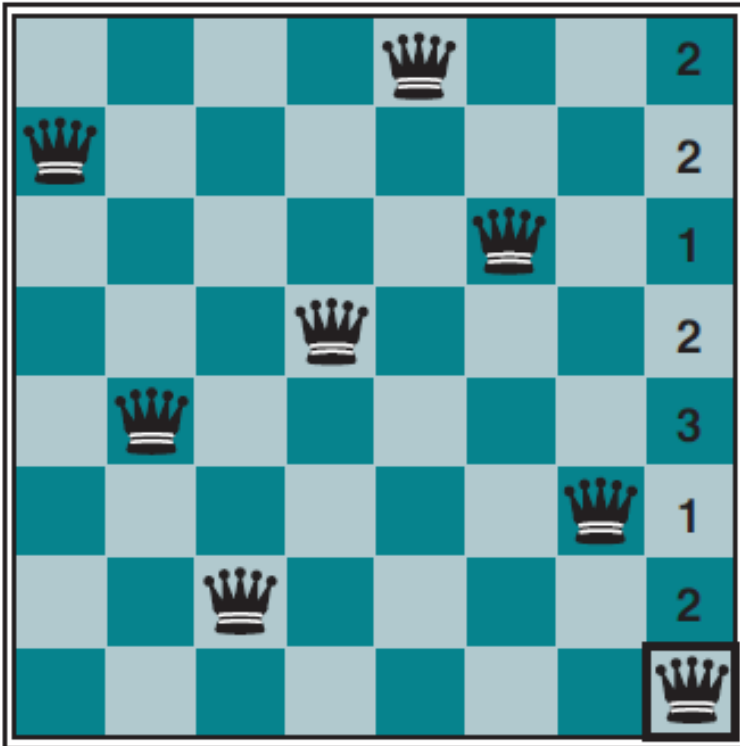
Min-conflicts heuristic:

- Start with a complete assignment.
- Randomly choose a conflicted variable.
- Select the value that results in the least conflicts with other variables.

Applying Min-Conflicts to 8-Queen

Variable set: $\mathcal{X} = \{Q_1, Q_2, \dots, Q_8\}$

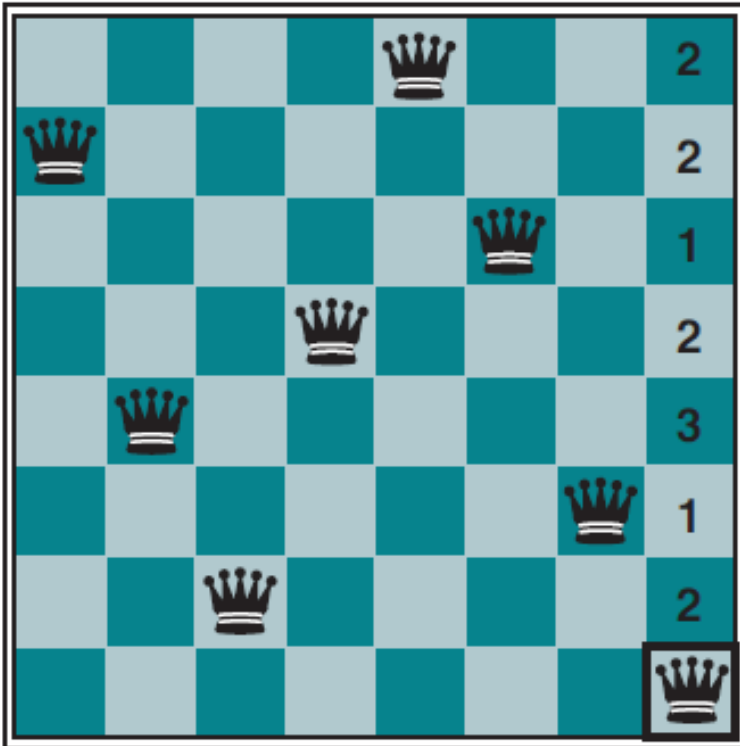
Q_i : the row number of the queen placed in the i th column.



Applying Min-Conflicts to 8-Queen

Variable set: $\mathcal{X} = \{Q_1, Q_2, \dots, Q_8\}$

Q_i : the row number of the queen placed in the i th column.



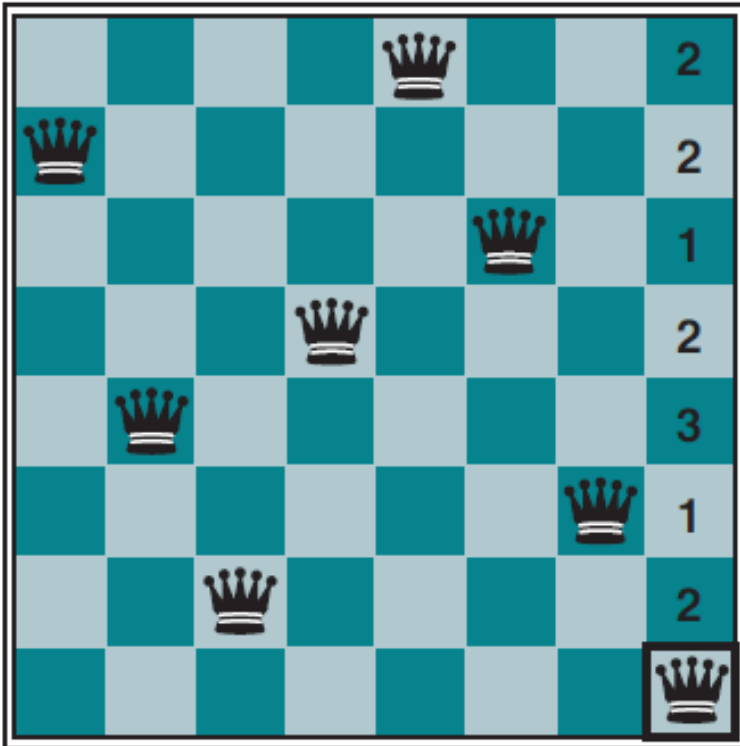
$$Q_8 = 8$$

Applying Min-Conflicts to 8-Queen

Variable set: $\mathcal{X} = \{Q_1, Q_2, \dots, Q_8\}$

Q_i : the row number of the queen placed in the i th column.

- Q_8 out of the set $\{Q_4, Q_8\}$ of conflicted variables by a random choice.



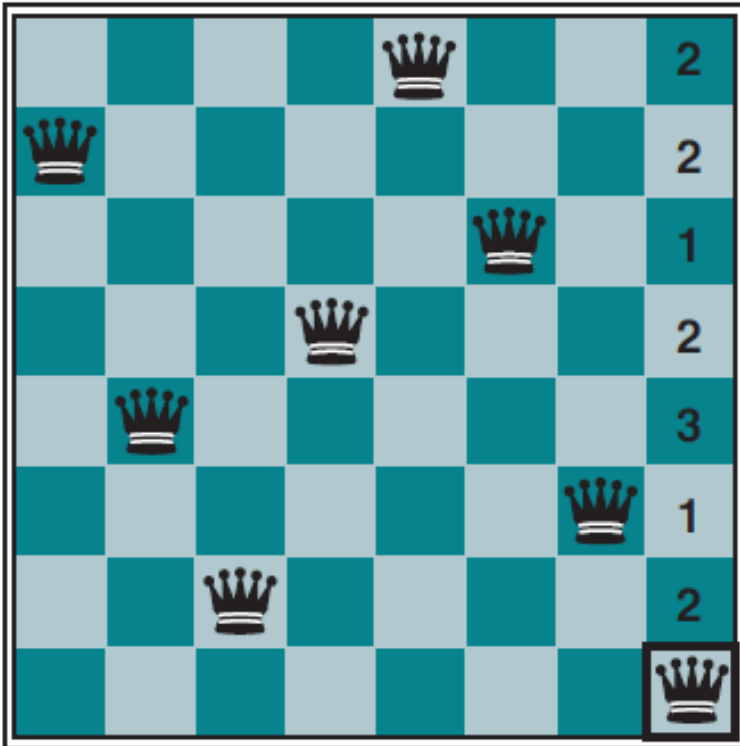
$$Q_8 = 8$$

Applying Min-Conflicts to 8-Queen

Variable set: $\mathcal{X} = \{Q_1, Q_2, \dots, Q_8\}$

Q_i : the row number of the queen placed in the i th column.

- Q_8 out of the set $\{Q_4, Q_8\}$ of conflicted variables by a random choice.



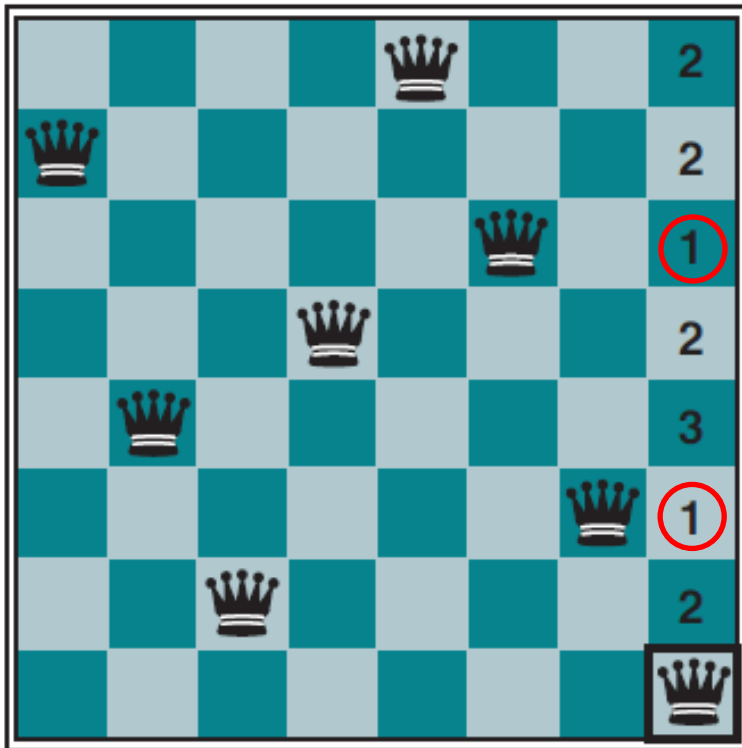
← 2 conflicts if $Q_8 = 7$

$Q_8 = 8$

Applying Min-Conflicts to 8-Queen

Variable set: $\mathcal{X} = \{Q_1, Q_2, \dots, Q_8\}$

Q_i : the row number of the queen placed in the i th column.



$Q_8 = 8$

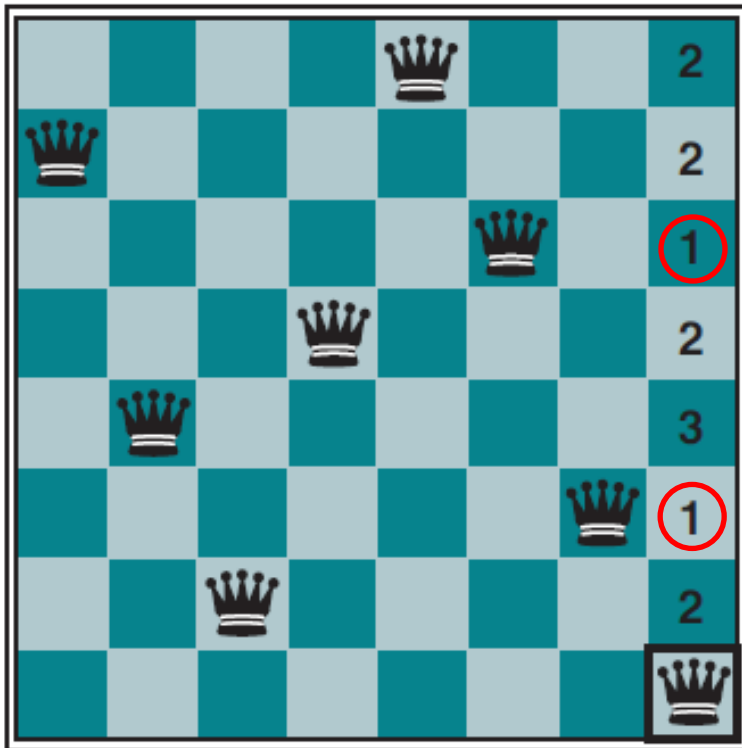
- Q_8 out of the set $\{Q_4, Q_8\}$ of conflicted variables by a random choice.
- The 8th queen in row 3 or 6 would violate only one constraint.

← 2 conflicts if $Q_8 = 7$

Applying Min-Conflicts to 8-Queen

Variable set: $\mathcal{X} = \{Q_1, Q_2, \dots, Q_8\}$

Q_i : the row number of the queen placed in the i th column.

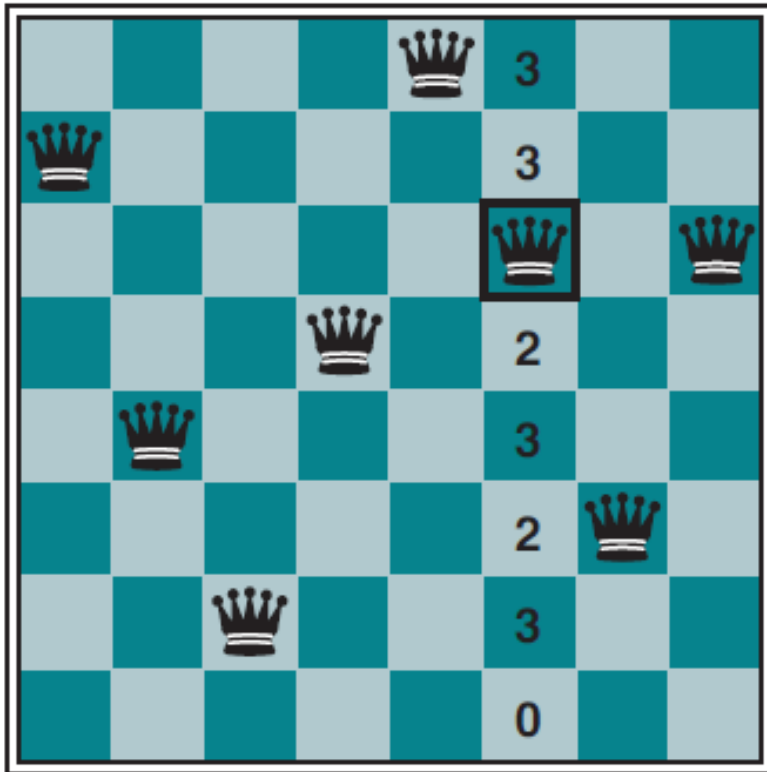


$Q_8 = 8$

- Q_8 out of the set $\{Q_4, Q_8\}$ of conflicted variables by a random choice.
- The 8th queen in row 3 or 6 would violate only one constraint.
- Move the queen to, say, row 3.

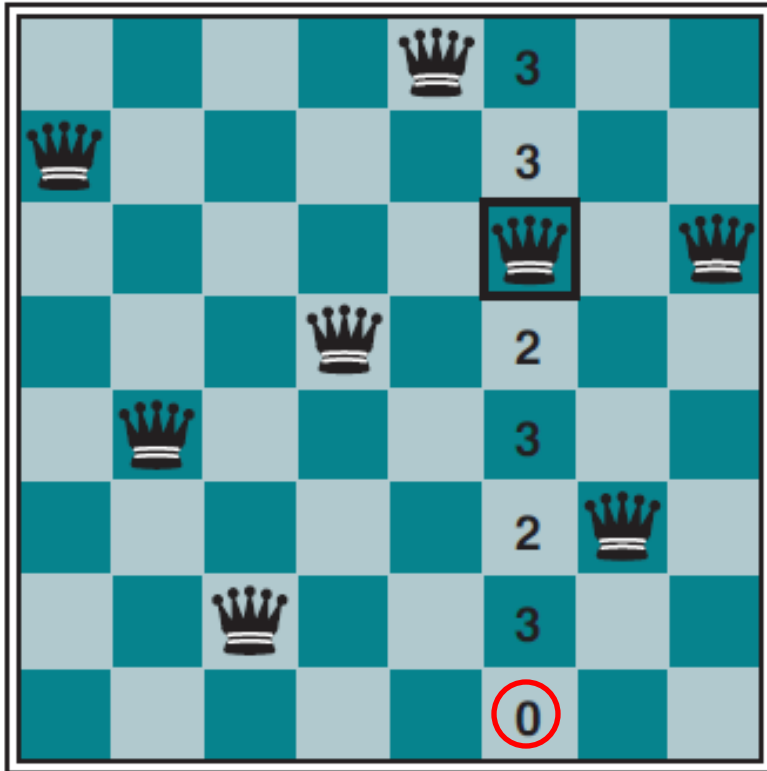
← 2 conflicts if $Q_8 = 7$

8- and n -Queen problems



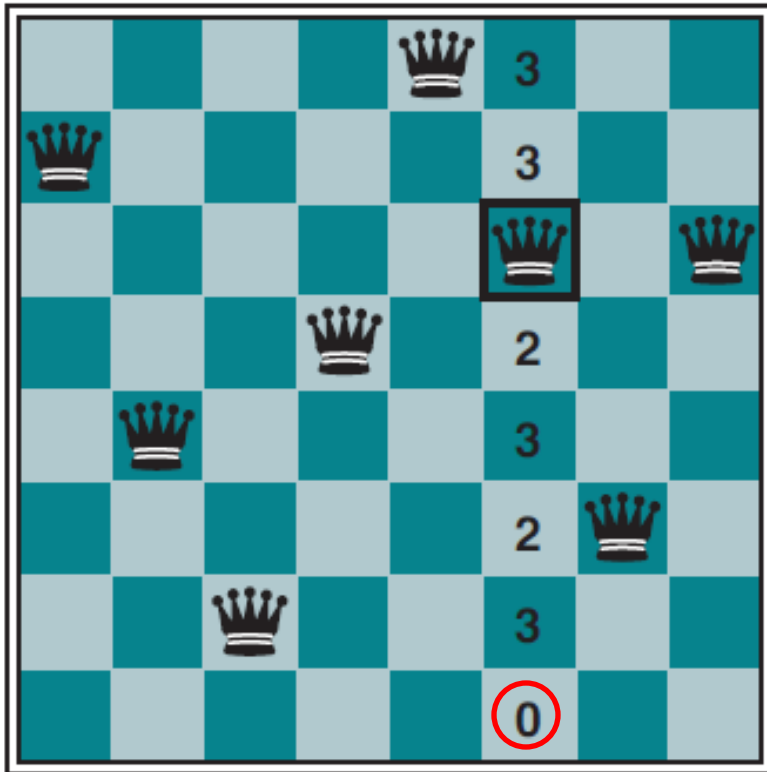
- Q_6 out of $\{Q_6, Q_8\}$.

8- and n -Queen problems



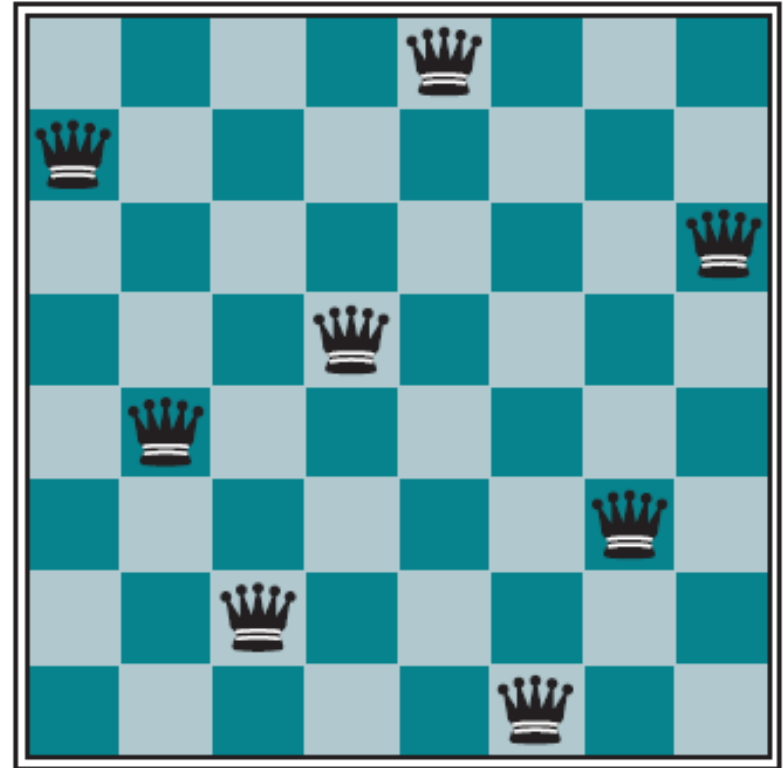
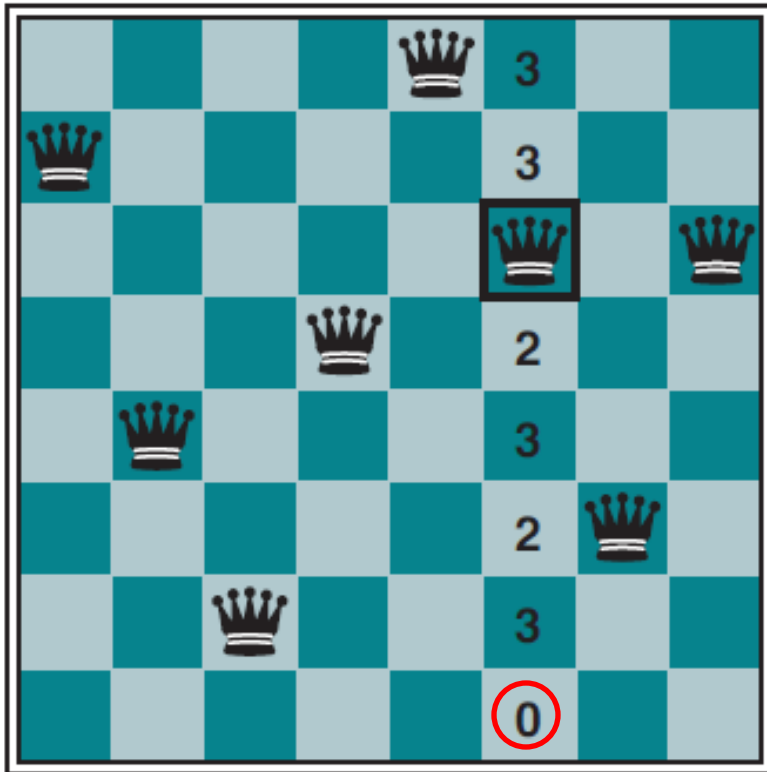
- Q_6 out of $\{Q_6, Q_8\}$.

8- and n -Queen problems



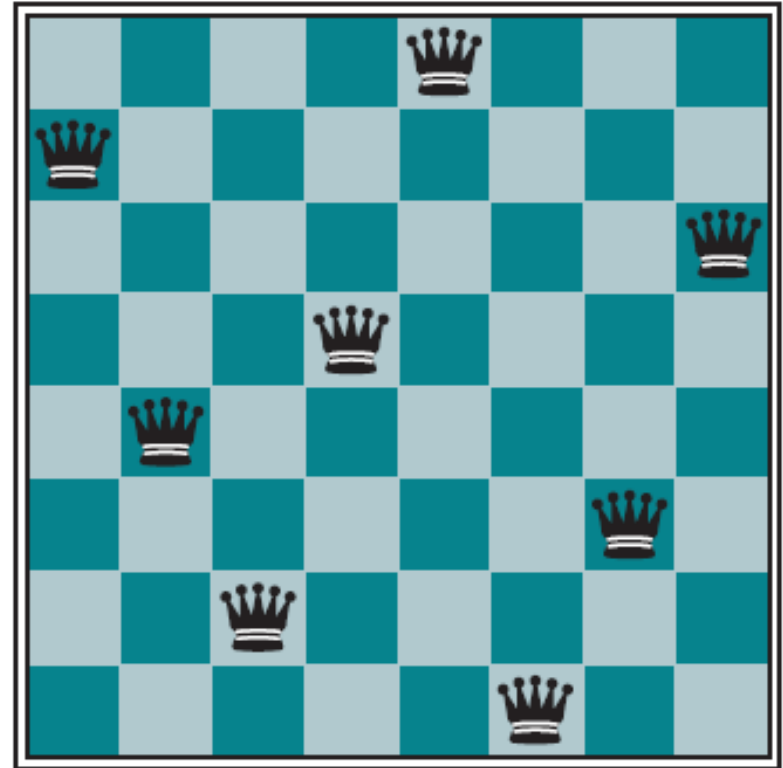
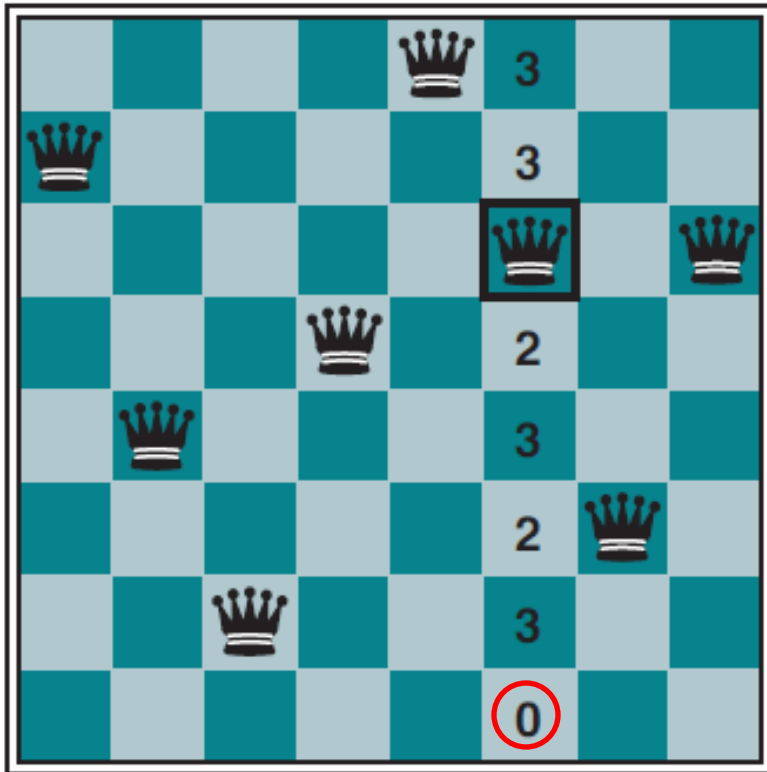
- Q_6 out of $\{Q_6, Q_8\}$.
- Reassignment: $Q_6 = 8$.

8- and n -Queen problems



- Q_6 out of $\{Q_6, Q_8\}$.
- Reassignment: $Q_6 = 8$.

8- and n -Queen problems



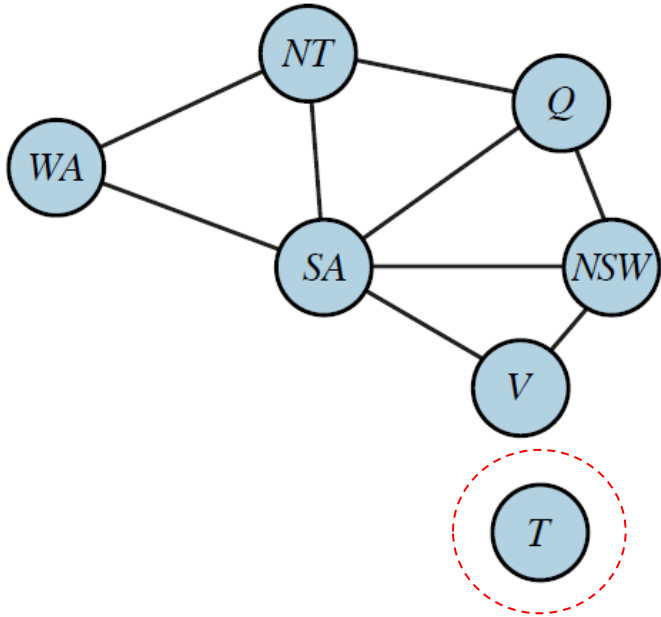
Solution

- Q_6 out of $\{Q_6, Q_8\}$.
- Reassignment: $Q_6 = 8$.

Local Search: n -Queen and Beyond

- ◆ Run time of min-conflicts on n -queen is roughly independent of n .
 - 10^6 -queen problems are solved in an average of 50 steps (after the initial assignment).
- ◆ Ease of solving n -queen due to dense distribution of solutions throughout the state space.
- ◆ Min-conflicts also effective on hard problems such as observation scheduling for the Hubble Space Telescope.
- ◆ Local search is applicable in an online setting (e.g., repairing the scheduling of an airline's weekly activities – in the advent of bad weather).

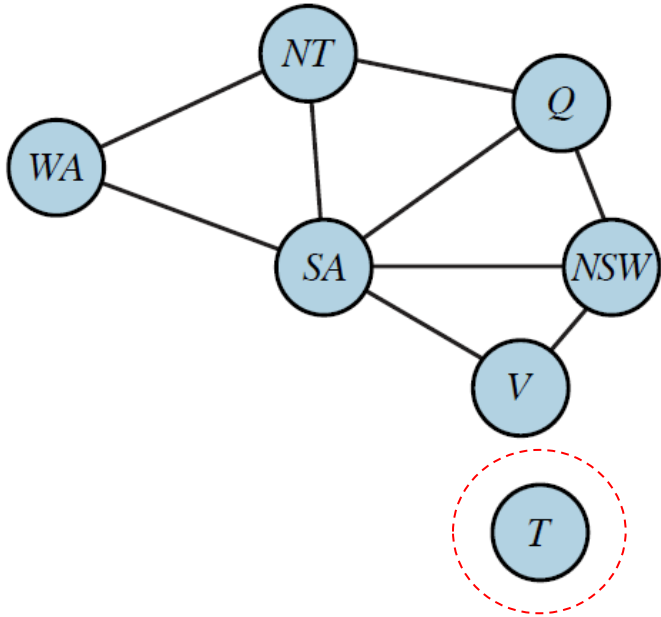
III. The Structure of CSP Problems



Independent subproblems

- **Connected components** in the constraint graph.
- Each subproblem can be solved independently.

III. The Structure of CSP Problems

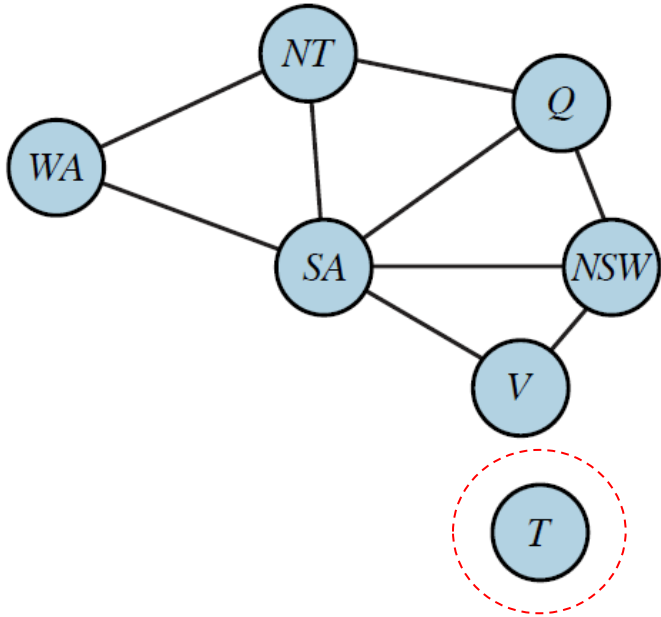


n variables
domain size d

Independent subproblems

- **Connected components** in the constraint graph.
- Each subproblem can be solved independently.

III. The Structure of CSP Problems



Independent subproblems

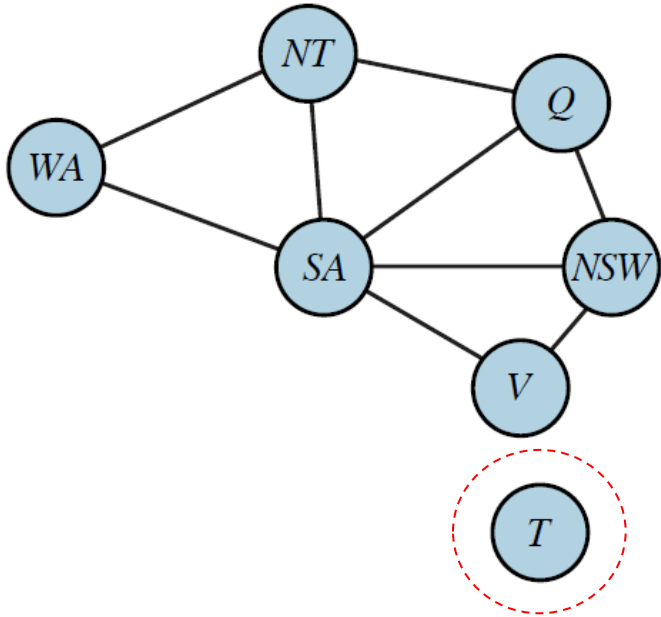
- **Connected components** in the constraint graph.
- Each subproblem can be solved independently.

n variables
domain size d



Total work $O(d^n)$ without problem decomposition.

III. The Structure of CSP Problems



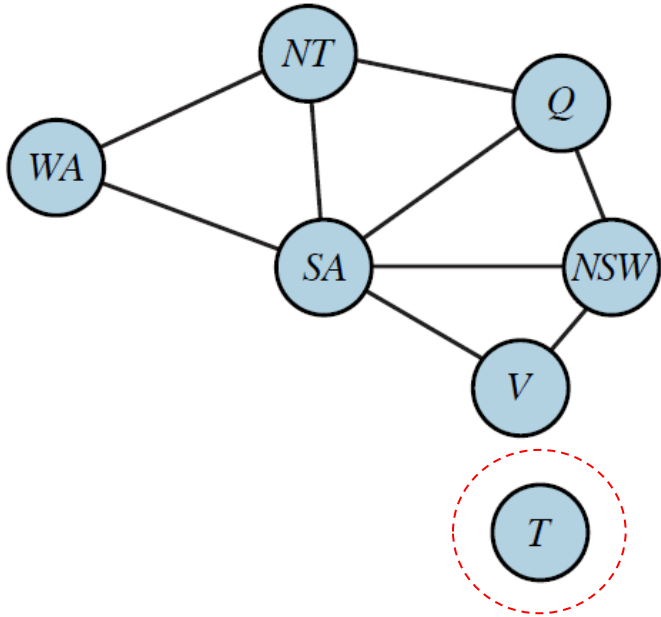
Independent subproblems

- **Connected components** in the constraint graph.
- Each subproblem can be solved independently.

n variables
domain size d } \implies Total work $O(d^n)$ without problem decomposition.

c variables for each subproblem

III. The Structure of CSP Problems



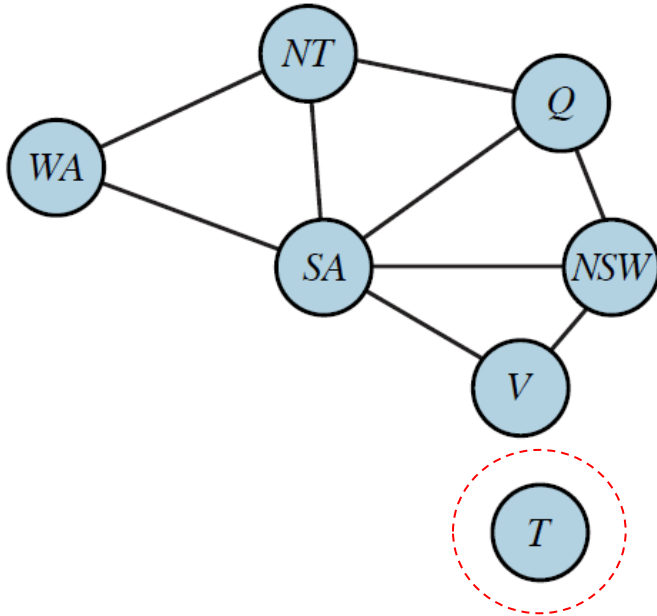
Independent subproblems

- **Connected components** in the constraint graph.
- Each subproblem can be solved independently.

n variables
domain size d } \implies Total work $O(d^n)$ without problem decomposition.

c variables for each subproblem $\implies n/c$ subproblems, each requiring $O(d^c)$

III. The Structure of CSP Problems



Independent subproblems

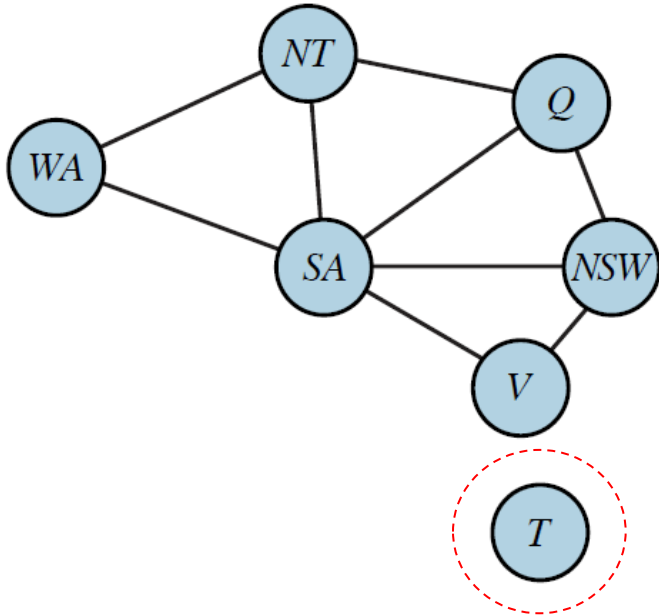
- **Connected components** in the constraint graph.
- Each subproblem can be solved independently.

n variables
domain size d } \implies Total work $O(d^n)$ without problem decomposition.

c variables for each subproblem $\implies n/c$ subproblems, each requiring $O(d^c)$

\implies Total work $O(d^c n/c)$

III. The Structure of CSP Problems



Independent subproblems

- **Connected components** in the constraint graph.
- Each subproblem can be solved independently.

n variables
domain size d } \implies Total work $O(d^n)$ without problem decomposition.

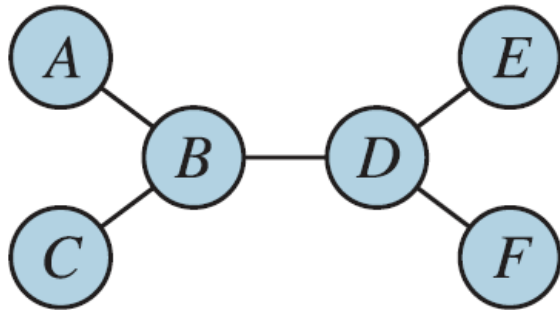
c variables for each subproblem $\implies n/c$ subproblems, each requiring $O(d^c)$

\implies Total work $O(d^c n/c)$

Linear in n .

Tree-Structured CSPs

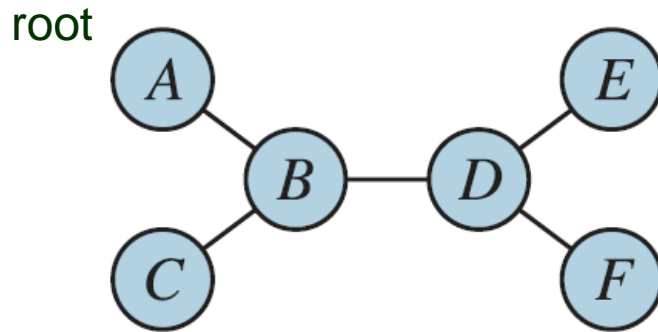
Constraint graph is a tree.



Tree-Structured CSPs

Constraint graph is a tree.

Solution:



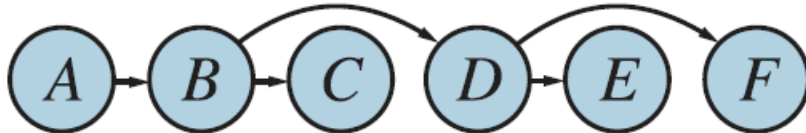
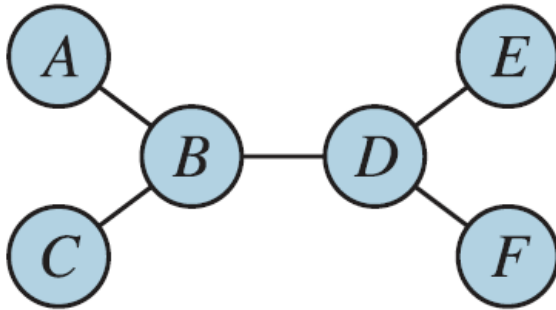
- ◆ Generate a topological order of the variables (pick any variable as the root).

Tree-Structured CSPs

Constraint graph is a tree.

Solution:

root



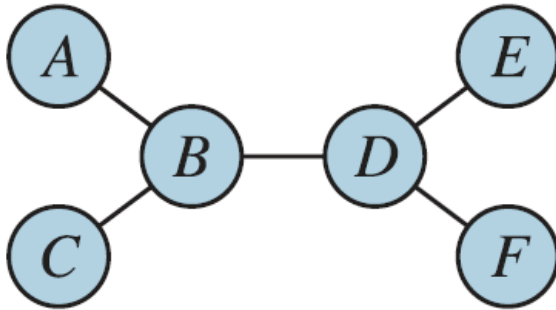
- ◆ Generate a topological order of the variables (pick any variable as the root).

Tree-Structured CSPs

Constraint graph is a tree.

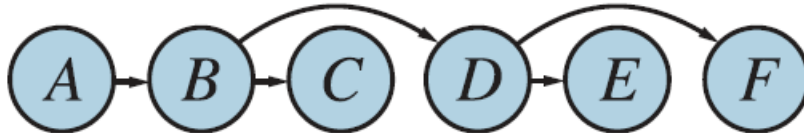
Solution:

root



- ◆ Generate a topological order of the variables (pick any variable as the root).

A, B, C, D, E, F

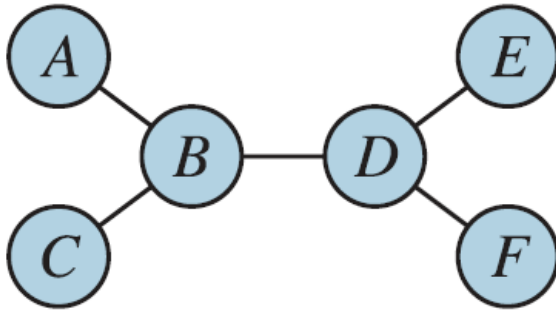


Tree-Structured CSPs

Constraint graph is a tree.

Solution:

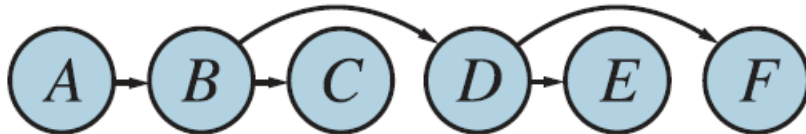
root



- ◆ Generate a topological order of the variables (pick any variable as the root).

A, B, C, D, E, F

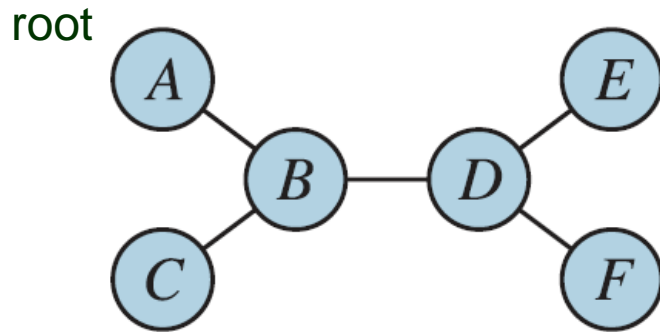
$O(n)$



Tree-Structured CSPs

Constraint graph is a tree.

Solution:



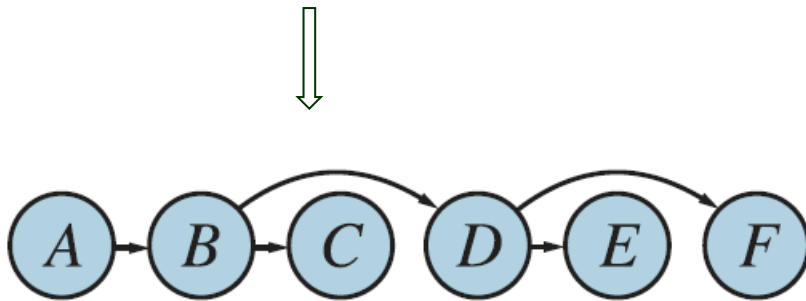
- ◆ Generate a topological order of the variables (pick any variable as the root).

A, B, C, D, E, F

$O(n)$

- ◆ Visit variables in the reverse order. $O(n)$

F, E, D, C, B, A

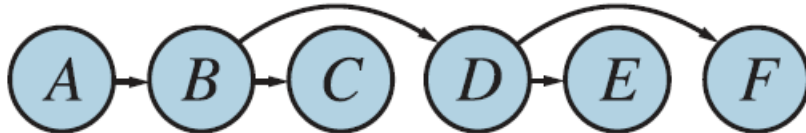
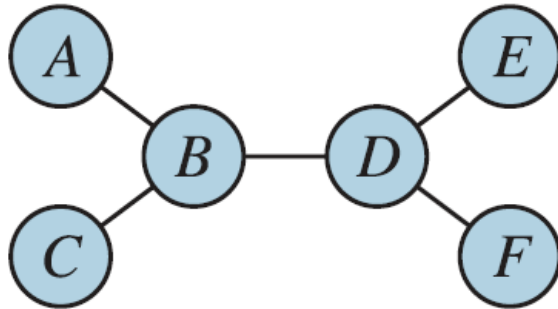


Tree-Structured CSPs

Constraint graph is a tree.

Solution:

root



- ◆ Generate a topological order of the variables (pick any variable as the root).

A, B, C, D, E, F

$O(n)$

- ◆ Visit variables in the reverse order. $O(n)$

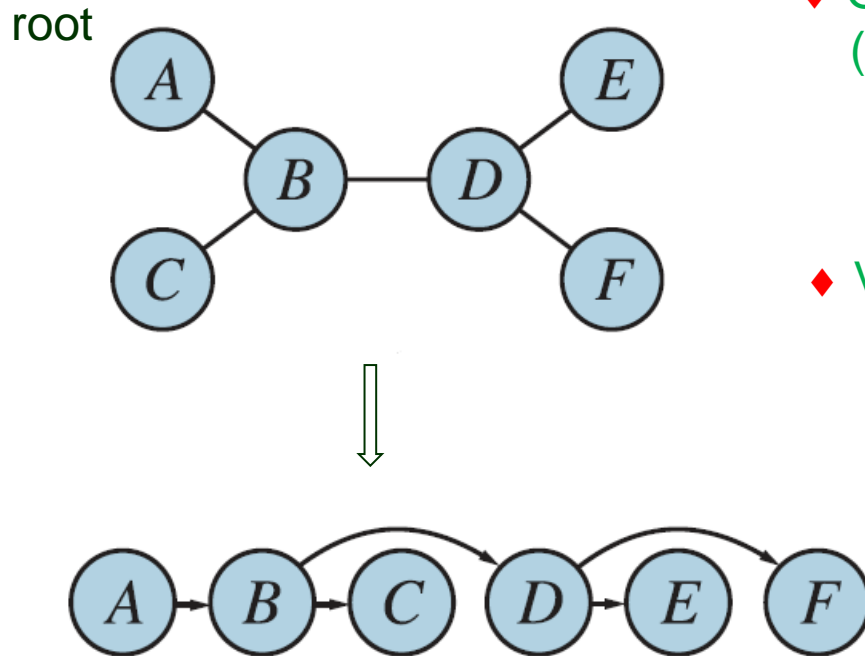
F, E, D, C, B, A

- At each visited vertex v , make every incoming edge (u, v) arc-consistent (i.e., u is arc-consistent w.r.t. v) by reducing the domain of u .

Tree-Structured CSPs

Constraint graph is a tree.

Solution:



- ◆ Generate a topological order of the variables (pick any variable as the root).

A, B, C, D, E, F

$O(n)$

- ◆ Visit variables in the reverse order. $O(n)$

F, E, D, C, B, A

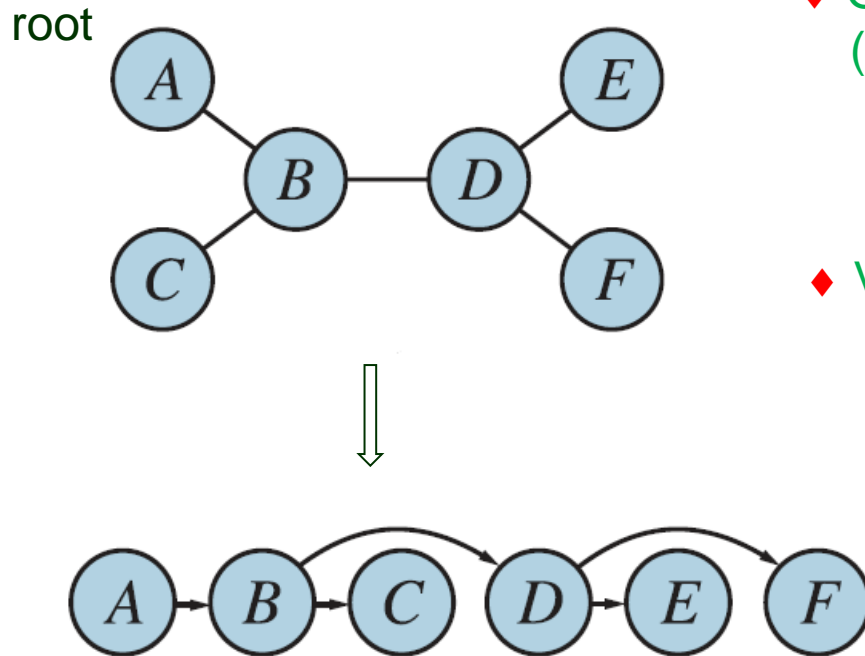
- At each visited vertex v , make every incoming edge (u, v) arc-consistent (i.e., u is arc-consistent w.r.t. v) by reducing the domain of u .

$O(d^2)$ for the edge

Tree-Structured CSPs

Constraint graph is a tree.

Solution:



- ◆ Generate a topological order of the variables (pick any variable as the root).

A, B, C, D, E, F

$O(n)$

- ◆ Visit variables in the reverse order. $O(n)$

F, E, D, C, B, A

- At each visited vertex v , make every incoming edge (u, v) arc-consistent (i.e., u is arc-consistent w.r.t. v) by reducing the domain of u .

$O(d^2)$ for the edge

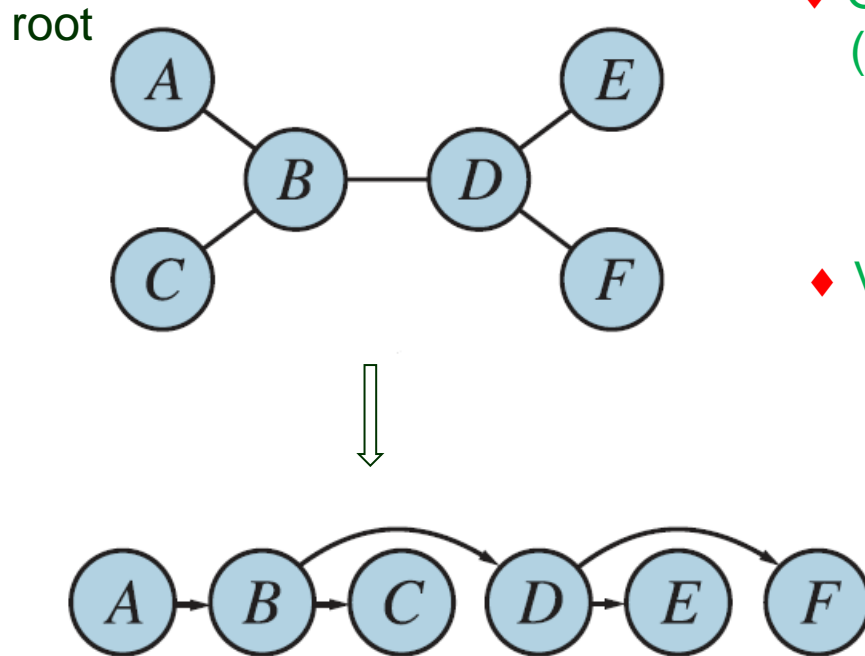
- ◆ Finally, visit variables in the topological order again to assign values sequentially.

A, B, C, D, E, F

Tree-Structured CSPs

Constraint graph is a tree.

Solution:



- ◆ Generate a topological order of the variables (pick any variable as the root).

A, B, C, D, E, F

$O(n)$

- ◆ Visit variables in the reverse order. $O(n)$

F, E, D, C, B, A

- At each visited vertex v , make every incoming edge (u, v) arc-consistent (i.e., u is arc-consistent w.r.t. v) by reducing the domain of u .

$O(d^2)$ for the edge

- ◆ Finally, visit variables in the topological order again to assign values sequentially.

A, B, C, D, E, F

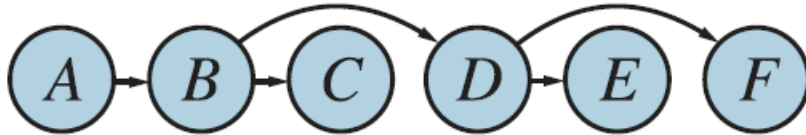
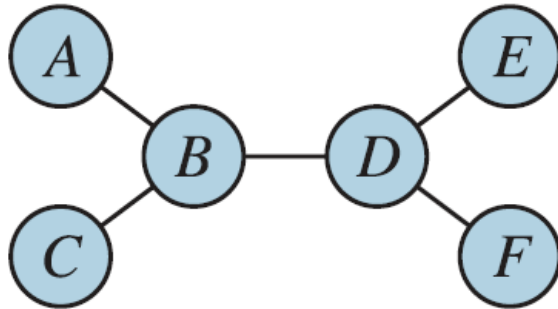
$O(n)$

Tree-Structured CSPs

Constraint graph is a tree.

Solution: $O(nd^2)$

root



- ◆ Generate a topological order of the variables (pick any variable as the root).

A, B, C, D, E, F

$O(n)$

- ◆ Visit variables in the reverse order. $O(n)$

F, E, D, C, B, A

- At each visited vertex v , make every incoming edge (u, v) arc-consistent (i.e., u is arc-consistent w.r.t. v) by reducing the domain of u .

$O(d^2)$ for the edge

- ◆ Finally, visit variables in the topological order again to assign values sequentially.

A, B, C, D, E, F

$O(n)$

Tree CSP Solver

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**

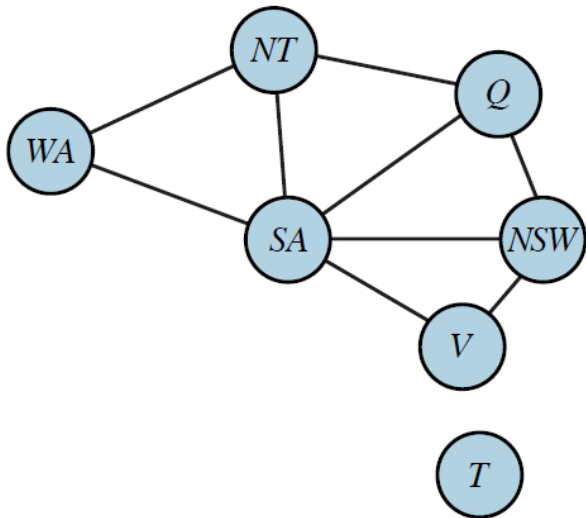
assignment[X_i] \leftarrow any consistent value from D_i

if there is no consistent value **then return** *failure*

return *assignment*

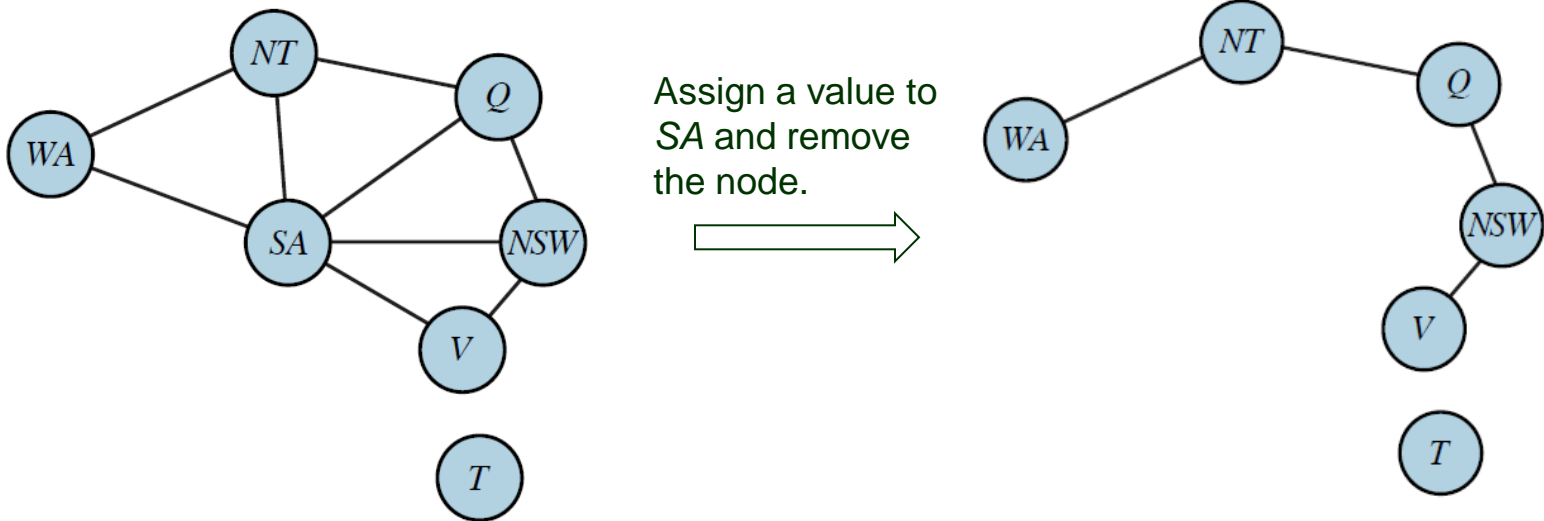
Cutset Conditioning

Reduce a constraint graph to a tree (or a forest) by assigning values to some variables.



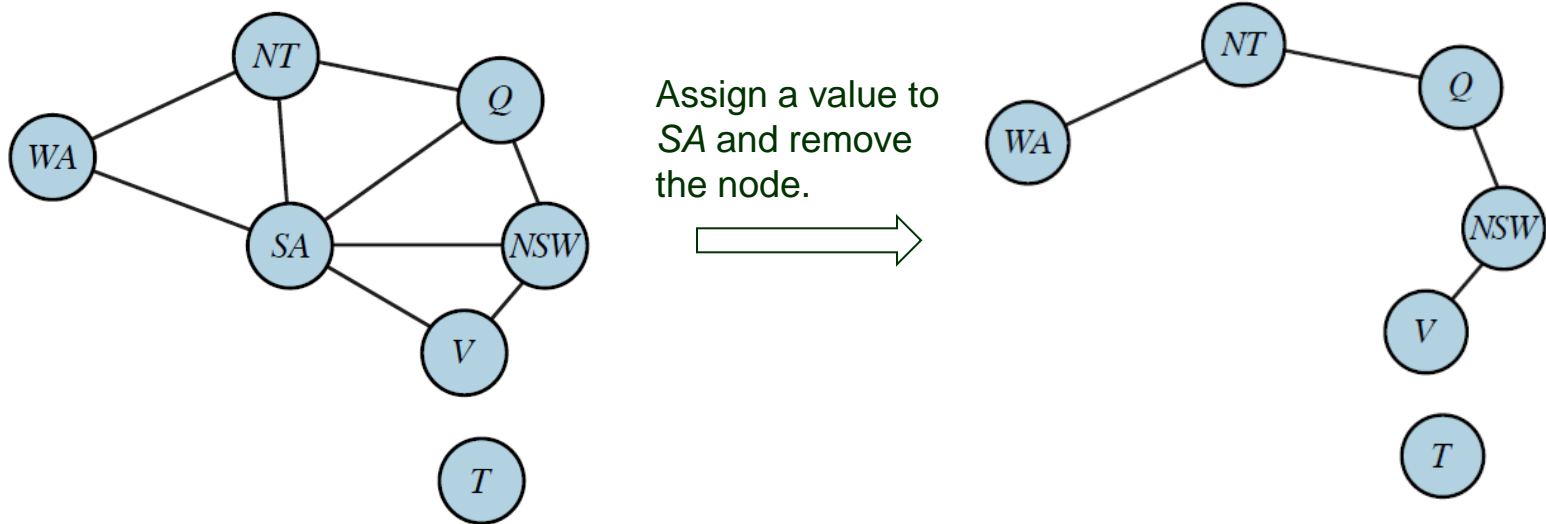
Cutset Conditioning

Reduce a constraint graph to a tree (or a forest) by assigning values to some variables.



Cutset Conditioning

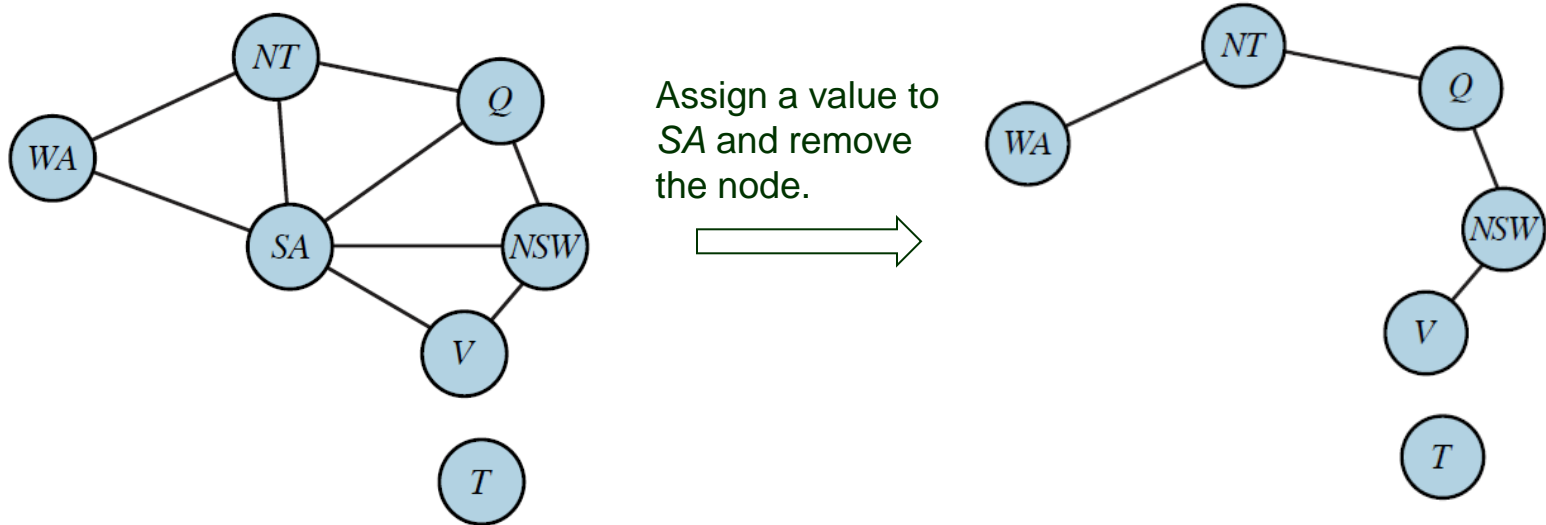
Reduce a constraint graph to a tree (or a forest) by assigning values to some variables.



1. Choose a subset $S \subset \mathcal{X}$ of variables whose removals reduce the constraint graph, now with the set $\mathcal{X} \setminus S$ of variables, to a tree (or a forest). E.g., $S = \{SA\}$.

Cutset Conditioning

Reduce a constraint graph to a tree (or a forest) by assigning values to some variables.



1. Choose a subset $S \subset \mathcal{X}$ of variables whose removals reduce the constraint graph, now with the set $\mathcal{X} \setminus S$ of variables, to a tree (or a forest). E.g., $S = \{SA\}$.
2. For every consistent assignment A to variables in S :
 - remove from the domain of every $X \in \mathcal{X} \setminus S$ all values that are inconsistent with A .
 - return the solution to the reduced CSP (if exists) along with A .

Tree Decomposition

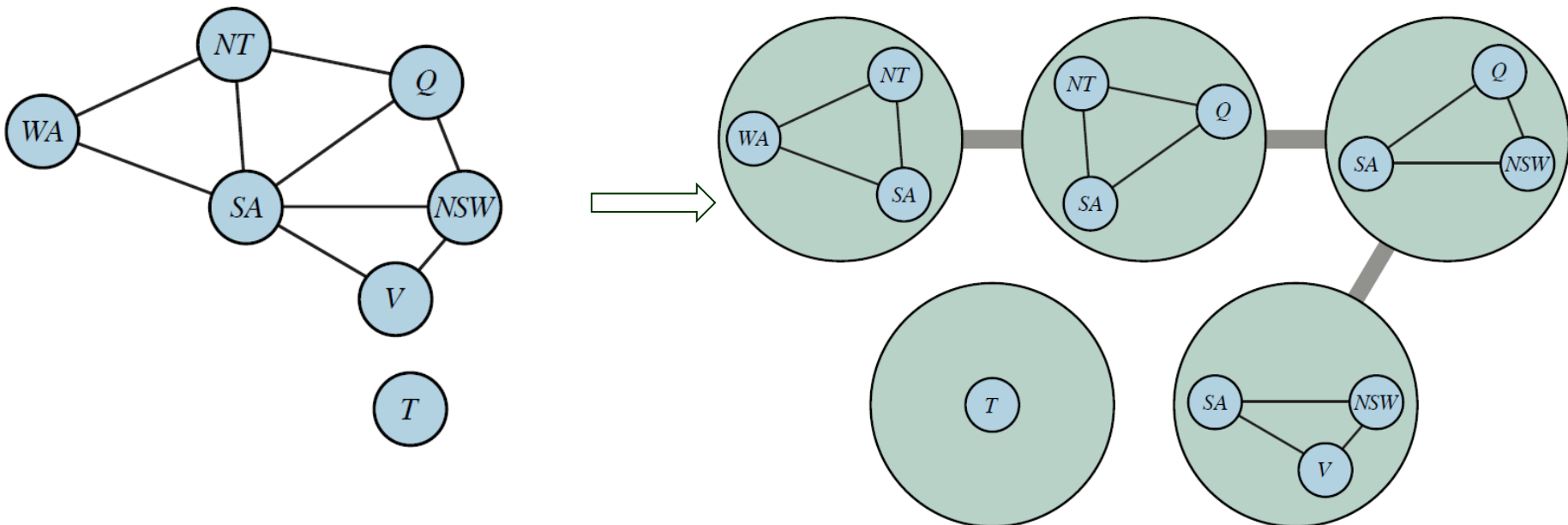
Transform the constraint graph into a tree where each node consists of a set of variables such that

- ♣ Every variable X must appear in at least one tree node n .
- ♣ Two variables X, Y sharing a constraint must appear together in at least one node n .
- ♣ If X appears in two nodes n_1 and n_2 , it must appear in every node on the path connecting n_1 and n_2 .

Tree Decomposition

Transform the constraint graph into a tree where each node consists of a set of variables such that

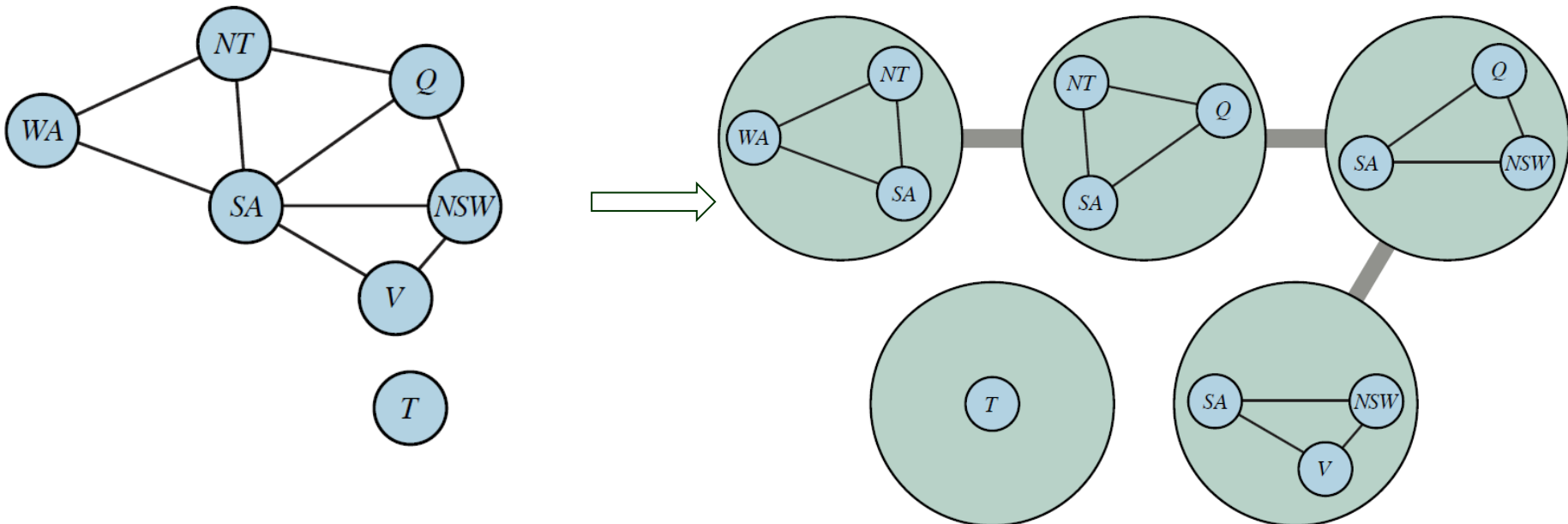
- ♣ Every variable X must appear in at least one tree node n .
- ♣ Two variables X, Y sharing a constraint must appear together in at least one node n .
- ♣ If X appears in two nodes n_1 and n_2 , it must appear in every node on the path connecting n_1 and n_2 .



Tree Decomposition

Transform the constraint graph into a tree where each node consists of a set of variables such that

- All variables & constraints are represented
- ♣ Every variable X must appear in at least one tree node n .
 - ♣ Two variables X, Y sharing a constraint must appear together in at least one node n .
 - ♣ If X appears in two nodes n_1 and n_2 , it must appear in every node on the path connecting n_1 and n_2 .



Tree Decomposition

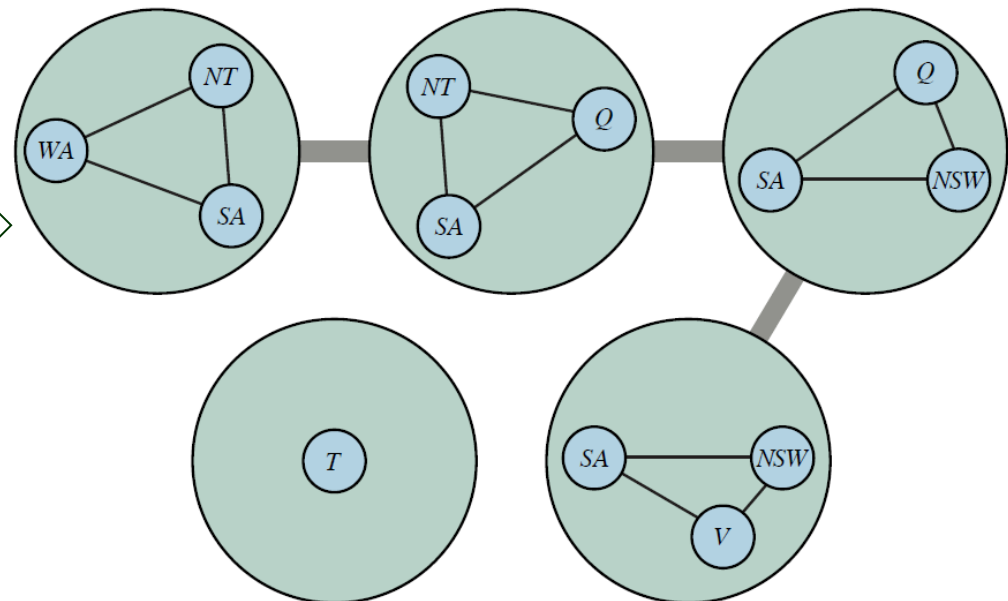
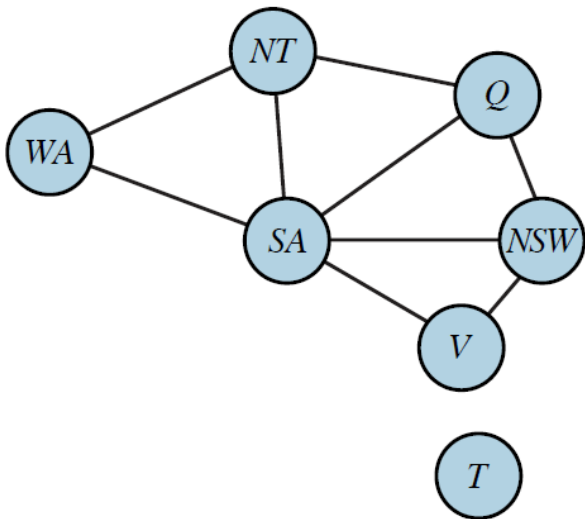
Transform the constraint graph into a tree where each node consists of a set of variables such that

All variables & constraints are represented

- ♣ Every variable X must appear in at least one tree node n .
- ♣ Two variables X, Y sharing a constraint must appear together in at least one node n .

A variable must have the same value everywhere it appears.

- ♣ If X appears in two nodes n_1 and n_2 , it must appear in every node on the path connecting n_1 and n_2 .



Tree Decomposition

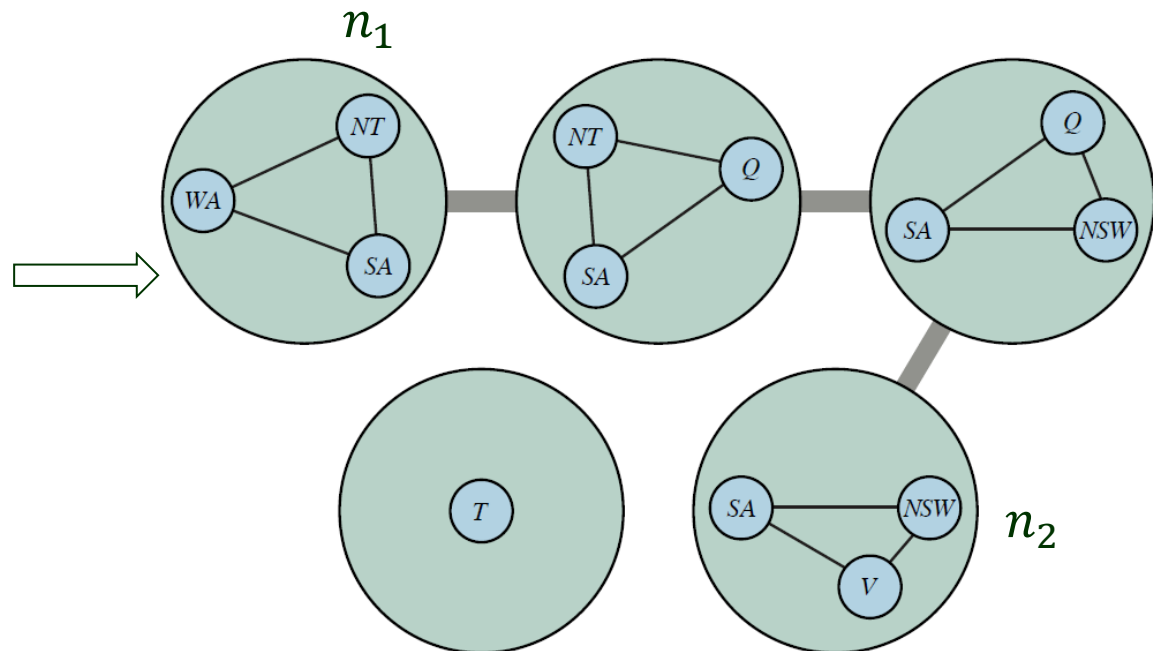
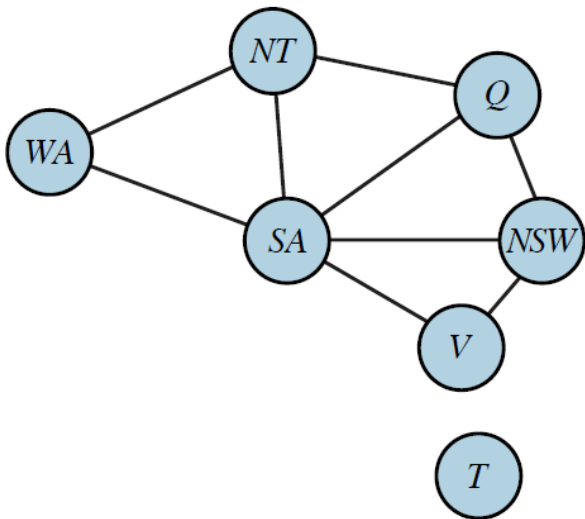
Transform the constraint graph into a tree where each node consists of a set of variables such that

All variables & constraints are represented

- ♣ Every variable X must appear in at least one tree node n .
- ♣ Two variables X, Y sharing a constraint must appear together in at least one node n .

A variable must have the same value everywhere it appears.

- ♣ If X appears in two nodes n_1 and n_2 , it must appear in every node on the path connecting n_1 and n_2 .



Tree Decomposition

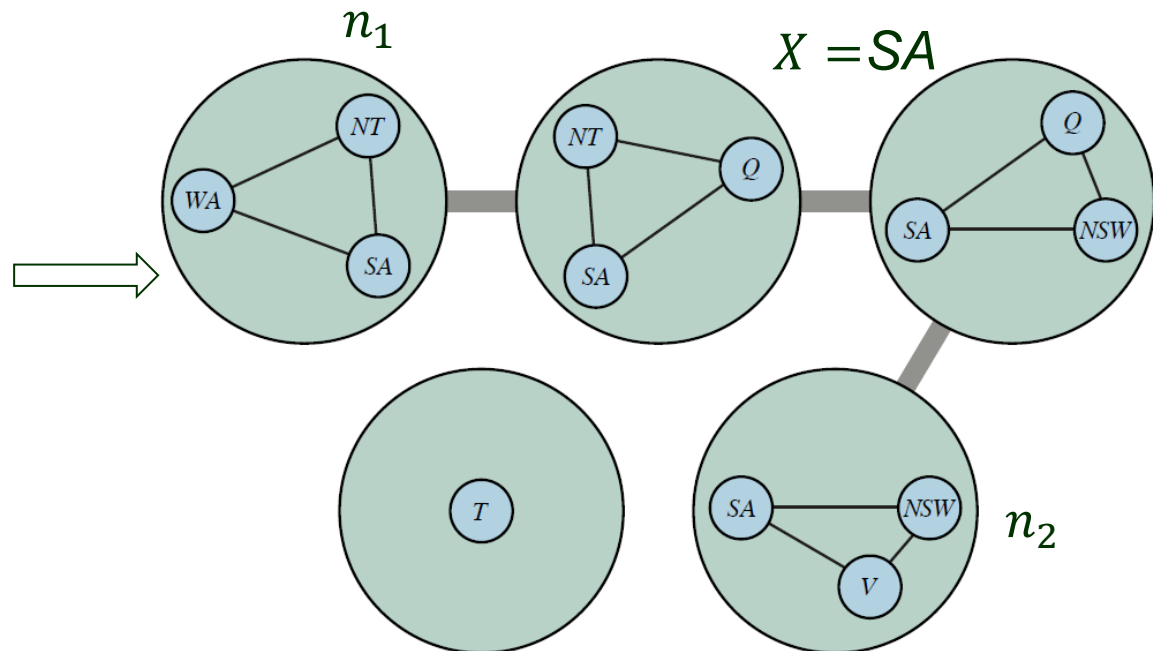
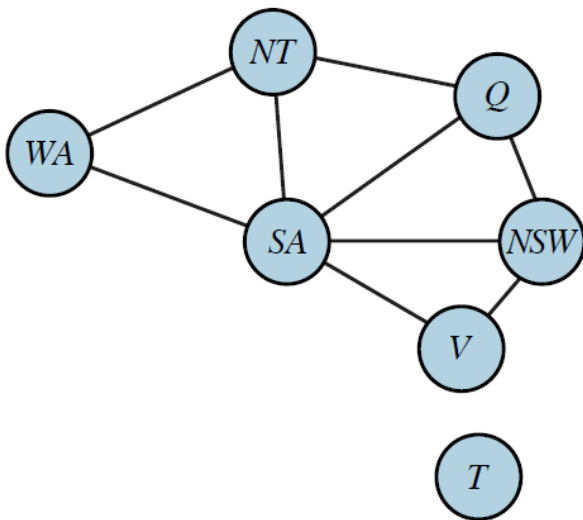
Transform the constraint graph into a tree where each node consists of a set of variables such that

All variables & constraints are represented

- ♣ Every variable X must appear in at least one tree node n .
- ♣ Two variables X, Y sharing a constraint must appear together in at least one node n .

A variable must have the same value everywhere it appears.

- ♣ If X appears in two nodes n_1 and n_2 , it must appear in every node on the path connecting n_1 and n_2 .



Tree Decomposition

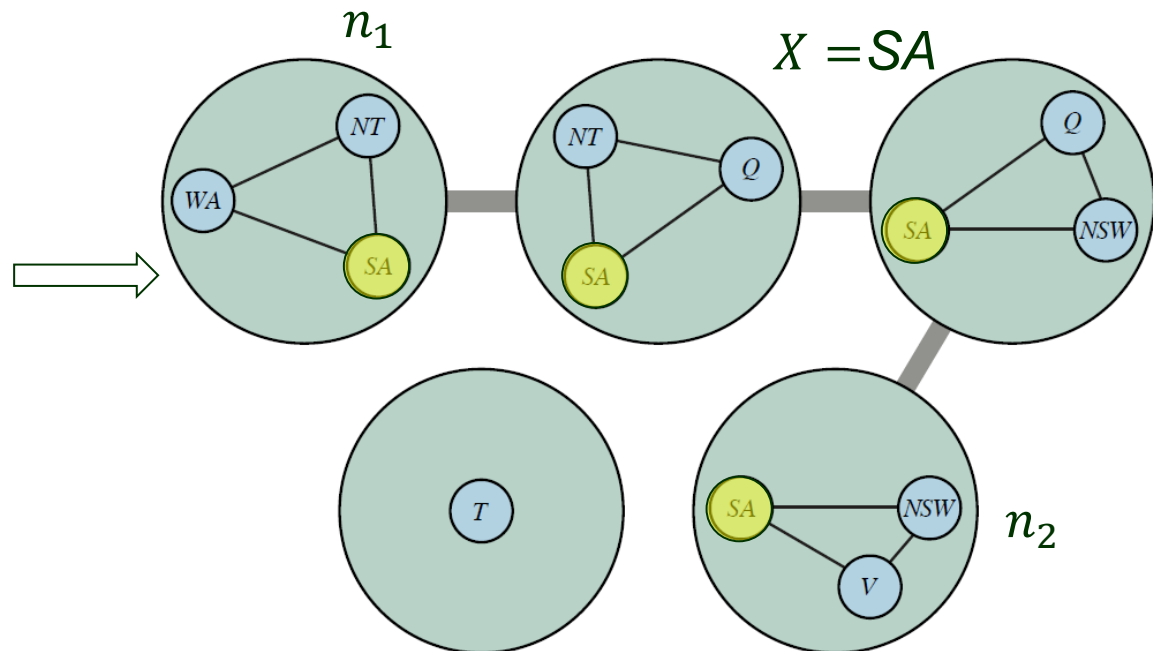
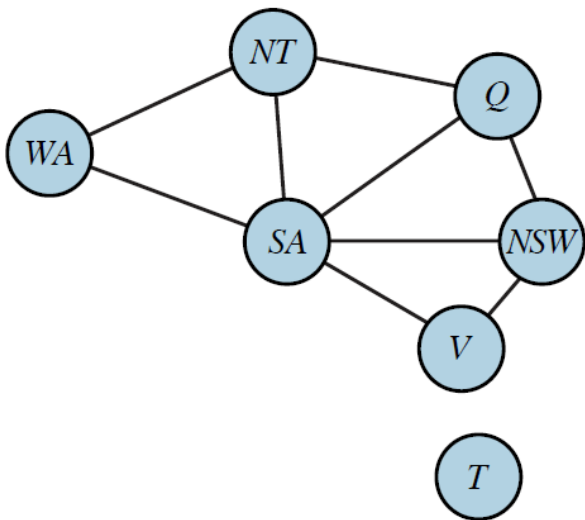
Transform the constraint graph into a tree where each node consists of a set of variables such that

All variables & constraints are represented

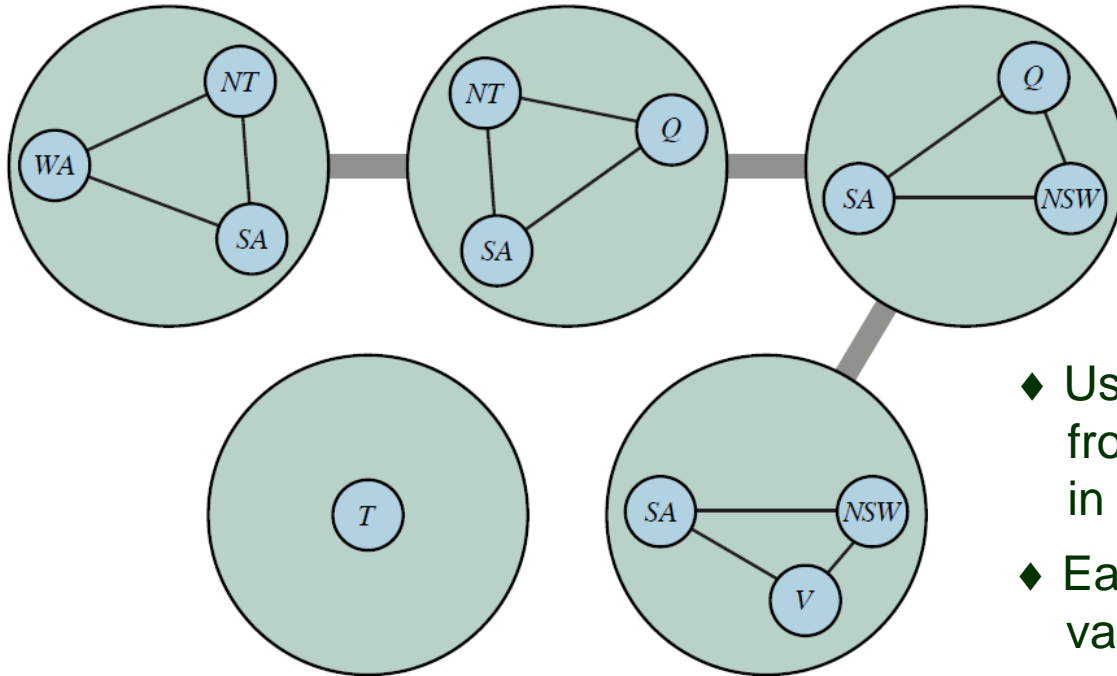
- ♣ Every variable X must appear in at least one tree node n .
- ♣ Two variables X, Y sharing a constraint must appear together in at least one node n .

A variable must have the same value everywhere it appears.

- ♣ If X appears in two nodes n_1 and n_2 , it must appear in every node on the path connecting n_1 and n_2 .



Solution After Tree Decomposition



- ◆ Use the CSP tree solver to move from one tree node to the next in some topological order.
- ◆ Each tree node represents a single variable with values that are m -tuples, where m is # variables in the node.
- ◆ At each tree node, solve the CSP subproblem represented at that node.