Search for CSPs

Outline

I. Backtracking algorithm

II. Local search

III. CSP structure

* Figures/images are from the textbook site (or by the instructor).
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.

\[ n \text{ variables of domain size } d \]
I. Backtracking Search

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

\[ X_1 = v_1 \]
\[ X_2 = v_1 \]
\[ \cdots \]
\[ X_n = v_d \]

Solve a CSP using depth-limited search.

\( n \) variables of 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 a depth-limited search.

\[ n \text{ variables of domain size } d \]

\[ nd \quad (\text{any value can be assigned to any variable at depth } 1) \]
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.

\[ X_1 = v_1 \quad \cdots \quad X_1 = v_d \quad X_2 = v_1 \quad \cdots \quad X_n = v_d \]

\[ (n - 1)d \]

\[ nd \] (any value can be assigned to any variable at depth 1)
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.

\[ \text{#leaves} = n! \cdot d^n \]
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.

\[ \text{ Solve a CSP using depth-limited search.} \]

\[ \begin{array}{c}
X_1 = v_1 & \cdots & X_1 = v_d & X_2 = v_1 & \cdots & X_n = v_d \\
\end{array} \]

\[ \text{start} \]

\[ (n-1)d \]

\[ nd \text{ (any value can be assigned to any variable at depth 1)} \]

\[ \text{#leaves} = n! \cdot d^n \]

But \text{#assignments} = d^n.
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 of domain size $d$

$\text{#leaves} = n! \cdot d^n$

But $\text{#assignments} = d^n$.

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.

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: $NSW = \text{red}$
Step 2: $SA = \text{blue}$ and
Step 1: $SA = \text{blue}$
Step 2: $NSW = \text{red}$

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

\[
\begin{align*}
\text{Step 1: } NSW &= \text{ red} \\
\text{Step 2: } SA &= \text{ blue}
\end{align*}
\]

and

\[
\begin{align*}
\text{Step 1: } SA &= \text{ blue} \\
\text{Step 2: } NSW &= \text{ red}
\end{align*}
\]

Need only consider a single variable at each node.

At the root choose between

\[
SA = \text{ blue, } SA = \text{ red, and } SA = \text{ green}
\]

but not between

\[
SA = \text{ blue and } NSW = \text{ red}
\]
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 ← 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 ← INERENCE(csp, var, assignment)
    if inferences ≠ failure then
      add inferences to csp
      result ← BACKTRACK(csp, assignment)
      if result ≠ failure then return result
      remove inferences from csp
    remove \{var = value\} from assignment
  return failure
Order of Variables

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

In what order should we choose the variables?
Order of Variables

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

In what order should we choose the variables?

- Static order: \( \{X_1, X_2, \ldots \} \)?
Order of Variables

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

In what order should we choose the variables?

- Static order: \( \{X_1, X_2, \ldots \} \)?
- Random order: \( \{X_1, X_2, \ldots \} \)?
Order of Variables

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

In what order should we choose the variables?

- Static order: \( \{X_1, X_2, \ldots \} \)?
- Random order: \( \{X_1, X_2, \ldots \} \)?

Neither is optimal!
Order of Variables

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

In what order should we choose the variables?

- Static order: \( \{X_1, X_2, \ldots \} \)?
- Random order: \( \{X_1, X_2, \ldots \} \)?

Neither is optimal!
Order of Variables

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

In what order should we choose the variables?

- Static order: \( \{X_1, X_2, \ldots \} \) ?
- Random order: \( \{X_1, X_2, \ldots \} \) ?

Neither is optimal!

It makes more sense to assign \( SA = blue \) than assigning \( Q \).
Minimum-remaining-values (MRV): Choose the variable with the fewest “legal” values.

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

*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.
**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.

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

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

\[
\text{deg}(SA) = 5 \\
\text{Others have degrees } \leq 3.
\]
**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.

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

\[ \text{deg}(SA) = 5 \]
\[ \text{Others have degrees } \leq 3. \]

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?

- *red*
- *green*
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** $X = v$

Diagram:
- **$X$** (unassigned)
- **$Y$** (unassigned)
- **Assigned**
Forward Checking

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

Assignment $X = \nu$

For every unassigned $Y$ connected to $X$, delete any value from $Y$’s domain that is inconsistent with $\nu$. 
Backtracking with Forward Checking

Initial domains

Diagram of constraints and domains for variables WA, NT, Q, NSW, V, SA, and T.
Backtracking with Forward Checking

Initial domains

- WA = red
Backtracking with Forward Checking

Initial domains

- **WA** = *red*

Deletes *red* for **NT** and **SA**.
Backtracking with Forward Checking

<table>
<thead>
<tr>
<th>WA</th>
<th>NT</th>
<th>Q</th>
<th>NSW</th>
<th>V</th>
<th>SA</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Initial domains

After $WA = red$

- $WA = red$
  
  Deletes *red* for $NT$ and $SA$.  

Diagram:

- $WA$ is connected to $NT$, $Q$, $SA$, and $NSW$.
- $NT$ is connected to $Q$ and $SA$.
- $Q$ is connected to $NSW$.
- $SA$ is connected to $NSW$.
- $V$ is connected to $NSW$.
- $T$ is isolated.
Backtracking with Forward Checking

- $WA = red$
  - Deletes red for $NT$ and $SA$.
- $Q = green$
Backtracking with Forward Checking

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

- **Q = green**
  Deletes *green* for *NT*, *SA*, and *NSW*.
### Backtracking with Forward Checking

<table>
<thead>
<tr>
<th></th>
<th>WA</th>
<th>NT</th>
<th>Q</th>
<th>NSW</th>
<th>V</th>
<th>SA</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial domains</strong></td>
<td><img src="image1.png" alt="Domains" /></td>
<td><img src="image2.png" alt="Domains" /></td>
<td><img src="image3.png" alt="Domains" /></td>
<td><img src="image4.png" alt="Domains" /></td>
<td><img src="image5.png" alt="Domains" /></td>
<td><img src="image6.png" alt="Domains" /></td>
<td><img src="image7.png" alt="Domains" /></td>
</tr>
<tr>
<td><strong>After WA=red</strong></td>
<td><img src="image1.png" alt="Domains" /></td>
<td><img src="image2.png" alt="Domains" /></td>
<td><img src="image3.png" alt="Domains" /></td>
<td><img src="image4.png" alt="Domains" /></td>
<td><img src="image5.png" alt="Domains" /></td>
<td><img src="image6.png" alt="Domains" /></td>
<td><img src="image7.png" alt="Domains" /></td>
</tr>
<tr>
<td><strong>After Q=green</strong></td>
<td><img src="image1.png" alt="Domains" /></td>
<td><img src="image2.png" alt="Domains" /></td>
<td><img src="image3.png" alt="Domains" /></td>
<td><img src="image4.png" alt="Domains" /></td>
<td><img src="image5.png" alt="Domains" /></td>
<td><img src="image6.png" alt="Domains" /></td>
<td><img src="image7.png" alt="Domains" /></td>
</tr>
</tbody>
</table>

- **WA = red**
  - Deletes *red* for *NT* and *SA*.
- **Q = green**
  - Deletes *green* for *NT*, *SA*, and *NSW*.
Backtracking with Forward Checking

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

- **Q = green**
  Deletes *green* for *NT, SA, and NSW*.
  *NT* & *SA* each have a single value.
Backtracking with Forward Checking

- **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

- **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

<table>
<thead>
<tr>
<th>Initial domains</th>
<th>WA</th>
<th>NT</th>
<th>Q</th>
<th>NSW</th>
<th>V</th>
<th>SA</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image1" alt="Initial domains" /></td>
<td><img src="image2" alt="Initial domains" /></td>
<td><img src="image3" alt="Initial domains" /></td>
<td><img src="image4" alt="Initial domains" /></td>
<td><img src="image5" alt="Initial domains" /></td>
<td><img src="image6" alt="Initial domains" /></td>
<td><img src="image7" alt="Initial domains" /></td>
</tr>
</tbody>
</table>

- **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

- **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.

<table>
<thead>
<tr>
<th>WA</th>
<th>NT</th>
<th>Q</th>
<th>NSW</th>
<th>V</th>
<th>SA</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Initial domains

After WA = red

- *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: $\mathcal{X} = \{Q_1, Q_2, ..., Q_8\}$

$Q_i$: the row number of the queen placed in the $i$th column.
Applying Min-conflicts to 8-Queen

Variable set: \( \mathcal{X} = \{Q_1, Q_2, \ldots, Q_8\} \)

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

\( Q_8 = 8 \)
Applying Min-conflicts to 8-Queen

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

$Q_i$: 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: $X = \{Q_1, Q_2, ..., Q_8\}$

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

$Q_8 = 8$ → 2 conflicts if $Q_8 = 7$

$Q_8$ out of the set $\{Q_4, Q_8\}$ of conflicted variables by a random choice.
Applying Min-conflicts to 8-Queen

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

$Q_i$: 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.
- The 8th queen in row 3 or 6 would violate only one constraint.

$Q_8 = 8$ ← 2 conflicts if $Q_8 = 7$
Applying Min-conflicts to 8-Queen

Variable set: \(X = \{Q_1, Q_2, \ldots, Q_8\}\)

\(Q_i\): 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.
- The 8\(^{th}\) queen in row 3 or 6 would violate only one constraint.
- Move the queen to, say, row 3.

\(Q_8 = 8\) ← 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

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

Solution
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

Independent subproblems

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

\( n \) variables
domain size \( d \)
III. The Structure of CSP Problems

Independent subproblems

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

\[ n \text{ variables} \quad \text{domain size} \quad d \\quad \rightarrow \quad \text{Total work } O(d^n) \text{ without problem decomposition.} \]
III. The Structure of CSP Problems

Independent subproblems

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

\[ n \text{ variables, domain size } d \rightarrow \text{ Total work } O(d^n) \text{ without problem decomposition.} \]

\[ c \text{ variables for each subproblem} \]
III. The Structure of CSP Problems

Independent subproblems

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

\[ n \text{ variables, domain size } d \quad \Rightarrow \quad \text{Total work } O(d^n) \text{ without problem decomposition.} \]

\[ c \text{ variables for each subproblem} \quad \Rightarrow \quad n/c \text{ subproblems, each requiring } O(d^c) \]
III. The Structure of CSP Problems

Independent subproblems

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

\[ n \text{ variables, domain size } d \quad \implies \quad \text{Total work } O(d^n) \text{ without problem decomposition.} \]

\[ c \text{ variables for each subproblem } \implies \quad n/c \text{ subproblems, each requiring } O(d^c) \]

\[ \implies \quad \text{Total work } O(d^c n/c) \]
III. The Structure of CSP Problems

Independent subproblems

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

\[ n \text{ variables, domain size } d \implies \text{Total work } O(d^n) \text{ without problem decomposition.} \]

\[ c \text{ variables for each subproblem} \implies n/c \text{ subproblems, each requiring } O(d^c) \]

\[ \implies \text{Total work } O(d^c n/c) \text{ Linear in } n. \]
Tree-Structured CSPs

Constraint graph is a tree.

root

- A
- B
- C
- D
- E
- F
Tree-Structured CSPs

Constraint graph is a tree.

Solution:

- Generate a topological order of the variables.
Tree-Structured CSPs

Constraint graph is a tree.

Solution:
- Generate a topological order of the variables.
Constraint graph is a tree.

Solution:
- Generate a topological order of the variables.

\[ O(n) \]
Tree-Structured CSPs

Constraint graph is a tree.

Solution:

- Generate a topological order of the variables. \( O(n) \)
- Visit variables in the order. \( O(n) \)
Tree-Structured CSPs

Constraint graph is a tree.

Solution:

- Generate a topological order of the variables. $O(n)$
- Visit variables in the order. $O(n)$
- At each visited vertex, make every outgoing edge arc-consistent by reducing the domains of its two vertices.
Tree-Structured CSPs

Constraint graph is a tree.

Solution:
- Generate a topological order of the variables. \(O(n)\)
- Visit variables in the order. \(O(n)\)
- At each visited vertex, make every outgoing edge arc-consistent by reducing the domains of its two vertices. \(O(d^2)\)
Tree-Structured CSPs

Constraint graph is a tree.

Solution:

- Generate a topological order of the variables. $O(n)$
- Visit variables in the order. $O(n)$
  - At each visited vertex, make every outgoing edge arc-consistent by reducing the domains of its two vertices. $O(d^2)$
- Finally, visit variables in the topological order again and choose any value from its reduced domain,
Tree-Structured CSPs

Constraint graph is a tree.

Solution:

- Generate a topological order of the variables. \( O(n) \)
- Visit variables in the order. \( O(n) \)
- At each visited vertex, make every outgoing edge arc-consistent by reducing the domains of its two vertices. \( O(d^2) \)
- Finally, visit variables in the topological order again and choose any value from its reduced domain, \( O(n) \)
Tree-Structured CSPs

Constraint graph is a tree.

Solution: $O(nd^2)$

- Generate a topological order of the variables. $O(n)$
- Visit variables in the order. $O(n)$
- At each visited vertex, make every outgoing edge arc-consistent by reducing the domains of its two vertices. $O(d^2)$
- Finally, visit variables in the topological order again and choose any value from its reduced domain, $O(n)$
function TREE-CSP-SOLVER(csp) returns a solution, or failure
inputs: csp, a CSP with components X, D, C

n ← number of variables in X
assignment ← an empty assignment
root ← any variable in X
X ← 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] ← 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.

Assign a value to \( SA \) and remove the node.
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 to a tree (or a forest).

Assign a value to $SA$ and remove the node.
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 to a tree (or a forest).

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 $A$. 

---

**Diagram:**

- **Before:**
  - Variables: WA, SA, NT, Q, NSW, V, T
  - Constraint graph with connections between variables.

- **After:**
  - Variables: WA, NT, Q, NSW, V, T
  - Constraint graph with nodes SA and T removed, indicating the removal of a cutset.

**Assign a value to SA and remove the node.**
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

- 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 $n_2$. 

All variables & constraints are represented.

![Diagram of Tree Decomposition]
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 $n_2$.

A variable must have the same value everywhere it appears.
Solution After Tree Decomposition

- Use the CSP tree solver to move from one tree node to the next in some topological order.

- At each tree node, solve the CSP subproblem represented at that node.
Solution After Tree Decomposition

- Use the CSP tree solver to move from one tree node to the next in some topological order.
- At each tree node, solve the CSP subproblem represented at that node.