Artificial Intelligence

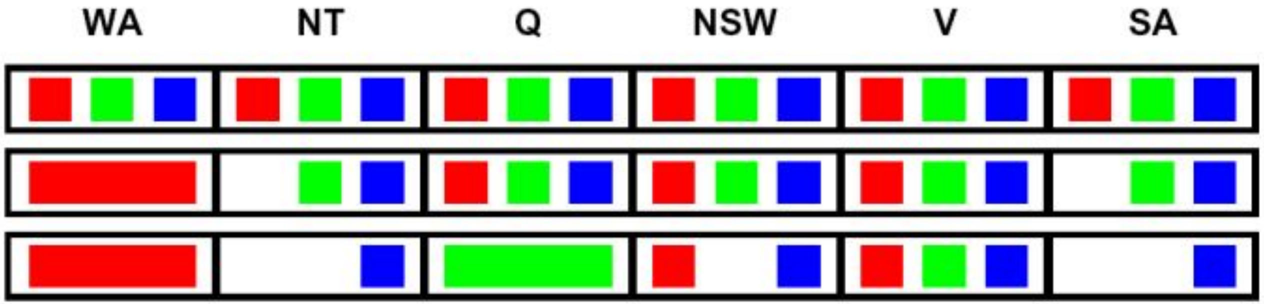
Lec 14: Constraint Satisfaction Problems (contd.)

Pratik Mazumder

Content taken from various sources including
CS188 Intro to AI at UC Berkeley

Filtering: Constraint Propagation

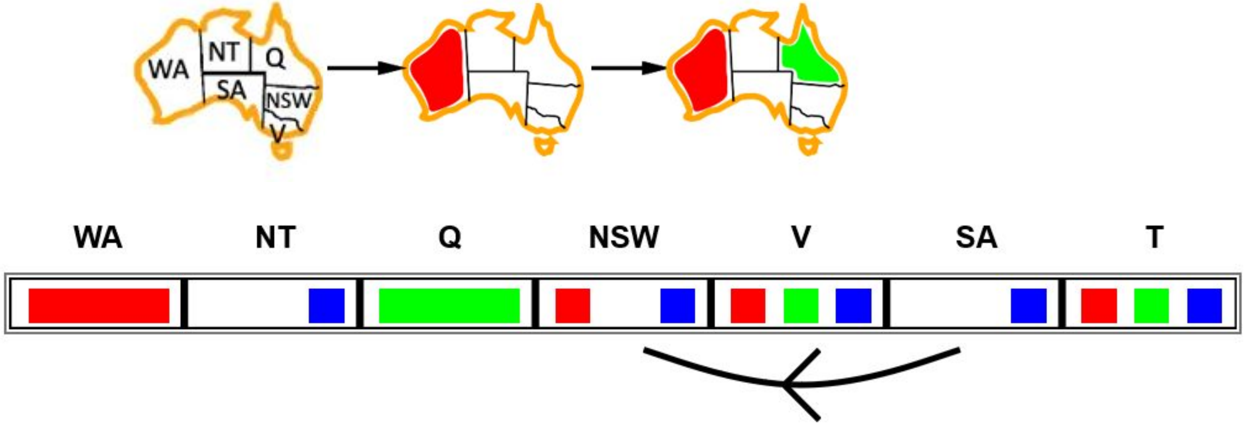
- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:



- NT and SA cannot both be blue!
- Why didn’t we detect this yet?
- Constraint propagation: reason from constraint to constraint

Consistency of A Single Arc

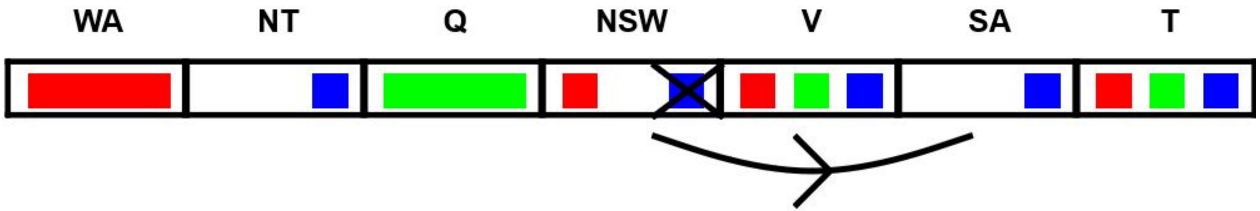
- Simplest form of propagation makes each arc consistent
- An arc X -> Y is consistent iff for every value x of X there is some y which could be assigned without violating a constraint



- arc X -> Y
    - X = SA, Y = NSW
    - If SA = blue: we could assign NSW = red

Single Arc Consistency to Arc Consistency of an Entire CSP

- Simplest form of propagation makes each arc consistent
- An arc X -> Y is consistent iff for every value x of X there is some y which could be assigned without violating a constraint



- arc X -> Y
  - X = NSW, Y = SA
  - If NSW = red: we could assign SA = blue
  - If NSW = blue: there is no remaining assignment to SA that we can use

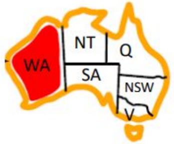
  - Deleting NSW = blue from X makes this arc consistent.
- **Important: If X loses a value, neighbors of X need to be rechecked.**

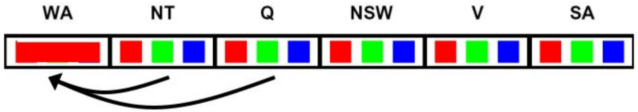
Arc Consistency

- An arc X -> Y is consistent iff for every x in the tail there is some y in the head which could be assigned without violating a constraint.

  OR

- An arc X -> Y is consistent iff for every value x of X there is some y which could be assigned without violating a constraint.

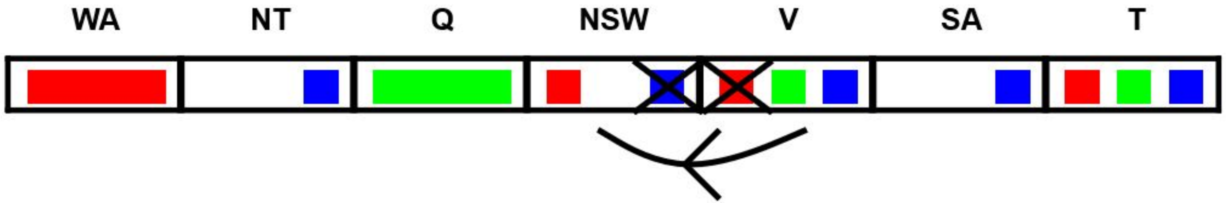




Delete from the tail!

Single Arc Consistency to Arc Consistency of an Entire CSP

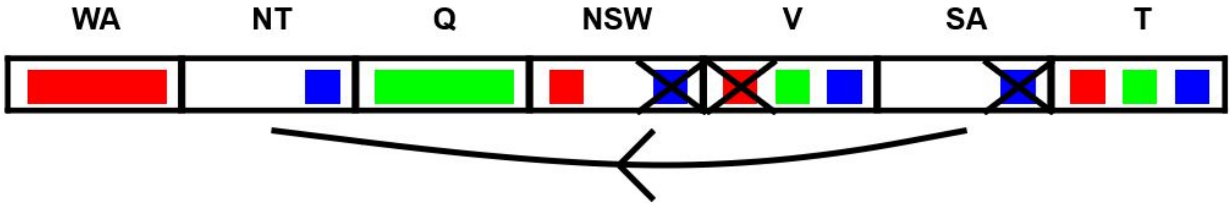
- Simplest form of propagation makes each arc consistent.
- An arc X -> Y is consistent iff for every value x of X there is some y which could be assigned without violating a constraint



- arc X -> Y
  - X = V, Y = NSW
  - If V = red: there is no remaining assignment to NSW that we can use.
  - If V = green: we could assign NSW= red.
  - If V = blue: we could assign NSW= red.
  - Deleting from **tail** V = red from X makes this arc consistent.
- **Important: If X loses a value, neighbors of X need to be rechecked.**

Single Arc Consistency to Arc Consistency of an Entire CSP

- Simplest form of propagation makes each arc consistent
- An arc X -> Y is consistent iff for every value x of X there is some y which could be assigned without violating a constraint



- arc X -> Y
  - X = SA, Y = NT
  - If SA = blue: there is no remaining assignment to NT that we can use.
  - Deleting from **tail** SA = red from X will result in no available colors for SA.
- **Arc consistency detects failure earlier than forward checking.**
- In fact, **forward checking** is **only enforcing arc consistency** for **arcs pointing to a new assignment**.
- Arc consistency can be run as a preprocessor or after each assignment.

Arc Consistency of an Entire CSP

Assignment: When a variable X is assigned a value during the CSP solving process, this assignment may affect the consistency of other variables that are connected to X through constraints.

Arc Consistency Checking:

- **Direct Arcs:**
  a. The algorithm first checks the arcs (constraints) directly involving X and any other variable Y (i.e., X→Y, and Y→X).
  b. The goal is to ensure that the current assignment of X does not violate these constraints.
- **Neighboring Variables:**
  a. If any values in the domain of Y (neighboring variables of X) are found to be inconsistent with the new value of X, those values are removed from Y's domain.
- **Propagation:**
  a. If the domain of Y is reduced, this reduction may, in turn, affect other variables connected to Y.
  b. The algorithm then needs to **check/re-check** the arcs involving these affected variables as well.
  c. This propagation continues until no more domain reductions occur.
  d. All Arcs Checked/Re-checked: **Even if no domain values are removed** during a consistency check, the algorithm **still needs to verify that all the arcs are consistent**.
     i. This ensures that all constraints are satisfied and no values violate the constraints across the entire CSP.

- **Termination:** The process of arc consistency checking and propagation continues **until no more values can be removed from any variable's domain and all arcs have been checked for consistency**.

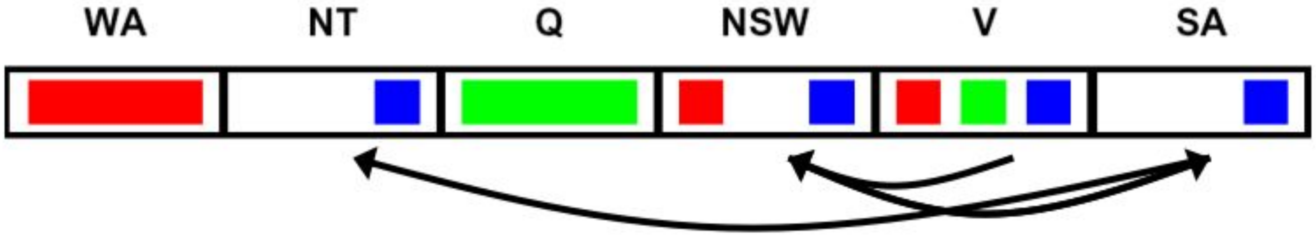
Enforcing Arc Consistency in a CSP

```
function AC-3( csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do
      (Xi, Xj) ← REMOVE-FIRST(queue)
      if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
         for each Xk in NEIGHBORS[Xi] do
            add (Xk, Xi) to queue
```

```
function REMOVE-INCONSISTENT-VALUES( Xi, Xj) returns true iff succeeds
   removed ← false
   for each x in DOMAIN[Xi] do
      if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint Xi ↔ Xj
         then delete x from DOMAIN[Xi];  removed ← true
   return removed
```

Arc Consistency of an Entire CSP

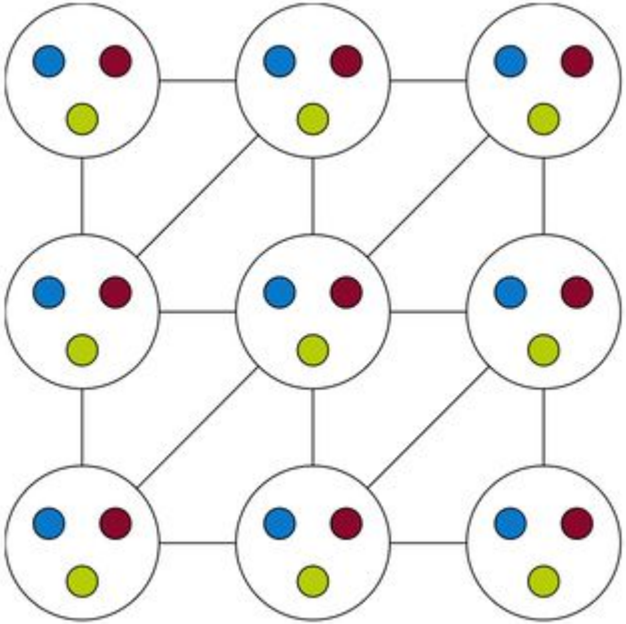


- What's the downside of enforcing arc consistency?
  - Slow

Remember: Delete from the tail!

X→Y

Arc Consistency of an Entire CSP



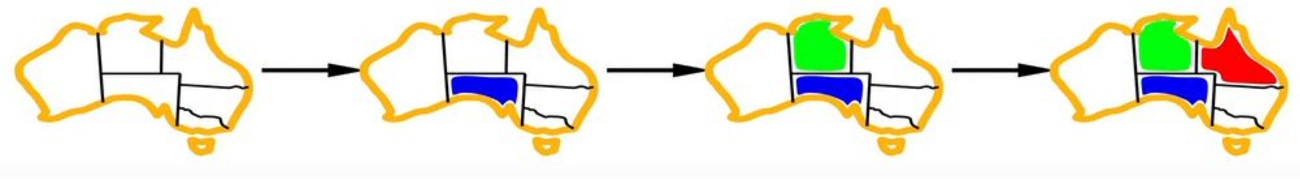
Improving Backtracking (contd.)

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
   return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
   if assignment is complete then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
   for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
       if value is consistent with assignment given CONSTRAINTS[csp] then
           add {var = value} to assignment
           result ← RECURSIVE-BACKTRACKING(assignment, csp)
           if result ≠ failure then return result
           remove {var = value} from assignment
   return failure
```

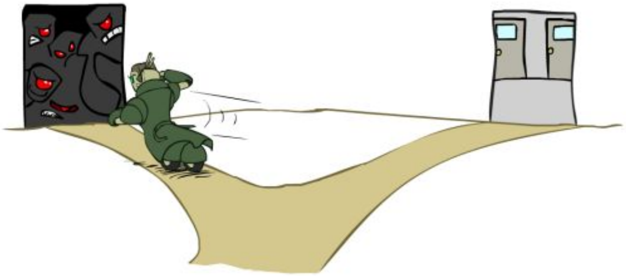
Improving Backtracking: Ordering- Minimum Remaining Values

- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal left values in its domain (most likely the neighbors).



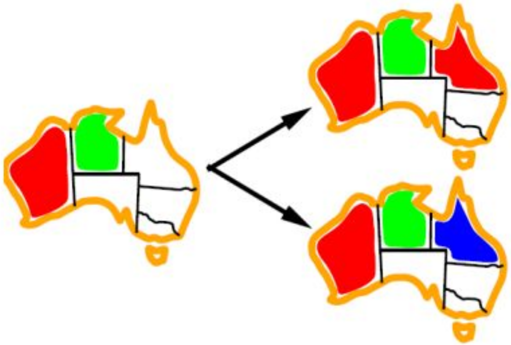
- Why min rather than max?
  - Charging into the hard problem first.
  - So that if I am wrong, then I can backtrack earlier.

- Also called “most constrained variable”.
- “Fail-fast” ordering.

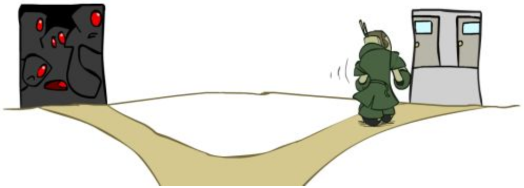


Improving Backtracking: Ordering- Least Constraining Value

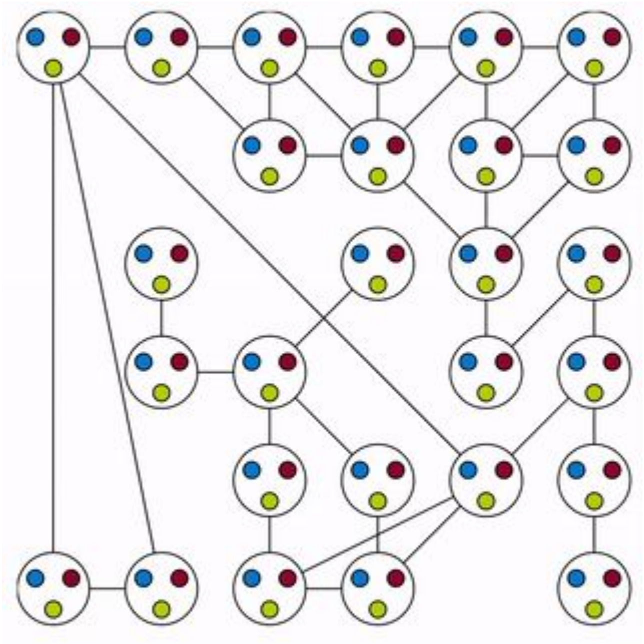
- Value Ordering: Least Constraining Value (LCV)
    - Given a choice of variable, choose the least constraining value.
    - i.e., the one that rules out the fewest values in the remaining variables
    - Note that it may take some computation to determine this! (E.g., re-running filtering)



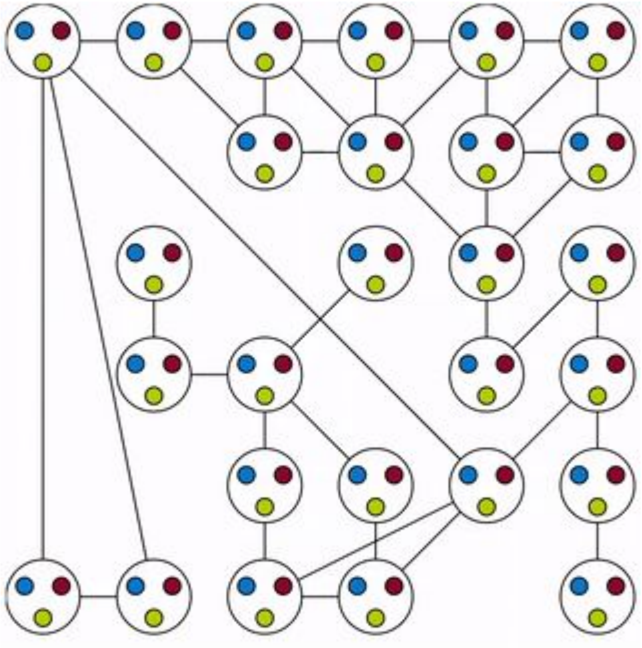
- Why least rather than most?
    - Less likely to remove all values of another variable and force a backtrack.
    - Also, in the hope that I don’t have to try the hard ones ever.



Improving Backtracking: Ordering



Backtracking with Arc Consistency

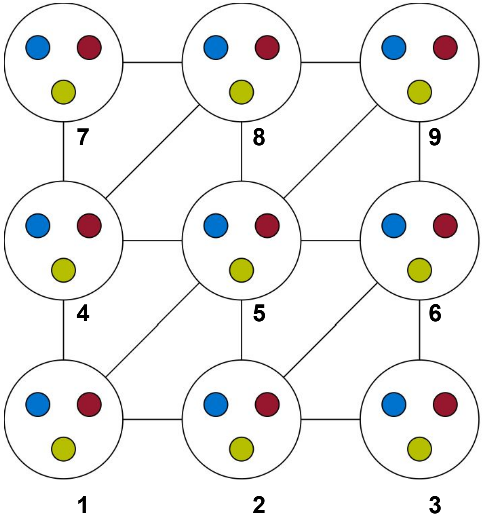


Backtracking with Arc Consistency with MRV and LCV

Practice Question

Consider the graph coloring problem with 9 nodes using only red, green and blue colors. No connected nodes should have the same color.

Sol: [Link]



	R	G	B	Order
1				
2				
3				
4				
5				
6				
7				
8				
9				

Solve using CSP with backtracking with **1) forward consistency 2) arc consistency + MRV**. Fill the Table. Stop at the first Failure and mention FAILURE. Order column should mention the order in which the nodes were colored, even if that resulted in a failure.

Structure

Can we use the structure of the problem to solve it more efficiently?

Problem Structure

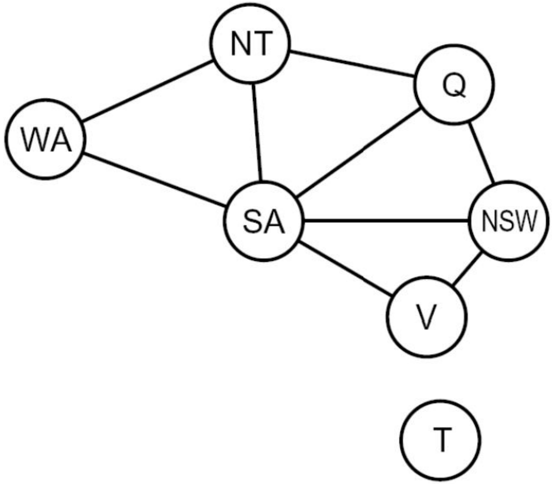
- Extreme case: independent subproblems
  - Example: Tasmania and the mainland do not interact

- Independent subproblems are identifiable as connected components of the constraint graph.

- Suppose a graph of n variables can be broken into subproblems of only c variables.

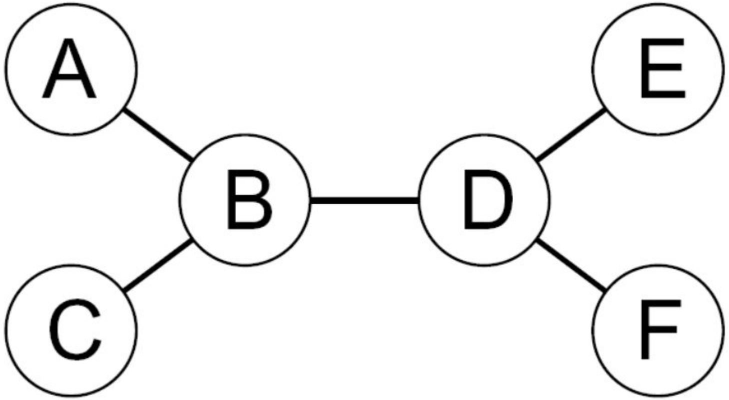
  - Solving smaller independent problems is much easier and faster.
  - e.g., variables n = 80, domain values d = 2.
  - 2^{80} possibilities = 4 billion years at 10 million nodes/sec.

  - Suppose if you can break this problem down into 4 independent subproblems containing 20 nodes each.
  - $(4)(2^{20})$ possibilities= 0.4 seconds at 10 million nodes/sec



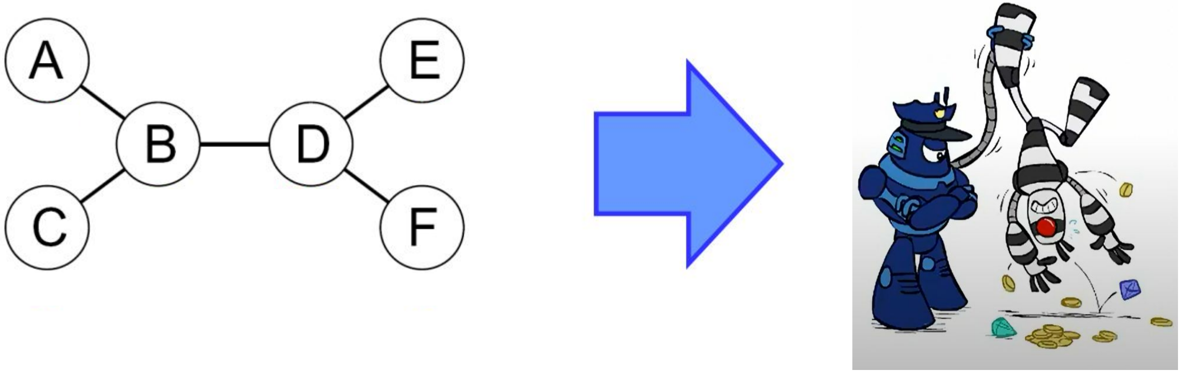
Tree-Structured CSPs

- If the constraint graph has no loops, the CSP can be solved in a significantly faster way compared to general CSPs



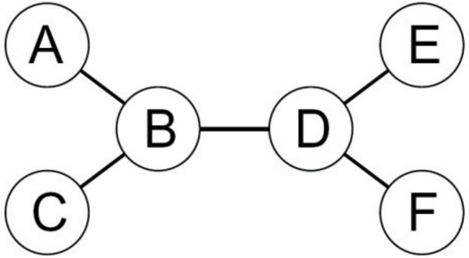
Tree-Structured CSPs

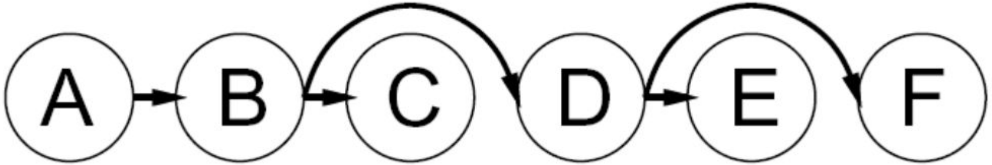
- Algorithm for tree-structured CSPs:
    - Order: Choose a root variable and order variables so that parents precede children.



Tree-Structured CSPs

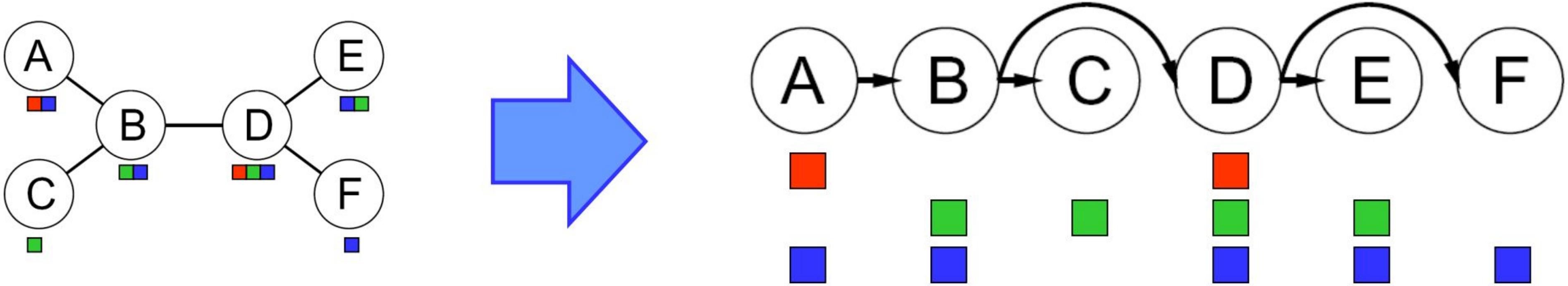
- Algorithm for tree-structured CSPs:
    - Order: Choose a root variable and order variables so that parents precede children.





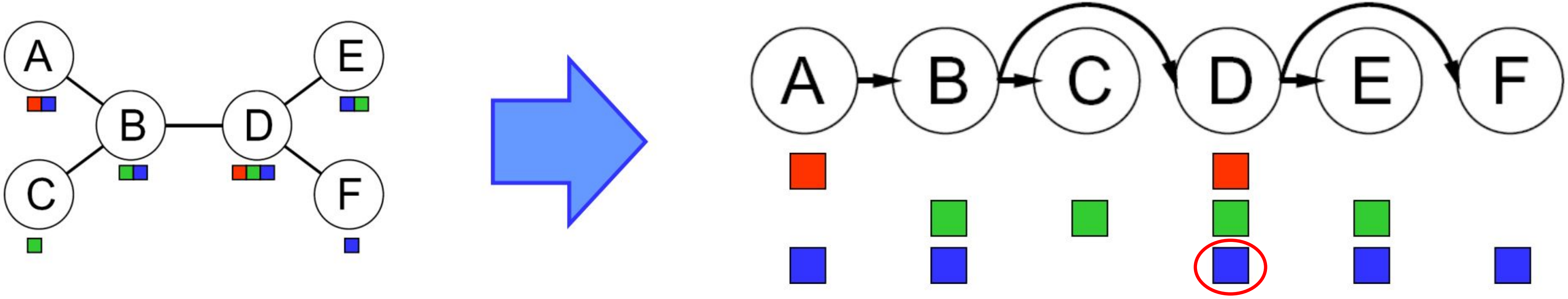
Tree-Structured CSPs

- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable and order variables so that parents precede children.



Tree-Structured CSPs

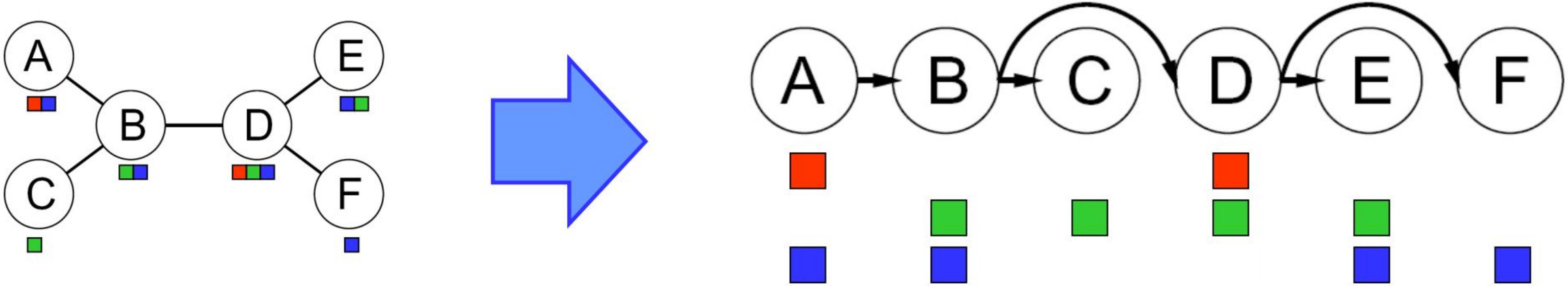
- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable and order variables so that parents precede children.



  - Remove backward: For i = n to 2, apply RemoveInconsistent(Parent(X_i),X_i)
    - One backward pass from F back to A and enforce arc consistency
      - Check F and D: Conflict on blue

Tree-Structured CSPs

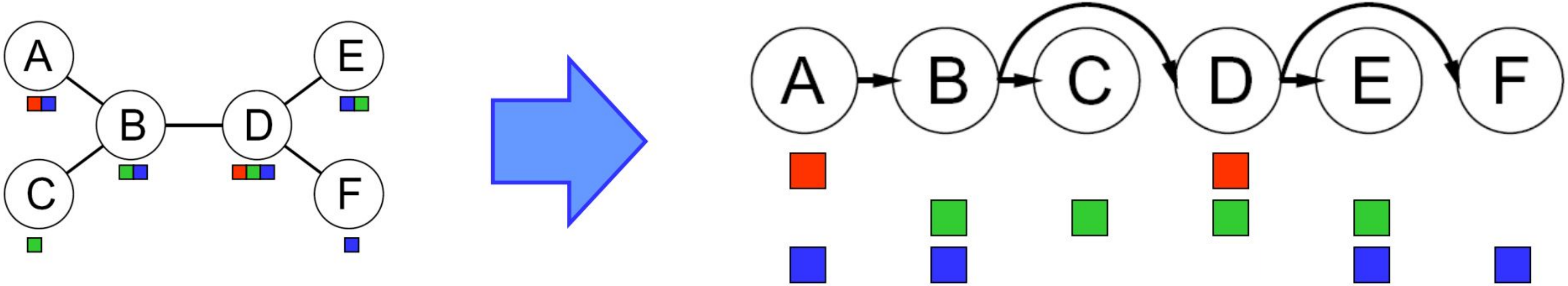
- Algorithm for tree-structured CSPs:
    - Order: Choose a root variable and order variables so that parents precede children.



    - Remove backward: For i = n to 2, apply RemoveInconsistent(Parent(X_i),X_i)
        - One backward pass from F back to A and enforce arc consistency
            - Check E and D: For each color of E, a non-conflicting color available at D

Tree-Structured CSPs

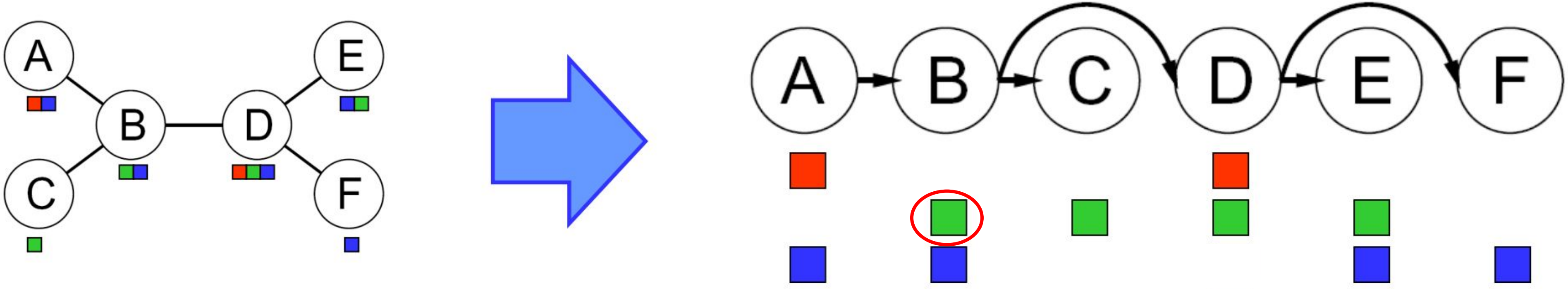
- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable and order variables so that parents precede children.



  - Remove backward: For i = n to 2, apply RemoveInconsistent(Parent(X_i),X_i)
    - One backward pass from F back to A and enforce arc consistency
      - Check D and B: For each color of D, a non-conflicting color available at B

Tree-Structured CSPs

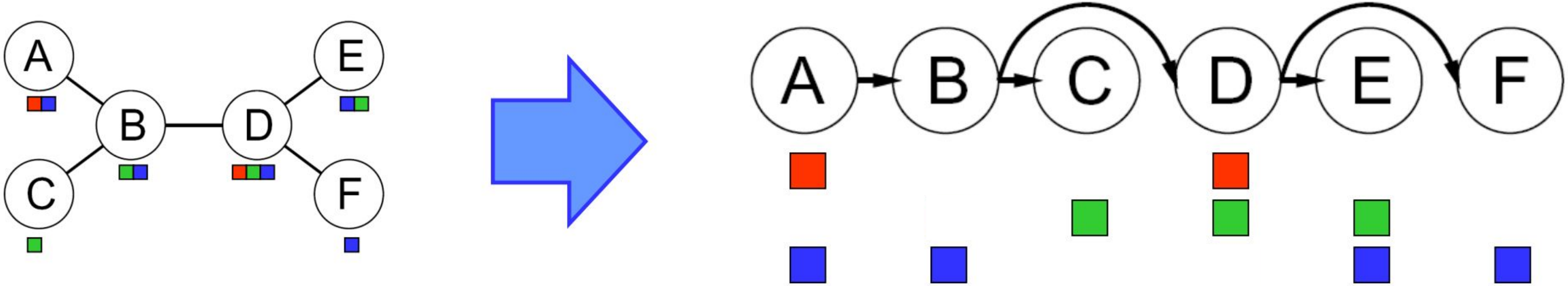
- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable and order variables so that parents precede children.



  - Remove backward: For i = n to 2, apply RemoveInconsistent(Parent(X_i),X_i)
    - One backward pass from F back to A and enforce arc consistency
      - Check C and B: C can only be green, therefore, remove green from B

Tree-Structured CSPs

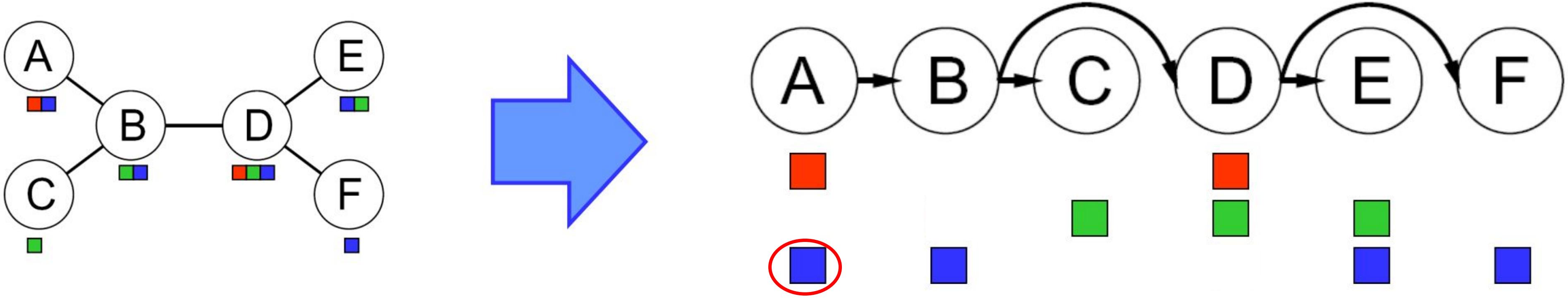
- Algorithm for tree-structured CSPs:
    - Order: Choose a root variable and order variables so that parents precede children.



    - Remove backward: For i = n to 2, apply RemoveInconsistent(Parent(X_i),X_i)
        - One backward pass from F back to A and enforce arc consistency
            - Check C and B: C can only be green, therefore, remove green from B

Tree-Structured CSPs

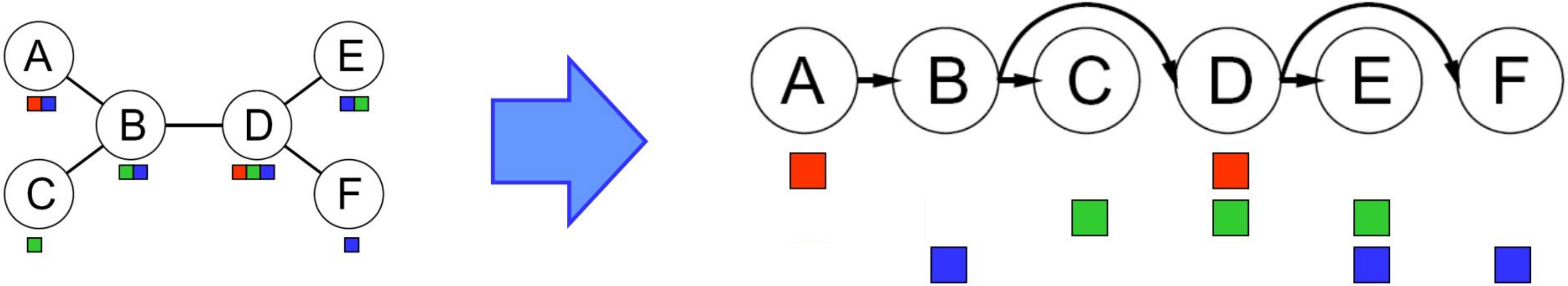
- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable and order variables so that parents precede children.



  - Remove backward: For i = n to 2, apply RemoveInconsistent(Parent(X_i),X_i)
    - One backward pass from F back to A and enforce arc consistency
      - Check B and A: B can only be blue, therefore, remove blue from A

Tree-Structured CSPs

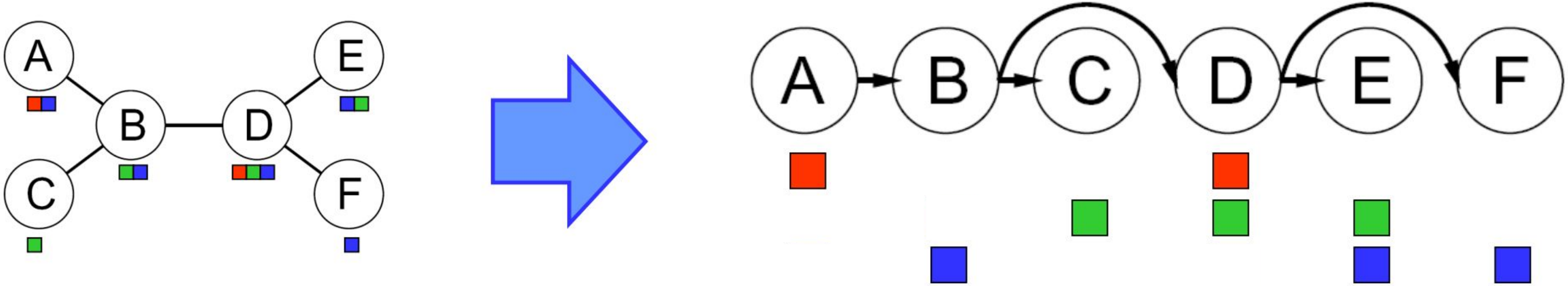
- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable and order variables so that parents precede children.



  - Remove backward: For i = n to 2, apply RemoveInconsistent(Parent(X_i),X_i)
    - One backward pass from F back to A and enforce arc consistency
      - Check B and A: B can only be blue, therefore, remove blue from A

Tree-Structured CSPs

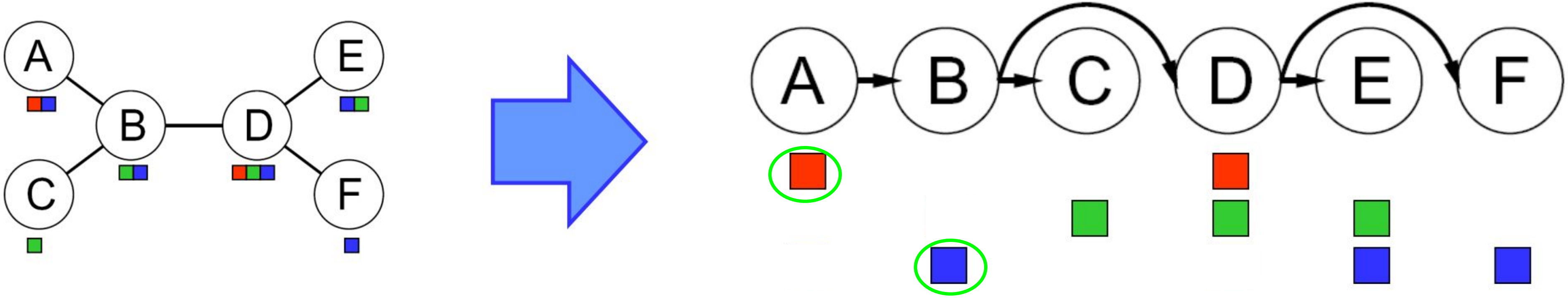
- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable and order variables so that parents precede children.



  - Remove backward: For i = n to 2, apply RemoveInconsistent(Parent(X_i),X_i)
    - One backward pass from F back to A and enforce arc consistency
  - Assign forward: For i = 1 to n, assign X_i consistently with Parent(X_i)
    - One forward pass from A to F and assign one of the available colors consistent with the parent

Tree-Structured CSPs

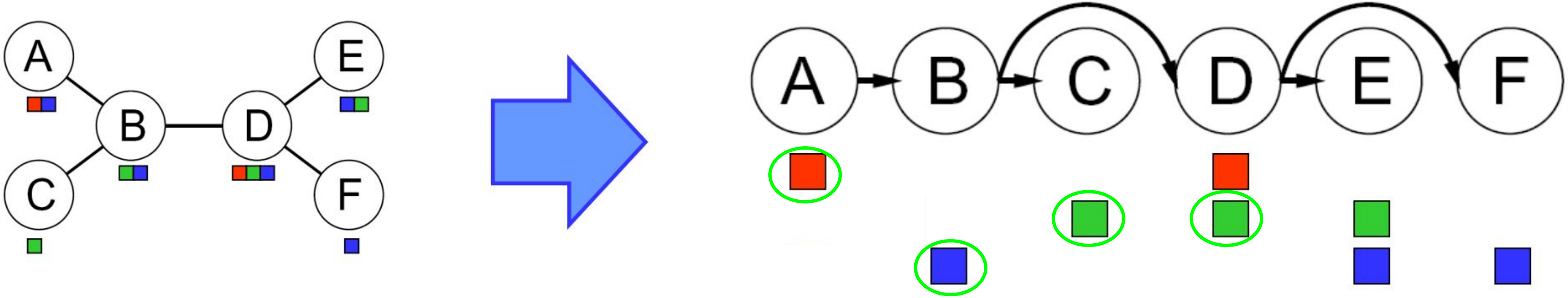
- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable and order variables so that parents precede children.



  - Remove backward: For i = n to 2, apply RemoveInconsistent(Parent(X_i),X_i)
    - One backward pass from F back to A and enforce arc consistency
  - Assign forward: For i = 1 to n, assign X_i consistently with Parent(X_i)
    - One forward pass from A to F and assign one of the available colors consistent with the parent

Tree-Structured CSPs

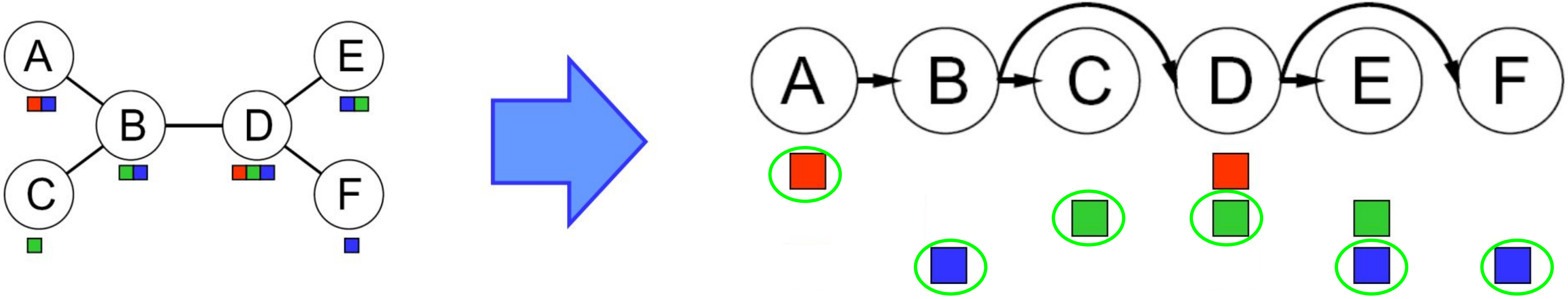
- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable and order variables so that parents precede children.



  - Remove backward: For i = n to 2, apply RemoveInconsistent(Parent(X_i),X_i)
    - One backward pass from F back to A and enforce arc consistency
  - Assign forward: For i = 1 to n, assign X_i consistently with Parent(X_i)
    - One forward pass from A to F and assign one of the available colors consistent with the parent

Tree-Structured CSPs

- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable and order variables so that parents precede children.



  - Remove backward: For i = n to 2, apply RemoveInconsistent(Parent(X_i),X_i)
    - One backward pass from F back to A and enforce arc consistency
  - Assign forward: For i = 1 to n, assign X_i consistently with Parent(X_i)
    - One forward pass from A to F and assign one of the available colors consistent with the parent