NTNU

Constraint Satisfaction Problems

Ole C. Eidheim

September 12, 2024

Department of Computer Science

Motivation: chess

- ChatGPT plays chess

Motivation: chess

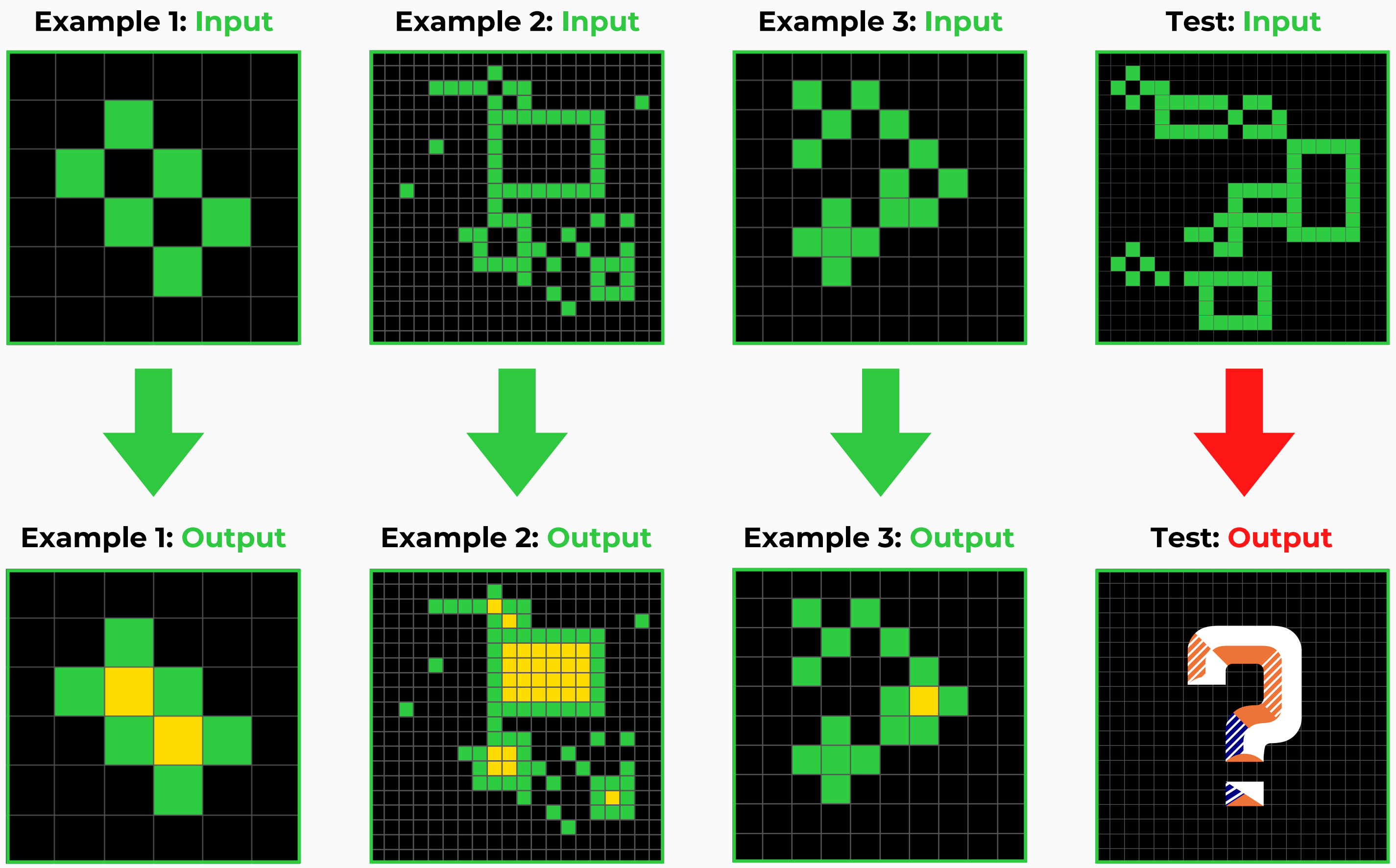
- ChatGPT plays chess
    - What is ChatGPT missing?

Motivation: chess

- ChatGPT plays chess
    - What is ChatGPT missing?
        - Search!

From ARC challenge:

- ARC evaluates an AI's ability to tackle each task from scratch, using only the kind of prior knowledge about the world that humans naturally possess, known as core knowledge.
- Modern deep-learning models and large language models score near zero on ARC, highlighting the need for innovative approaches to reach human-level AI.



Motivation: the Zebra Puzzle

Five persons of different nationalities and with different jobs live in consecutive houses. The houses are painted in different colors, and the persons have different pets and favorite drinks. Additionally:

- The English lives in a red house
- The Spaniard owns a dog
- The Japanese is a painter
- The Ukrainian drinks tea
- The Norwegian lives in the first house
- The green house immediately to the right of the white one
- The photographer owns snails
- The diplomat lives in the yellow house
- Milk is drunk in the middle house
- The owner of the green house drinks coffee
- The Norwegian's house is next to the blue one
- The violinist drinks orange juice
- The fox is in a house next to that of the physician
- The horse is in a house next to that of the diplomat

Who owns a zebra? And whose favorite drink is water?

Outline

Review

Constraint Satisfaction Problems

Inference: reducing domains

Heuristics: selecting variable and ordering values

Looking backward

Local search

Tree-structured CSPs

Reducing the constraint graph to a tree

Search Problems

Definitions

- *initialState*: starting state
- $actions(s)$: possible actions at state s
- $cost(s, a)$: cost of taking action a at state s
- $result(s, a)$: next state after taking action a at state s
- $isGoal(s)$: is state s an end state?

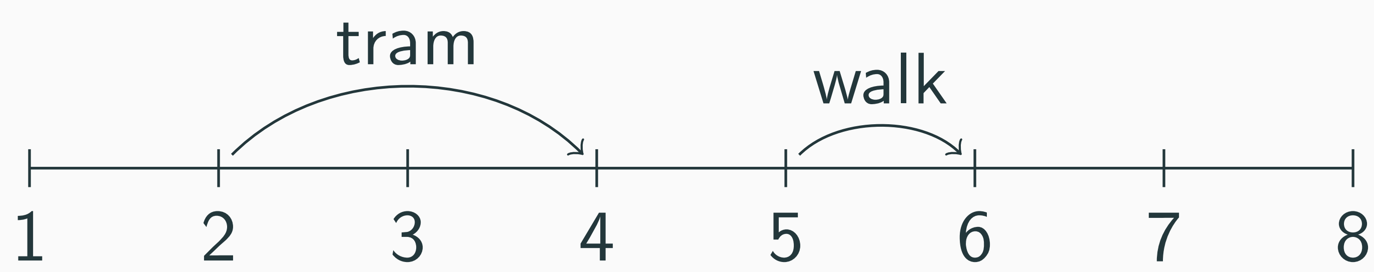
Objective: find a state path with the lowest sum of costs from *initialState* to a state s that satisfies $isGoal(s)$

Example: transportation problem

- Places numbered from 1 to P
- Walking from place p to $p+1$ takes 1 minute
- Taking the magic tram from place p to $2p$ takes 3 minutes

How to travel from place 1 to P in the least amount of time?

Example actions:

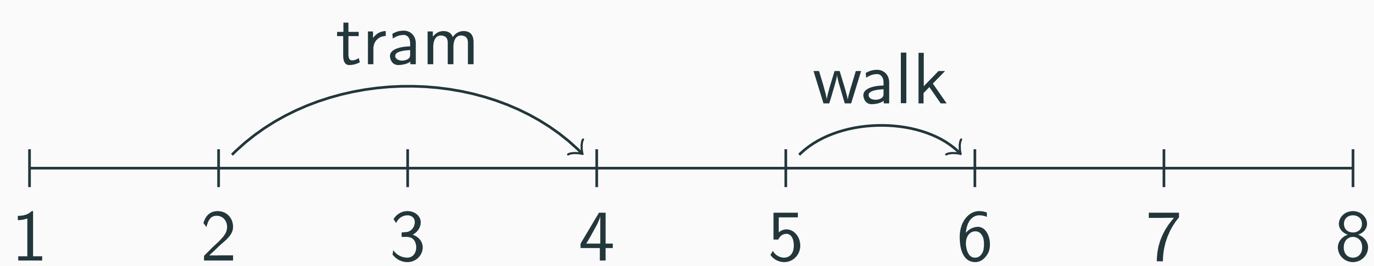


Example: transportation problem

- Places numbered from 1 to P
- Walking from place p to $p + 1$ takes 1 minute
- Taking the magic tram from place p to $2p$ takes 3 minutes

How to travel from place 1 to P in the least amount of time?

Example actions:



Example code solution using backtracking

Search Algorithms

Search Algorithm	Cost assumptions
Backtracking	None
Depth-first search	$cost(s, a) = 0$
Breadth-first search	$cost(s, a) = c, c \geq 0$
Depth-first search with iterative deepening	$cost(s, a) = c, c \geq 0$
Dynamic programming	None
Uniform-cost search	$cost(s, a) \geq 0$
A*	$cost(s, a) \geq 0$

where c is a constant

Outline

Search problems vs constraint satisfaction problems

- Search problems
  - Problem specific states, actions, next states, costs, end state, search function, and heuristics
- Constraint satisfaction problems
  - A set of constraints that must be satisfied
  - Problem specific variables, variable domains, and constraints
  - Problem independent search function, inference and heuristics
  - Example problem: Sudoku
    - Variables: the cells
    - Variable domains (values a cell may have): $\{1, \ldots, 9\}$
    - Constraints: different values horizontally, vertically, and within 3x3 blocks

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

Constraint Satisfaction Problems

Definitions

- Variables $X_1, \ldots, X_n$
  - The variables might have different names, e.g. $A, B, \ldots$
- Domains $D_1, \ldots, D_n$ of each variable $X_i \in D_i$, i.e. values each variable can have
- Constraints $C_1, \ldots, C_m$ with each $C_j \in \{0, 1\}$

$$
\begin{aligned}
\text{For example: } C_1(X_1, X_2) &= \begin{cases} 1, & \text{if } X_1 \neq X_2 \\ 0, & \text{otherwise} \end{cases} \\
\text{or in book notation: } \quad C_1 &: \quad \langle (X_1, X_2), X_1 \neq X_2 \rangle \\
\text{or simply } \quad C_1 &: \quad X_1 \neq X_2
\end{aligned}
$$

Objective: find a complete assignment $\{X_1 = v_1, \ldots, X_n = v_n\}$ that satisfies all constraints

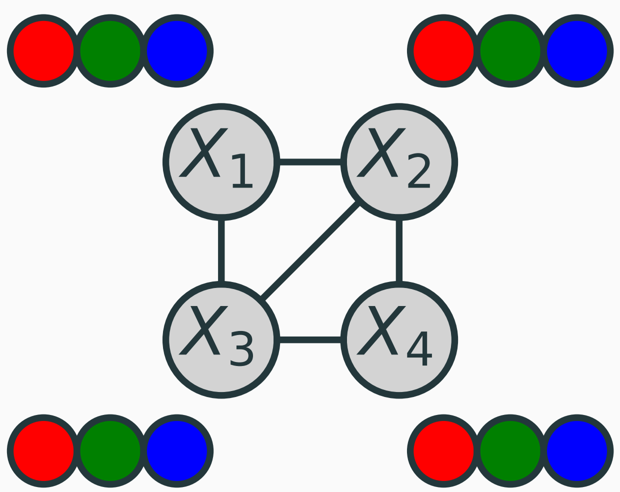
Example: coloring problem

Assign red, green or blue to each region, but neighboring regions cannot have the same color

Example: coloring problem

Assign red, green or blue to each region, but neighboring regions cannot have the same color

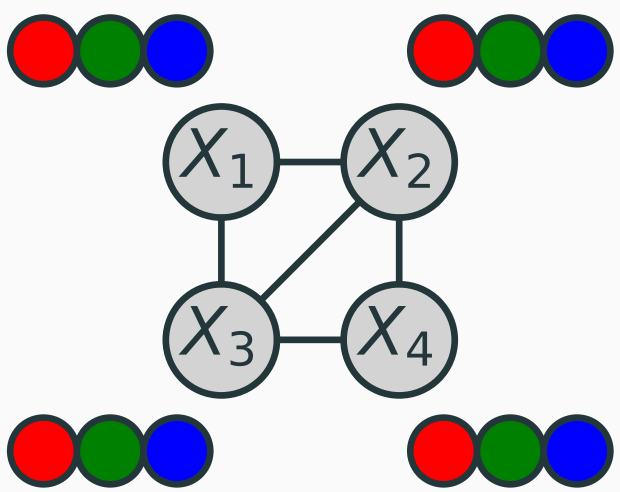
For simplicity, ignoring NSW, V, T



Constraint graph with domains

Example: coloring problem

Assign red, green or blue to each region, but neighboring regions cannot have the same color

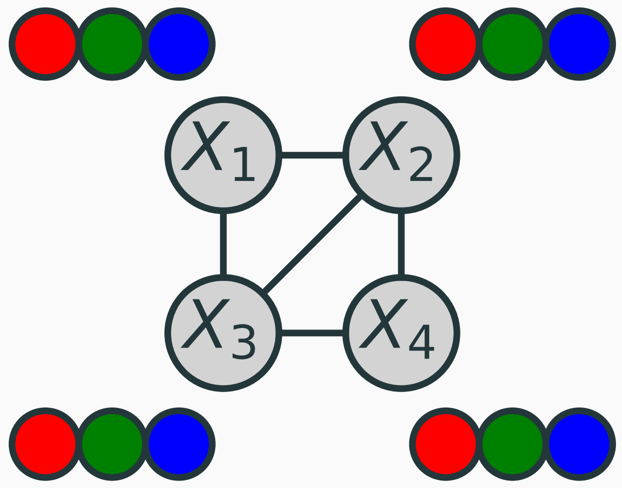


For simplicity, ignoring NSW, V, T

Constraint graph with domains

Example code solution (CSP implementation not shown)

More on constraints: coloring problem



- Variables: X_1, X_2, X_3, X_4
- Domains: $D_1 = \ldots = D_4 = \{red, green, blue\}$
- Constraints: $\{X_1 \neq X_2, X_1 \neq X_3, X_2 \neq X_3, X_2 \neq X_4, X_3 \neq X_4\}$

  or as tuples of values:

$$
\begin{aligned}
\{&\langle (X_1, X_2), \{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\}\rangle, \\
&\langle (X_1, X_3), \{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\}\rangle, \\
&\langle (X_2, X_3), \{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\}\rangle, \\
&\langle (X_2, X_4), \{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\}\rangle, \\
&\langle (X_3, X_4), \{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\}\rangle\}
\end{aligned}
$$

More on constraints: a simpler problem

- Variables: X_1, X_2
- Domains: $D_1 = \{1, 2, 3, 4, 5\}, D_2 = \{1, 2\}$
- Constraints: $\{X_1 + X_2 = 4\}$
  or as tuples of values:
  $\{\langle (X_1, X_2), \{(2, 2), (3, 1)\}\rangle\}$

Global constraints

- *alldiff* (*variables*): inequality constraints between the *variables*
    - For example: *alldiff* $(X_1, X_2, X_3) = \{X_1 \neq X_2, X_1 \neq X_3, X_2 \neq X_3\}$

Backtracking search for Constraint Satisfaction Problems

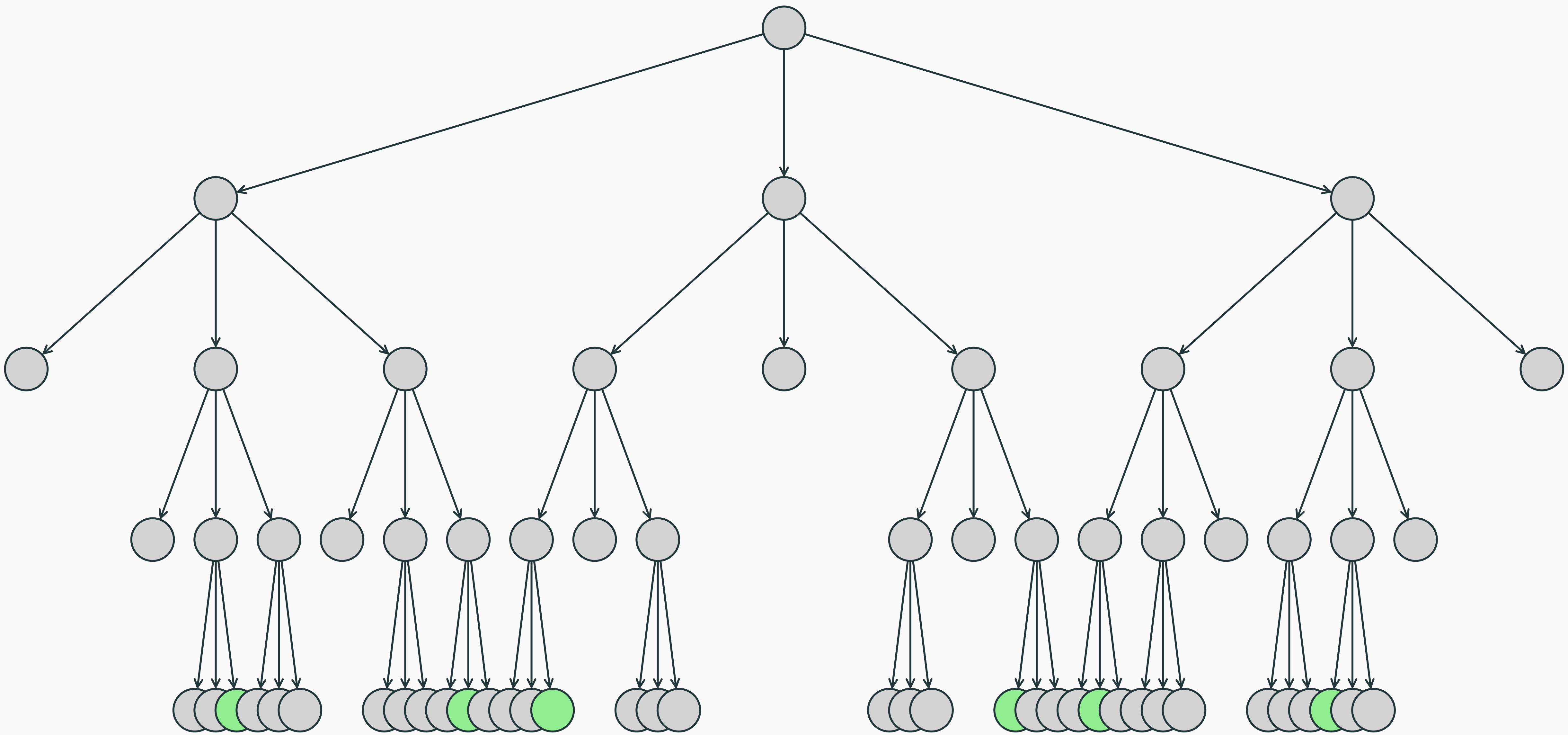
Algorithm

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 ← INFERENCE(*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*

Outline

Motivation: simplify search by solving subnetworks

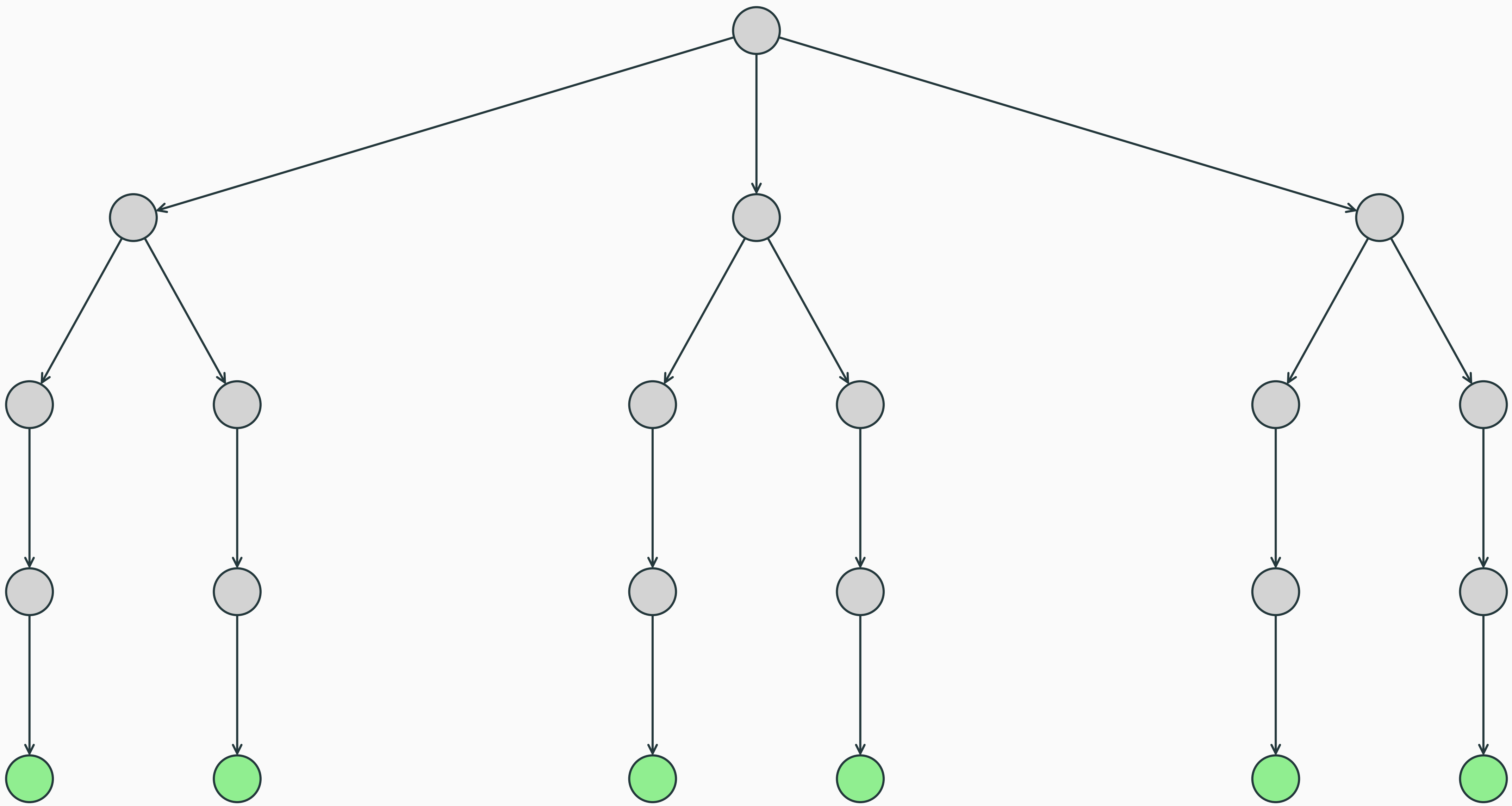
Without inference:



Example search tree with all solutions shown

Motivation: simplify search by solving subnetworks

With inference:



Example search tree with all solutions shown

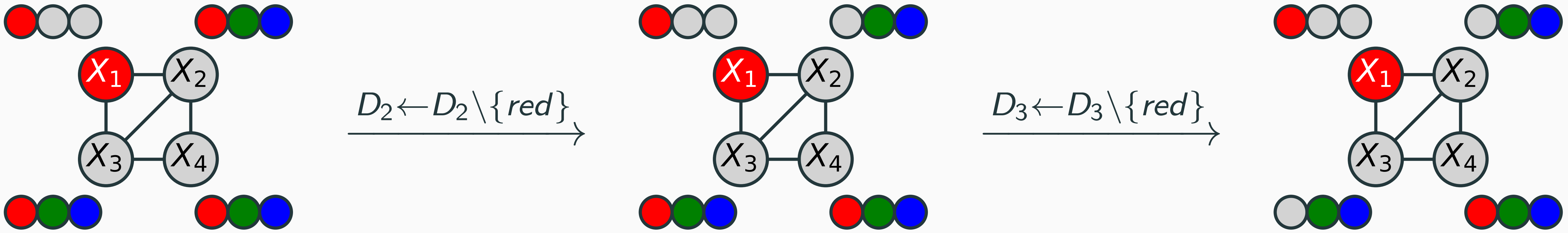
Forward checking

Algorithm

After assigning a value to a variable, explore the domains of neighboring unassigned variables, and remove values whose assignments would violate a constraint

- Neighboring variables: variables that share constraint dependencies
- During inference, domains are also set to the assigned value

For example, after assigning $\{X_1 = red\}$ (and $D_1 \leftarrow \{red\}$):



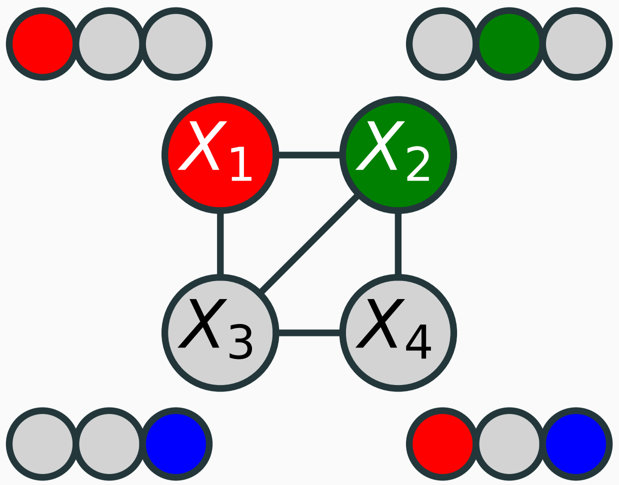
Arc Consistency Algorithm #3 (AC-3)

Arc(directed edge)-consistency

X_i is arc-consistent with X_j if for every value in D_i there exists a value in D_j whose assignments would not violate any constraint

- Only binary (two-variable) constraints considered

For example, is (X_3, X_4) arc-consistent (is X_3 arc-consistent with X_4)?



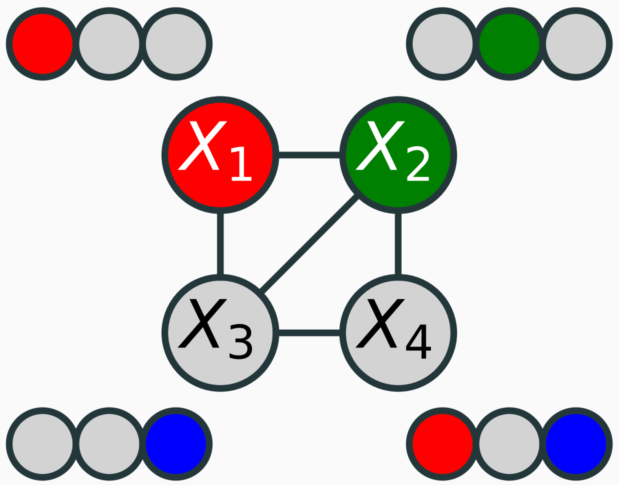
Arc Consistency Algorithm #3 (AC-3)

Arc(directed edge)-consistency

X_i is arc-consistent with X_j if for every value in D_i there exists a value in D_j whose assignments would not violate any constraint

- Only binary (two-variable) constraints considered

For example, is (X_3, X_4) arc-consistent (is X_3 arc-consistent with X_4)?



Answer: yes

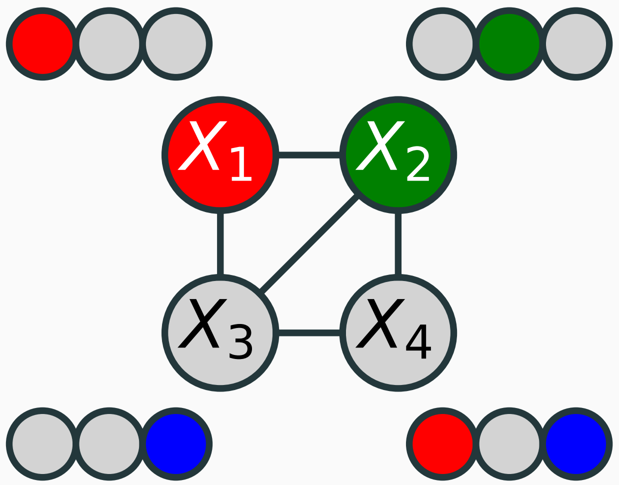
Arc Consistency Algorithm #3 (AC-3)

Arc(directed edge)-consistency

X_i is arc-consistent with X_j if for every value in D_i there exists a value in D_j whose assignments would not violate any constraint

- Only binary (two-variable) constraints considered

For example, is (X_4, X_3) arc-consistent?



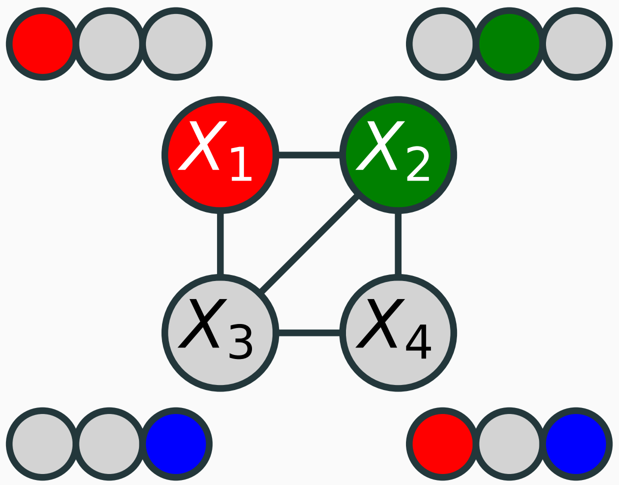
Arc Consistency Algorithm #3 (AC-3)

Arc(directed edge)-consistency

X_i is arc-consistent with X_j if for every value in D_i there exists a value in D_j whose assignments would not violate any constraint

- Only binary (two-variable) constraints considered

For example, is (X_4, X_3) arc-consistent?



Answer: no

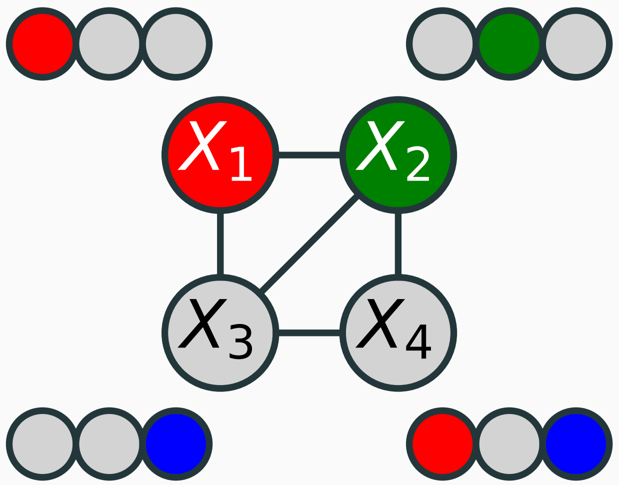
Arc Consistency Algorithm #3 (AC-3)

Arc(directed edge)-consistency

X_i is arc-consistent with X_j if for every value in D_i there exists a value in D_j whose assignments would not violate any constraint

- Only binary (two-variable) constraints considered

How do we make (X_4, X_3) arc-consistent?



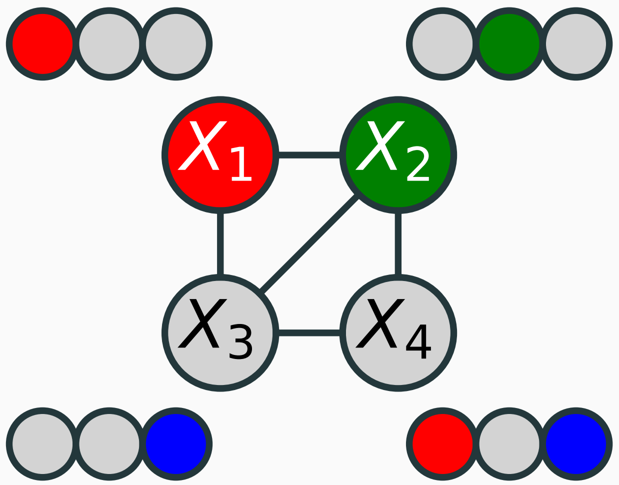
Arc Consistency Algorithm #3 (AC-3)

Arc(directed edge)-consistency

X_i is arc-consistent with X_j if for every value in D_i there exists a value in D_j whose assignments would not violate any constraint

- Only binary (two-variable) constraints considered

How do we make (X_4, X_3) arc-consistent?



Answer: $D_4 \leftarrow D_4 \setminus blue$

Arc Consistency Algorithm #3 (AC-3)

Arc(directed edge)-consistency

X_i is arc-consistent with X_j if for every value in D_i there exists a value in D_j whose assignments would not violate any constraint

- Only binary (two-variable) constraints considered

Make (X_i, X_j) arc-consistent

Remove values from D_i to make X_i arc-consistent with X_j

Algorithm

- start with a queue of initial arc(s)
- while queue is not empty
  - pop an arc and make arc-consistent
  - if the domain was reduced for a variable, add arcs between its unassigned neighbors and the variable

Arc Consistency Algorithm #3 (AC-3)

The algorithm in Figure 5.3 in the book starts with all neighboring unassigned variables in the queue (both directions initially)

This can for example solve simple Sudoku problems without search:

	4		1
3			
			4

Domains after setting the initial values:

$D_{12} = \{4\}, D_{14} = \{1\}, D_{21} = \{3\}, D_{34} = \{4\},$

$D_{11} = D_{13} = D_{22} = \ldots = D_{33} = D_{41} = \ldots = D_{44} = \{1, 2, 3, 4\}$

Domains after running the AC-3 algorithm:

$D_{11} = \{2\}, D_{12} = \{4\}, D_{13} = \{3\}, D_{14} = \{1\}, D_{21} = \{3\}, D_{22} = \{1\}, D_{23} = \{4\}, D_{24} = \{2\},$

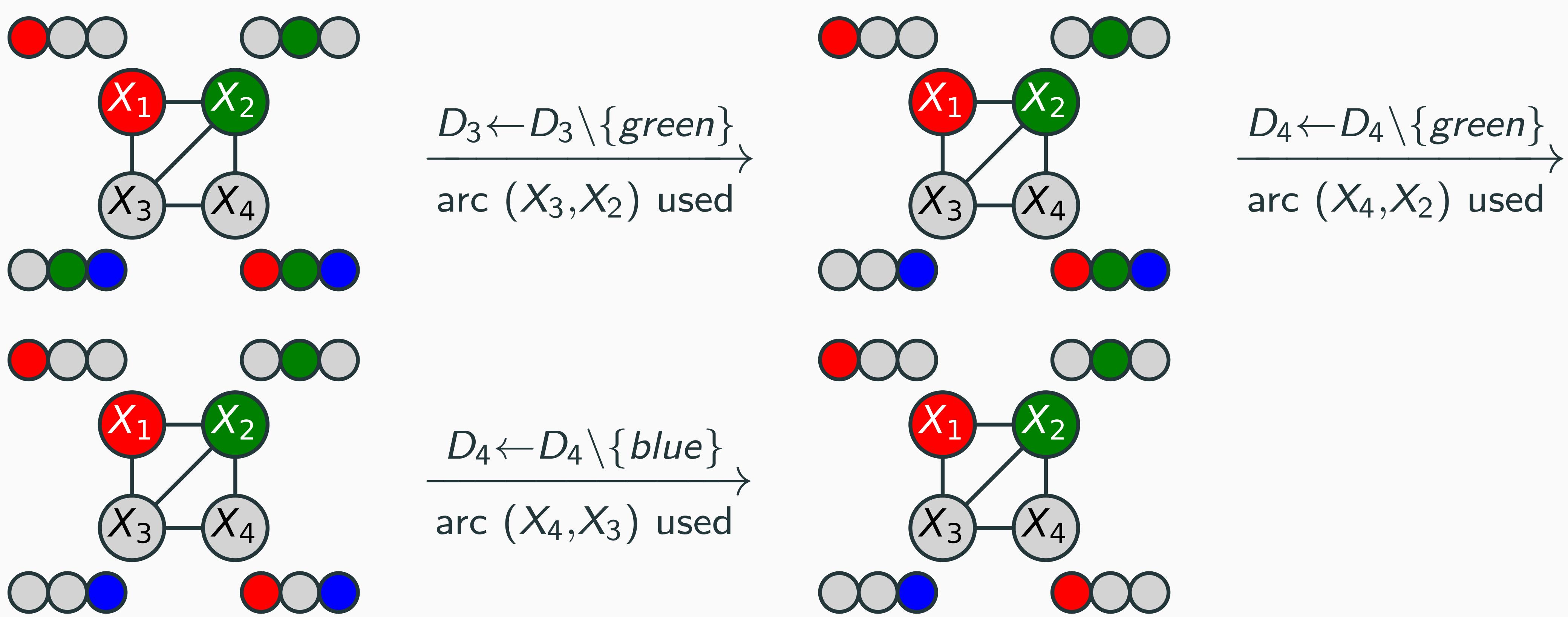
$D_{31} = \{1\}, D_{32} = \{3\}, D_{33} = \{2\}, D_{34} = \{4\}, D_{41} = \{4\}, D_{42} = \{2\}, D_{43} = \{1\}, D_{44} = \{3\}$

Arc Consistency Algorithm #3 (AC-3)

Another possibility is to run AC-3 for each newly assigned variable and its neighboring unassigned variables (in opposite directions)

Example, after assigning $\{X_2 = green\}$ (and $D_2 \leftarrow \{green\}$):

- Starts with two arcs in the queue: (X_3, X_2) and (X_4, X_2)
- When D_3 is updated, the arc (X_4, X_3) is added to queue
- When D_4 is updated, the arc (X_3, X_4) is added to queue



Outline

Motivation

Possibly faster search by focusing on likely choices

Algorithm

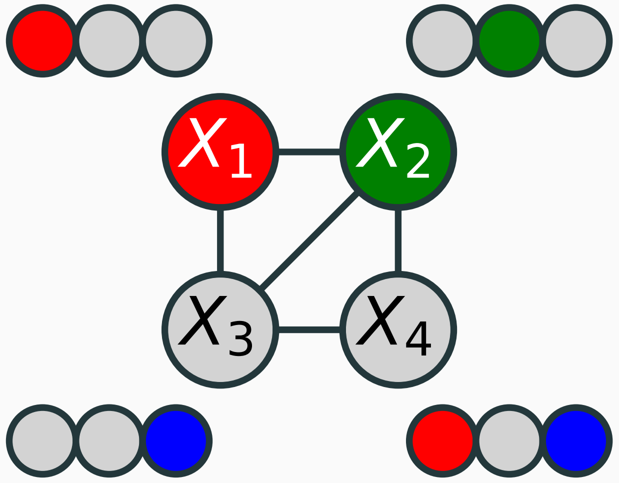
```
function BACKTRACK(csp, assignment) returns a solution or failure
   if assignment is complete then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
   for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
      if value is consistent with assignment then
         add {var = value} to assignment
         inferences ← INFERENCE(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
```

Selecting variable

Choose the most constrained variable

Select the unassigned variable with the smallest remaining domain

For example, will X_3 or X_4 be chosen:

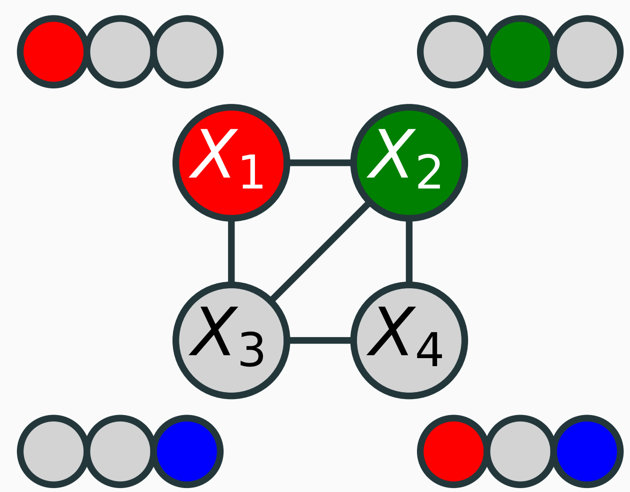


Selecting variable

Choose the most constrained variable

Select the unassigned variable with the smallest remaining domain

For example, will X_3 or X_4 be chosen:



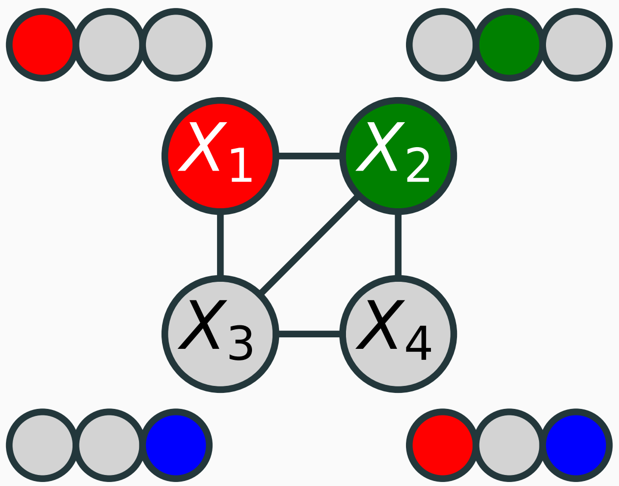
Answer: X_3

Ordering values

Choose the least constrained values first

Prefer values that would cause the smallest reduction in the unassigned variable domains during forward checking

For example, given variable X_4 was chosen, will red or blue be chosen?

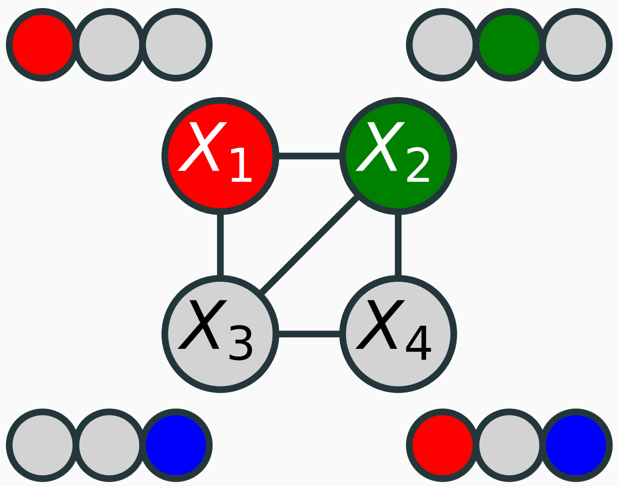


Ordering values

Choose the least constrained values first

Prefer values that would cause the smallest reduction in the unassigned variable domains during forward checking

For example, given variable X_4 was chosen, will red or blue be chosen?



Answer: red

Example: cryptarithmetic problem

Assign distinct digits to each letter such that the sum is correct, and none of the numbers have leading zeros, i.e. $T \neq 0$ and $F \neq 0$

$$\begin{array}{cccc} & T & W & O \\ + & T & W & O \\ \hline F & O & U & R \end{array}$$

Example solution:

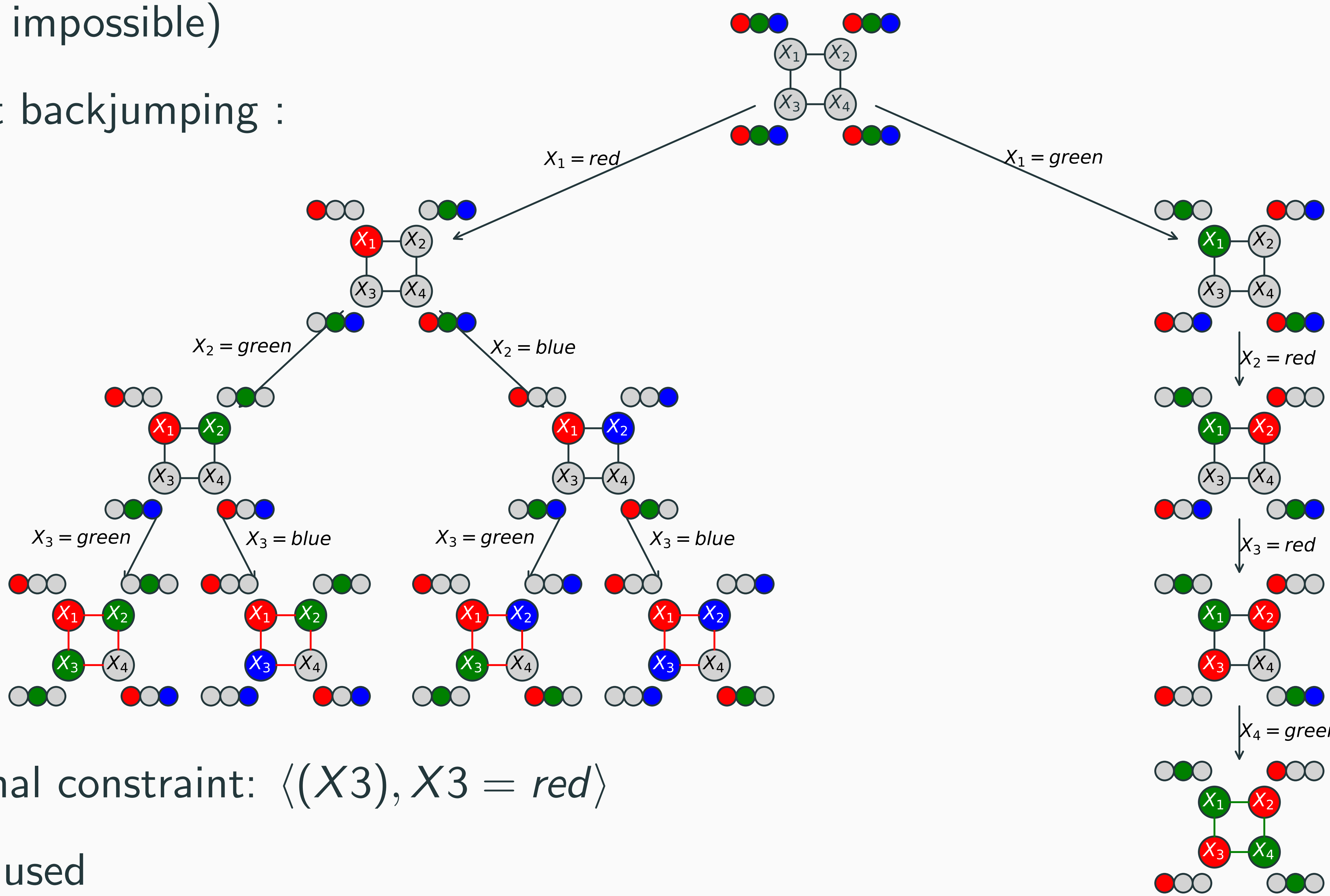
$$\begin{array}{cccc} & 7 & 6 & 5 \\ + & 7 & 6 & 5 \\ \hline 1 & 5 & 3 & 0 \end{array}$$

Outline

Backjumping

Jumps back to the most recent conflicting assignment (assignment that makes a solution impossible)

Without backjumping :



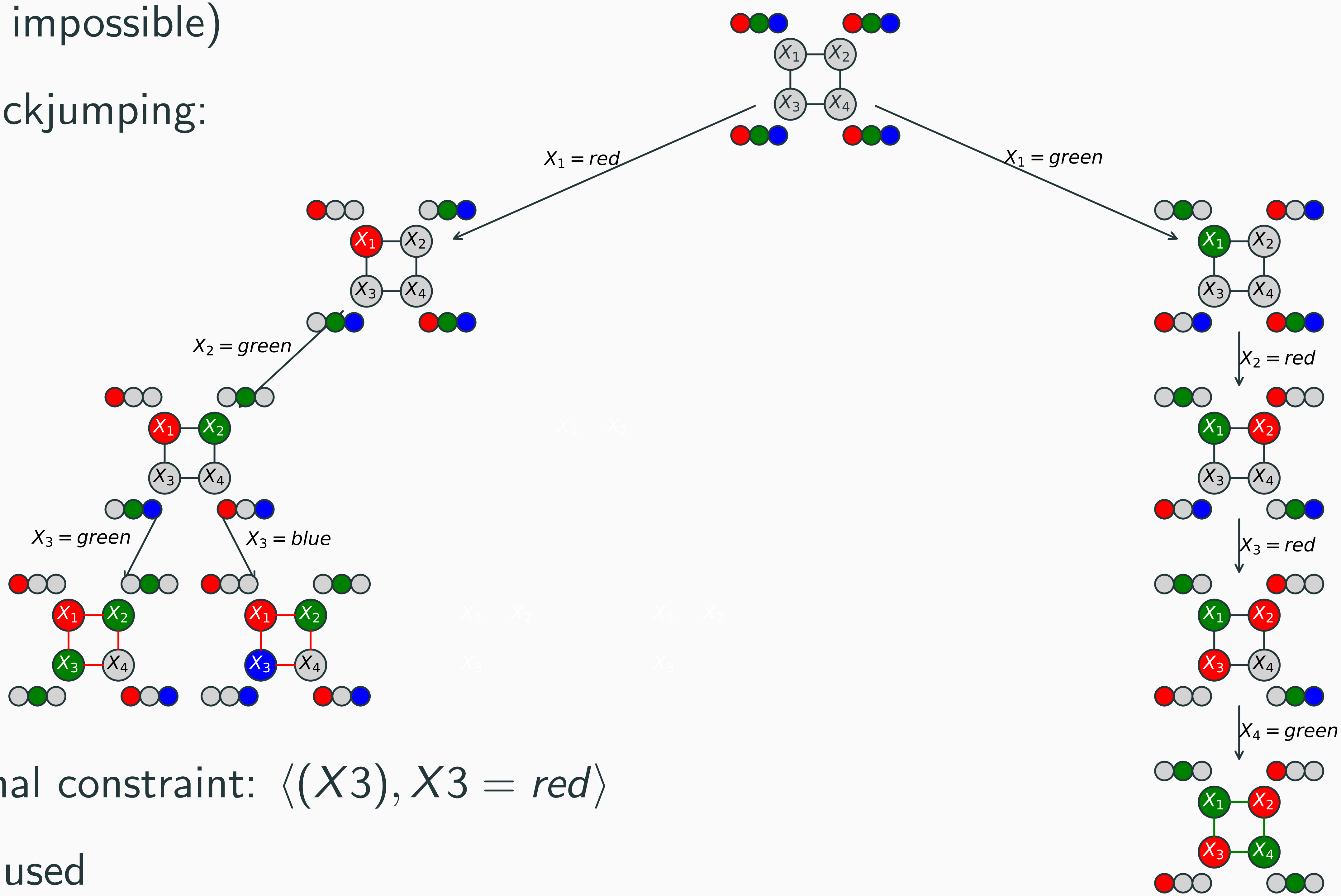
Additional constraint: $\langle (X3), X3 = \mathit{red} \rangle$

AC-3 is used

Backjumping

Jumps back to the most recent conflicting assignment (assignment that makes a solution impossible)

With backjumping:

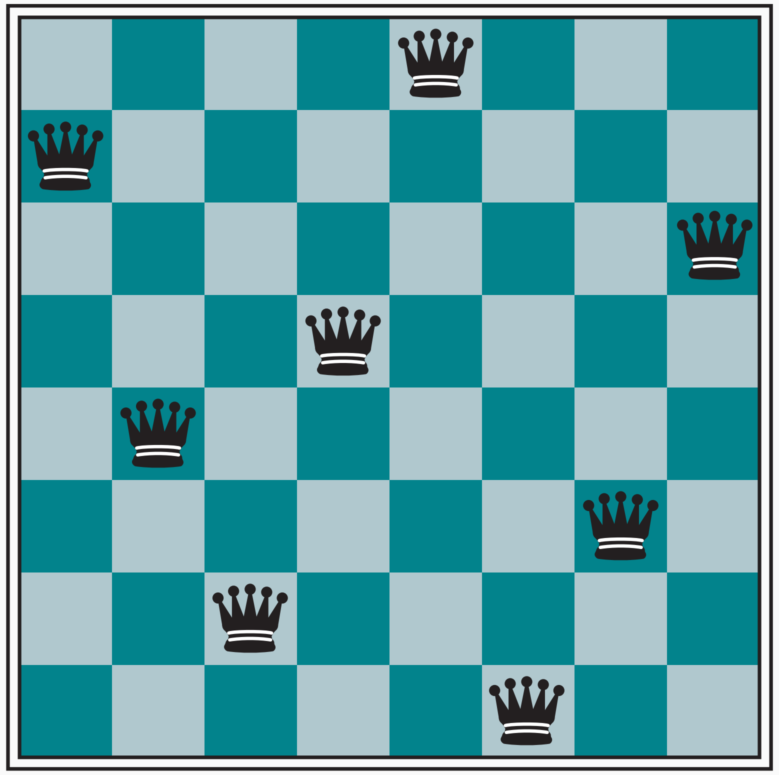


Additional constraint: $\langle (X3), X3 = red \rangle$

AC-3 is used

Outline

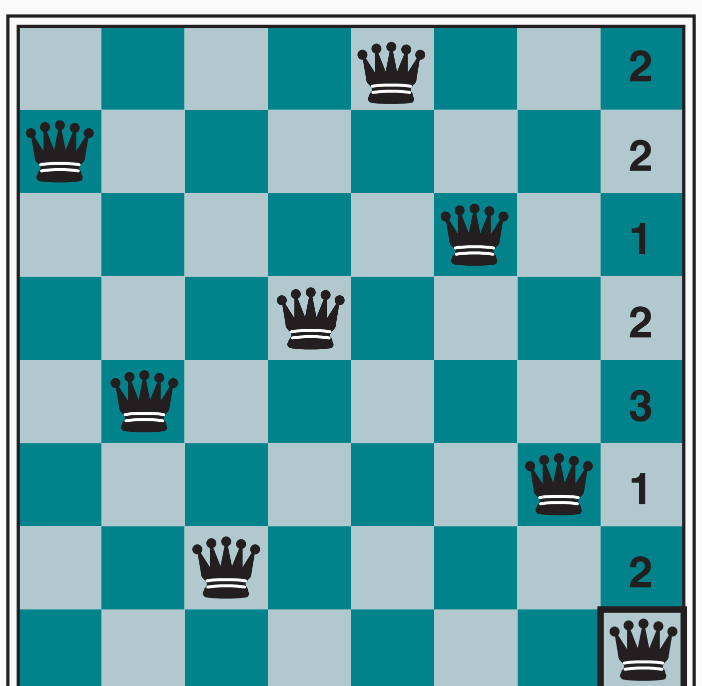
Example: 8-queens problem

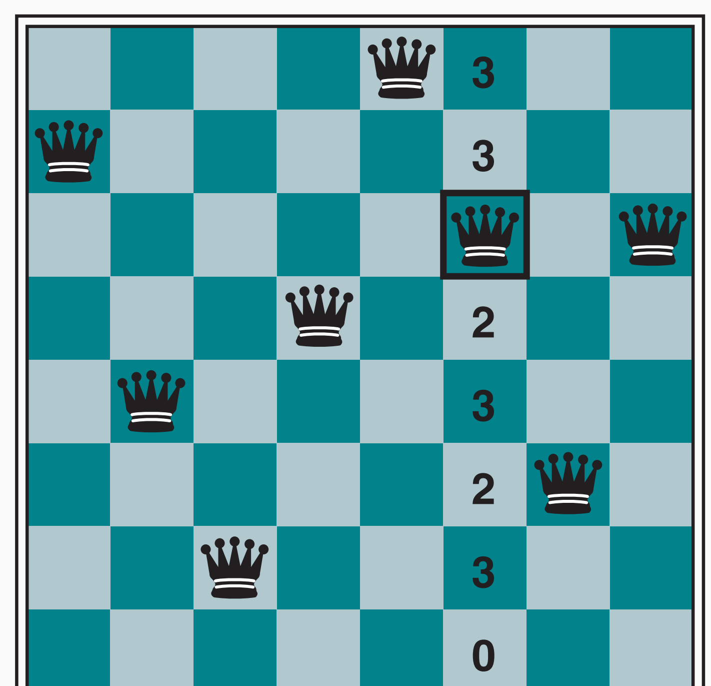


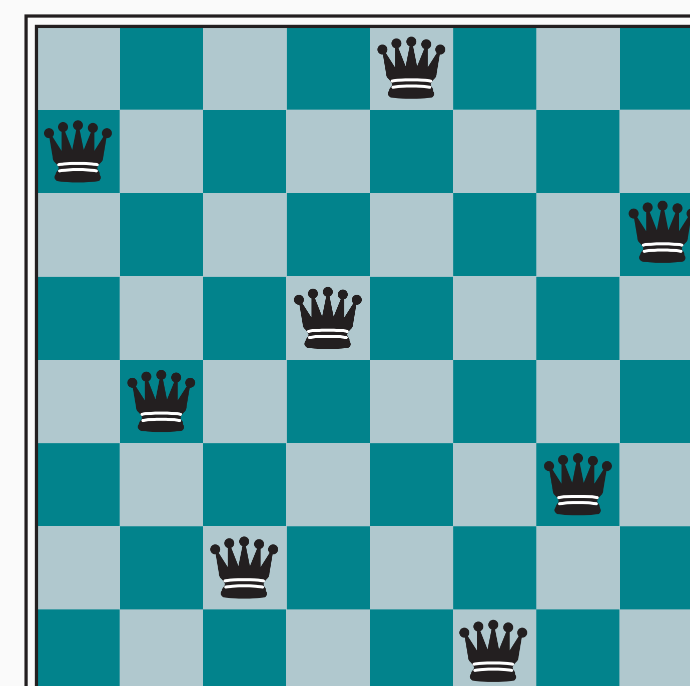
Min-Conflicts

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

  current ← an initial complete assignment for *csp*
  for i = 1 to *max_steps* **do**
    if *current* is a solution for *csp* **then return** *current*
    var ← a randomly chosen conflicted variable from *csp*.VARIABLES
    value ← the value v for *var* that minimizes CONFLICTS(*csp*, *var*, v, *current*)
    set *var* = *value* in *current*
  return *failure*







Outline

Review

Constraint Satisfaction Problems

Inference: reducing domains

Heuristics: selecting variable and ordering values

Looking backward

Local search

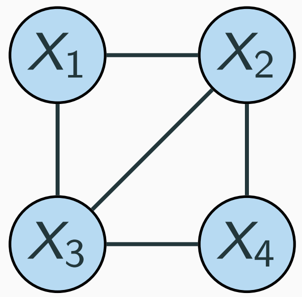
Tree-structured CSPs

Reducing the constraint graph to a tree

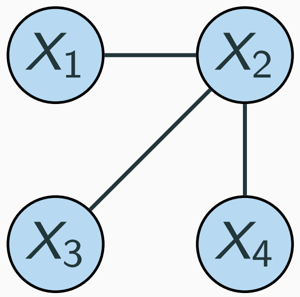
Tree-structured CSPs

Definition

Any two variables are connected by exactly one path



Not tree-structured

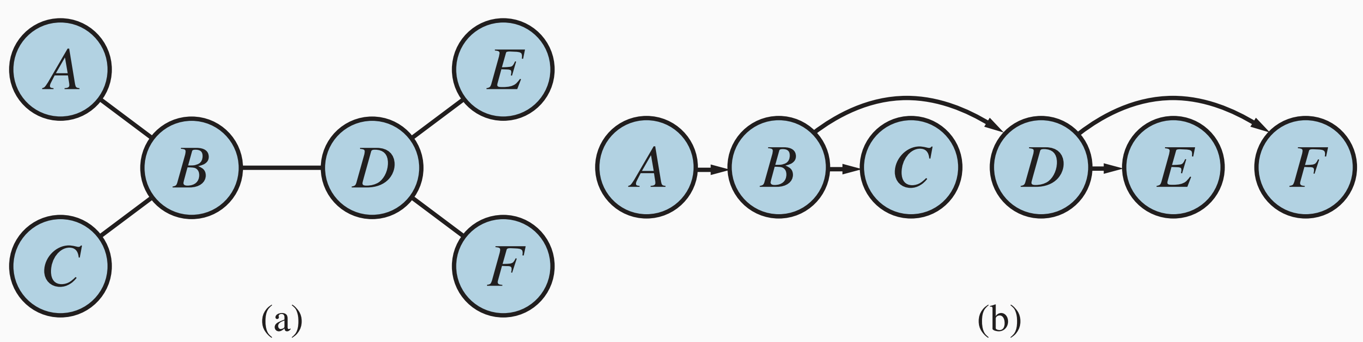


Tree-structured

The Tree-CSP-Solver algorithm

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 $assignment$

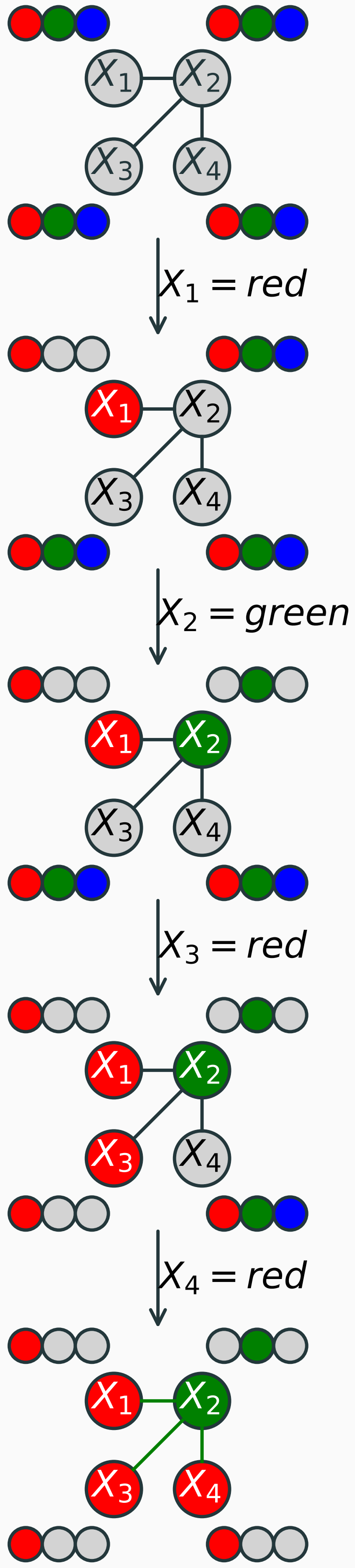


Topological sort of a) is shown in b)

If a tree-structured CSP has a solution, it will be found in linear time

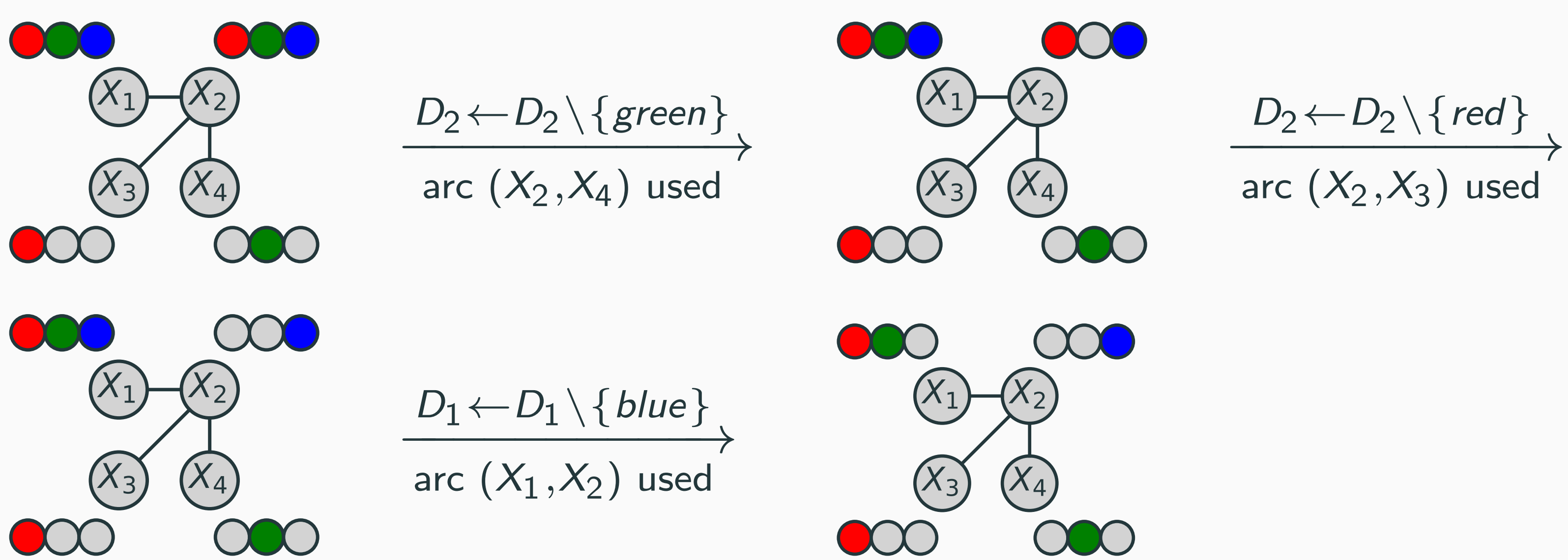
The Tree-CSP-Solver algorithm

- For example:
  - $D_1 = \ldots = D_4 = \{red, green, blue\}$
  - $root = X_1$
  - $parent(X_2) = X_1$
  - $parent(X_3) = X_2$
  - $parent(X_4) = X_2$
- No changes in the first for-loop of the algorithm
- The second for-loop of the algorithm $\rightarrow$

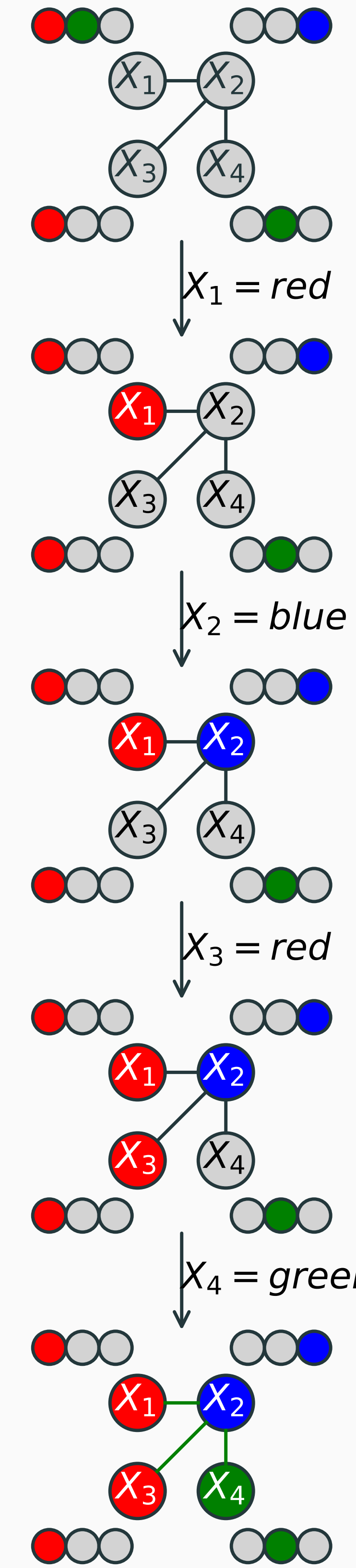


The Tree-CSP-Solver algorithm

- For example:
    - $D_1 = D_2 = \{red, green, blue\}$
    - $D_3 = \{red\}, D_4 = \{green\}$
    - $root = X_1$
    - $parent(X_2) = X_1$
    - $parent(X_3) = X_2$
    - $parent(X_4) = X_2$
- The first for-loop of the algorithm:

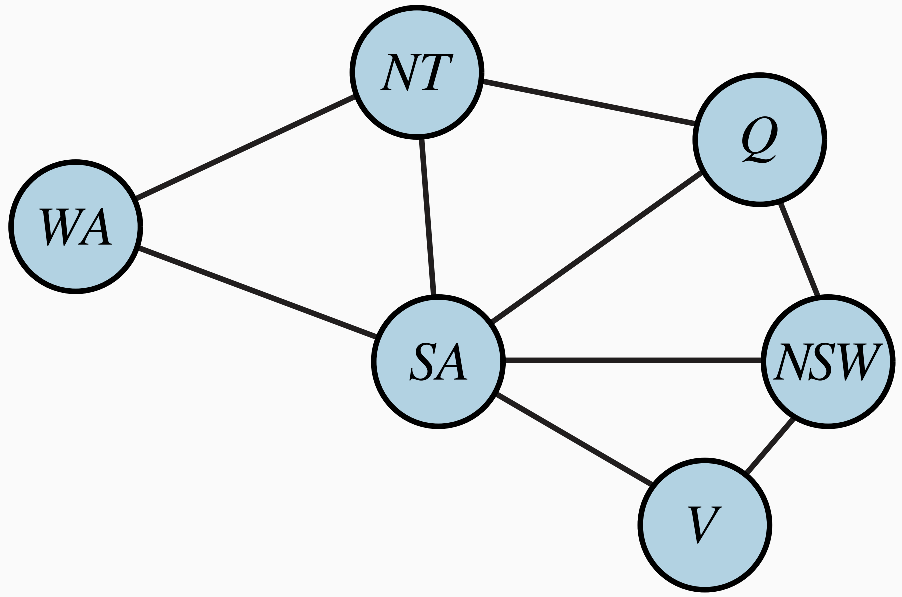


- The second for-loop of the algorithm $\rightarrow$

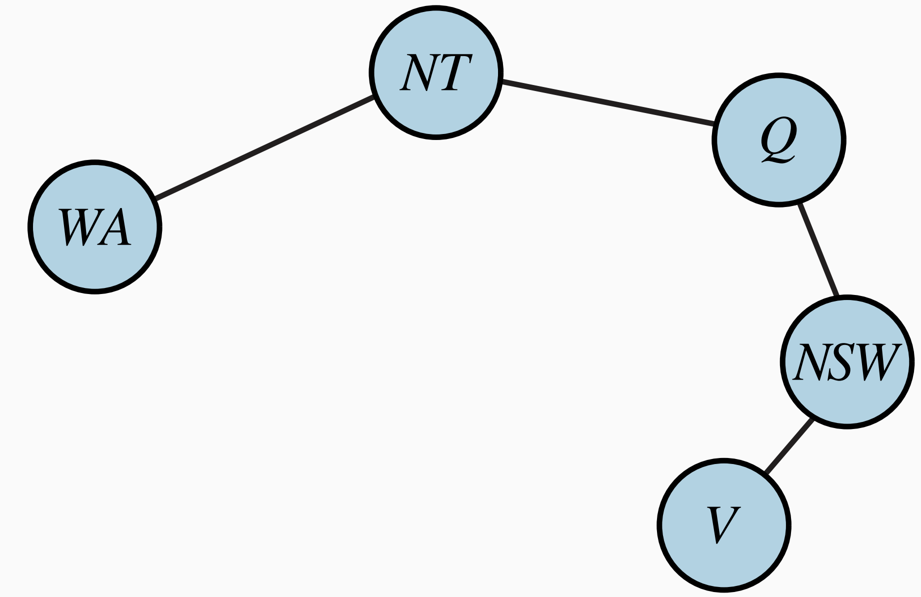


Outline

Cutset conditioning



The original graph



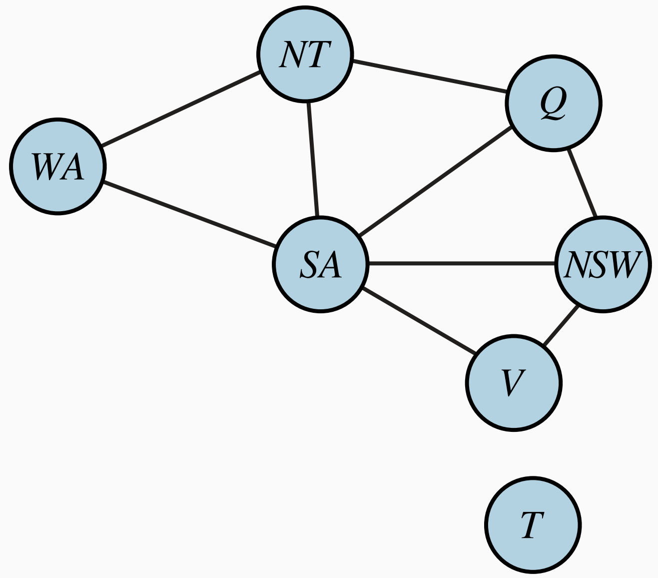
After removing the **cycle cutset** $= \{\text{SA}\}$, the graph becomes a tree

Algorithm

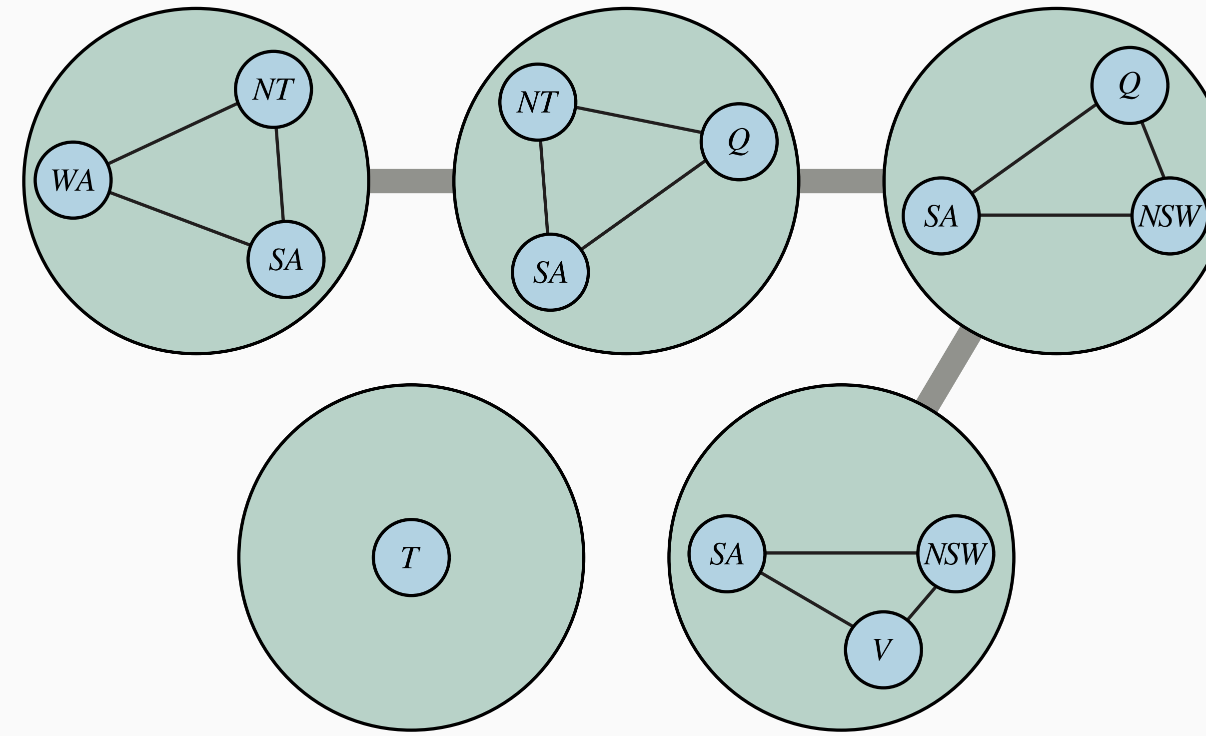
For each possible assignment to the **cycle cutset**:

- remove values from the remaining domains that would violate a constraint with the **cycle cutset**
- if the remaining tree-structured CSP has a solution, return the solution together with the assignment to the **cycle cutset**

Tree decomposition



The original graph



Decomposition of the graph

Algorithm

- Decompose the original graph into a tree where each node consists of overlapping subproblems that are solved independently
- Solve the tree-structured CSP