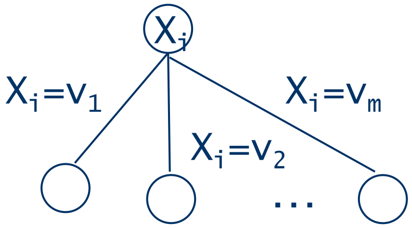
PART III: Search

Constraint Solver

- Enumerates all possible variable-value combinations via a systematic backtracking tree search.
  - Guesses a value for each variable.
- During search, examines the constraints to remove inconsistent values from the domains of the future (unexplored) variables, via propagation.
  - Shrinks the domains of the future variables.

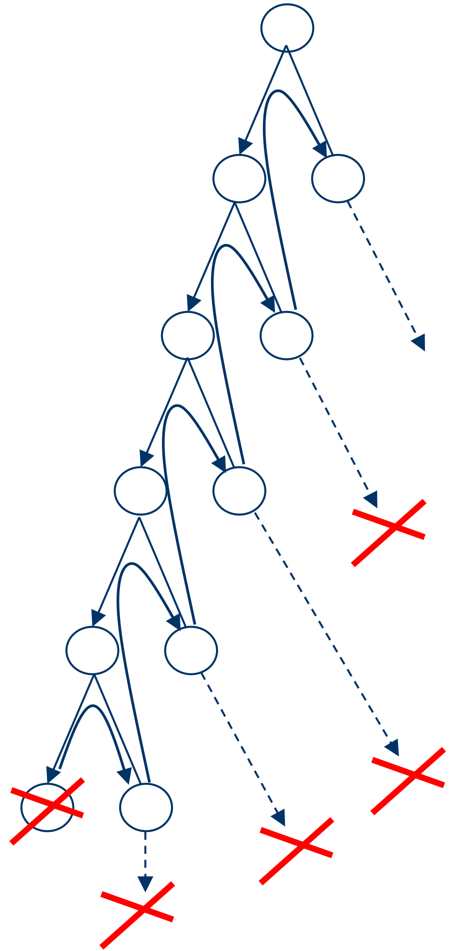
Backtracking Tree Search (BTS)

- Node → variable X_i
- Branch → decision on X_i
  - E.g., enumeration with single values from $D(X_i)$



Backtracking Tree Search (BTS)

- Variables are instantiated sequentially.
- By default depth-first traversal.

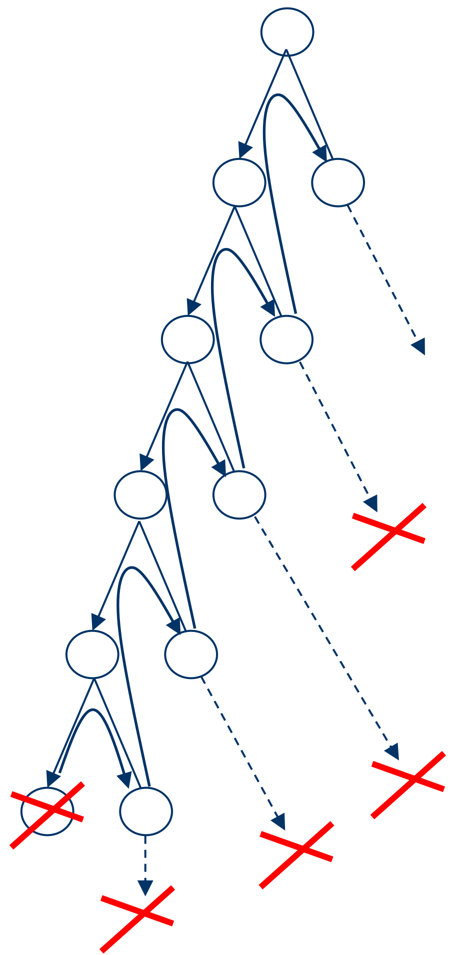


BTS without Propagation

- Enumerates all possible variable-value combinations via a systematic backtracking tree search.
  - Guesses a value for each variable.
- [redacted]
  - [redacted]

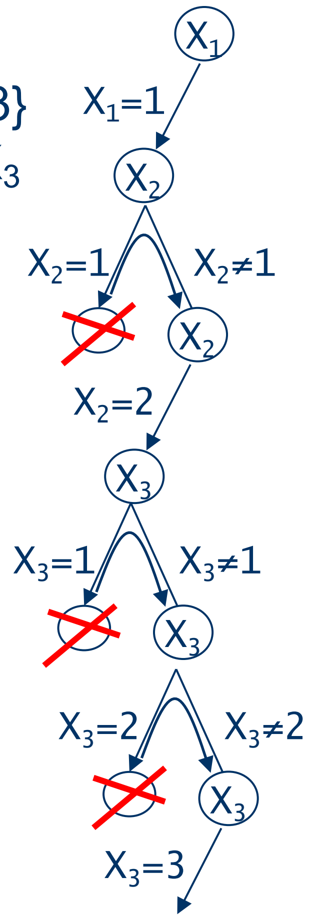
BTS without Propagation

- Whenever all the variables of a constraint is instantiated, the validity of the constraint is checked.
  - In case of dead-end, the most recently posted branching decision is retracted (chronological backtracking).
- Systematic search.
  - Eventually finds a solution or proves unsatisfiability.
  - Complexity $O(d^n)$, exponential!



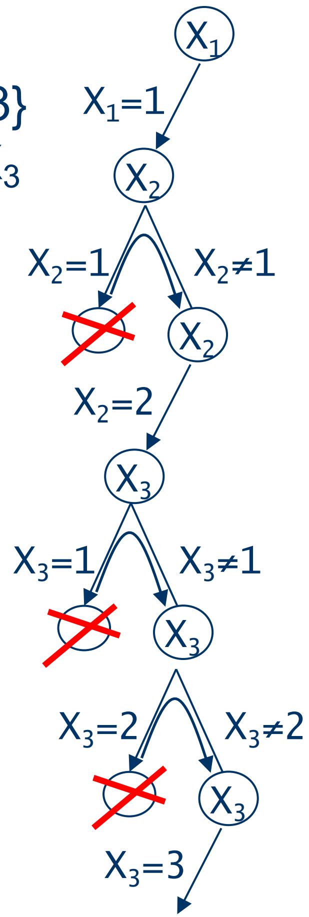
BTS without Propagation

$D(X_1) = \{1,2\}$
$D(X_2) = D(X_3) = \{1,2,3\}$
$C_1: X_1 \neq X_2$ $C_2: X_2 < X_3$



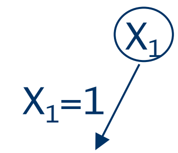
BTS interleaved with Propagation

$D(X_1) = \{1,2\}$
$D(X_2) = D(X_3) = \{1,2,3\}$
$C_1: X_1 \neq X_2$ $C_2: X_2 < X_3$



Propagation

$C_2: D(X_2) = \{1,2,\cancel{3}\}, D(X_3) = \{\cancel{1},2,3\}$

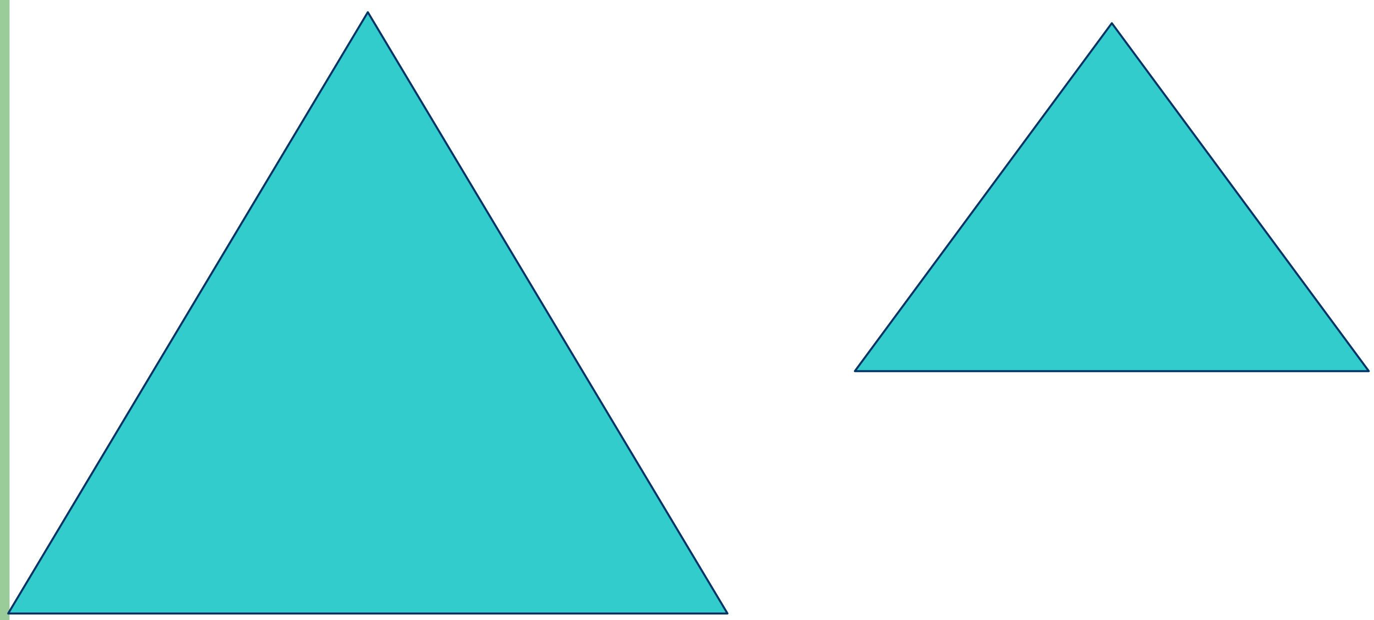


Propagation

$C_1: D(X_2) = \{\cancel{1},2\}$
$C_2: D(X_3) = \{\cancel{2},3\}$

BTS interleaved with Propagation

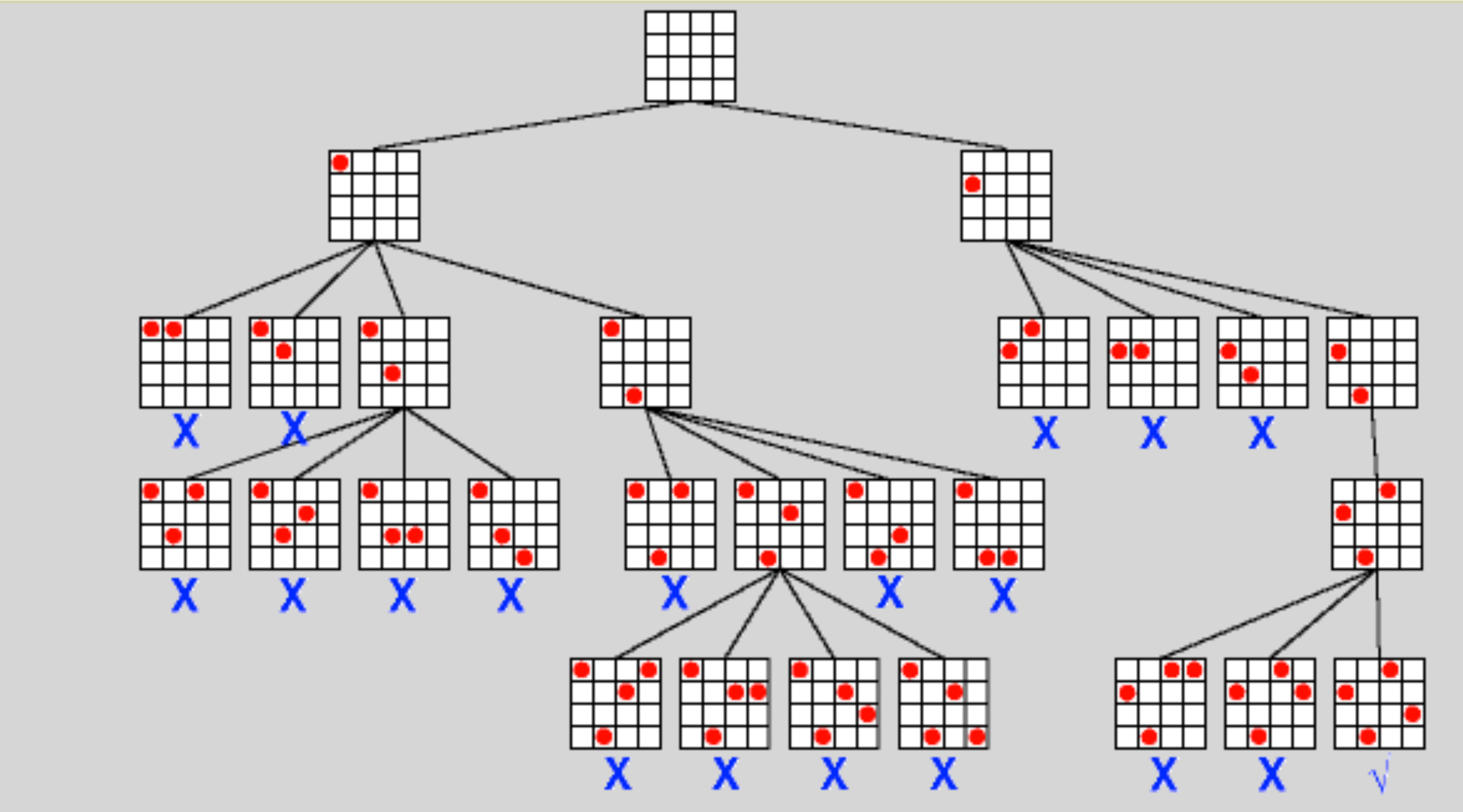
Search and Propagation



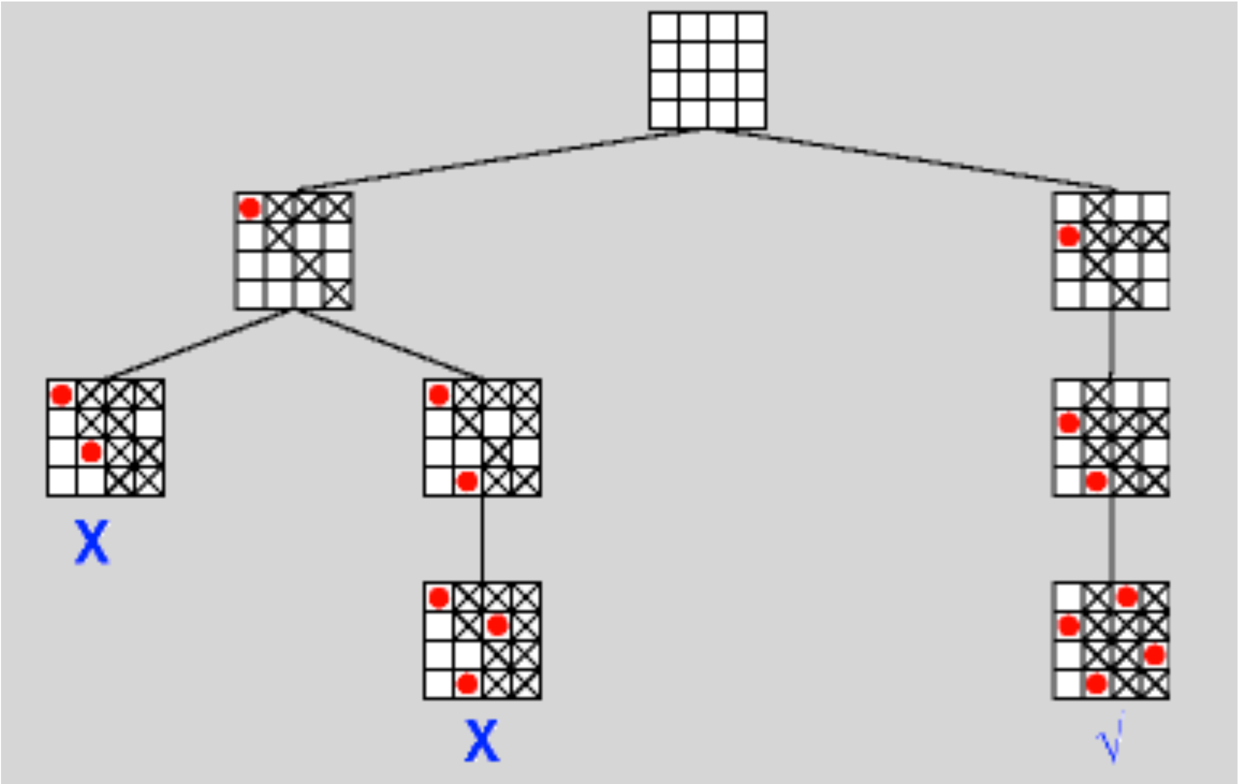
Exponential size

Reduction of the search tree size

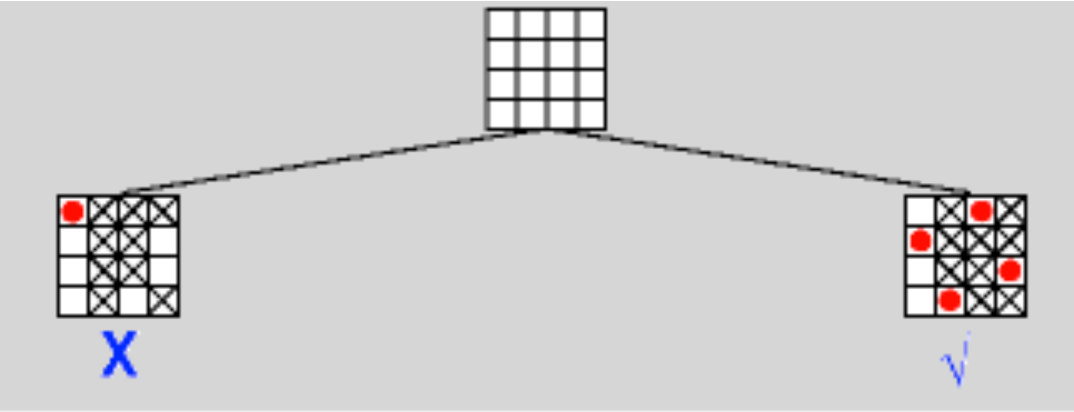
BTS without Propagation



BTS + Forward Checking Propagation



BTS + AC Propagation

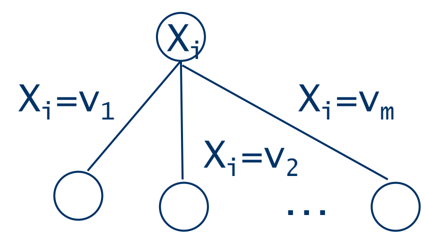


Outline

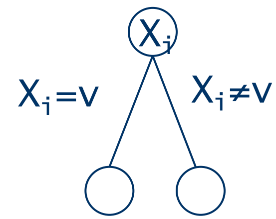
- Depth-first Search (DFS)
  - Branching Decisions
  - Branching/Search Heuristics
  - Randomization and Restarts
- Best-First Search (BFS)
  - Limited Discrepancy Search (LDS)
  - Depth-bounded Discrepancy Search (DDS)
- Constraint Optimization Problems

Branching Decisions

- Usually consists of posting a unary constraint on a chosen variable X_i.
- Enumeration (or labelling) with single values from $D(X_i)$.
  - d-way branching:
    - One branch is generated by $X_i = v_j$ for each $v_j \in D(X_i)$.

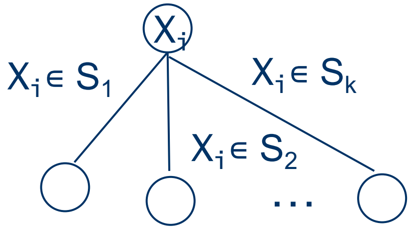


  - 2-way branching:
    - 2 branches are generated by $X_i = v$ and $X_i \neq v$ for some $v \in D(X_i)$.

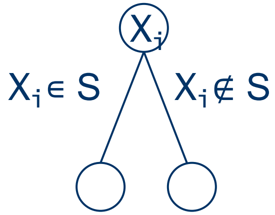


Branching Decisions

- Usually consists of posting a unary constraint on a chosen variable X_i.
- Domain partitioning of $D(X_i)$.
  - k-way branching:
    - One branch is generated by $X_i \in S_j$ for each partition S_j of D_i.



  - 2-way branching:
    - 2 branches are generated by $X_i \in S$ and $X_i \notin S$ for some $S \subseteq D_i$.

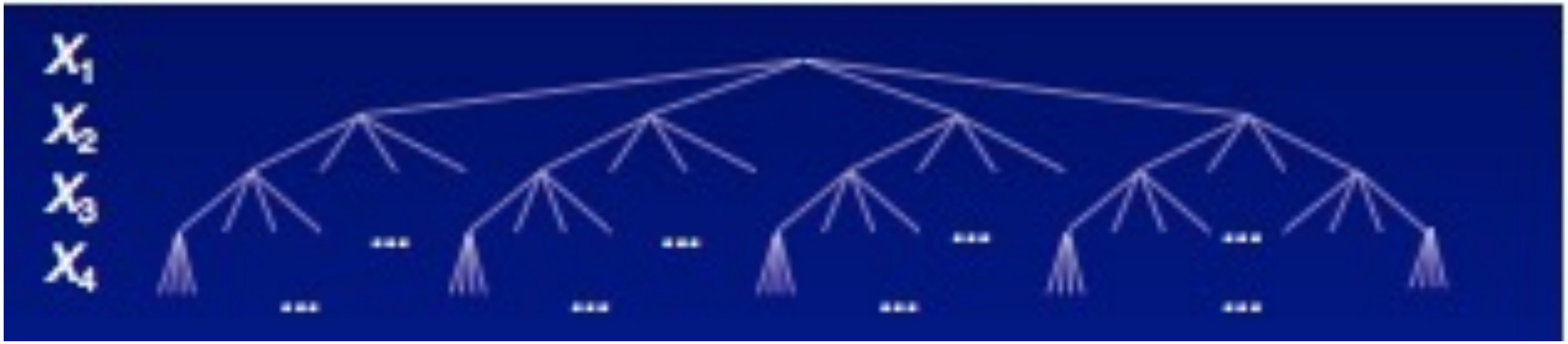


Branching/Search Heuristics

- Guide the search.
  - For a branching decision, need to choose a variable X_i and a (set of) value v_j.
  - Which variable next? Which value(s) next?
- Known also as variable and value ordering (vvo) heuristics.
- Static vs dynamic heuristics.
- Problem specific vs generic heuristics.

Static Variable Ordering Heuristics

- A variable is associated with each level.
- Branches are generated in the same order all over the tree.
- Calculated once and for all before search starts, hence cheap to evaluate.



Some Static Generic VOHs

- Lexicographic: The order of definition in case of a sequence of variables:
  - $X_1, X_2, \ldots, X_n$
- Top down, left to right, row by row in case of a matrix of variables:
  - $X_{11}, X_{12}, \ldots, X_{1m}$

    $X_{21}, X_{22}, \ldots, X_{2m}$

    …

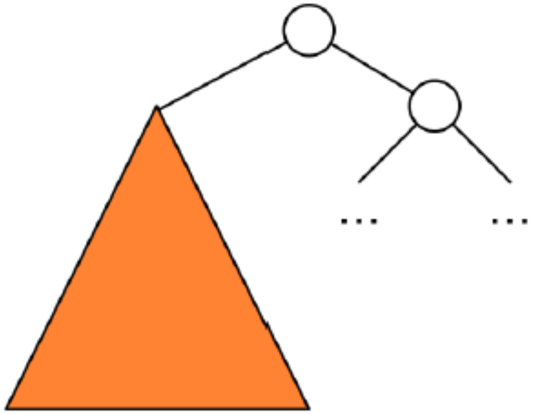
    $X_{n1}, X_{n2}, \ldots, X_{nm}$

Dynamic Variable Ordering Heuristics

- At any node, any variable & branch can be considered.
- Decided dynamically during search, hence costly.
- Takes into account the current state of the search tree.

Search Heuristics

- For feasible problems, choose variables and values that are likely to yield a solution.
  - In general, no guarantee of feasibility.
- What if we make a mistake?
  - Infeasible sub-problem!
  - We need to explore the whole sub-tree before backtracking!
  - We should explore the sub-tree as quickly as possible.



Heuristics for Infeasible Problems

- Fail-first (FF) principle: Try first where you are most likely to fail.
  - Aims at proving, as soon as possible, that the search is in a sub-tree with no feasible solutions.
- How do we know if a CSP is feasible or not?
- Trade-off:
  - choose next the variable that is most likely to cause failure;
  - choose next the value that is most likely to be part of a solution (least constrained value).
- Main focus on Variable Ordering Heuristics (VOHs).
  - To backtrack from an infeasible sub-problem, we need to explore all the values in the domain of a variable.

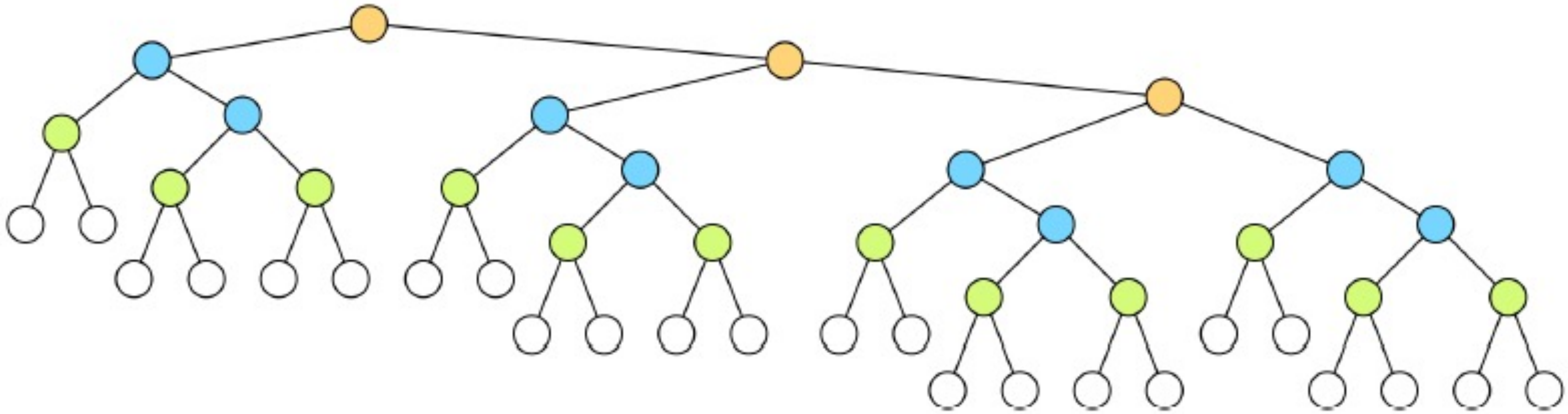
Generic Dynamic VOHs based on FF

- Minimum domain (dom)
  - Choose next the variable with minimum domain size.
  - Idea: minimize the search tree size.

Dom Heuristic

- Consider the order X_1, X_2, X_3.

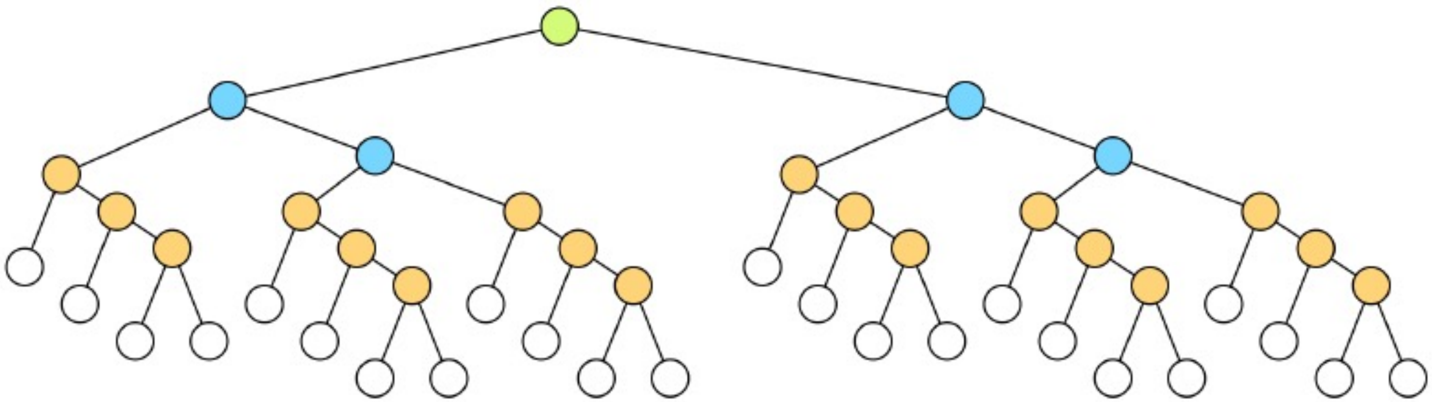
$X_1 \in \{0, 1, 2, 3\}$, $X_2 \in \{0, 1, 2\}$, $X_3 \in \{0, 1\}$



Dom Heuristic

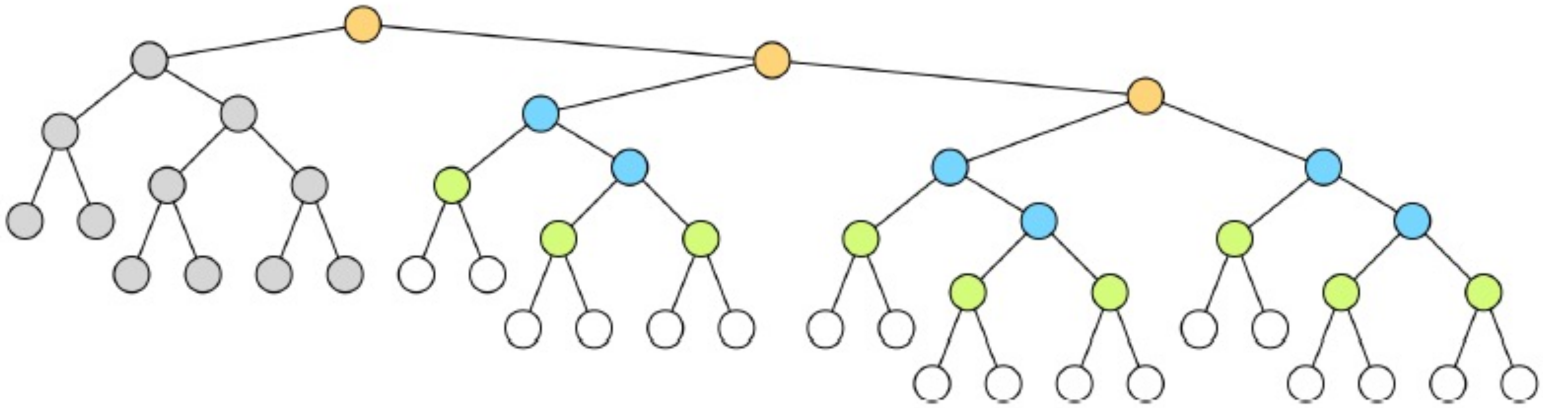
- Consider the order X_3, X_2, X_1.

$X_3 \in \{0, 1\}$, $X_2 \in \{0, 1, 2\}$, $X_1 \in \{0, 1, 2, 3\}$



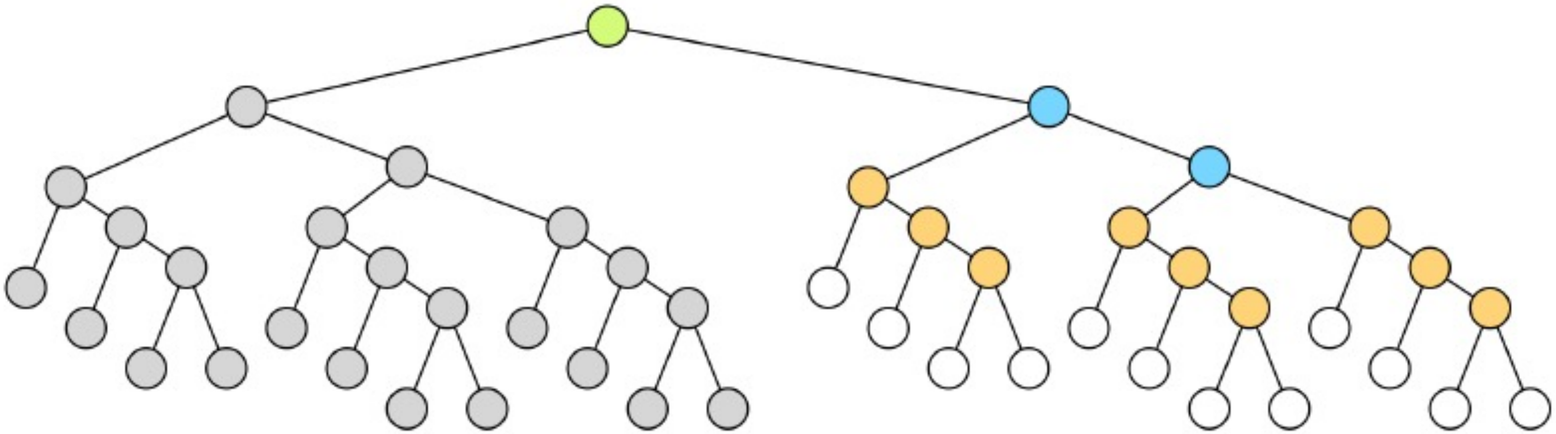
Dom Heuristic

- If propagation prunes a value at depth 1…



Dom Heuristic

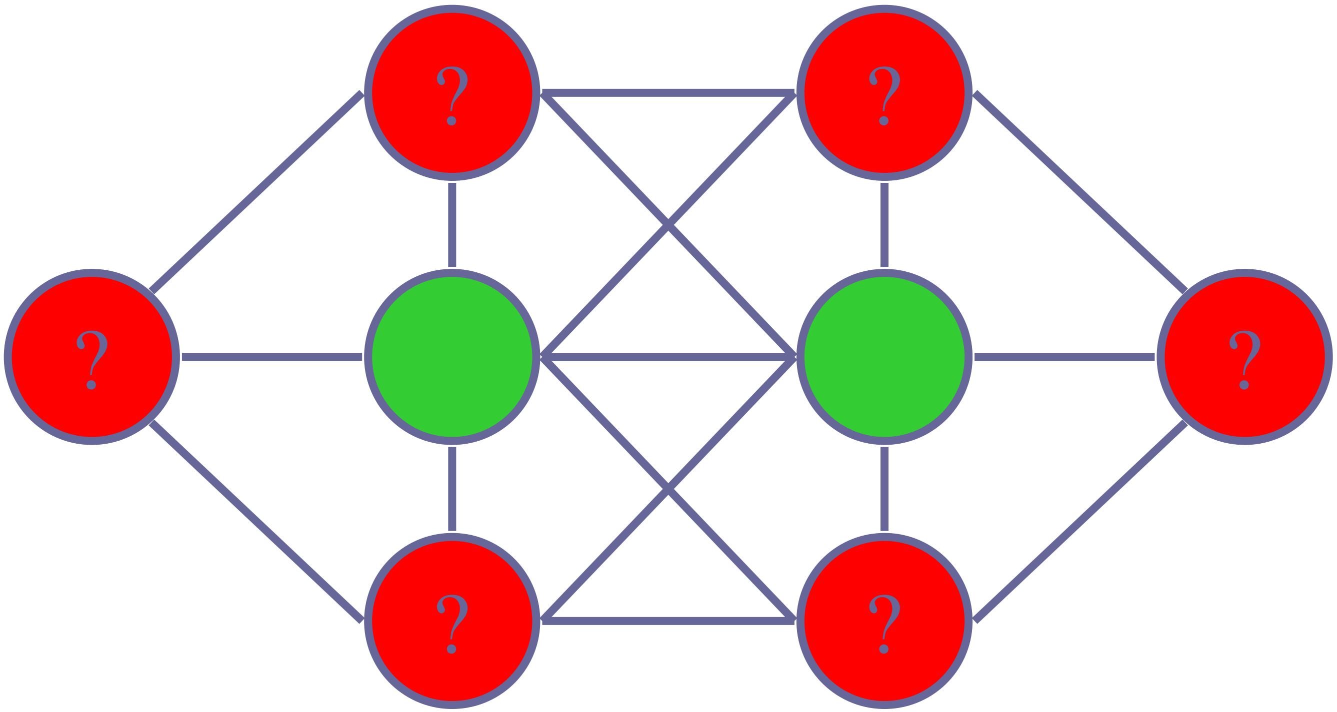
- …the effect is much stronger with the second ordering!



Generic Dynamic VOHs based on FF

- Minimum domain (dom)
  - Choose next the variable with minimum domain size.
  - Idea: minimize the search tree size.
- Most constrained (deg)
  - Choose next the variable involved in most number of constraints.
  - Idea: maximize constraint propagation.

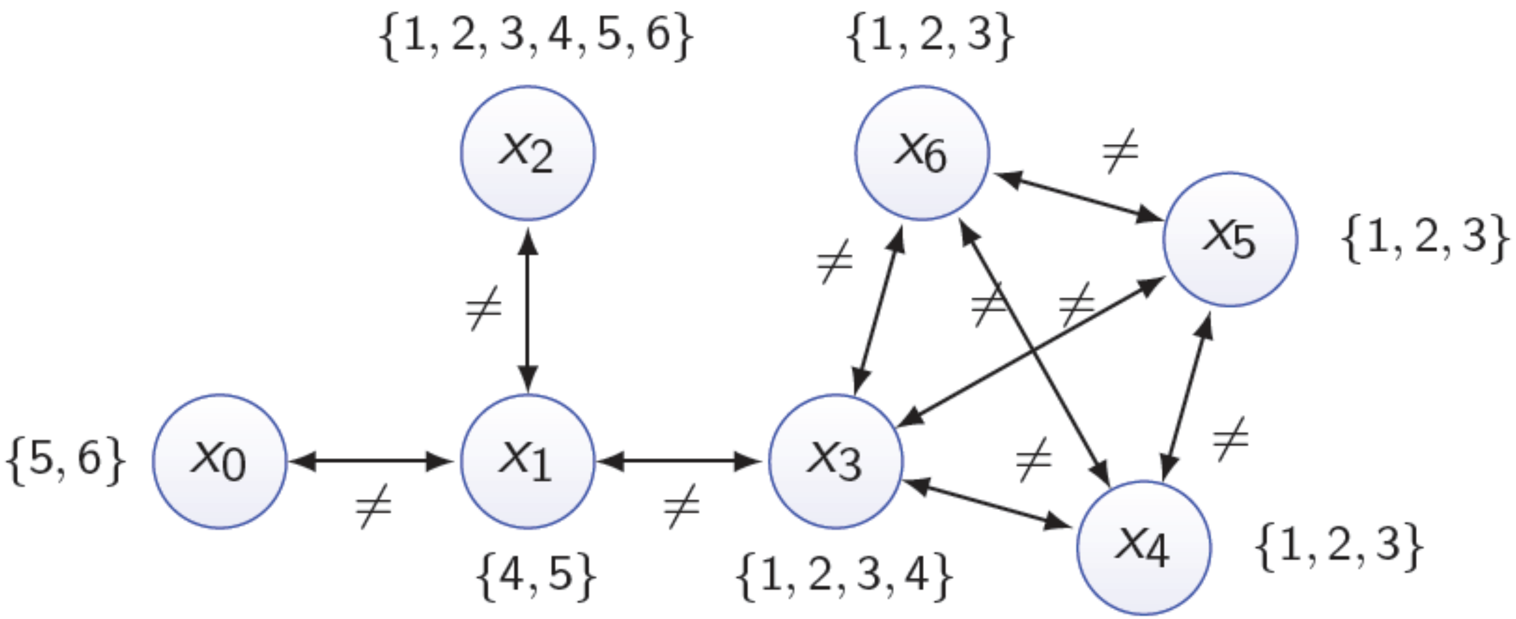
Most Constrained Variables



Generic Dynamic VOHs based on FF

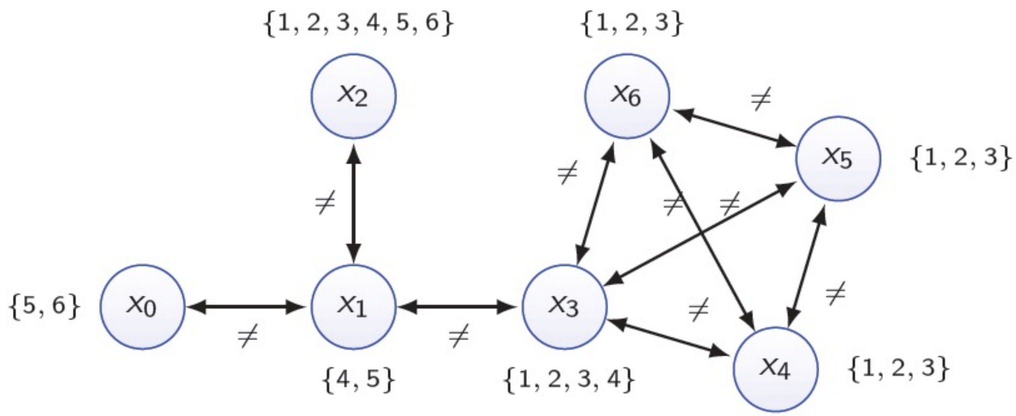
- Minimum domain (dom)
  - Choose next the variable with minimum domain.
  - Idea: minimize the search tree size.
- Most constrained (deg)
  - Choose next the variable involved in most number of constraints.
  - Idea: maximize constraint propagation.
- Combination
  - Minimize dom / deg

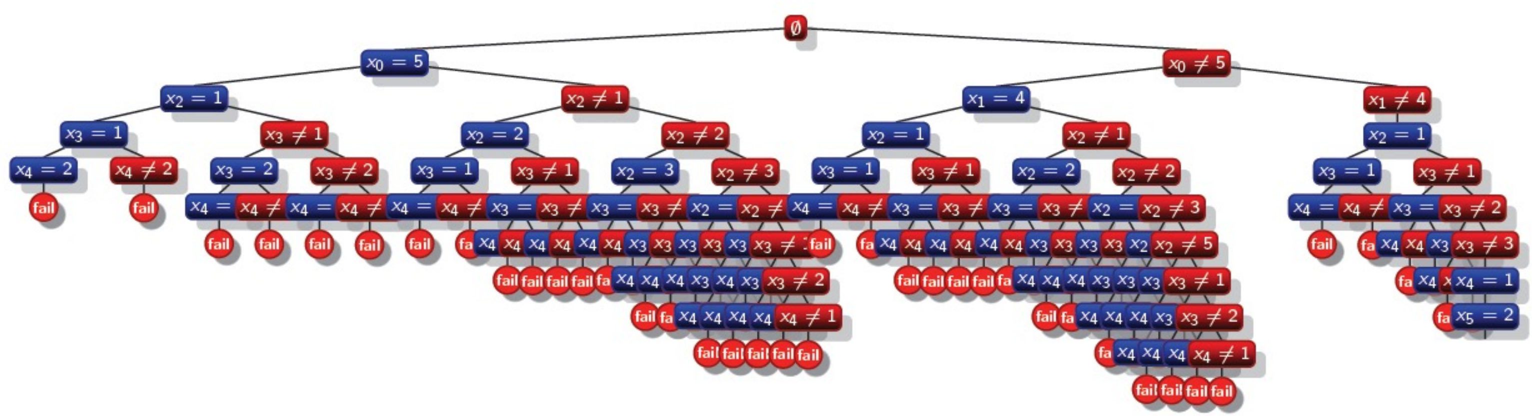
Map Colouring



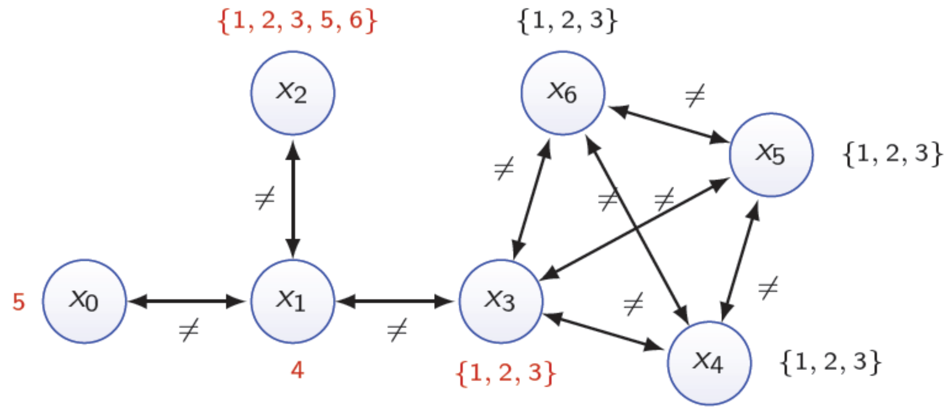
- Maintain AC during search with 2-way branching using various heuristics.

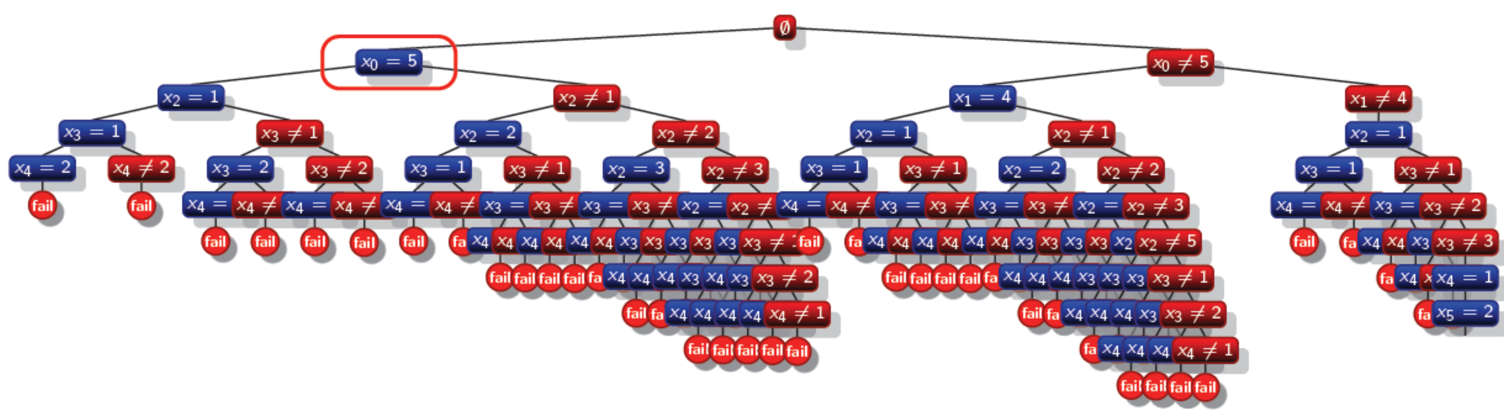
Lexicographic Ordering



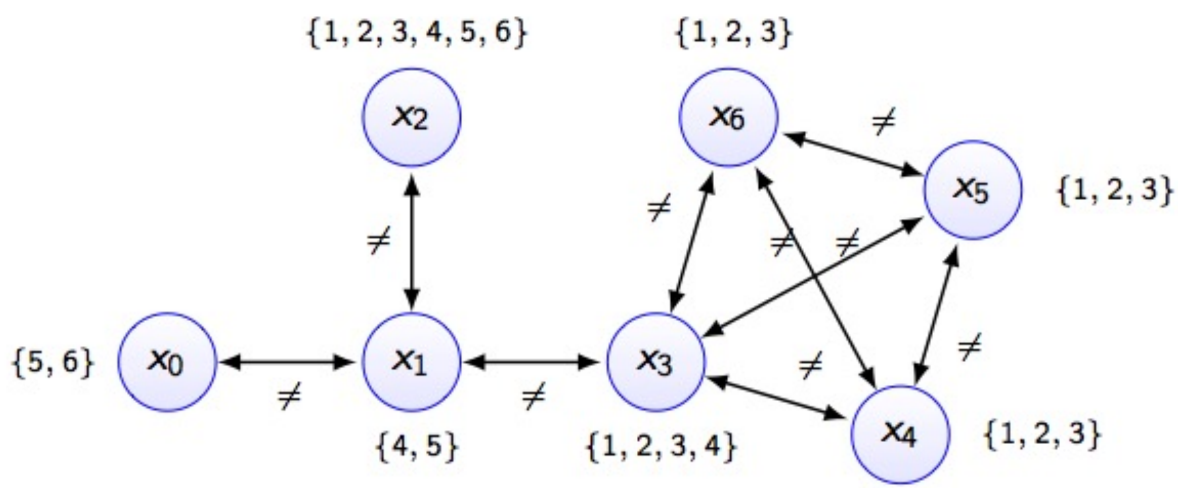


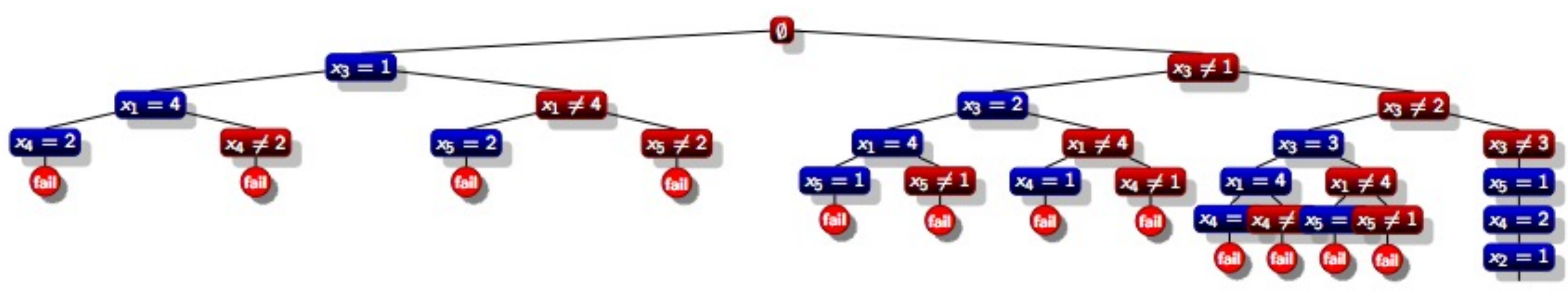
Lexicographic Ordering



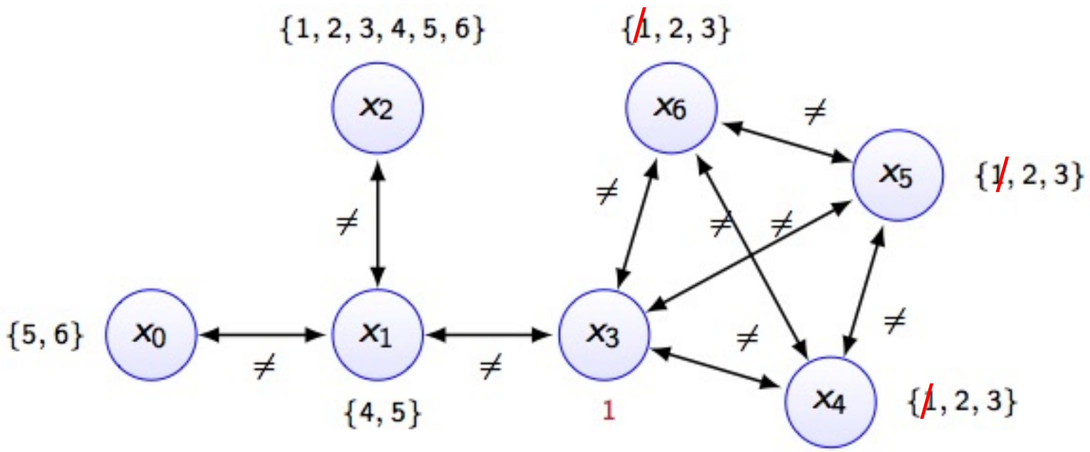


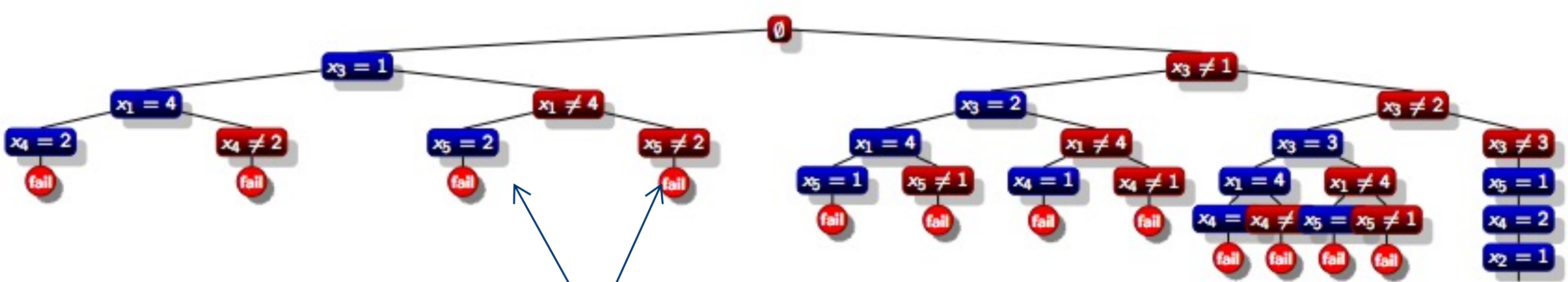
Maximum Degree





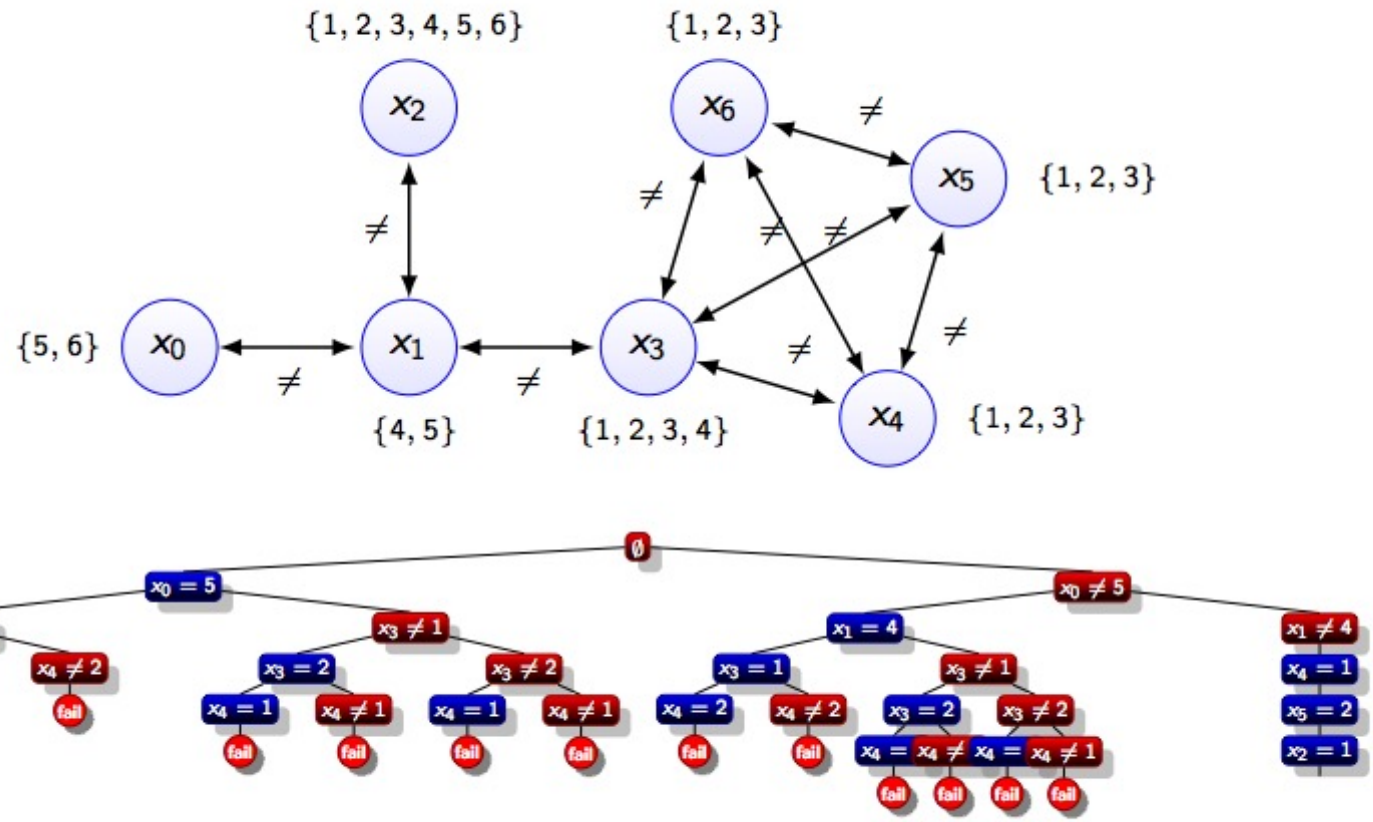
Maximum Degree



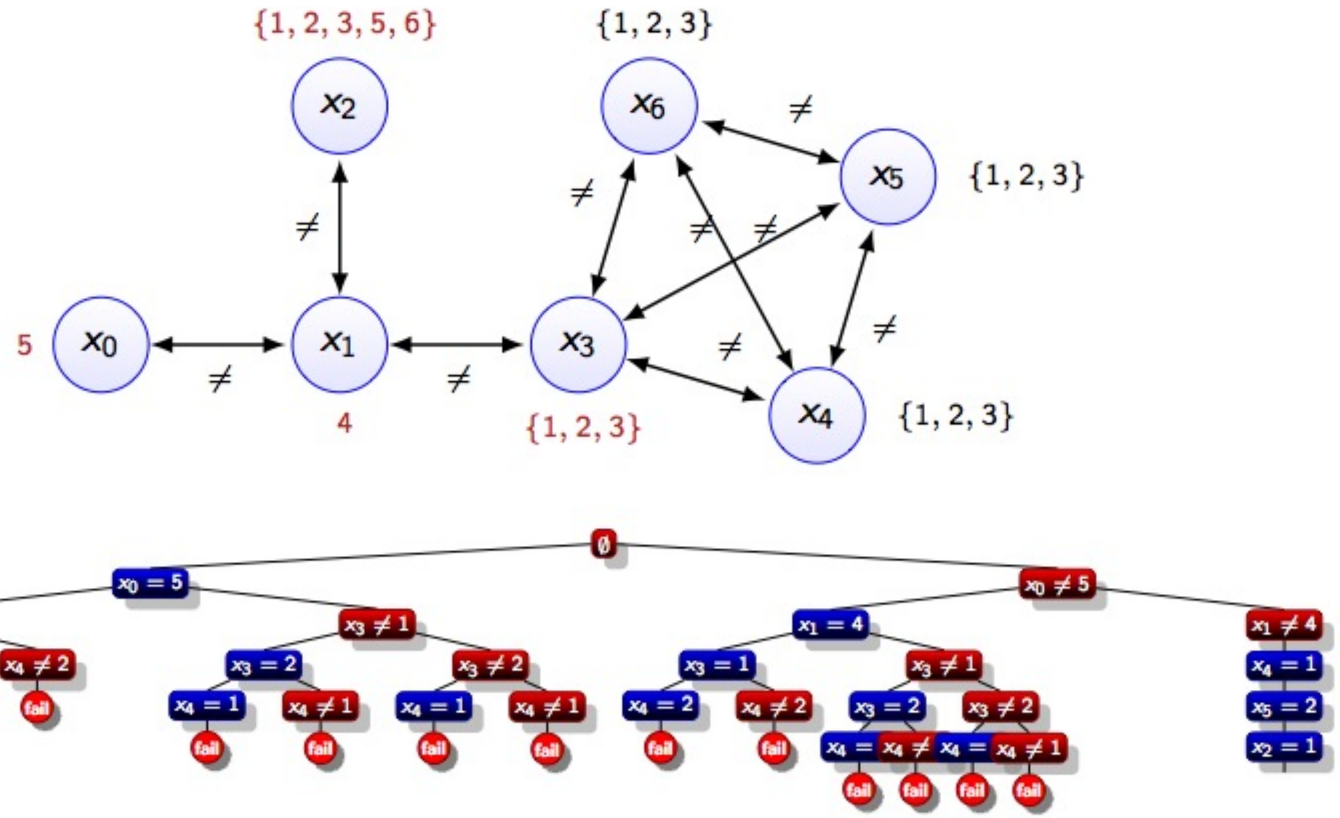


Correction: x_4

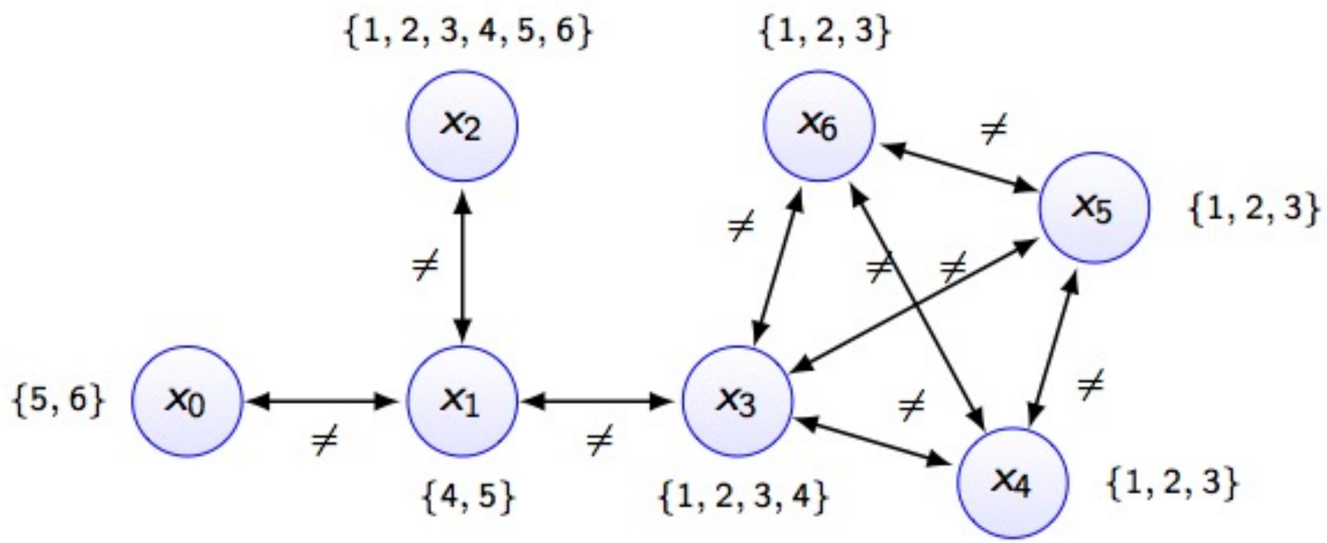
Minimum Domain

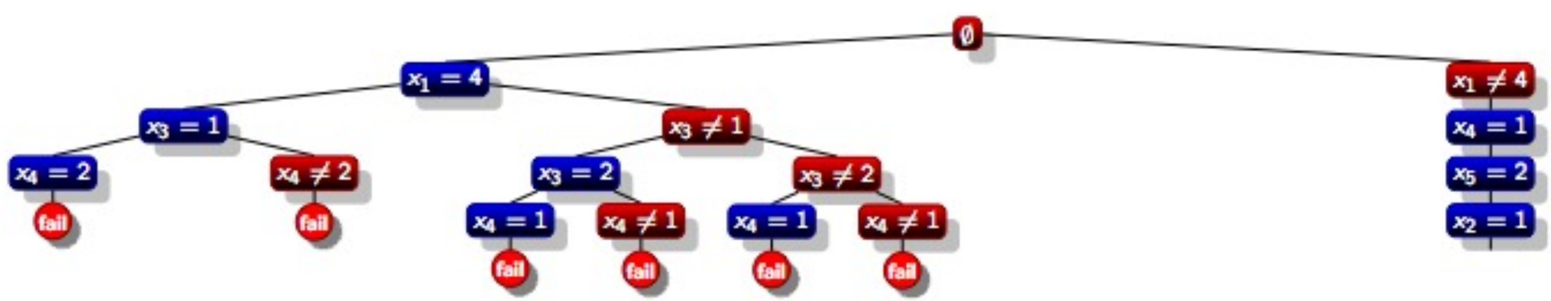


Minimum Domain

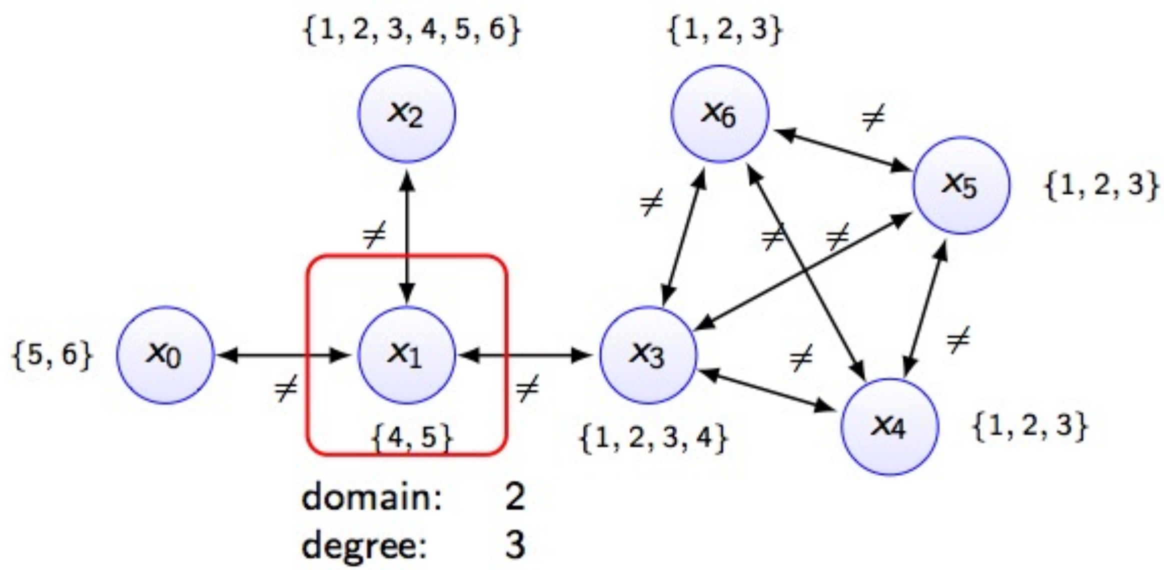


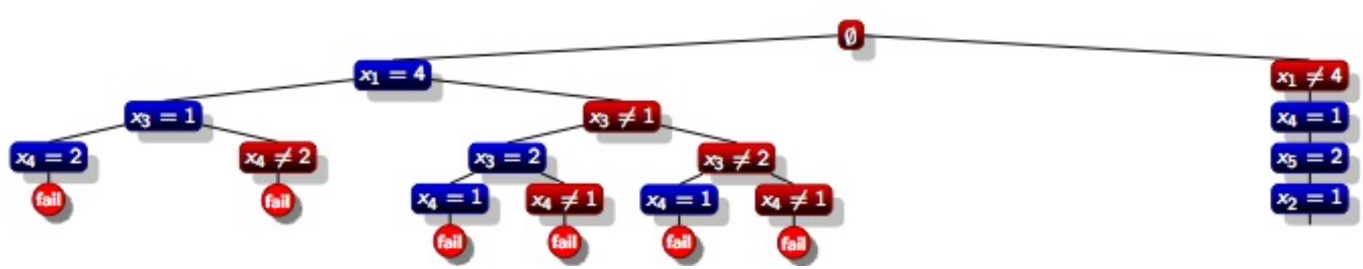
Minimum Domain / Degree



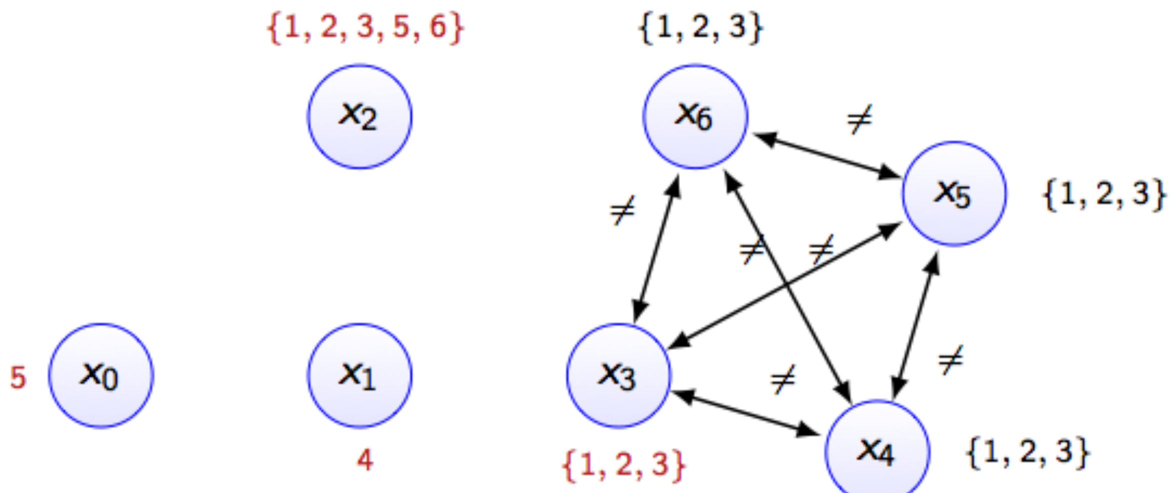


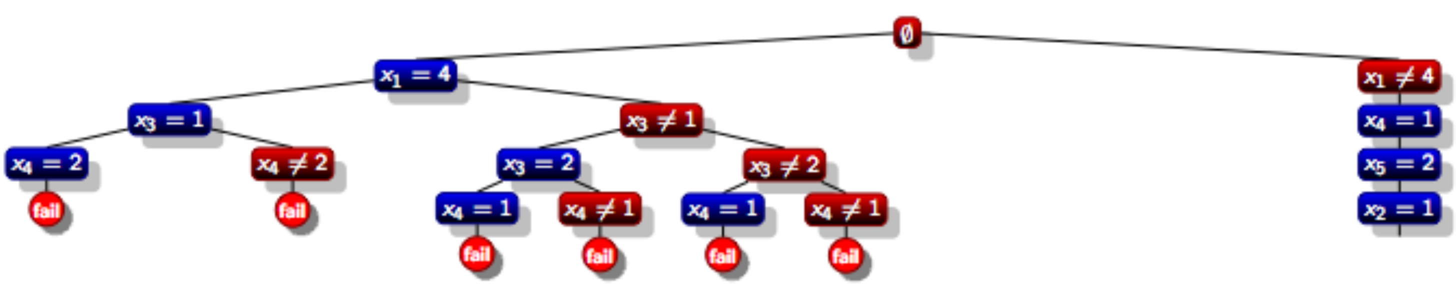
Minimum Domain / Degree



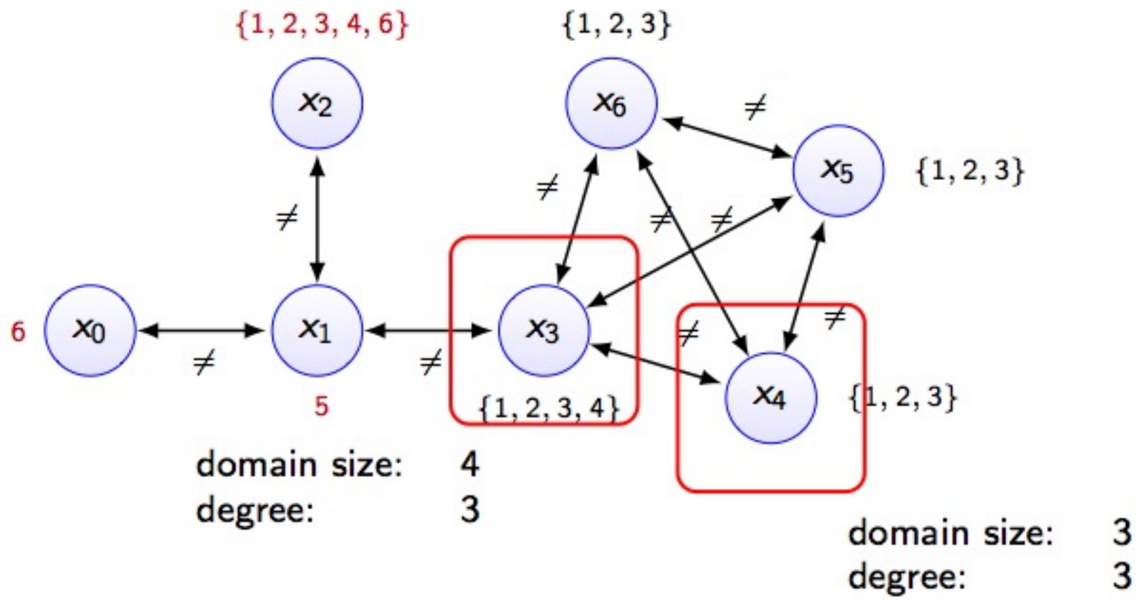


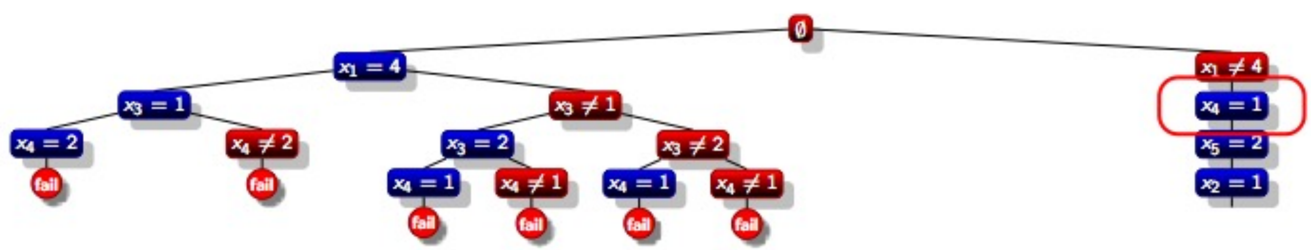
Minimum Domain / Degree





Minimum Domain / Degree





Weighted Degree Heuristic

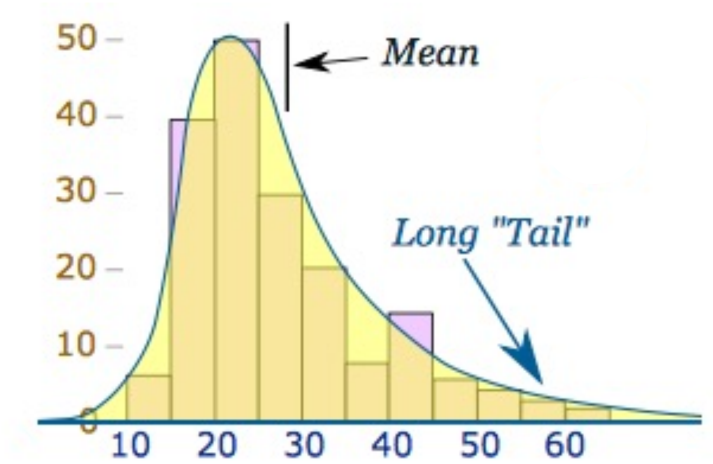
- Constraints are weighted.
  - Initially set to 1.
- During the propagation of a constraint c, its weight w(c) is incremented by 1 if the constraint fails.
- The weighted degree of a variable X_i:

$$w(X_i) = \sum_{c \ s.t.\ X_i \in \mathcal{X}(c)} w(c)$$

- Domain over weighted degree heuristic (domWdeg):
  - Choose the variable X_i with minimum $dom(X_i) / w(X_i)$.

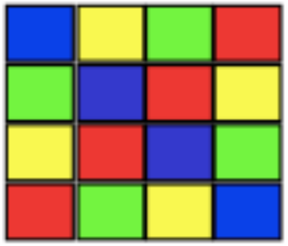
Heavy Tail Behaviour

- Given a collection of instances of a problem, we often observe some exceptionally hard instances that take exceptionally longer time to solve.
  - Large impact on the runtime distributions for a given set of instances.



Latin Squares

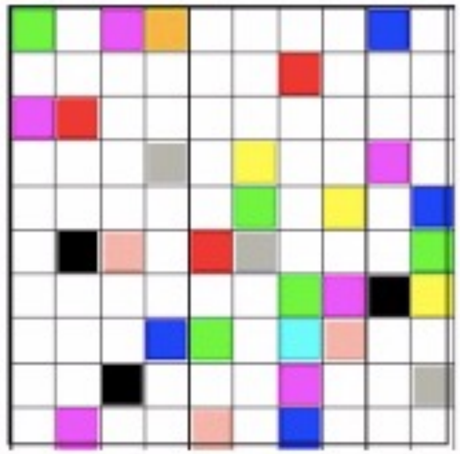
- Given an nxn matrix and n colours, a Latin square of order n is a coloured matrix such that all cells are coloured, each colour appears exactly once in each row and in each column.

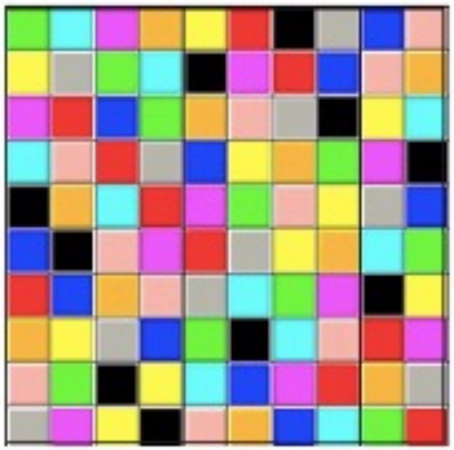


- Applications in fiber optic networks, design of statistical experiments, scheduling and timetabling.

Quasigroup Completion Problem

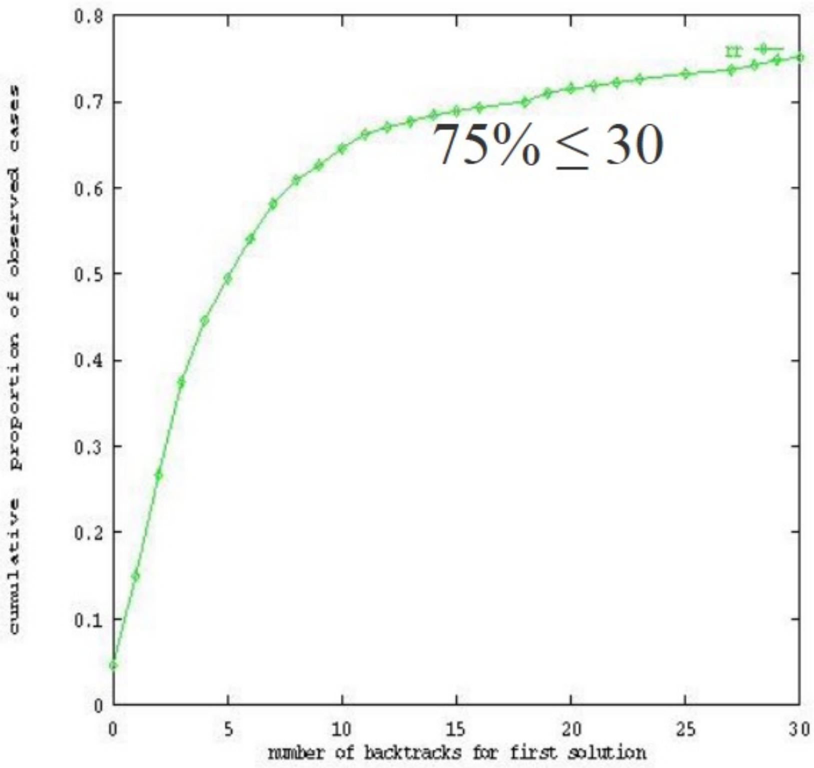
- Given a partial assignment of colours, can the partial Latin square (quasigroup) be completed so that we obtain a Latin square?

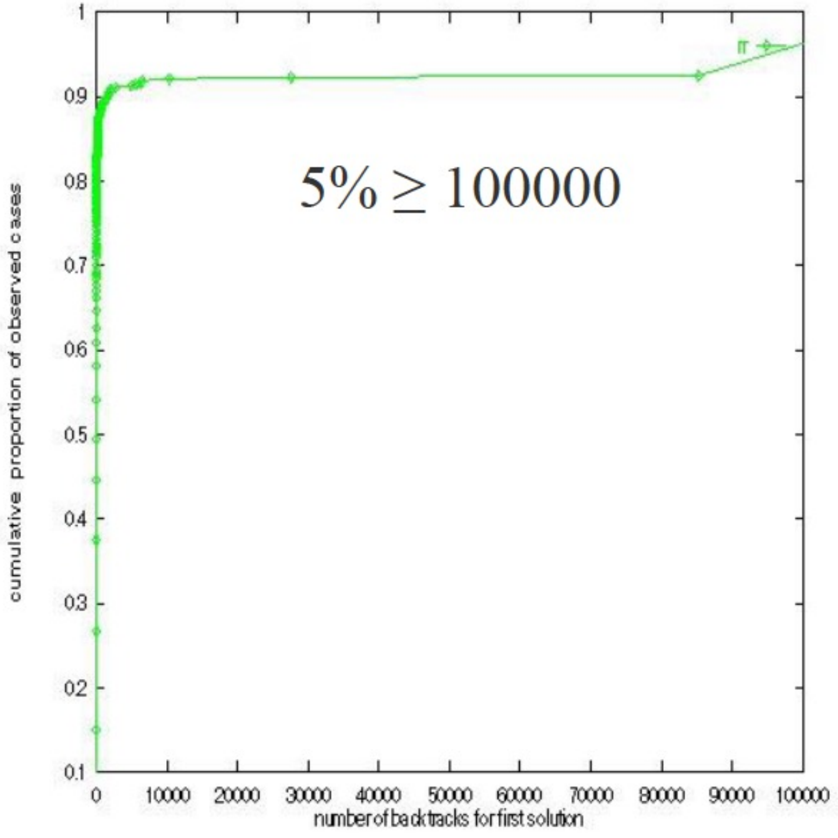




Quasigroup Completion Problem

- 11x11 matrix with 30% pre-assignments





Heavy Tail Behaviour

- Not a characteristic of the instance!
  - The same behaviour is observed if we run several times the same instance while varying some parameter (like the variable ordering) of the solver.
  - For some combination instance + solver parameters, we get trapped into an exponential subtree.
- Intuitive reason:
  - If we make a mistake early during search, we get stuck in trashing.
    - Remember the puzzle example!
  - Different heuristics lead to “bad” mistakes on different instances.
- Observation: such mistakes are seemingly random.

Heavy Tail Behaviour

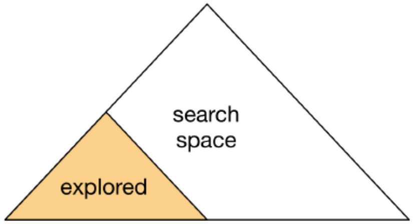
- Randomization
  - Add some randomized parameter in search. E.g.,
    - Pick (some) variables/values at random.
    - Break ties randomly.
  - Given the same random seed, the solver will explore the same tree, however it will never explore two identical subproblems in the same way.

Heavy Tail Behaviour

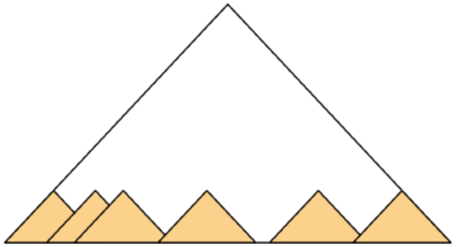
- Restarting
  - Restart the search, after certain amount of resources are consumed.
    - Usually in the form of search steps, such as the number of visited nodes.
  - In the subsequent runs, search differently.
    - Introduce randomization.
    - Learn from the accumulated experiences of previous runs.

Heavy Tail Behaviour

- Randomization + restarts eliminates the huge variance in solver performance.
- Without randomization + restarts



- With randomization + restarts



Restart Strategies

- Constant restart
  - Restart after using L resources.
- Geometric restart
  - Restart after L resources, with the new limit α*L.
  - Ends up being L, α*L, α^{2*}L, α^{3*}L, …
- Luby restart
  - Restart after s[i]*L resources where s[i] is the i^{th} number in the Luby sequence = [1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, …], which repeats two copies of the sequence ending in 2^i before adding the number 2^{i+1}.

domWdeg & Restarts

- domWdeg heuristic works well with restart.
  - Collected fail counts can be carried over to subsequent runs.
- domWdeg combined with random choice of values can be very effective!

Problems with DFS

- For many problems, heuristics are more accurate at deep nodes.
  - Often first decision is wrong.
- DFS:
  - puts tremendous burden on the heuristics early in the search and light burden deep in the search;
  - consequently mistakes made near the root of the tree can be costly to discover and undo.
  - Remember the puzzle example!

Problems with DFS

- Best-first search (BFS) strategy is of interest.
- BFS explores first the nodes that are most promising according to some heuristic evaluation.

Outline

- Depth-first Search (DFS)
  - Branching Decisions
  - Branching Heuristics
  - Randomization and Restarts
- Best-First Search (BFS)
  - Limited Discrepancy Search (LDS)
  - Depth-bounded Discrepancy Search (DDS)
- Constraint Optimization Problems

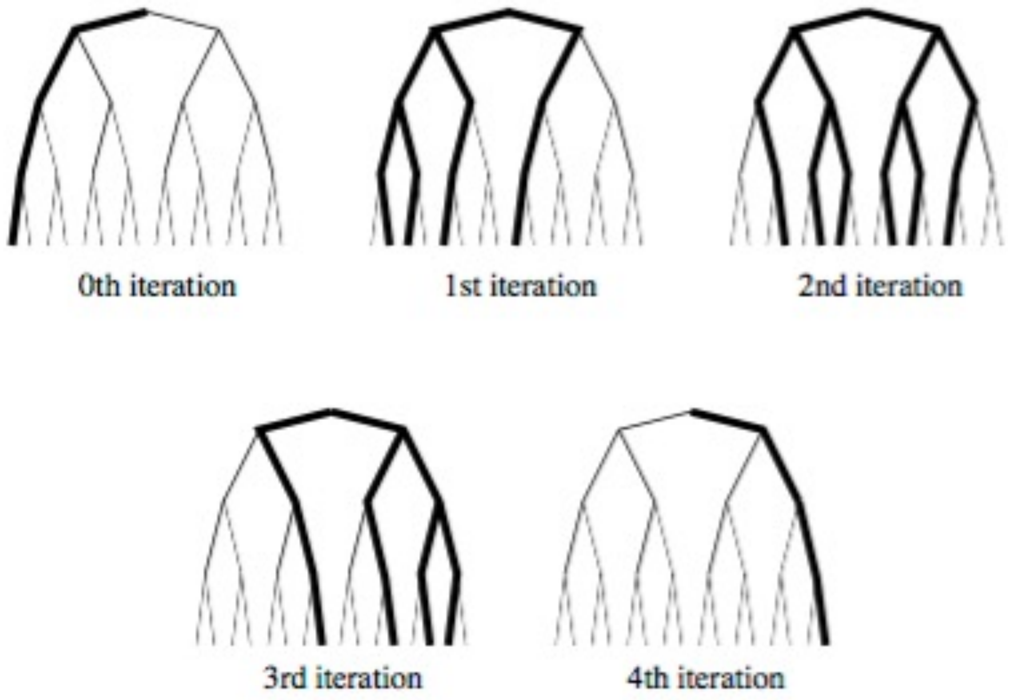
Limited Discrepancy Search

- A discrepancy is any decision in a search tree that does not follow the heuristic (any right branch out of a node).
- LDS
  - Trusts the heuristic and gives priority to the left branches.
  - Iteratively searches the tree by increasing number of discrepancies.
    - On the 0^{th} iteration, explore the leftmost branches.
    - On the 1^{sth} iteration, explore all left branches except 1 branch.
    - On the 2^{nd} iteration, explore all left branches except 2 branches.
    - …

Limited Discrepancy Search

- LDS
  - On the i^{th} iteration, LDS visits all leaf nodes with i discrepancies.
  - **Motivation**: the branching heuristic has hopefully made a few mistakes, and LDS allows a small number of mistakes to be corrected at little cost.
  - By contrast, DFS needs to explore a significant fraction of the tree before undoing an early mistake.

Limited Discrepancy Search



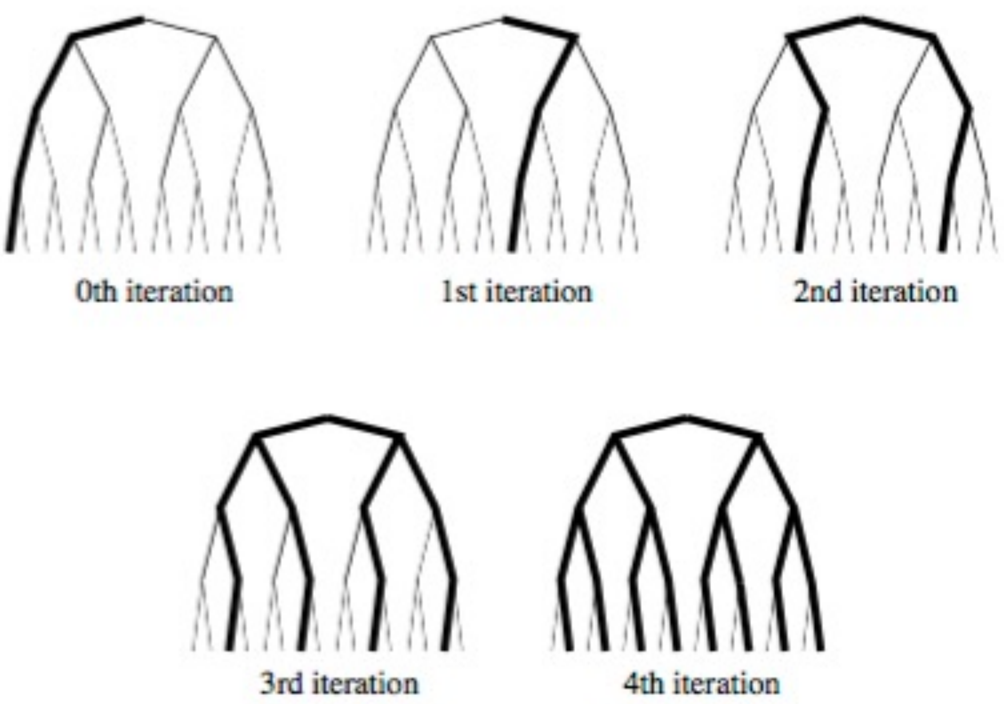
Problems with LDS

- All discrepancies are alike, irrespective of their depth.
- Heuristics tend to be less informed and make more mistakes at the top of the search tree.
- It is worth exploring discrepancies at the top of the tree before those at the bottom.

Depth-bounded Discrepancy Search

- Biases search to discrepancies high in the tree via an iteratively increasing depth bound.
  - Discrepancies below this depth are prohibited.
  - On the 0^{th} iteration, DDS = LDS.
  - On the i^{th} iteration, DDS explores those branches on which discrepancies occur at a depth of i or less.
  - At lesser depths, DDS explores more discrepancies.
  - At greater depths, DDS follows the heuristic.

Depth-bounded Discrepancy Search



Outline

- Depth-first Search (DFS)
  - Branching Decisions
  - Branching Heuristics
  - Randomization and Restarts
- Best-First Search (BFS)
  - Limited Discrepancy Search (LDS)
  - Depth-bounded Discrepancy Search (DDS)
- Constraint Optimization Problems

Constraint Optimization Problems (COPs)

- CSP enhanced with an optimization criterion, e.g.:
  - minimum cost;
  - shortest distance;
  - fastest route;
  - maximum profit.
- Formally, **<X,D,C,f>** where **f** is the formalization of the optimization criterion as an objective function/variable. Goal: minimize **f** (maximize **–f**).

Optimal Map Colouring

- What is the minimum number of colours necessary to colour the neighbouring regions differently?

Optimal Map Colouring

- Variables and Domains
  - X_i for each of n regions with domain [1..n]
- Constraints
  - $X_i \neq X_j$ for each neighbour region i and j
- Objective function/variable
  - $f = \max(X_i)$
- Objective: minimize f

Solving COPs

- Enumeration.
  - Doesn't scale up in case of too many solutions.
- Search over D(**f**).
- Branch & bound.

Searching over D(f)

- Destructive lower bound
  - Iterate over the values **v** ∊ D(**f**), starting from min(D(**f**)).
  - At each iteration, post the constraint **f ≤ v** and solve the CSP.
  - The first feasible solution is guaranteed to be optimal.
  - Why destructive?
    - Intermediate computation results are discarded.

Destructive Lower Bound

- Solve with 1 colour → fail
- Solve with 2 colours → fail
- Solve with 3 colours → success (optimal)

Searching over D(f)

- Destructive upper bound
  - Iterate over (some of) the values **v** ∊ D(**f**), starting from max(D(**f**)).
  - At each iteration, post the constraint **f ≤ v** and solve the CSP.
  - For the next iteration, set **v** = **f** -1.
  - When the problem is infeasible, the last solution is proven optimal.

Destructive Upper Bound

- Solve with 8 colours → success with 5 colours
- Solve with 4 colours → success with 4 colours
- Solve with 3 colours → success with 3 colours
- Solve with 2 colours → fail (optimality with 3 colours proven)

Upper or Lower Bounds?

- Destructive lower bound
  - CON: not an any time algorithm
  - CON: small steps
  - PRO: tighter constraints → more propagation
  - PRO: provides lower bounds

Upper or Lower Bounds?

- Destructive lower bound
  - CON: not an any time algorithm
  - CON: small steps
  - PRO: tighter constraints → more propagation
  - PRO: provides lower bounds
- Destructive upper bound
  - PRO: anytime algorithm
  - PRO: larger steps
  - CON: less propagation
  - CON: no lower bounds

Binary Search

- Combine the advantages of both!
  - Binary search over D(**f**).

Binary Search

- Main idea:
  - keep both a (feasible) upper bound **ub** and an (infeasible) lower bound **lb**;
  - solve by posting **lb** < **f** < (**lb** + **ub**)/2

Binary Search

- Main idea:
  - keep both a (feasible) upper bound **ub** and an (infeasible) lower bound **lb**;
  - solve by posting **lb** < **f** < (**lb** + **ub**)/2;
  - if feasible, update **ub**

Binary Search

- Main idea:
  - keep both a (feasible) upper bound **ub** and an (infeasible) lower bound **lb**;
  - solve by posting **lb** < **f** < (**lb** + **ub**)/2;
  - if feasible, update **ub**

Binary Search

- Main idea:
  - keep both a (feasible) upper bound **ub** and an (infeasible) lower bound **lb**;
  - solve by posting **lb** < **f** < (**lb** + **ub**)/2;
  - if feasible, update **ub**; if infeasible, update **lb**

Binary Search

- Main idea:
  - keep both a (feasible) upper bound **ub** and an (infeasible) lower bound **lb**;
  - solve by posting **lb** < **f** < (**lb** + **ub**)/2;
  - if feasible, update **ub**; if infeasible, update **lb**

Binary Search

- Main idea:
  - keep both a (feasible) upper bound **ub** and an (infeasible) lower bound **lb**;
  - solve by posting **lb** < **f** < (**lb** + **ub**)/2;
  - if feasible, update **ub**; if infeasible, update **lb**;
  - stop if a solution with **f** = **lb**+1 is found.

Binary Search

- A compromise between destructive lower and upper bounding.
  - Anytime algorithm.
  - Lower bounds.
  - Tight(ish) constraints on **f** → good propagation.
  - Large steps.

Binary Search

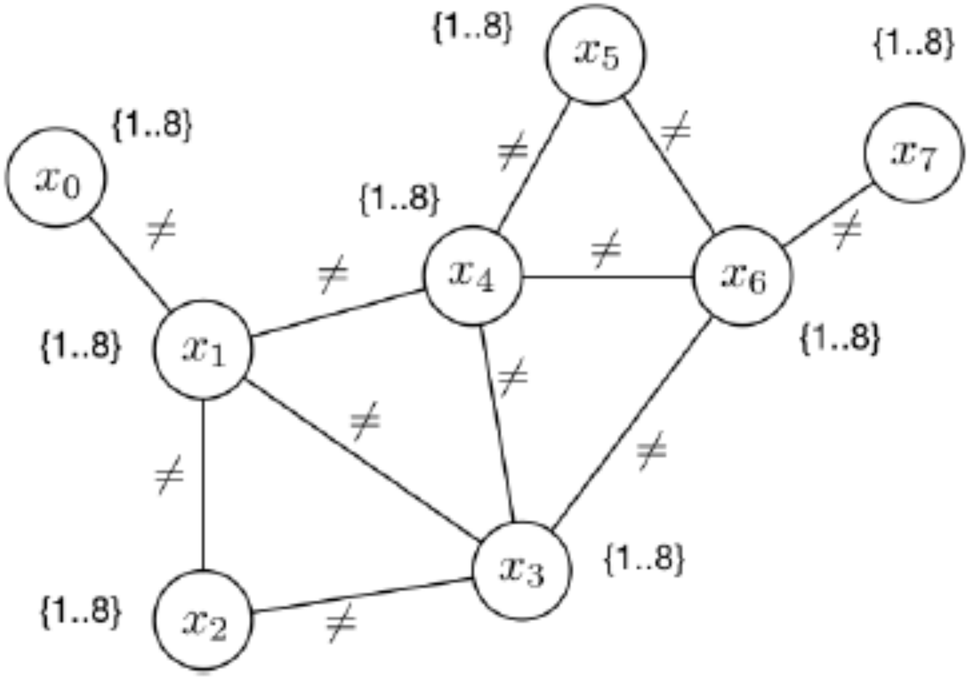
- Almost all information is discarded between each attempt.
  - → A lot of repeated work!
- Is there a more efficient method?

Branch & Bound Algorithm

- Solves a sequence of CSPs via a single search tree and incorporates bounding in the search.
- How?
  - Each time a feasible solution is found, posts a new bounding constraint which ensures that a future solution must be better than it.
  - Backtracks and looks for a new solution with the additional bounding constraint, using the same search tree.
  - Repeats until infeasible: the last solution found is optimal.

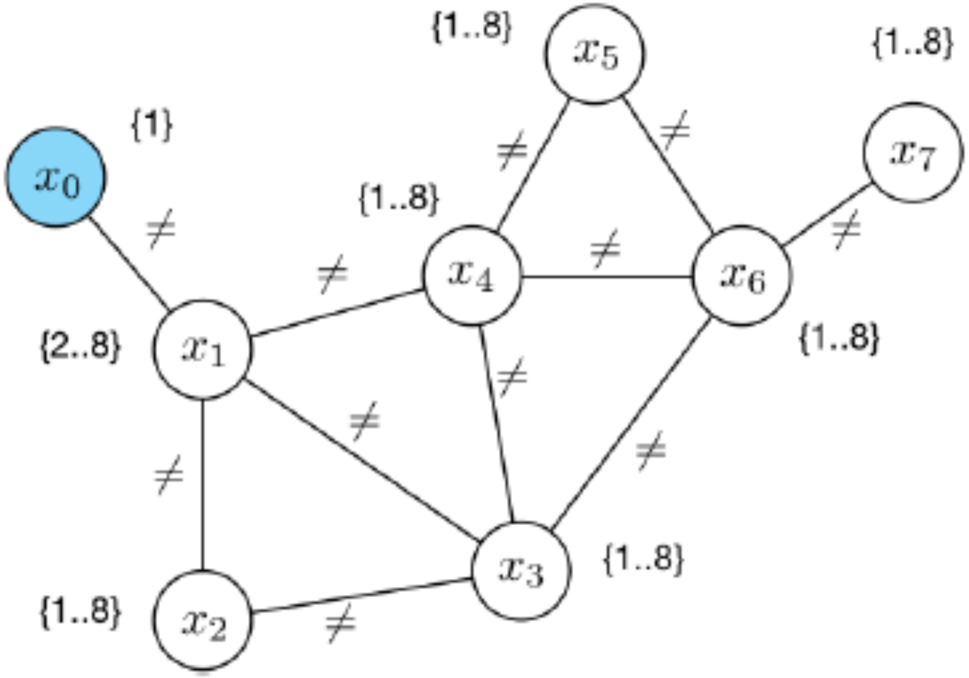
Optimal Map Colouring with B&B

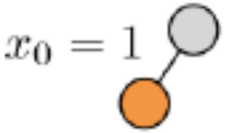
$$\max_{i=0..7}(x_i) \in \{1..8\}$$



Optimal Map Colouring with B&B

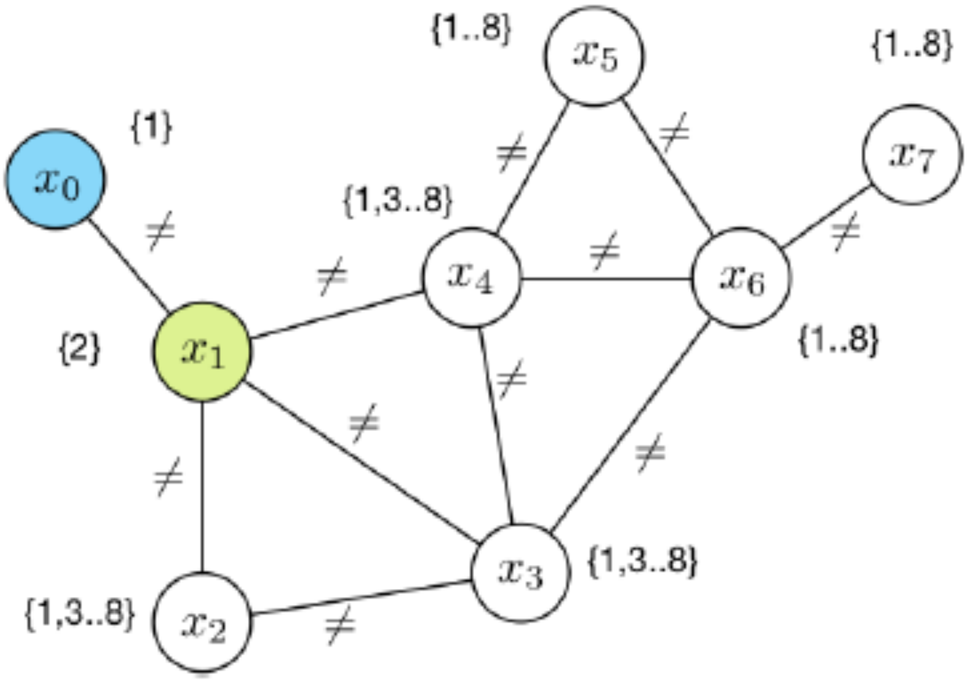
$$\max_{i=0..7}(x_i) \in \{1..8\}$$

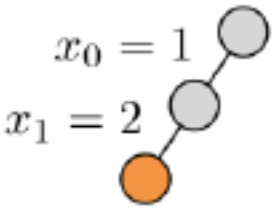




Optimal Map Colouring with B&B

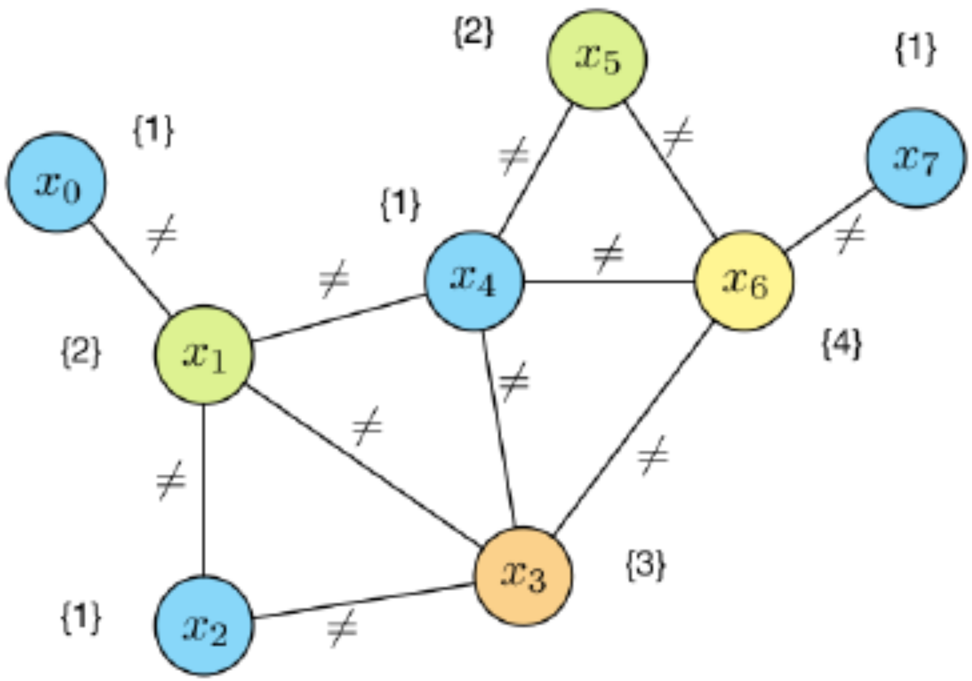
$$\max_{i=0..7}(x_i) \in \{2..8\}$$

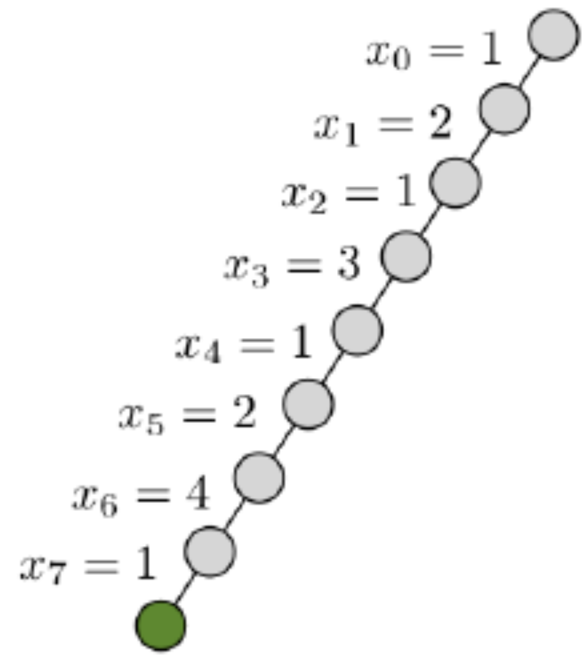




Optimal Map Colouring with B&B

$$\max_{i=0..7}(x_i) \in \{4\}$$

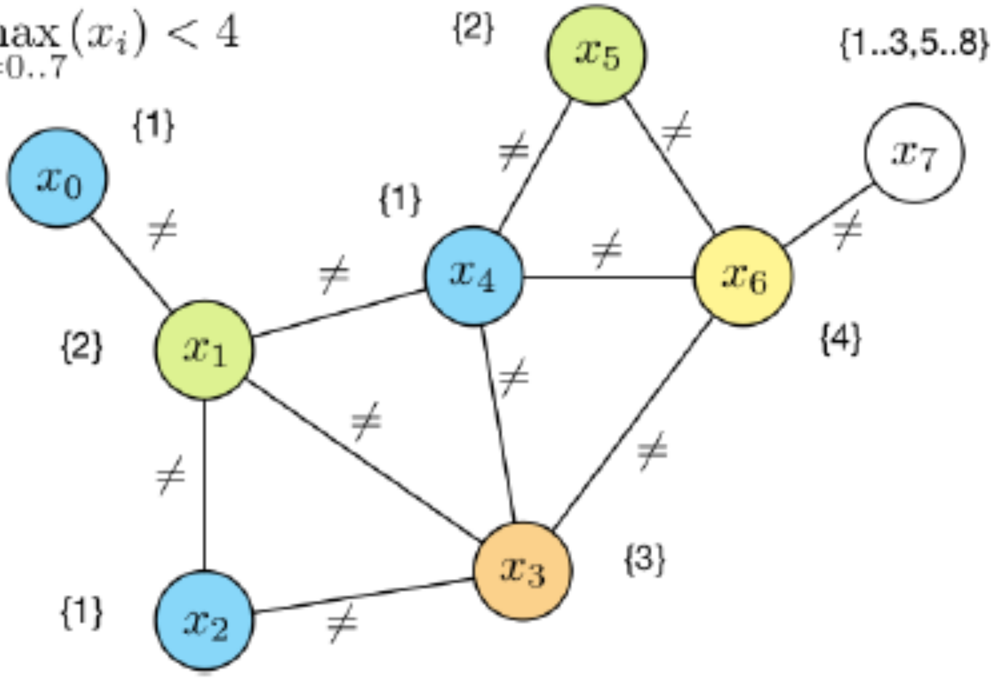


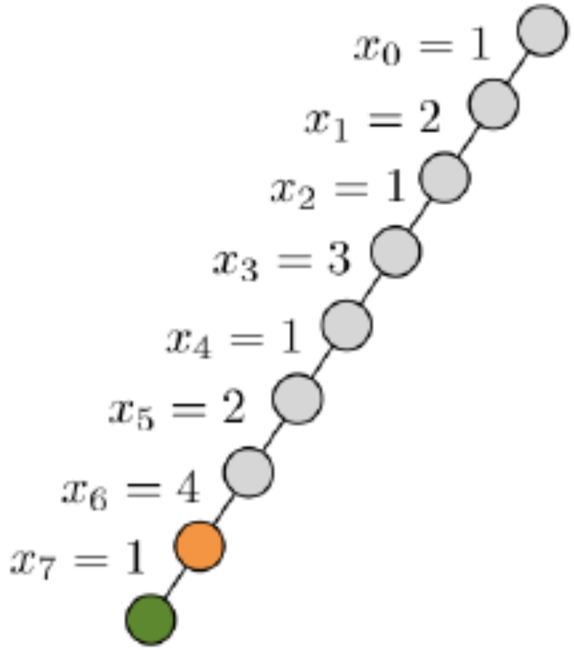


Optimal Map Colouring with B&B

$$\max_{i=0..7}(x_i) \in \{4..8\}$$

$$\max_{i=0..7}(x_i) < 4$$

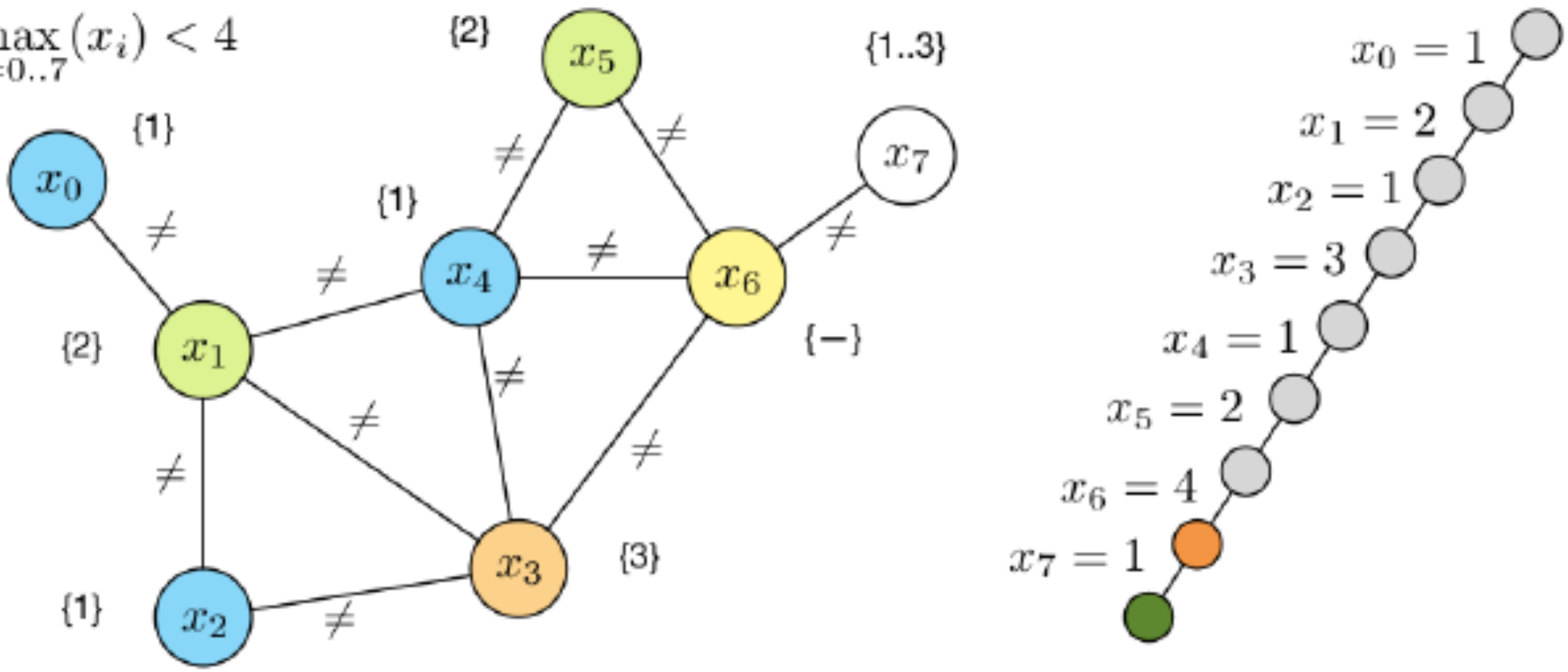




Optimal Map Colouring with B&B

$$\max_{i=0..7}(x_i) \in \{-\}$$

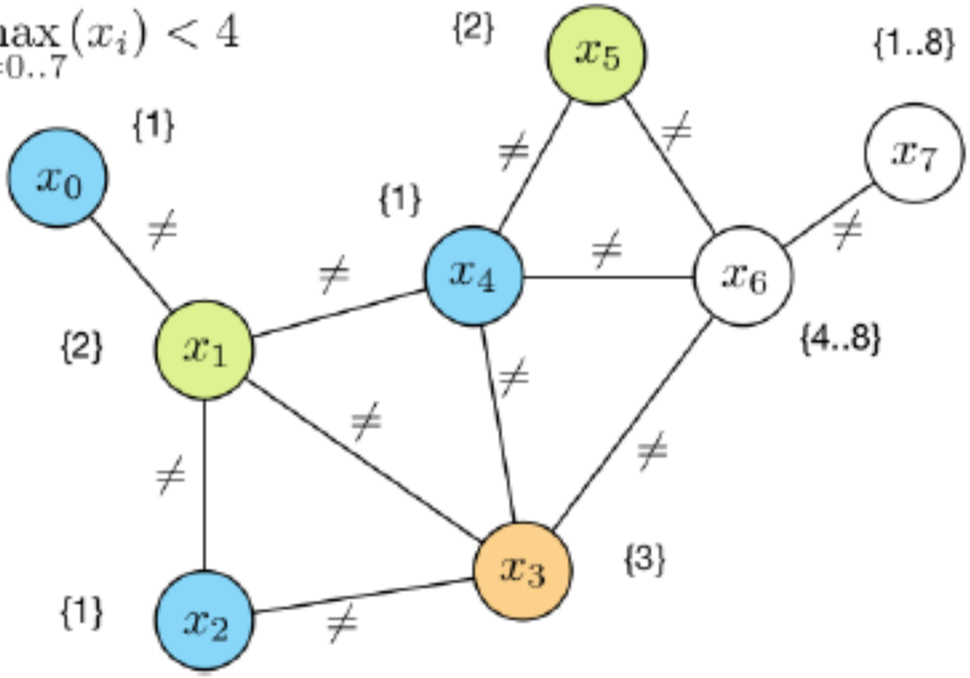
$$\max_{i=0..7}(x_i) < 4$$

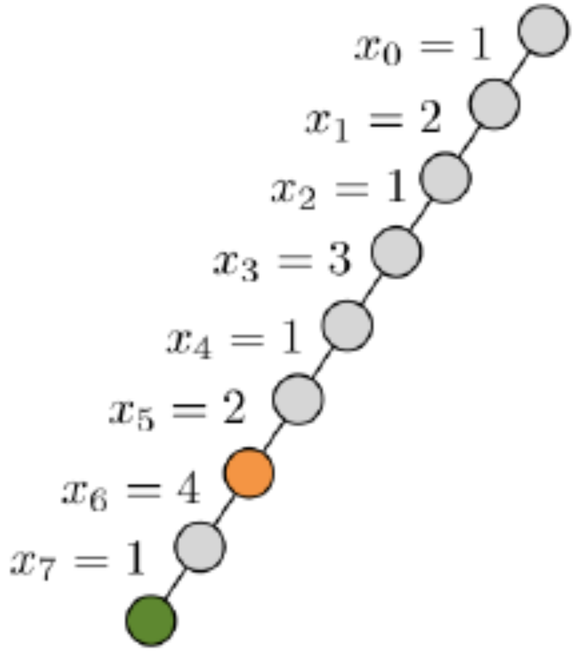


Optimal Map Colouring with B&B

$$\max_{i=0..7}(x_i) \in \{4..8\}$$

$$\max_{i=0..7}(x_i) < 4$$

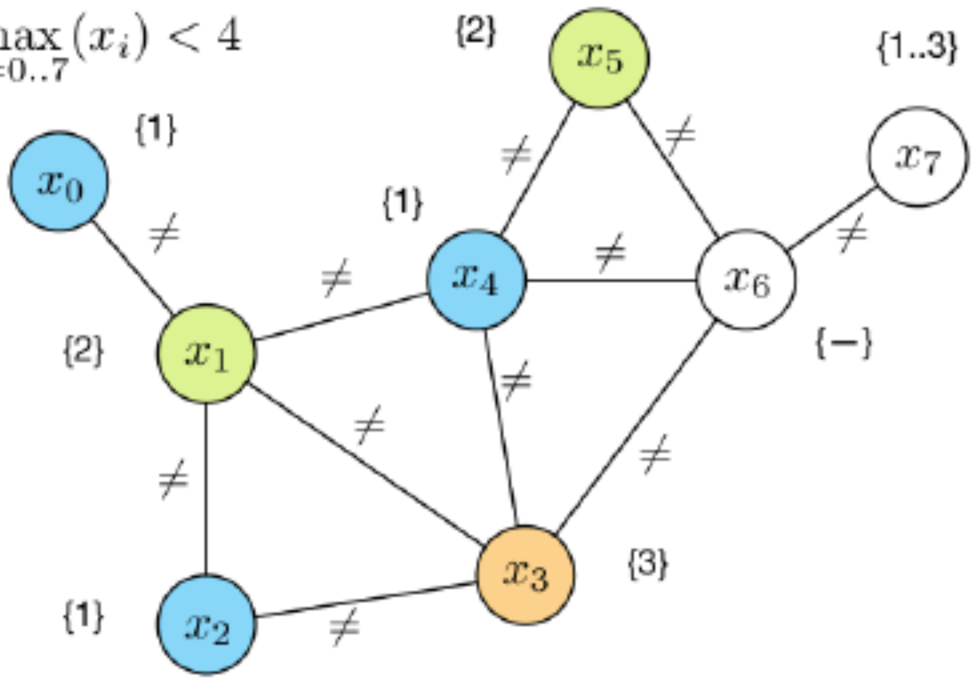


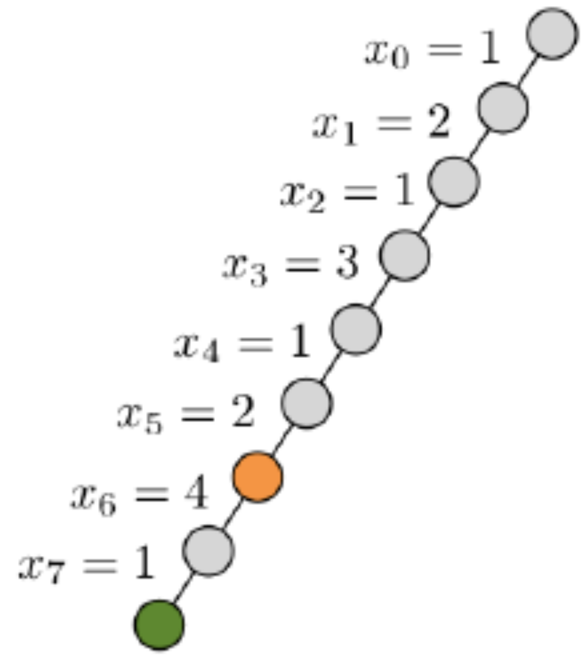


Optimal Map Colouring with B&B

$$\max_{i=0..7}(x_i) \in \{-\}$$

$$\max_{i=0..7}(x_i) < 4$$

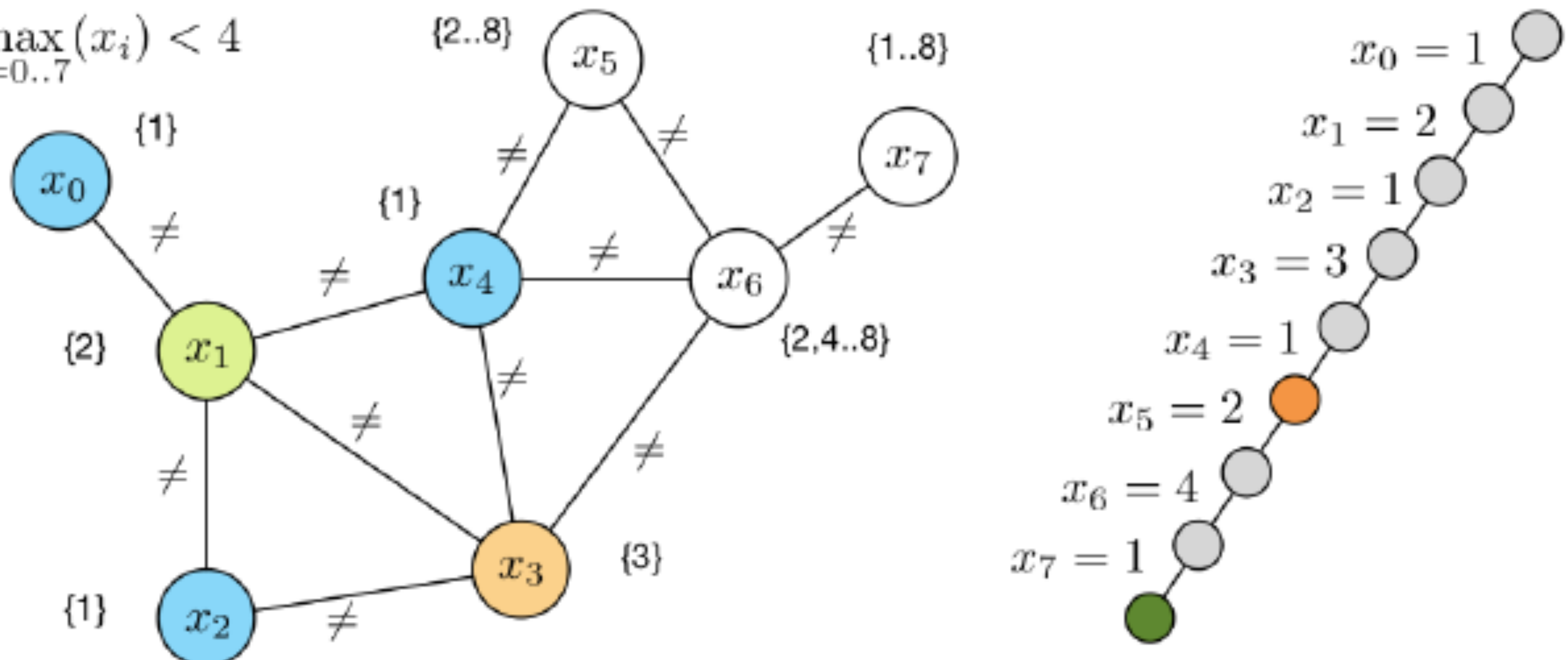




Optimal Map Colouring with B&B

$$\max_{i=0..7}(x_i) \in \{3..8\}$$

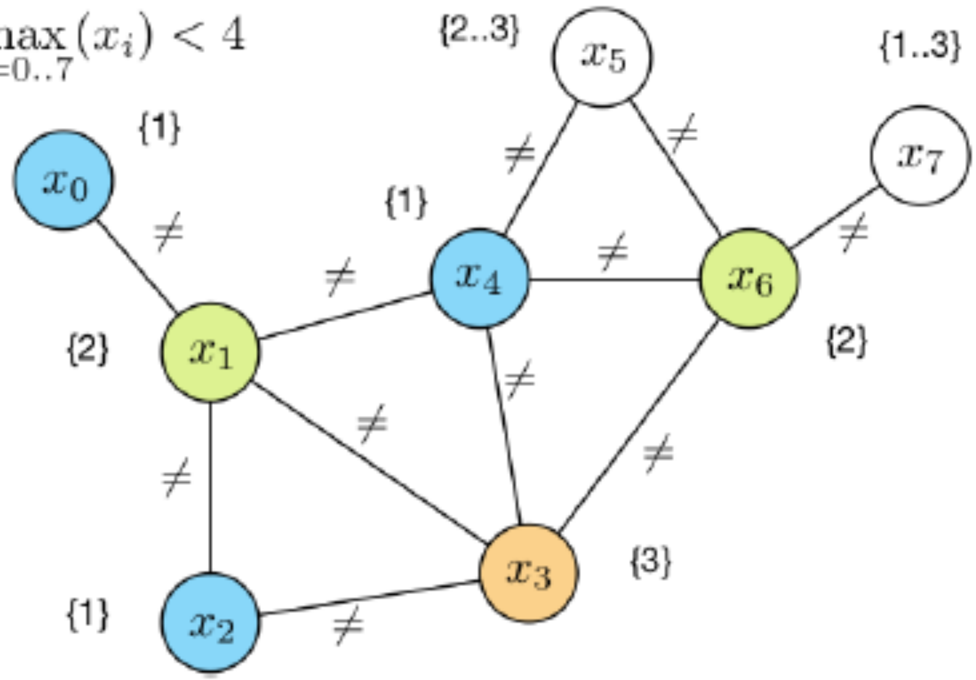
$$\max_{i=0..7}(x_i) < 4$$

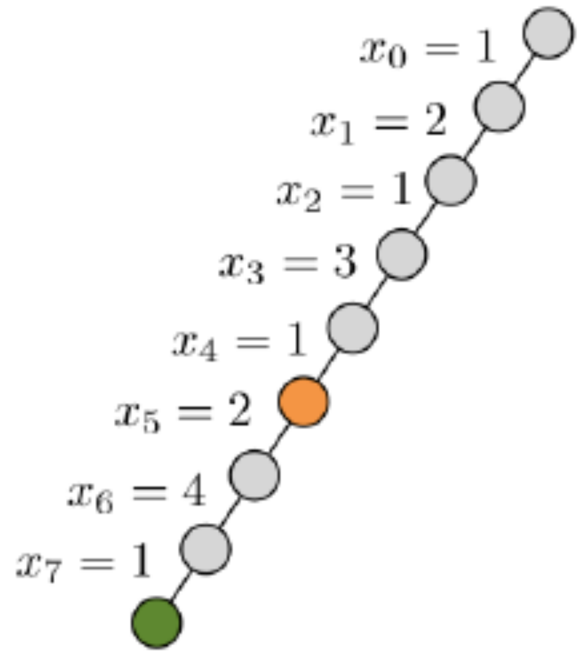


Optimal Map Colouring with B&B

$$\max_{i=0..7}(x_i) \in \{3\}$$

$$\max_{i=0..7}(x_i) < 4$$

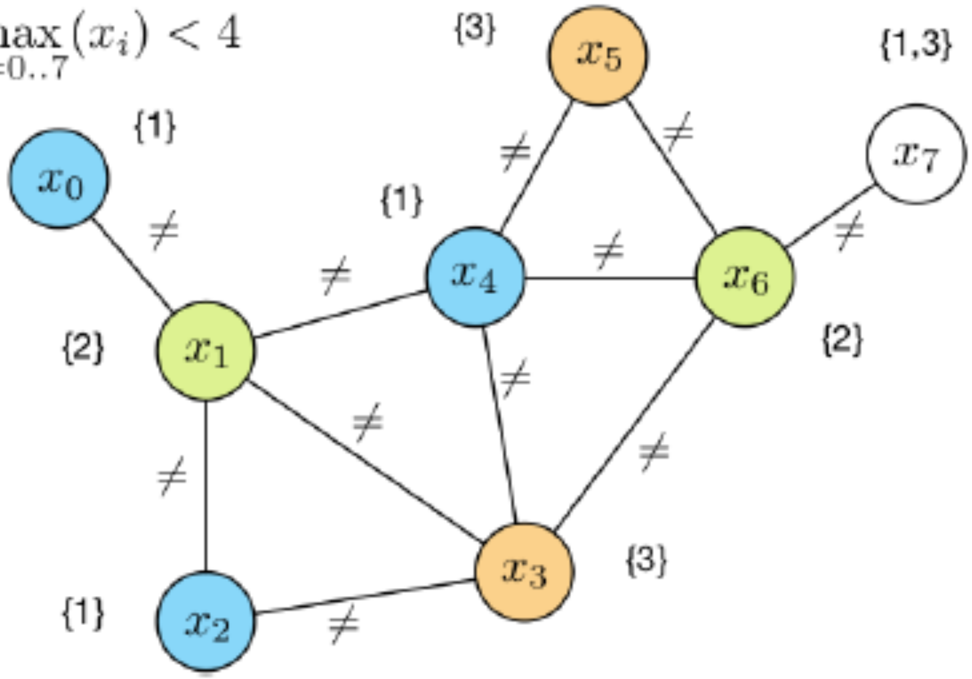


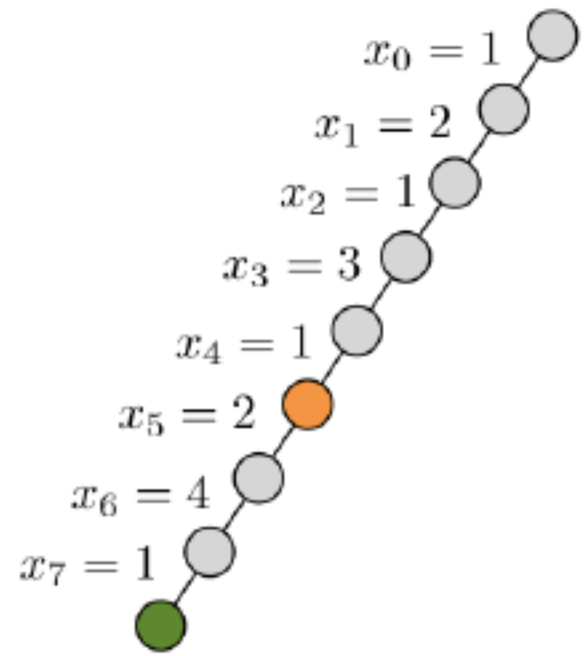


Optimal Map Colouring with B&B

$$\max_{i=0..7}(x_i) \in \{3\}$$

$$\max_{i=0..7}(x_i) < 4$$

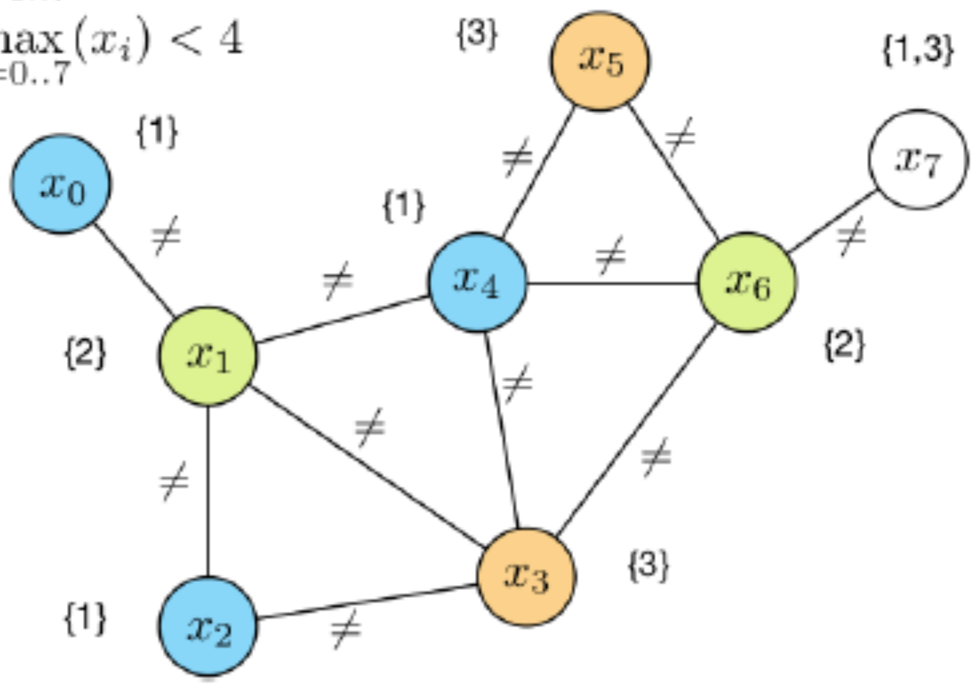


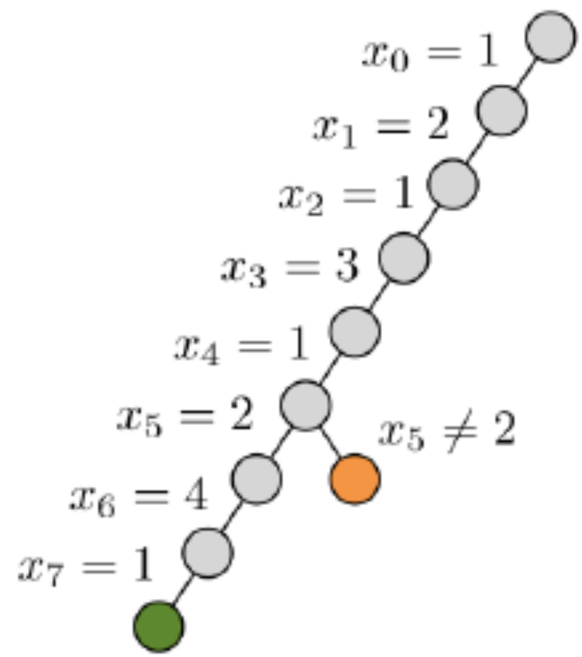


Optimal Map Colouring with B&B

$$\max_{i=0..7}(x_i) \in \{3\}$$

$$\max_{i=0..7}(x_i) < 4$$

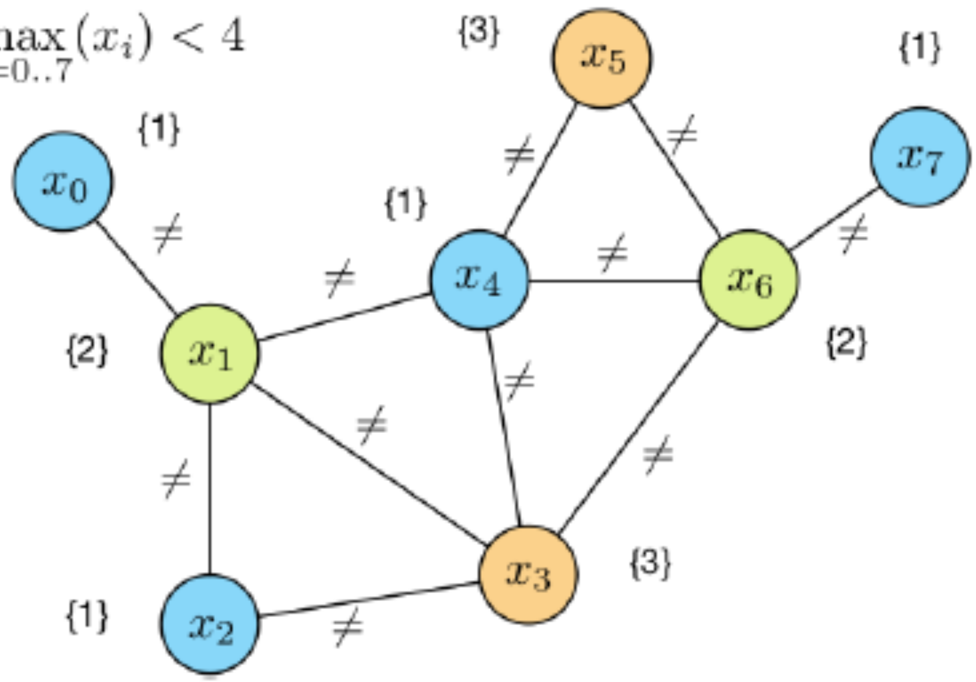


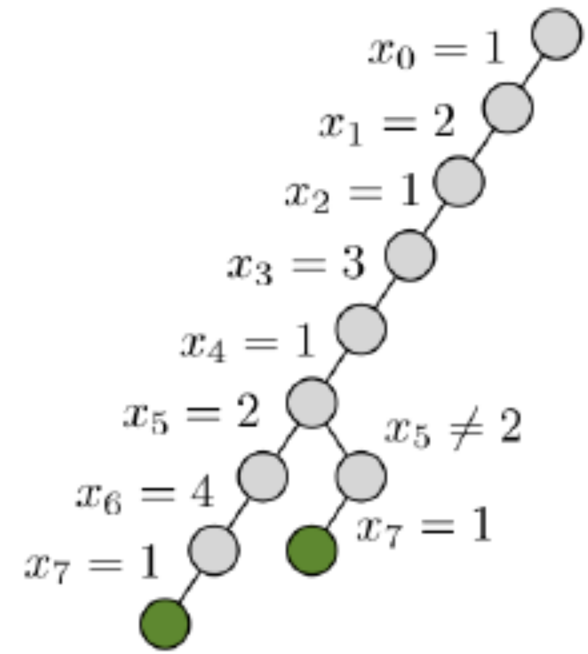


Optimal Map Colouring with B&B

$$\max_{i=0..7}(x_i) \in \{3\}$$

$$\max_{i=0..7}(x_i) < 4$$





Optimal Map Colouring with B&B

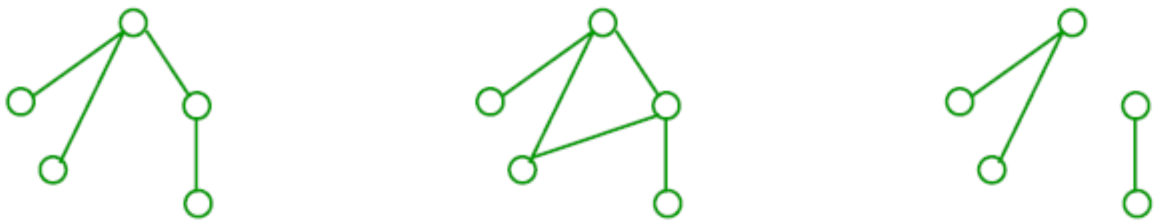
- The solution is optimal, but we don't know it yet!
- We need to finish exploring the search tree.
  - Often called optimality proof.

Conclusions on Optimization

- Main idea: solve a sequence of CSPs to solve a COP.
- 2 main approaches:
  - Search over D(**f**)
    - Destructive bounding and binary search.
    - Different trade-offs.
  - Branch and bound
    - PRO: No waste of information (and a bit of more propagation).
    - PRO: Anytime algorithm.
    - CON: (Almost) no lower bounds.

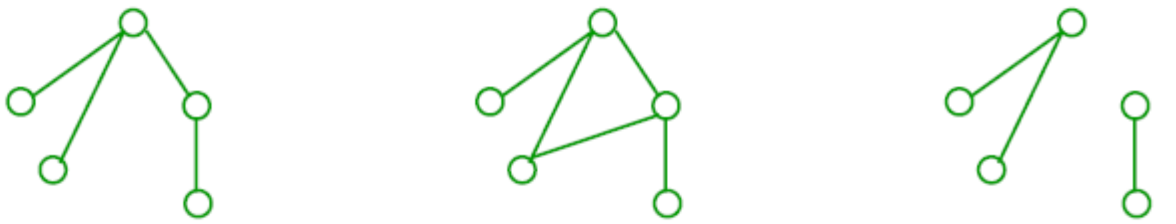
Tree: Formal Definition

- *Def:* A tree is a set of **nodes** (vertices) connected by **edges** (links) s.t. there is **exactly one** way to get from any node to any other node
- Which of the following are trees?



Tree: Formal Definition

- *Def:* A tree is a set of **nodes** (vertices) connected by **edges** (links) s.t. there is **exactly one** way to get from any node to any other node
- Which of the following are trees?



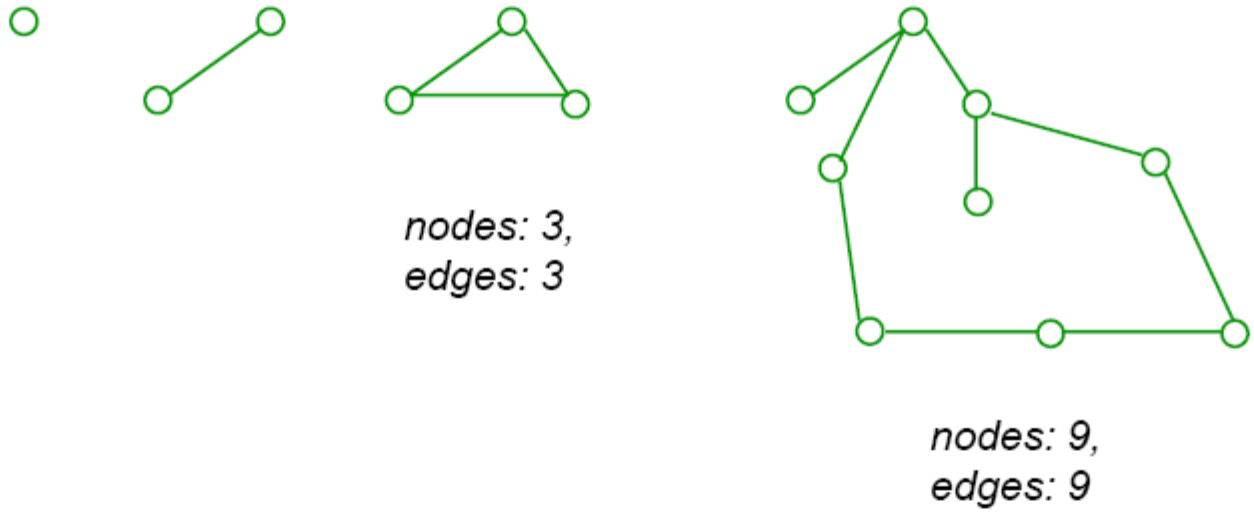
YES!

No, it is a graph

No, it is a forest (i.e. multiple trees)

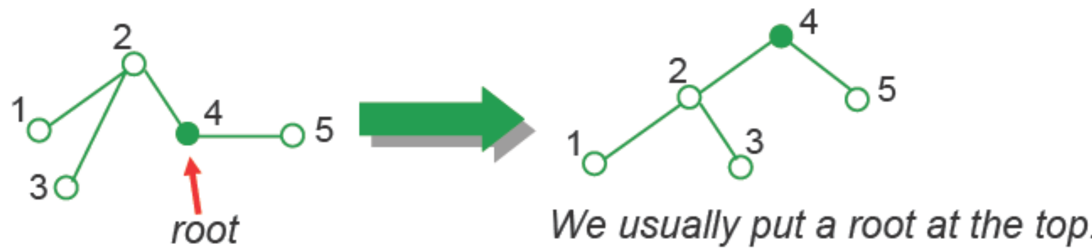
Fundamental Property

- Every non-empty tree with **n** nodes has exactly **n-1** edges
- This property can also be used to demonstrate that a given data structure is NOT a tree

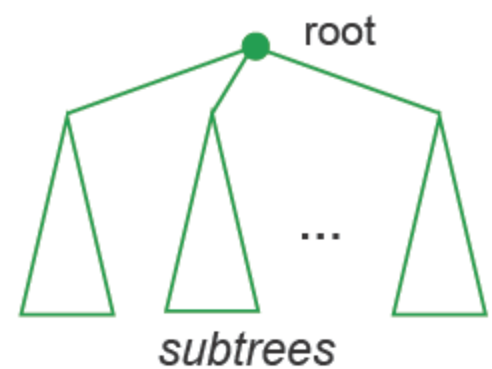


Rooted Tree

- A tree is a **rooted tree** if one of its nodes is distinguished as **root**

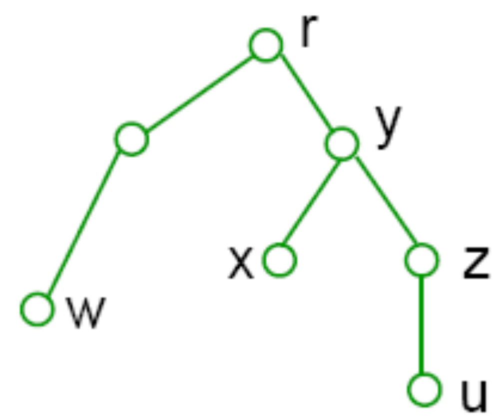


- This definition can be used in a recursive way
  - A rooted tree consists of a **root node** and a finite set of sub-trees, which are themselves rooted trees
  - Base case when the set of sub-trees is empty



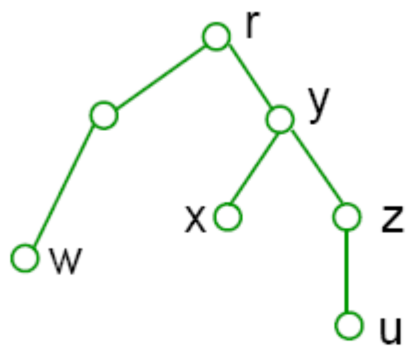
Some Terminology

- r is **root**
- y is a **parent** of x and z; r is **a parent** of y
- r, y and x are **ancestors** of x
- r, y are **proper ancestors** of x
- x, z are **children** of y
- x, y, z and u are **descendants** of r
- x and z are **siblings**
- all ancestors of u form a **path** from u to the root (u ➔ z ➔ y ➔r)
- w, x and u are **leaf nodes** (others are called **internal nodes**)
- The **depth** of a node is the length of the path to the root
  - The depth of w, x and z is 2, of u is 3
- The **height** of a node is the length of the longest path to a leaf
  - The height of y is 2
  - What is the height of the tree?



Sub-tree & Ordered Tree

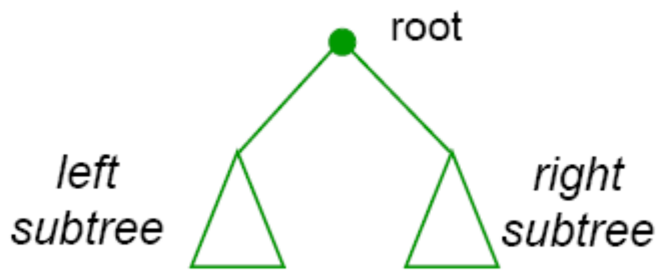
- A **sub-tree** is a node ***i*** plus all its descendants
  - ***i*** is the root of the sub-tree
    - In our example: **y** is the root of a sub-tree



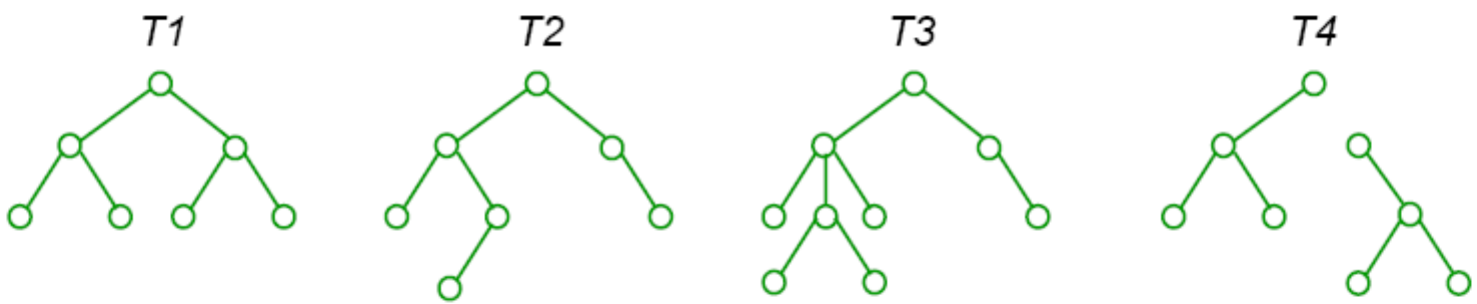
- An **ordered tree** is a rooted tree in which the children of each node are ordered
  - Order is important!

Binary Tree

- *Def:* A **binary tree** is an ordered tree which is either empty or consists of a root node and two sub-trees (left and right) which are themselves binary trees



- Which of the following are binary trees?



- Are these binary trees the same?
  - Binary trees are ordered trees!