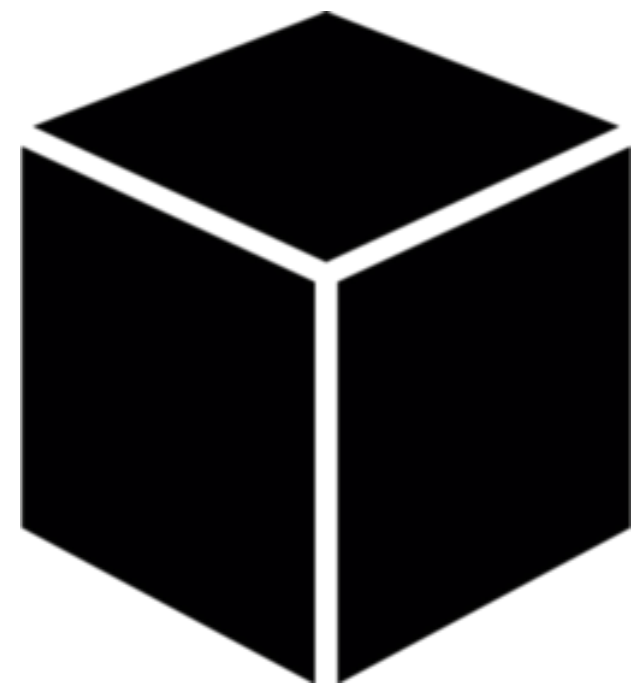


Constraint Programming



Search

- ▶ Motivation
- ▶ **Heuristics**
 - Value Selection Heuristics
 - Variable Selection Heuristics
 - First Fail
 - Degree
 - Impact
 - Activity
 - Most Recent Conflicts
 - Proving that there is no solution: Failure Directed Search
- ▶ **Strategies (a.k.a. Metaheuristics)**
 - Discrepancy Search
 - ² – LNS and Restarts

Motivation

Black-Box Search: The Vision

- Constraint Program = Model + Search

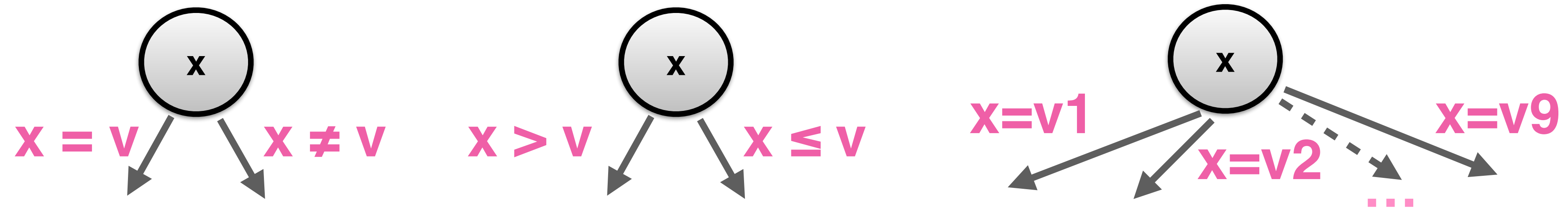
The vision is that we should not have to worry too much about the search. A CP solver should be smart enough to learn or select a good search for the model.

- Since ~2004, many research efforts have been devoted to making this vision true.

Refresher: Branching = 2-Step Choice

- Variable selection.
- Value selection & domain partition selection.

Branching does not need to be binary:

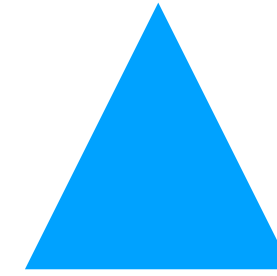


The branching decisions can have a strong impact on the size of the search tree.

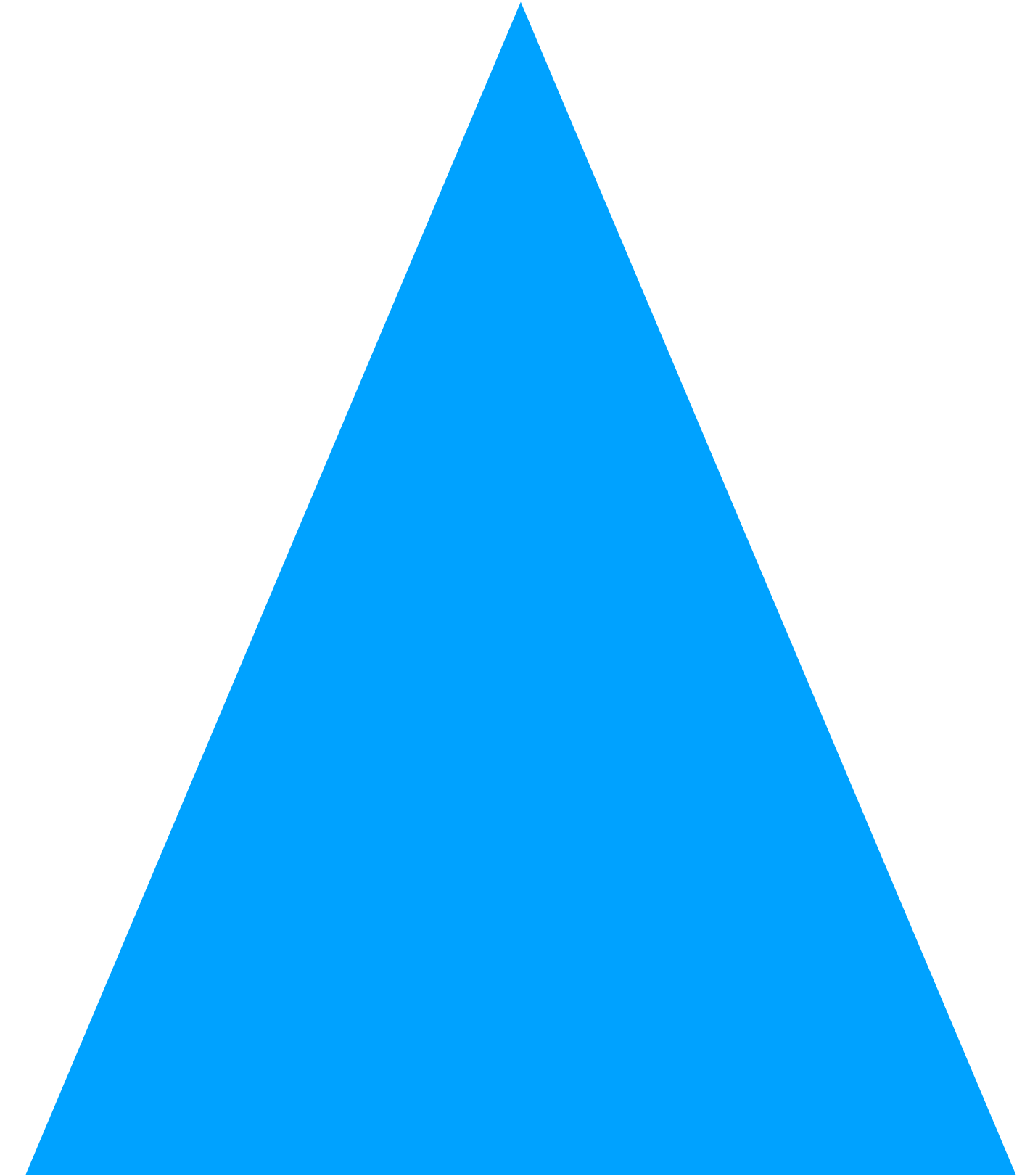
Searching...

- ▶ What is the big deal?
- ▶ It is all about the tree!
 - Size or shape
 - Depth of solutions
 - Location of solutions

Small Tree

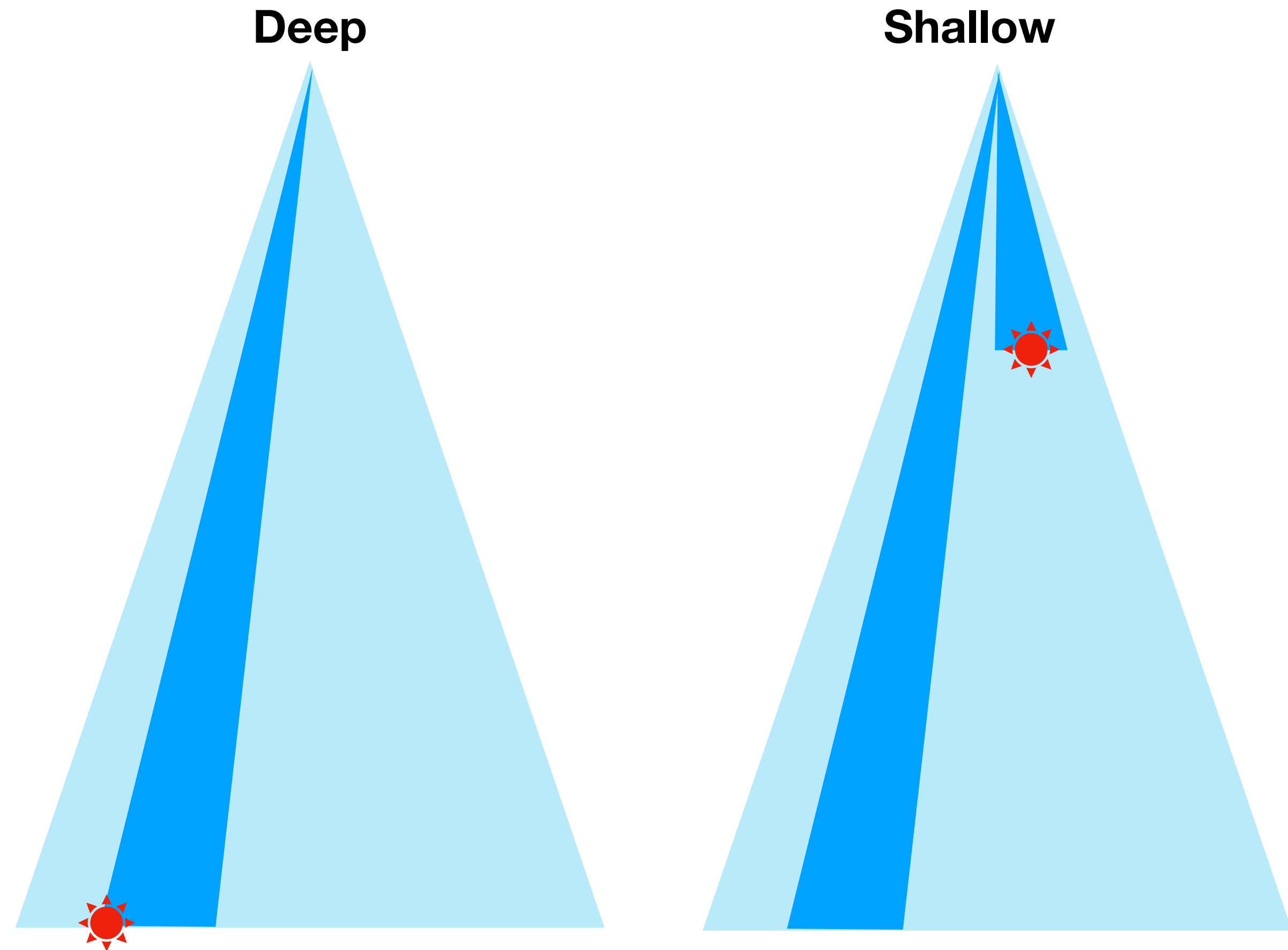


Large Tree



Searching...

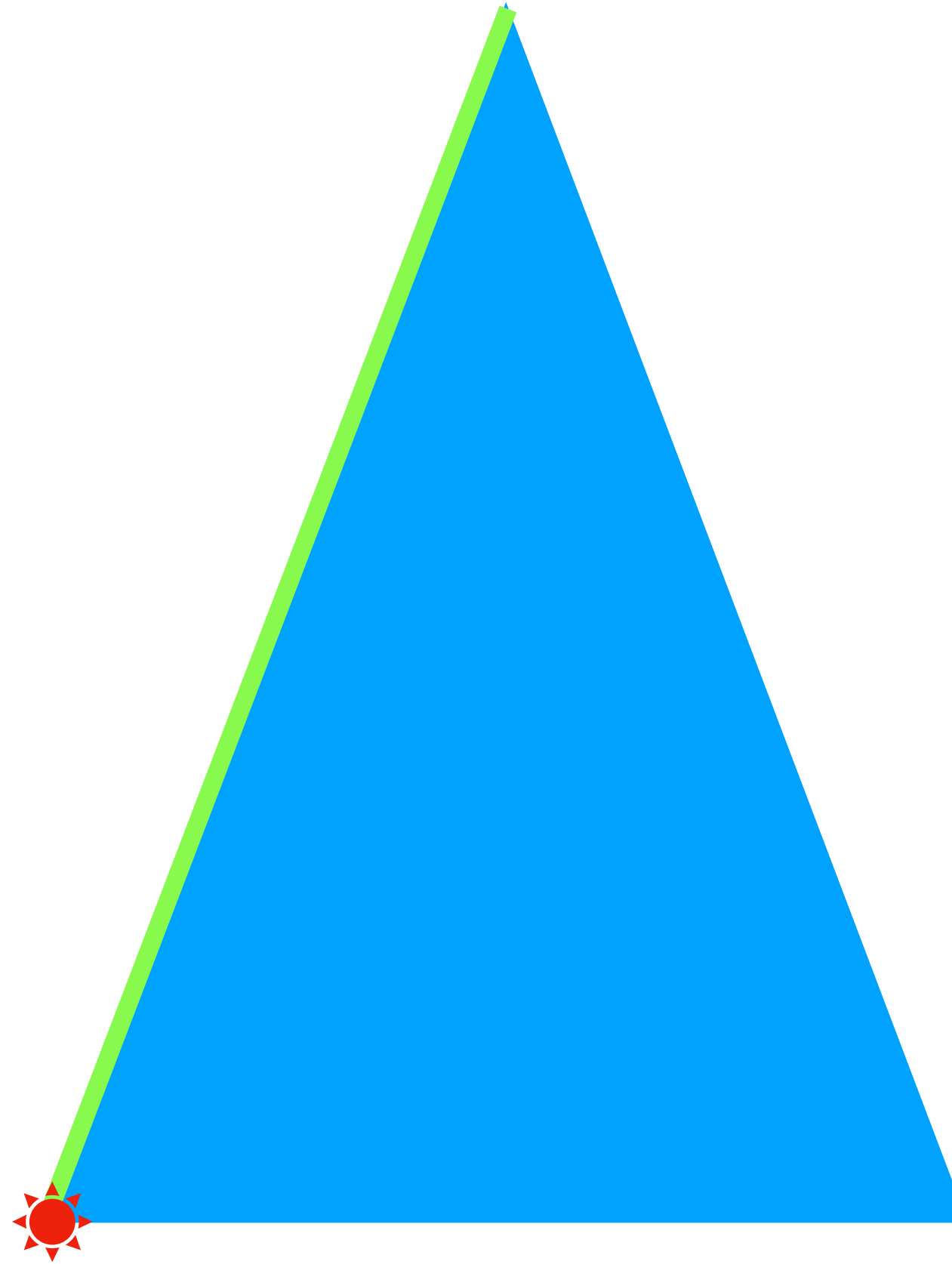
- ▶ What is the big deal?
- ▶ It is all about the tree!
 - Size or shape
 - Depth of solutions
 - Location of solutions



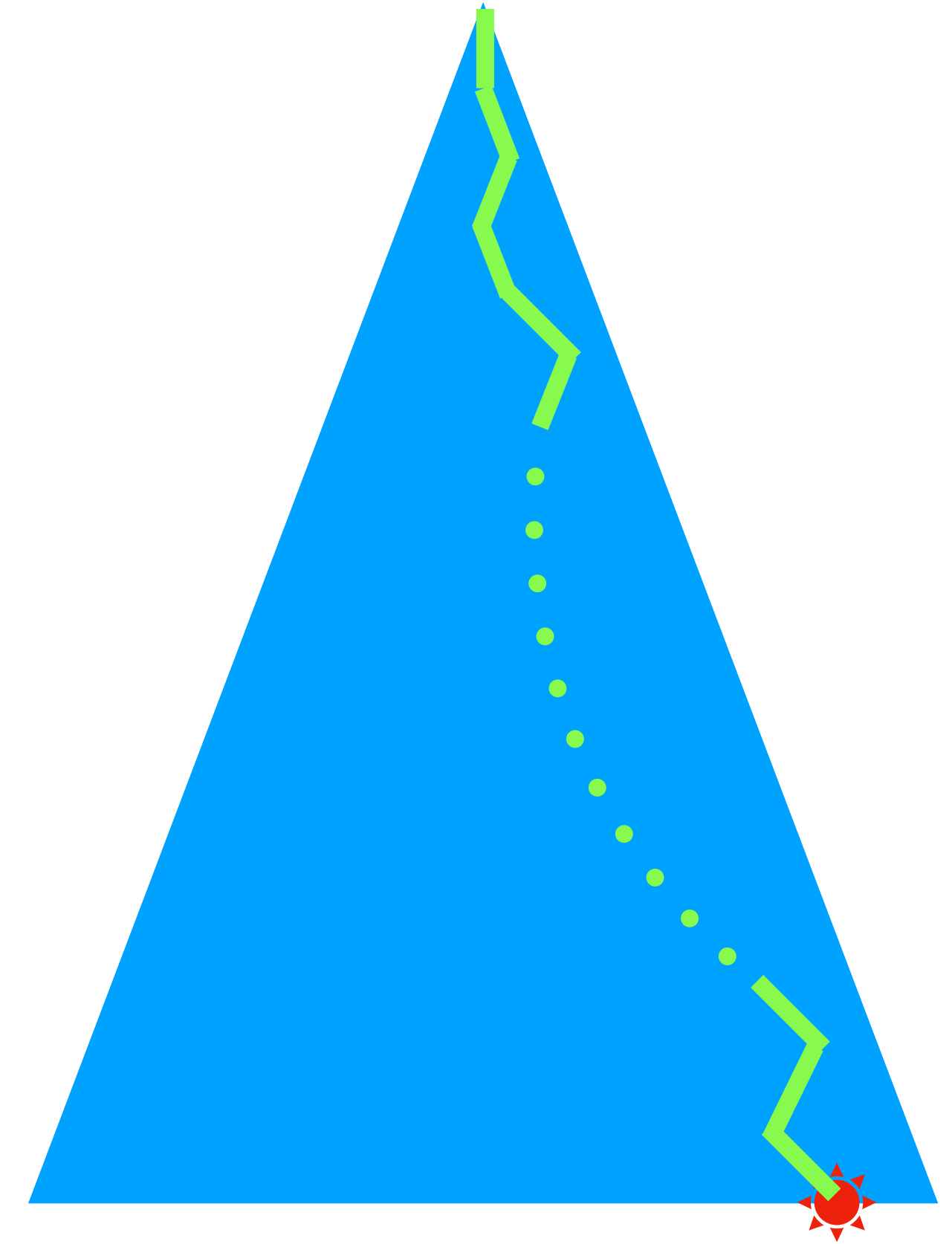
Searching...

- ▶ What is the big deal?
- ▶ It is all about the tree!
 - Size or shape
 - Depth of solutions
 - Location of solutions

Easy to reach...



Hard to reach...



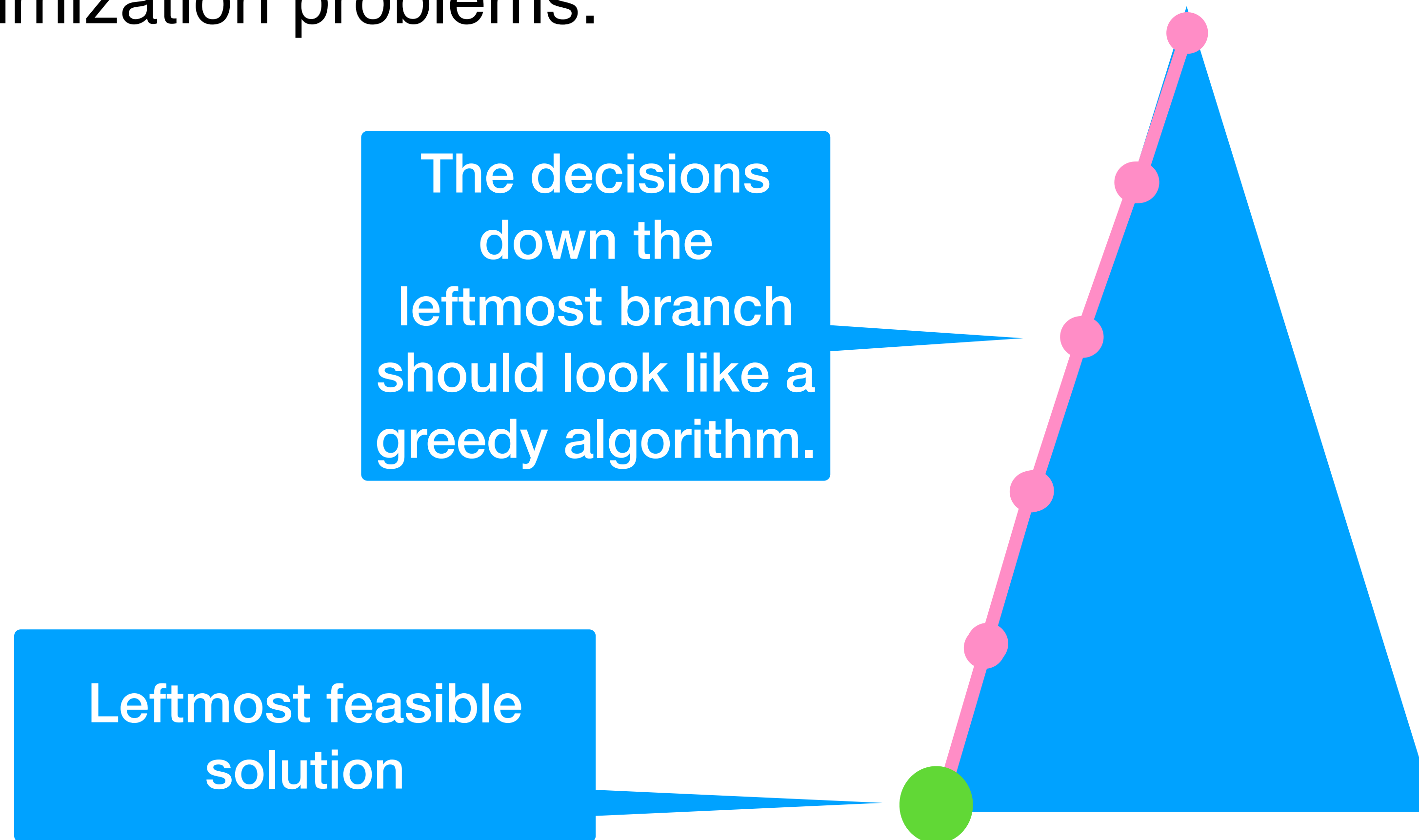
- ▶ Variable selection has an impact
 - on the size of the tree (because of propagation)
 - on the quality of solutions to a COP when timing out
 - on the depth where (good-quality) solutions are
- ▶ Value selection has an impact
 - on the size of the tree
 - on the quality of solutions to a COP when timing out
 - on the location of solutions (heuristic recommendations are “far left”)
- ▶ Strategies have an impact
 - on how quickly you hit a solution that is good or otherwise hard to reach

Value Selection

Here: for optimization problems

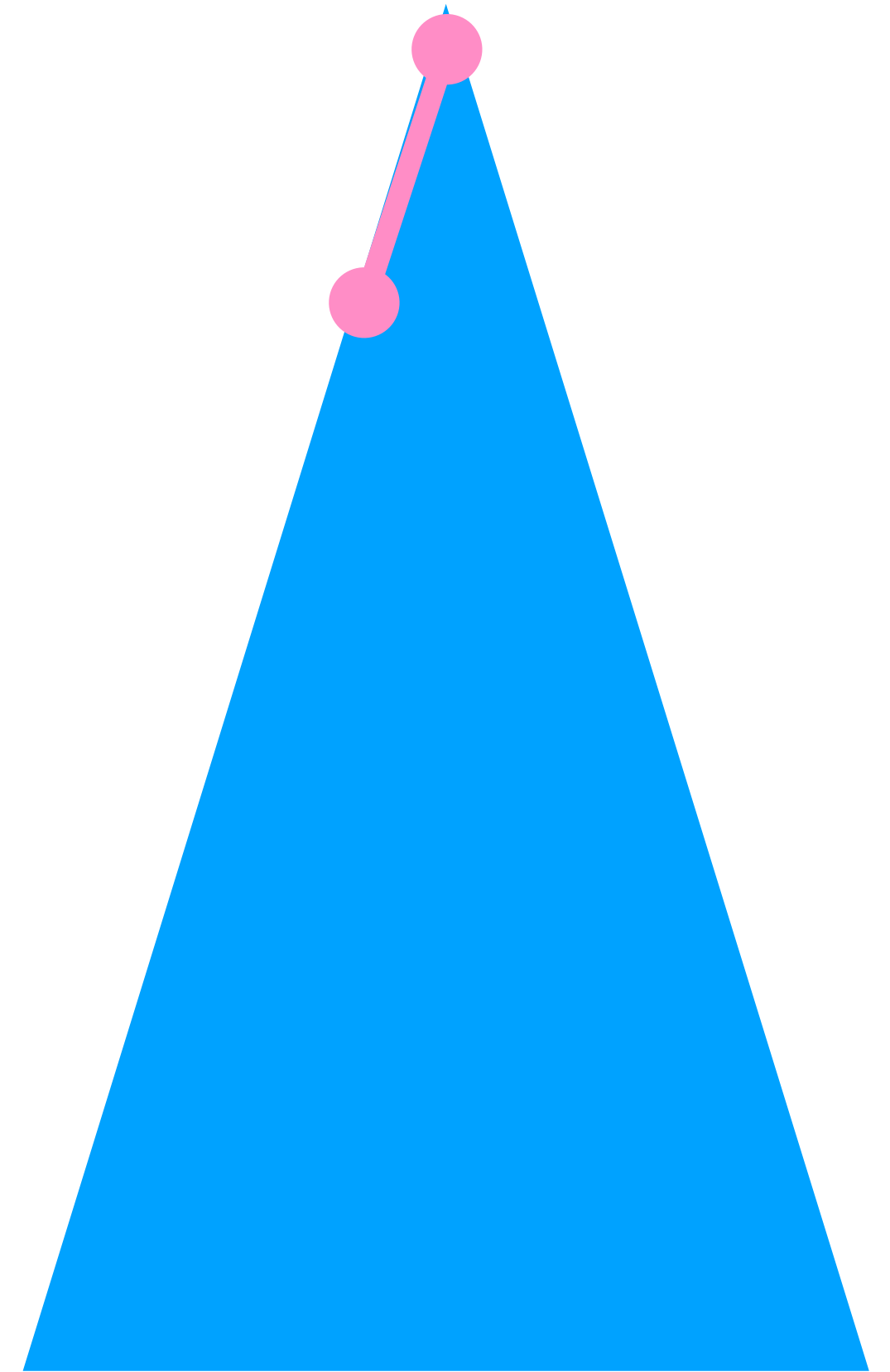
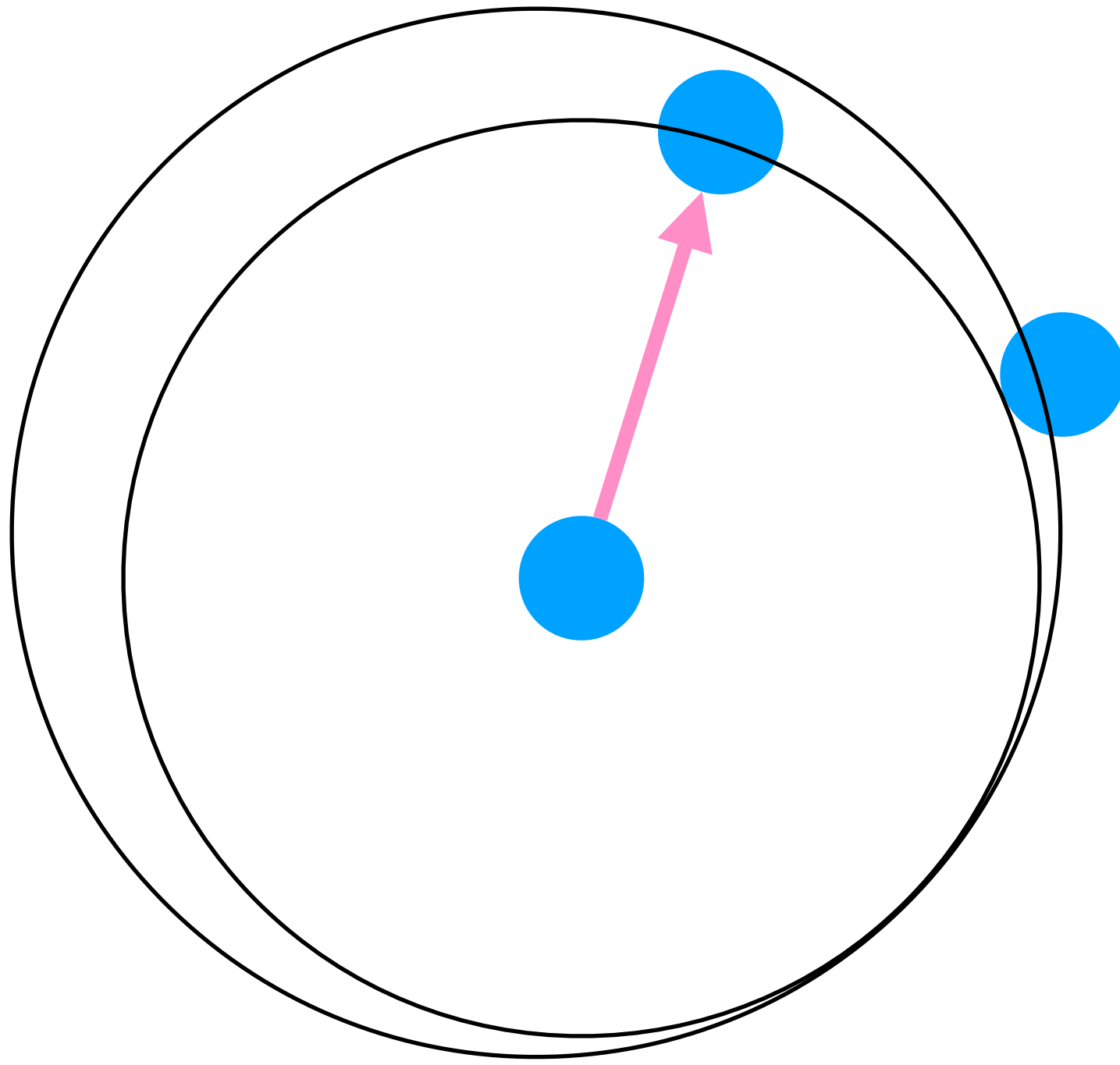
Value Selection for Optimization Problems

- ▶ The leftmost feasible solution is very important for pruning during branch-and-bound search.
- ▶ Solution quality is mostly impacted by the value selection heuristic for optimization problems.

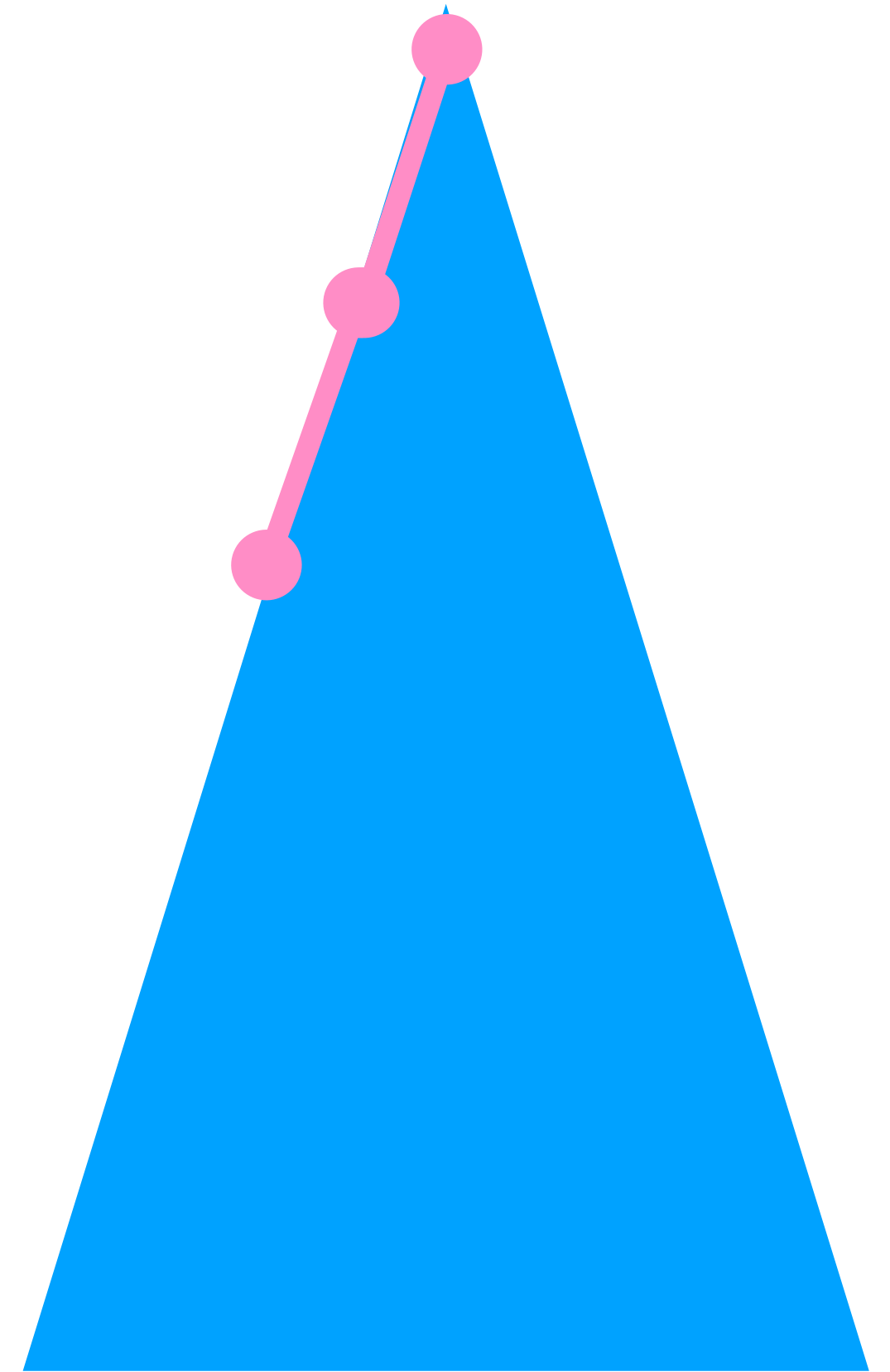
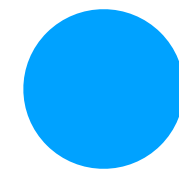
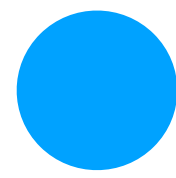
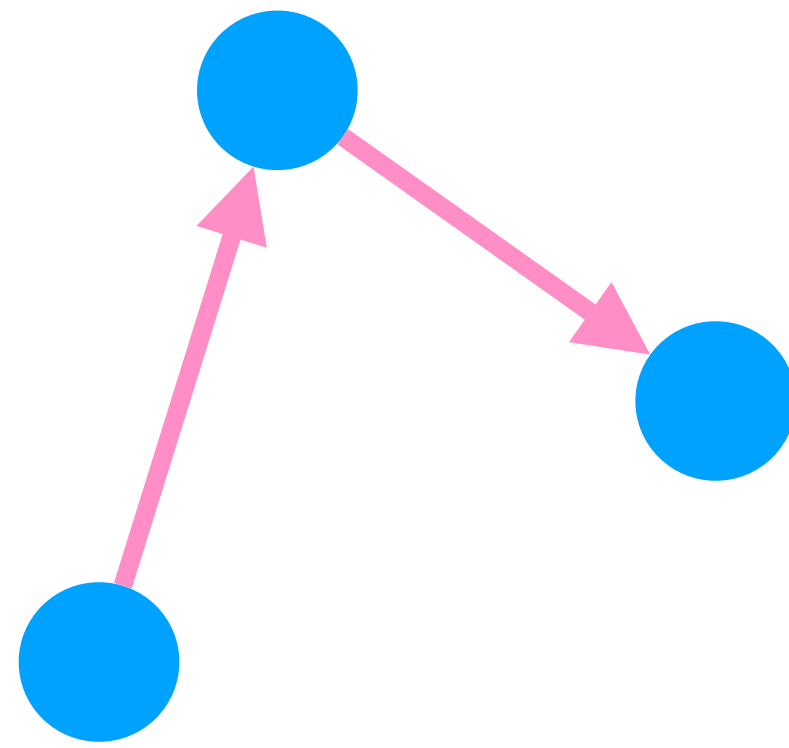


Example: Traveling Salesperson Problem

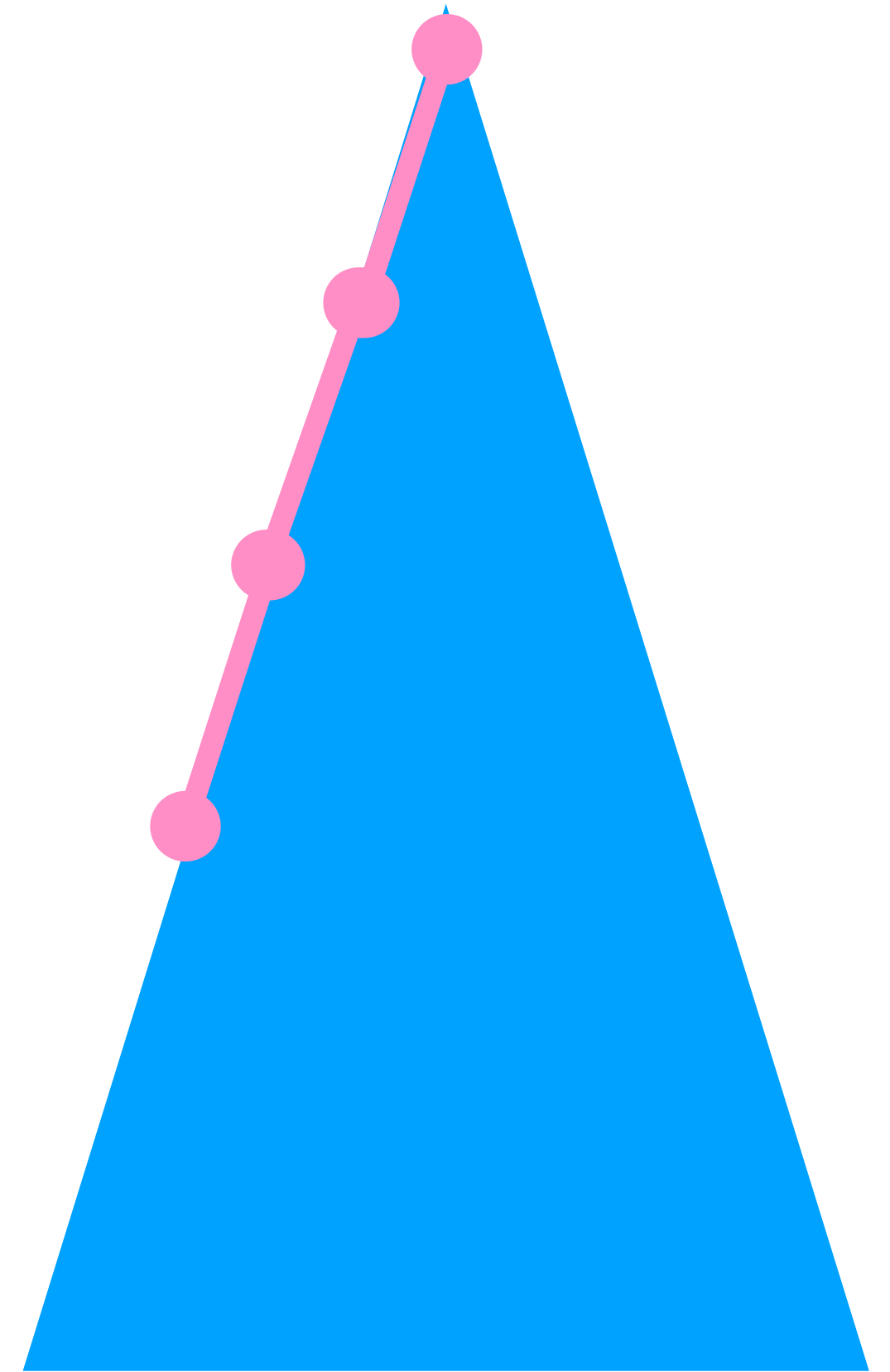
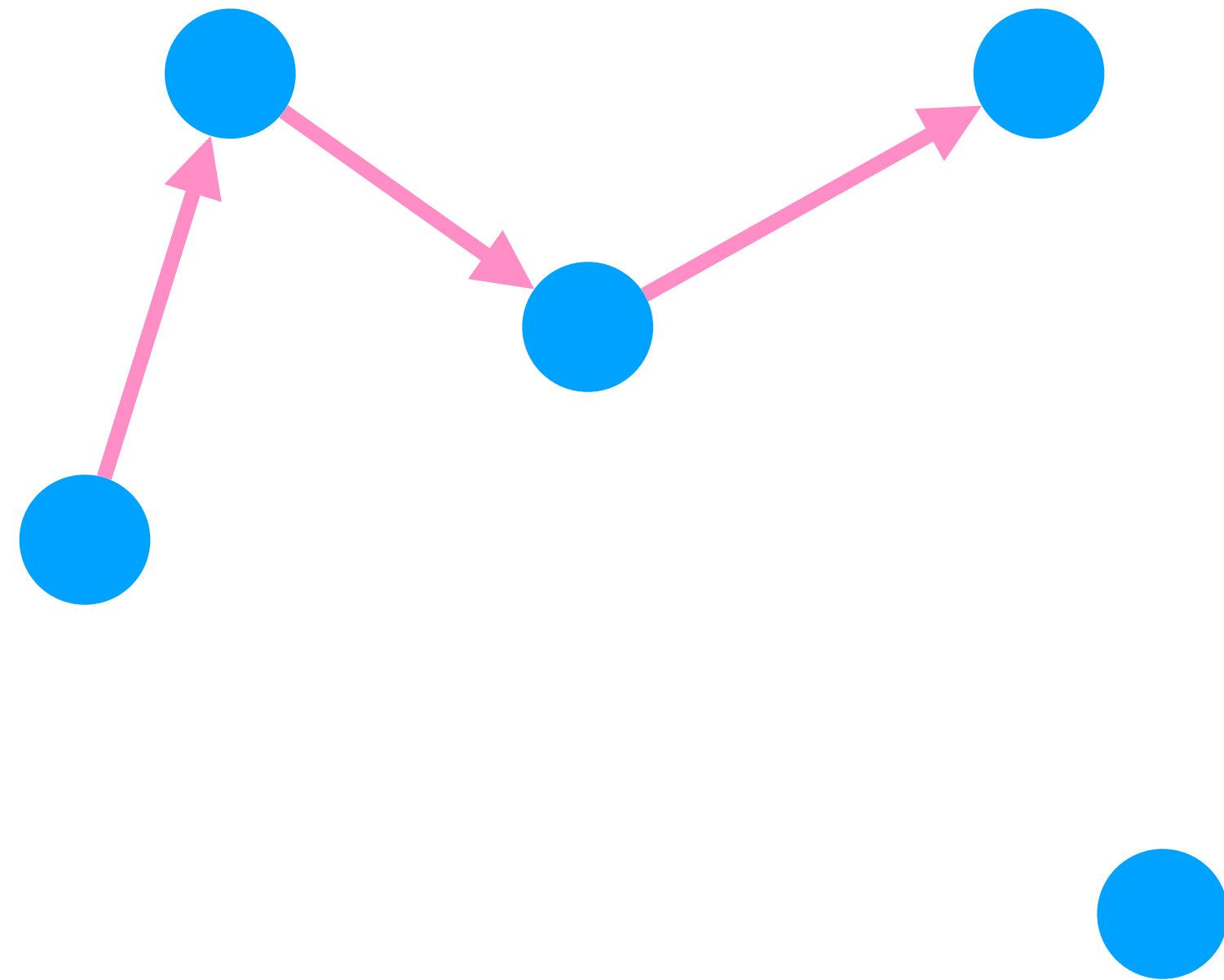
Greedily, we would adopt a nearest-neighbor value selection heuristic:



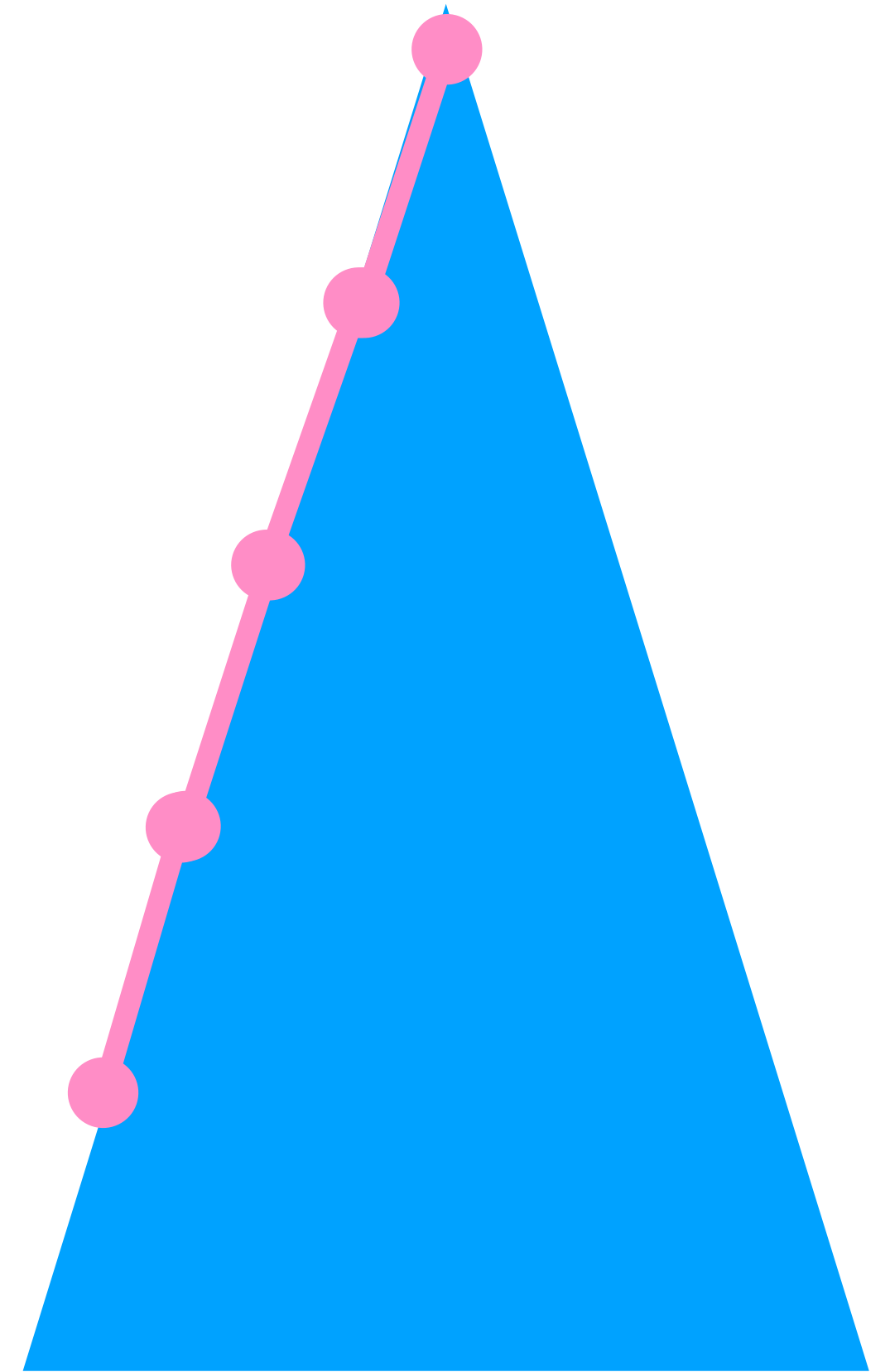
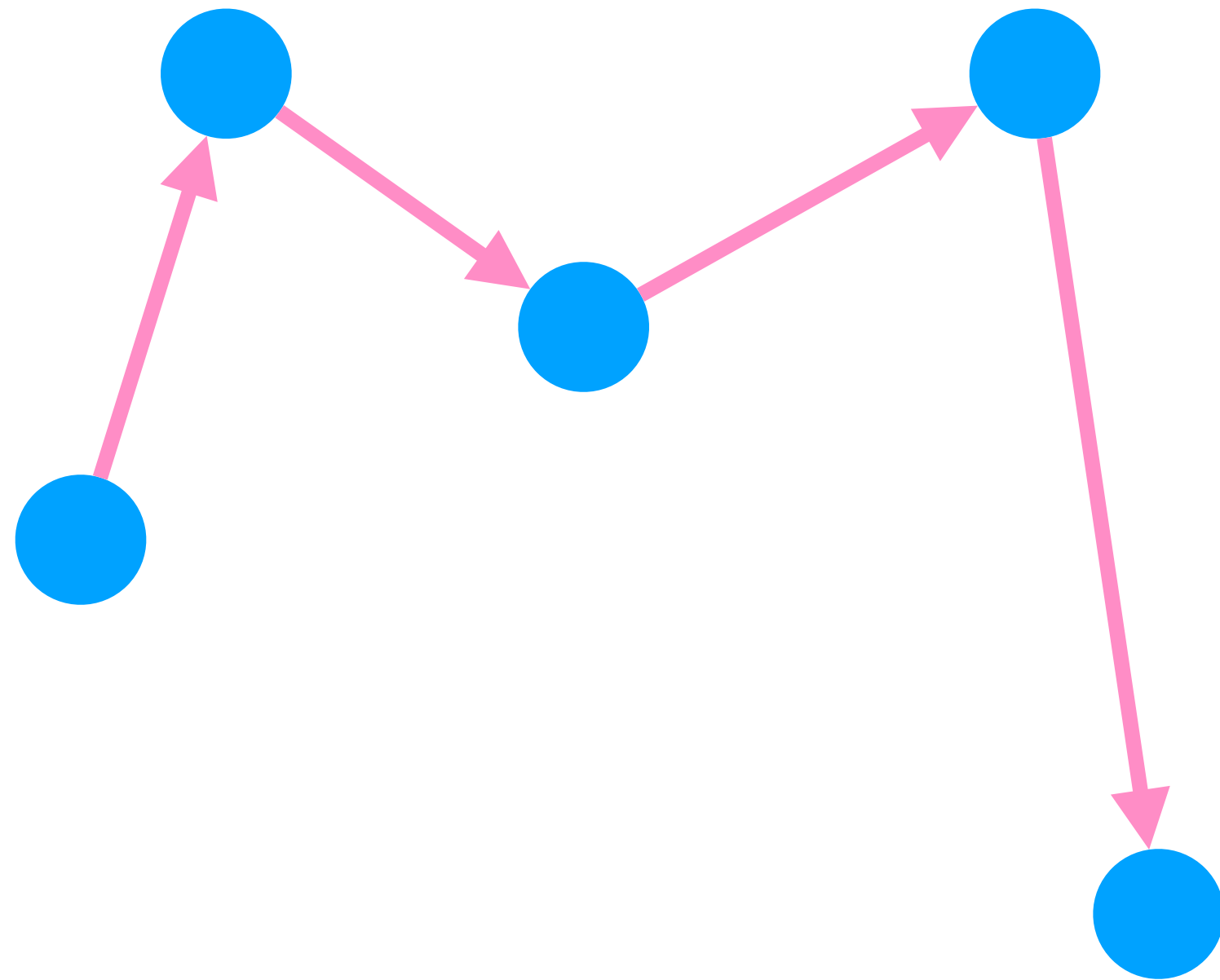
Example: Traveling Salesperson Problem



Example: Traveling Salesperson Problem

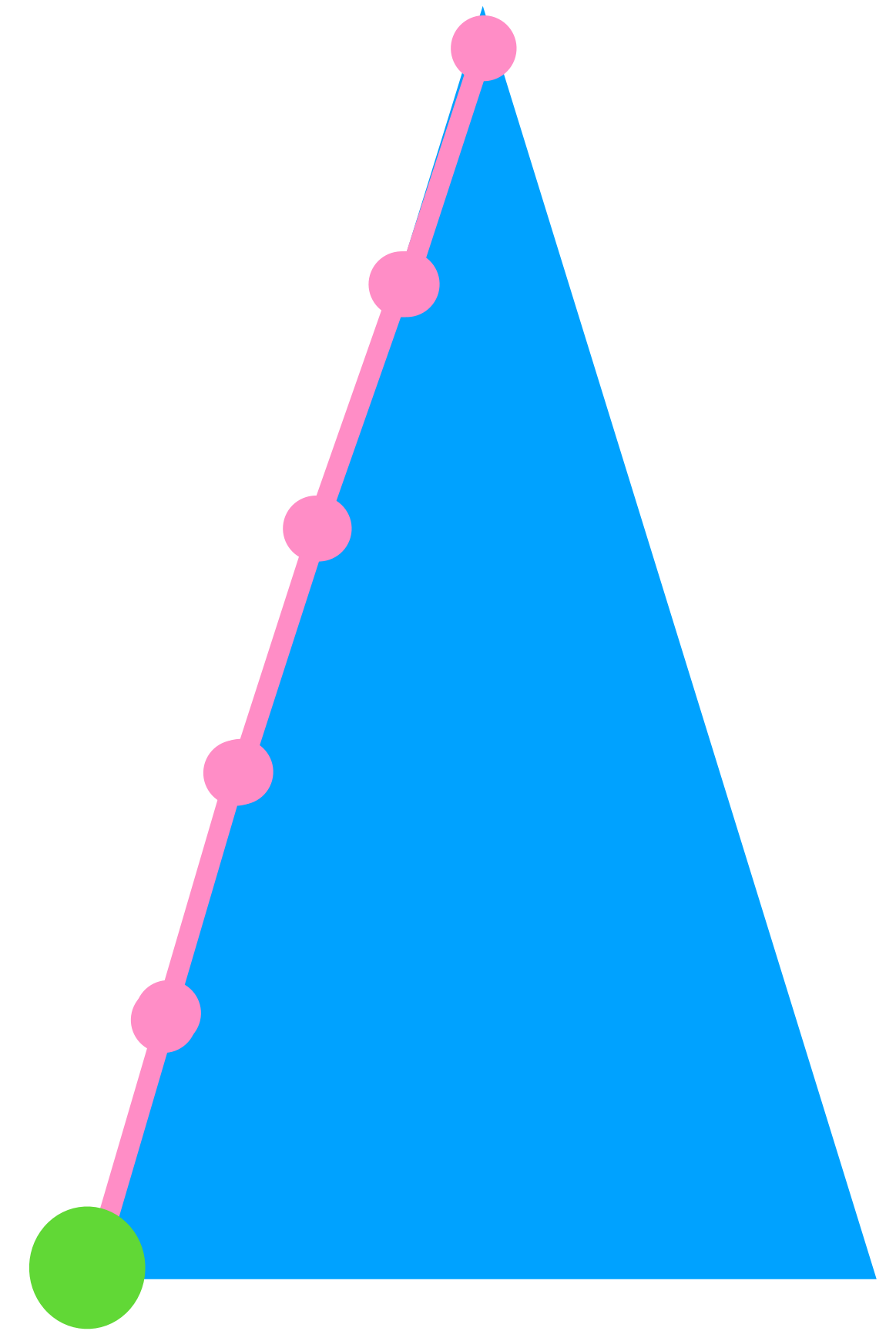
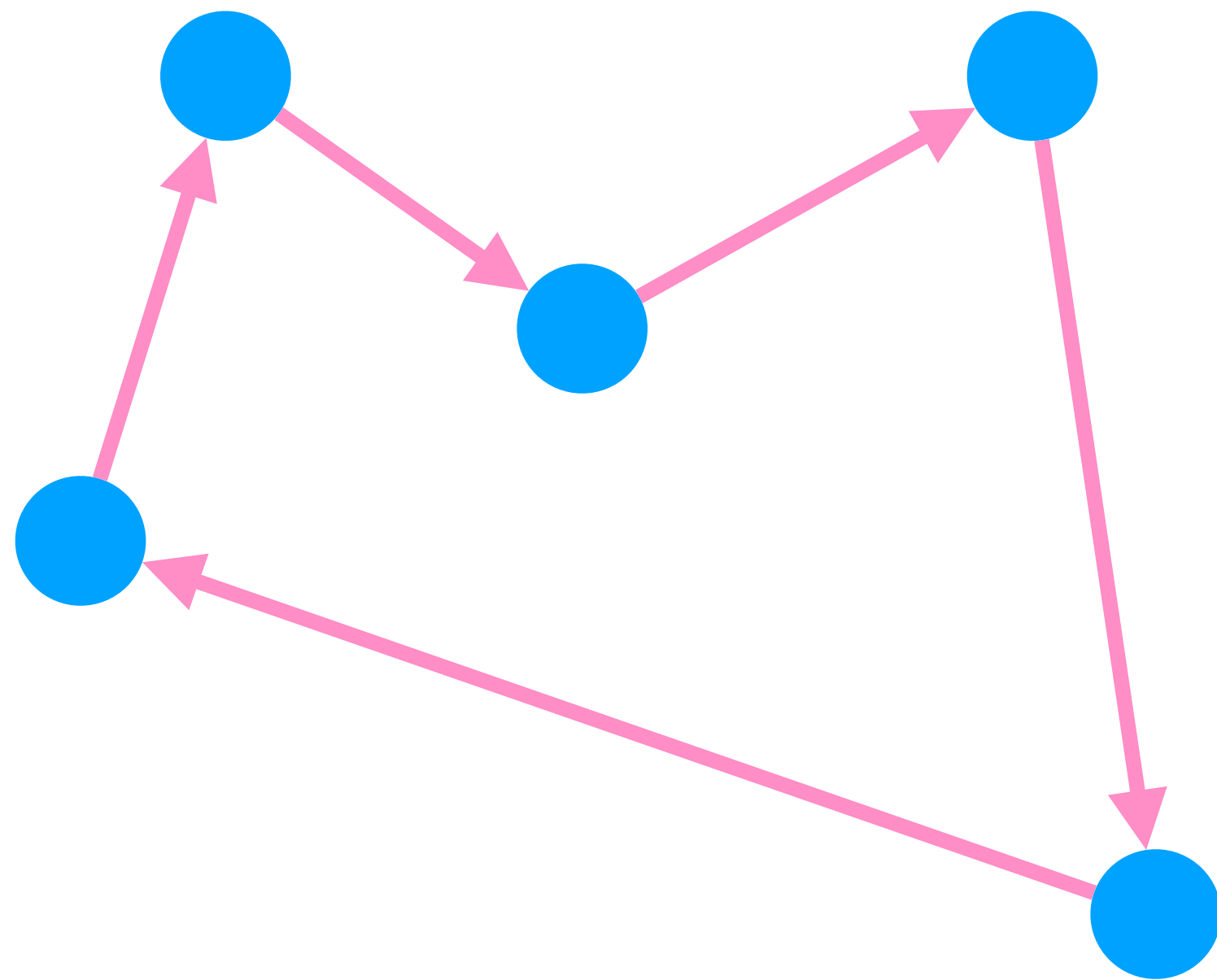


Example: Traveling Salesperson Problem



Example: Traveling Salesperson Problem

Can we automate this for *any* problem in order to find a good first feasible solution?



Making the First Solution Good!



Branching = 2 steps (in general: not necessarily in this order):

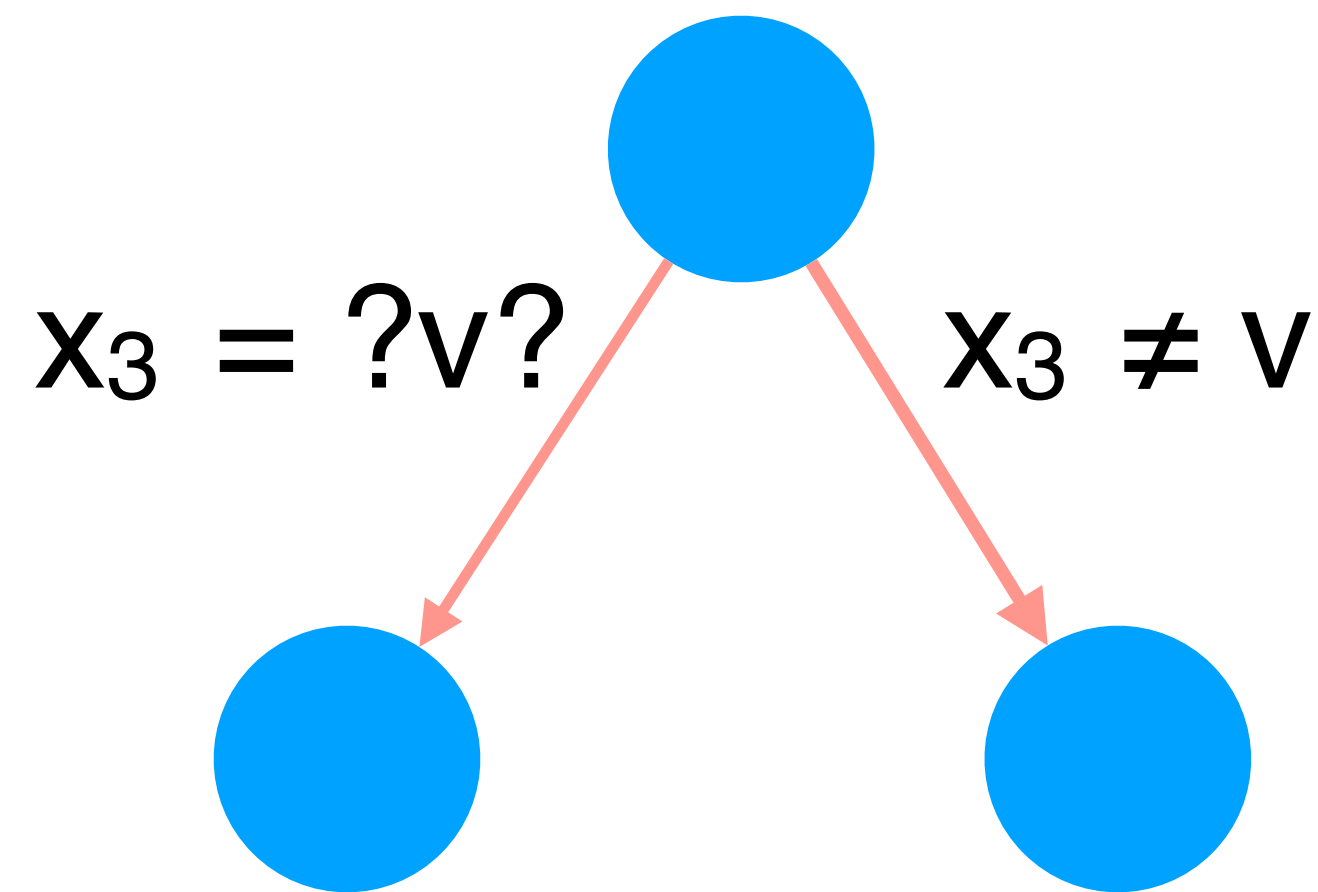
1. Variable selection.
2. Value selection & partition selection: the bound-impact value selector (BIVS) selects a value with the smallest objective lower bound after propagation of the fixing of the selected variable to this value (when minimizing).

Jean-Guillaume Fages and Charles Prud'homme.
Making the first solution good!

International Conference on Tools with Artificial Intelligence (ICTAI), 2017.

Making the First Solution Good!

- ▶ Minimize $O = f(x_1, x_2, \dots, x_{10})$ subject to some constraints.
- ▶ The variable selection heuristic selects x_3 .
- ▶ Its current domain is $D(x_3) = \{v_1, v_2, v_3, v_4\}$.



```

best = null
bestObj = +inf
for v in {v1, v2, v3, v4}
    sm.saveState()
    cp.post(x3 = v)
    if min(O) < bestObj:
        best = v
        bestObj = min(O)
    sm.restoreState()
branch {cp.post(x3 = best)}
        {cp.post(x3 != best)}
  
```

Making the First Solution Good: Advice



- ▶ This procedure (for making the first solution good) is quite costly: fixpoint computation for *every* value + saving & restoring the state.
- ▶ The advice is to do this only for discovering the *first* feasible solution and then to use a more naïve (but faster) value selection heuristic.

Value Selection: Phase Saving

- If a value v was successfully used for fixing a selected variable x — that is: if branching on $x=v$ did not lead to failure of the fixpoint algorithm — then store the value v as the last success value of x .
- Each time the variable x is selected by the variable selection heuristic, first try its last success value.

```
public static Supplier<Procedure[]> firstFail(IntVar... x) {
    return () -> {
        IntVar xs = selectMin(x,
            xi -> xi.size() > 1,
            xi -> xi.size());
        if (xs == null)
            return EMPTY;
        else {
            int v = xs.min();
            return branch(() -> xs.getSolver().post(equal(xs, v)),
                () -> xs.getSolver().post(notEqual(xs, v)));
        }
    };
}
```

One can easily implement
a default search
doing phase saving.

```
public static <T, N extends Comparable<N>> T selectMin(T[] x,
    Predicate<T> p, Function<T, N> f) {
    T sel = null;
    for (T xi : x) {
        if (p.test(xi)) {
            sel = sel == null ||
                f.apply(xi).compareTo(f.apply(sel)) < 0 ? xi : sel;
        }
    }
    return sel;
}
```

Variable Selection

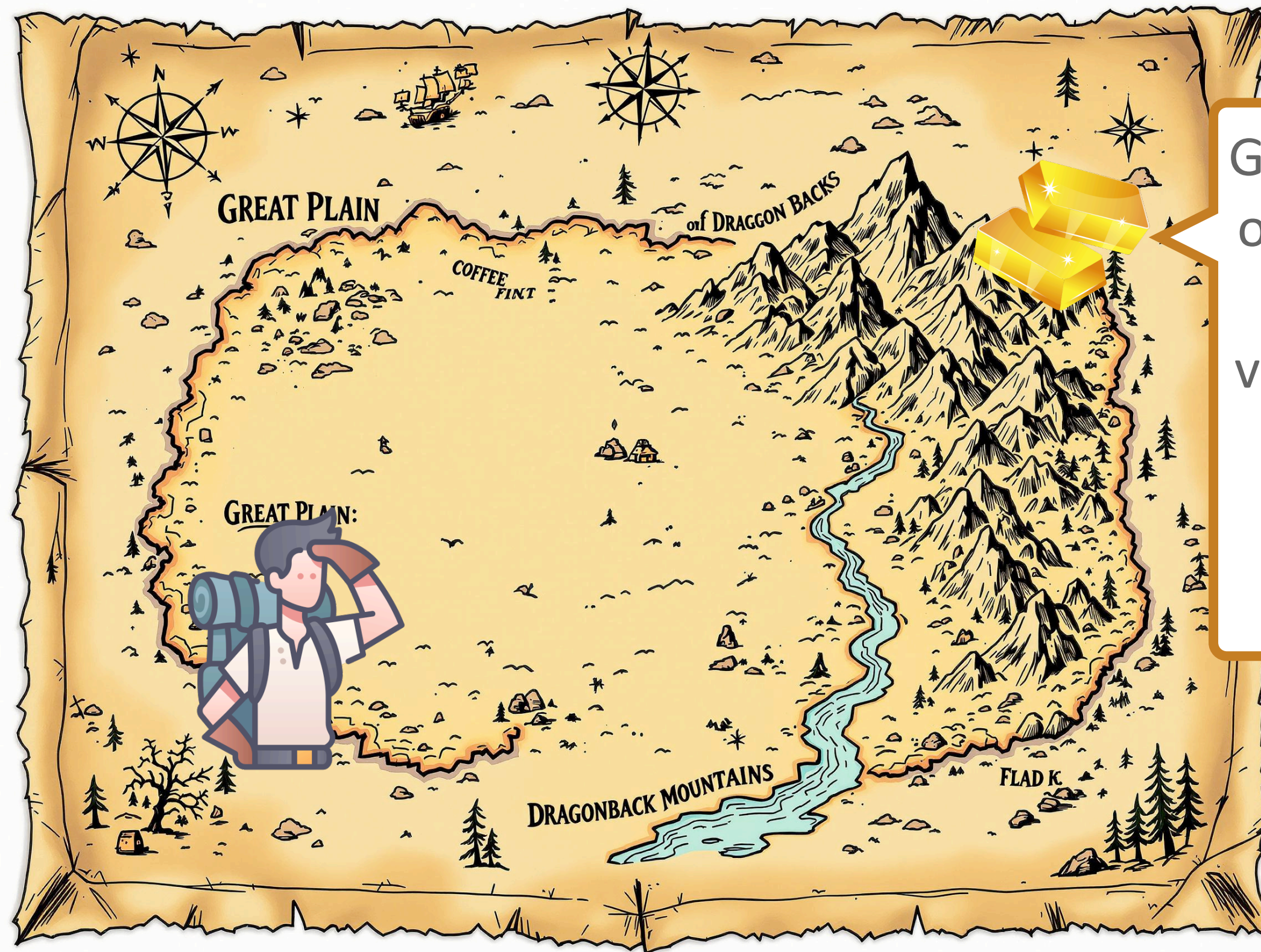
- Introduction to the first-fail (FF) principle
- A first instantiation of FF based on domain size

Reminder:

- Variable selection.
- Value selection & domain partition selection.

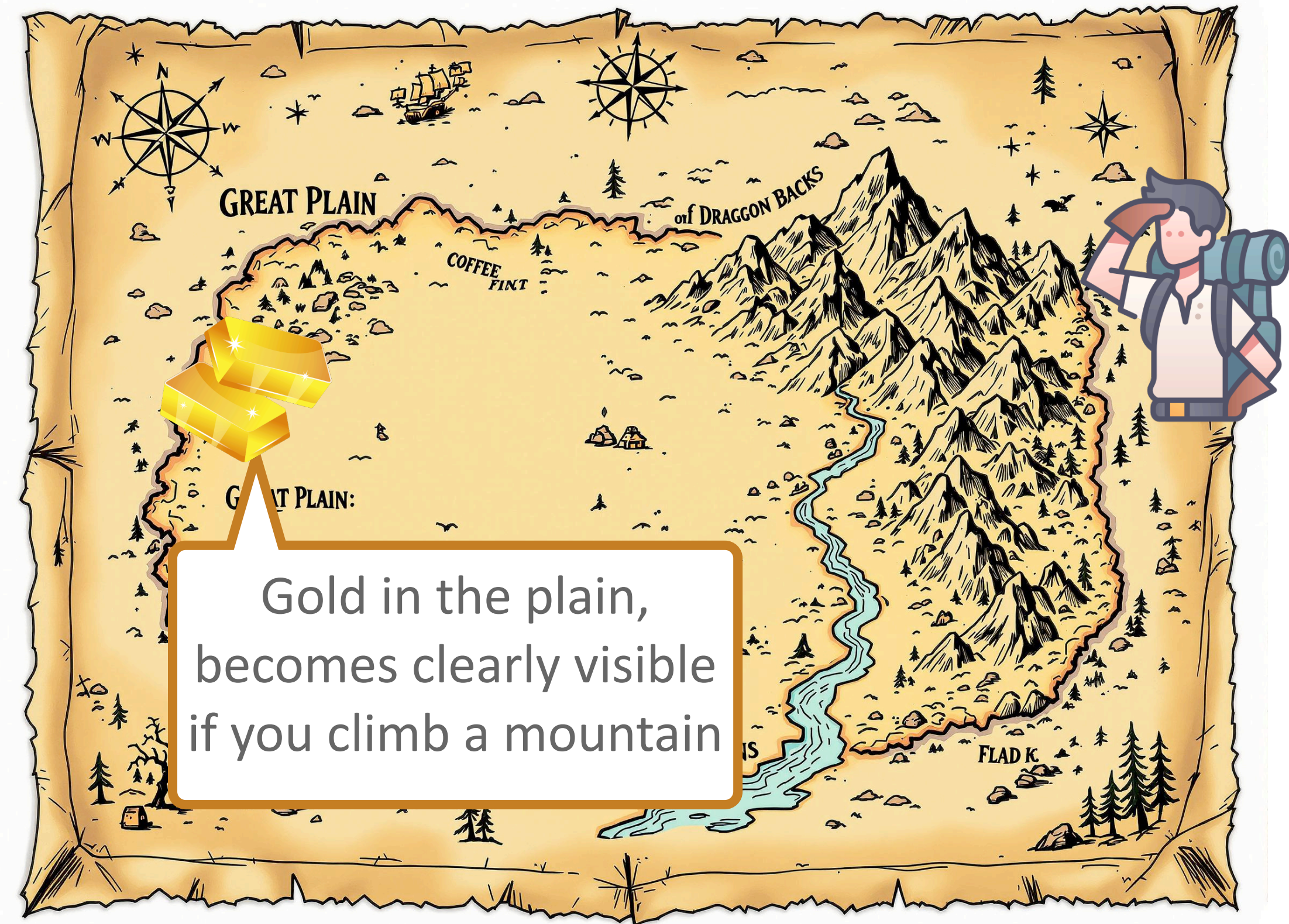
For the rest of this lecture, we focus on *variable* selection.

What situation is best to find the gold?



Gold in the middle of the mountains, not directly visible, unless you are lucky and climb the right mountain

The adventurer starts in a vast plain with no visibility of the treasure's location. He must blindly choose a mountain to climb, and if it's the wrong one, he must backtrack and try another. This mirrors a naive search strategy where decisions are made without guidance, leading to potentially many failed attempts before finding a solution.

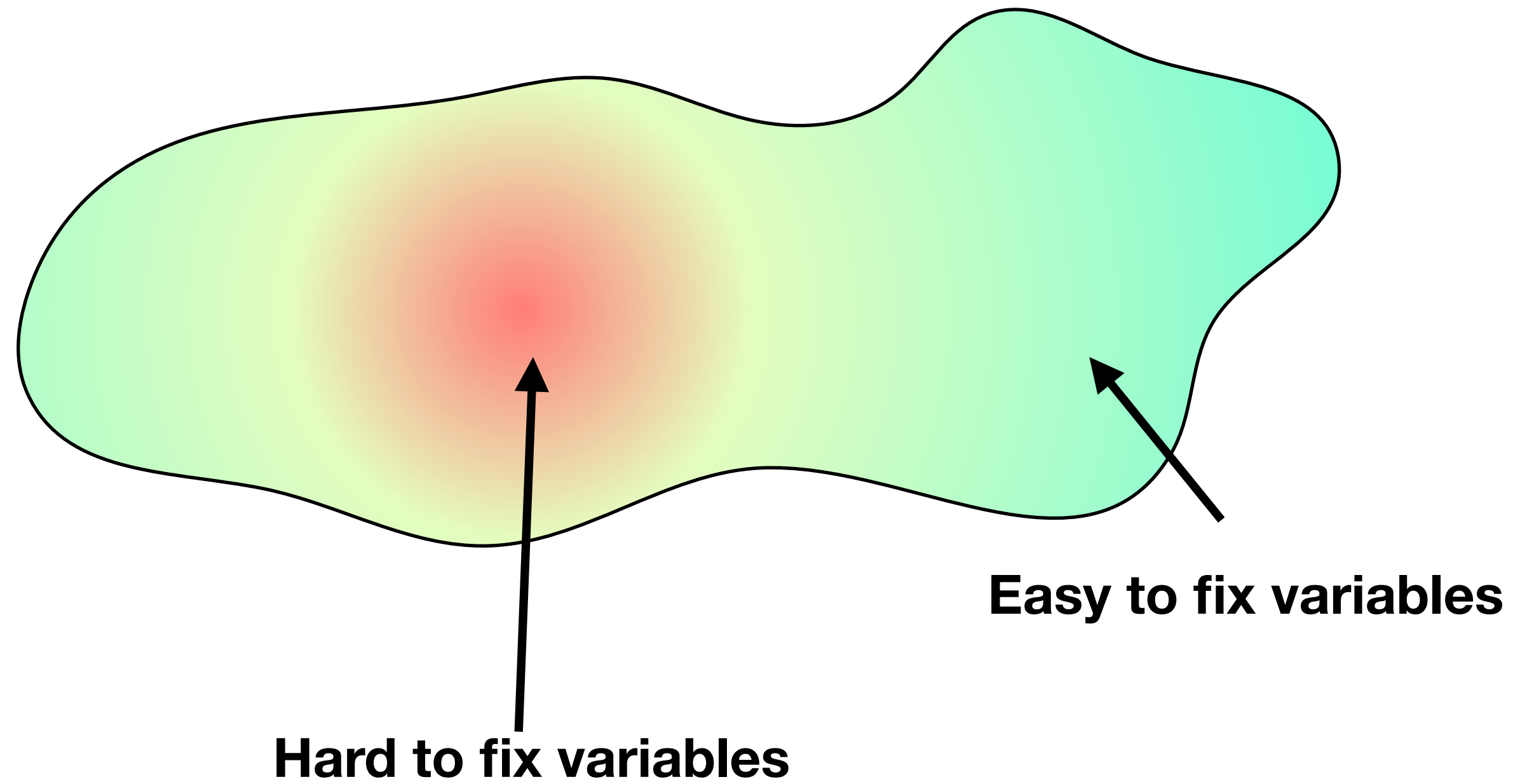


Gold in the plain, becomes clearly visible if you climb a mountain

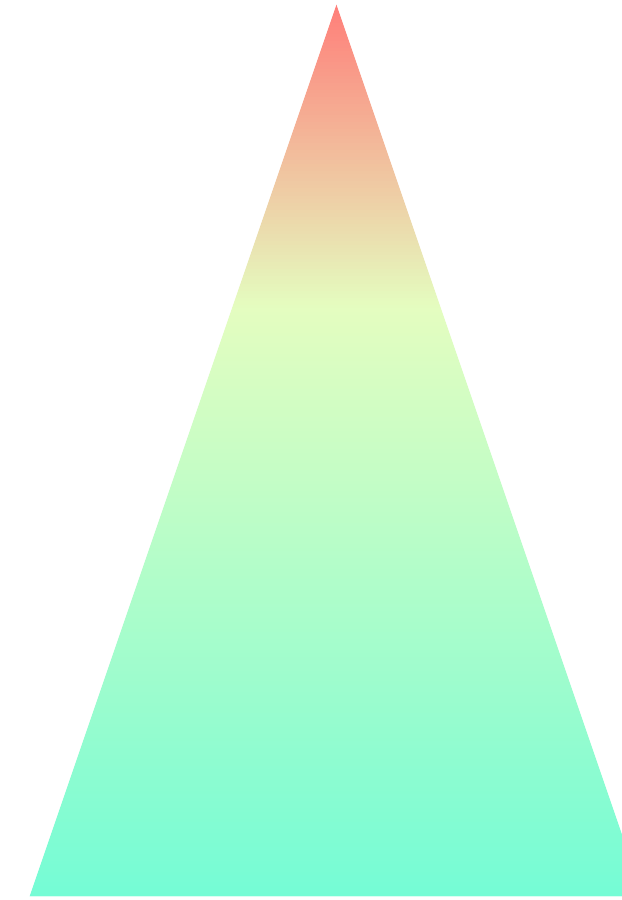
The adventurer starts by climbing a mountain, gaining visibility over the entire landscape. Even if the first mountain isn't the correct one, the vantage point reveals critical information about where the treasure is located. This mirrors the First Fail Principle, which selects the variable with the highest risk of failure first. By failing early on these critical choices, the search prunes large parts of the search space quickly, reducing unnecessary exploration and accelerating convergence to a solution.

First-Fail Visually

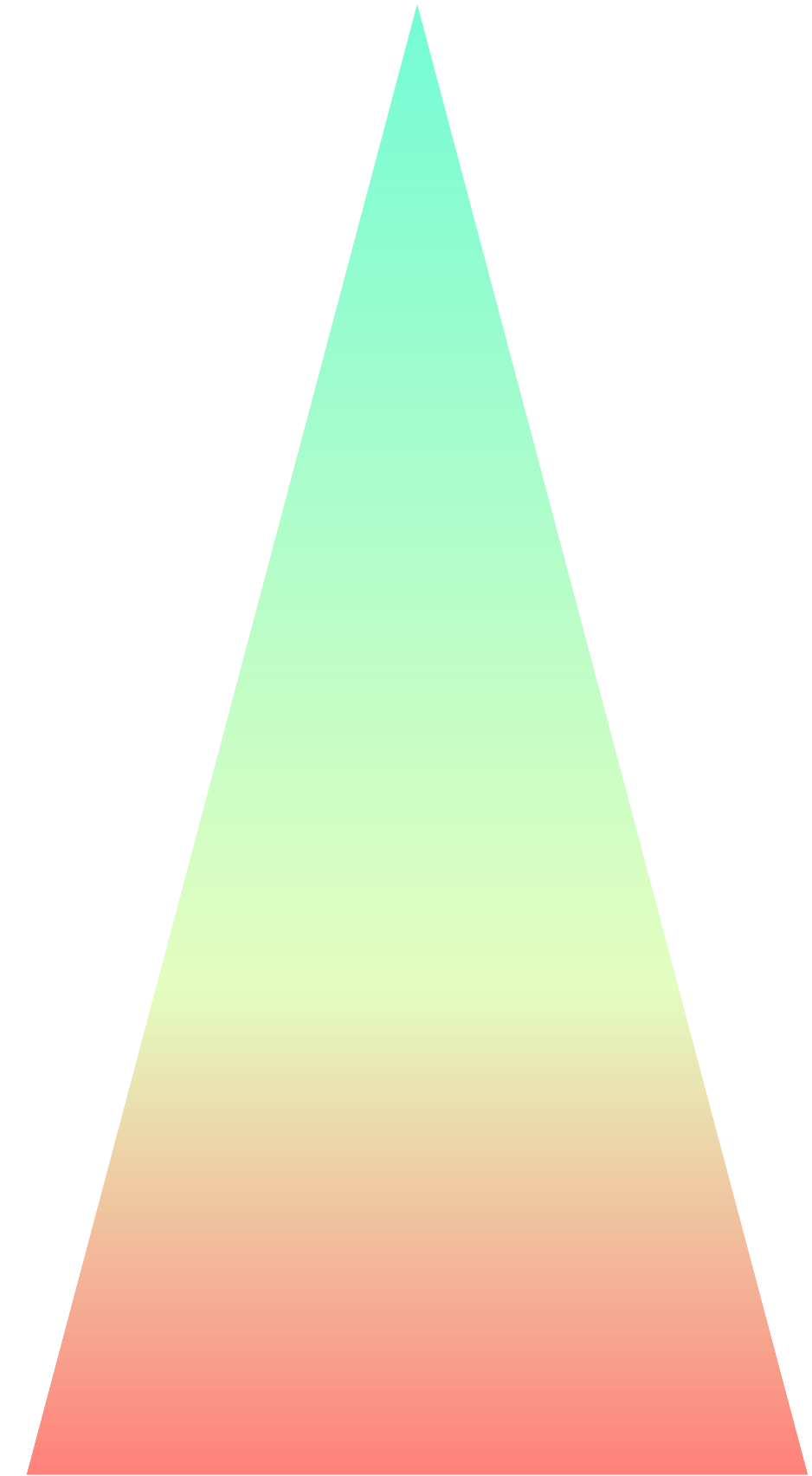
- Assume this is your problem



First-Fail:
spend a limited
time on the hard
part

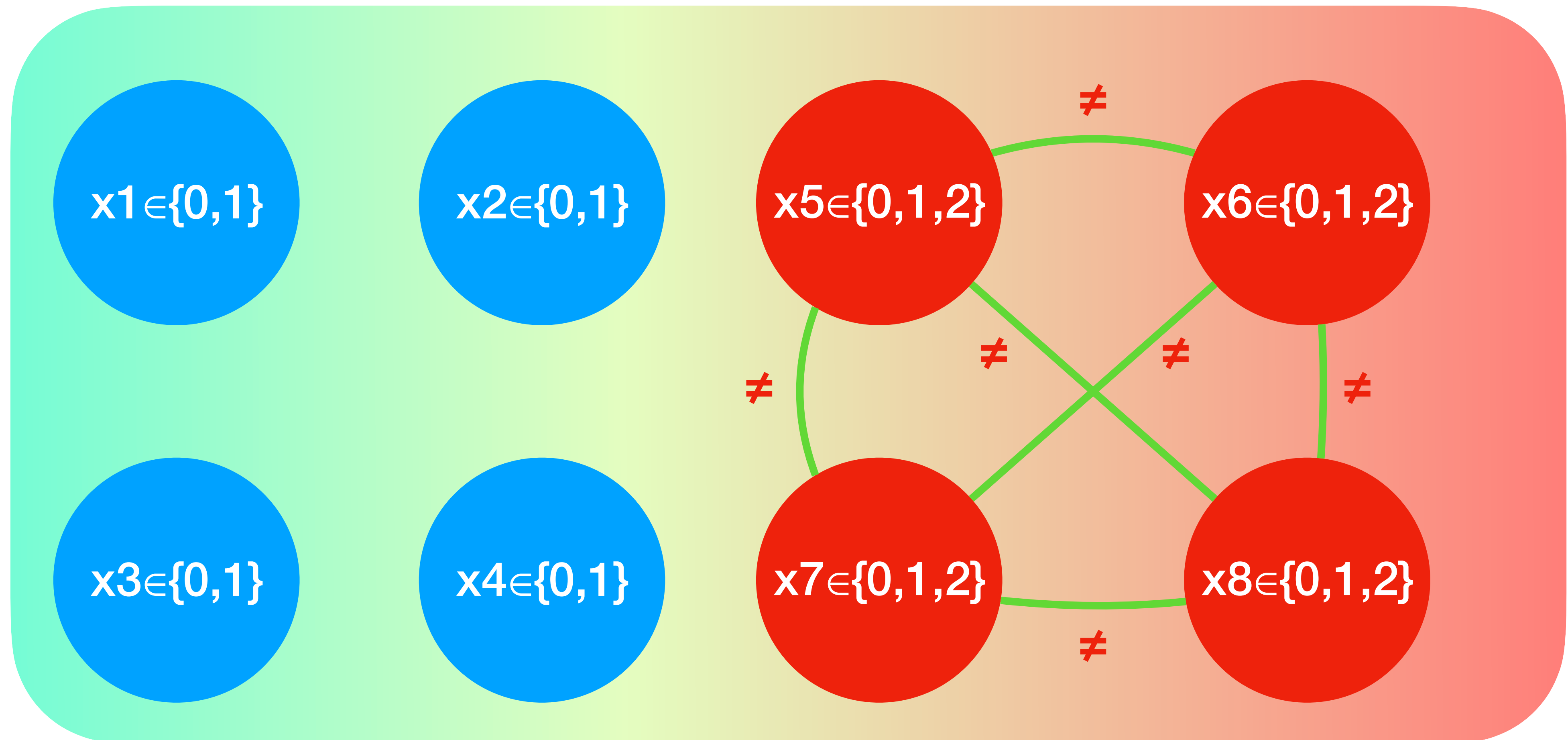


Not First-Fail:
spend most of your time
on the hard part

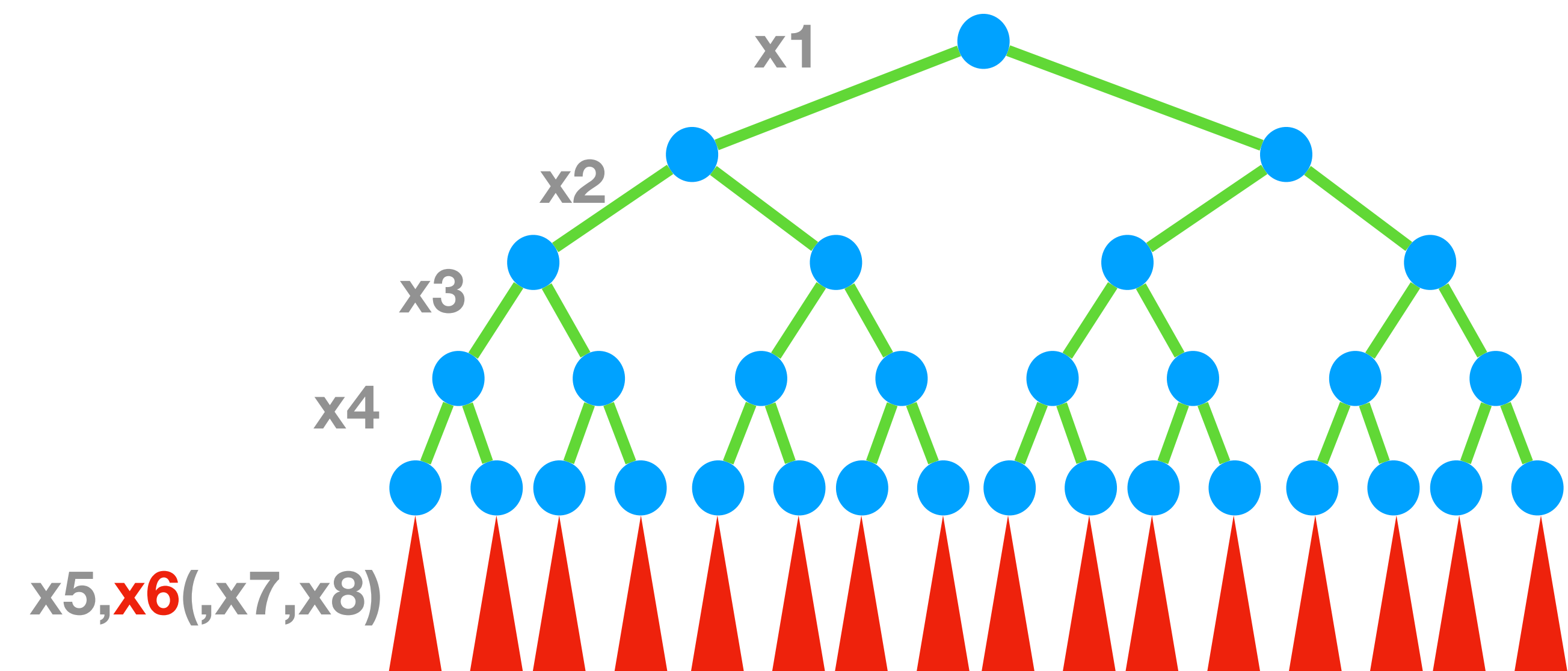


Example

Consider the CSP below, whose infeasibility is not detected by the fixpoint algorithm because of the AllDifferent decomposition. Assume we branch in the order $x_1, \dots, x_8$.

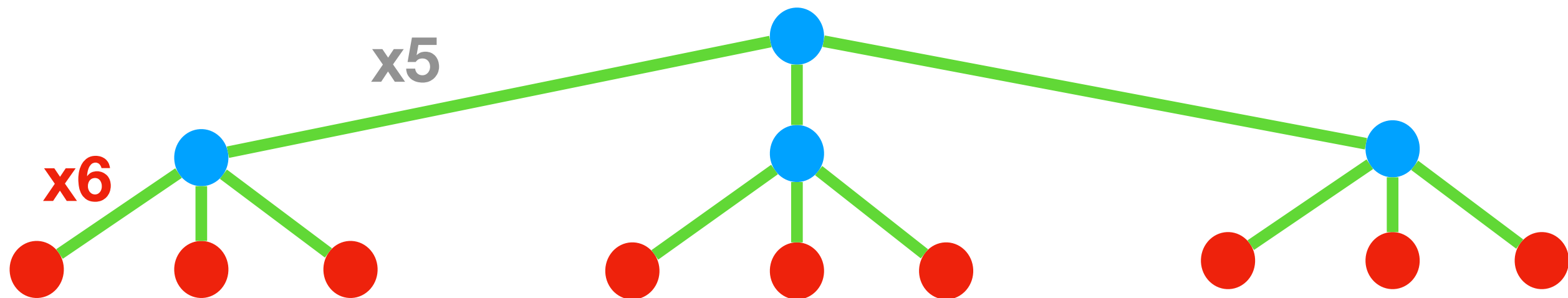


Example: Search Trees with 2 Static Orders



Not First Fail Order
[x1,x2,x3,x4,x5,x6,x7,x8].

First Fail Order
[x5,x6,x7,x8,x1,x2,x3,x4]

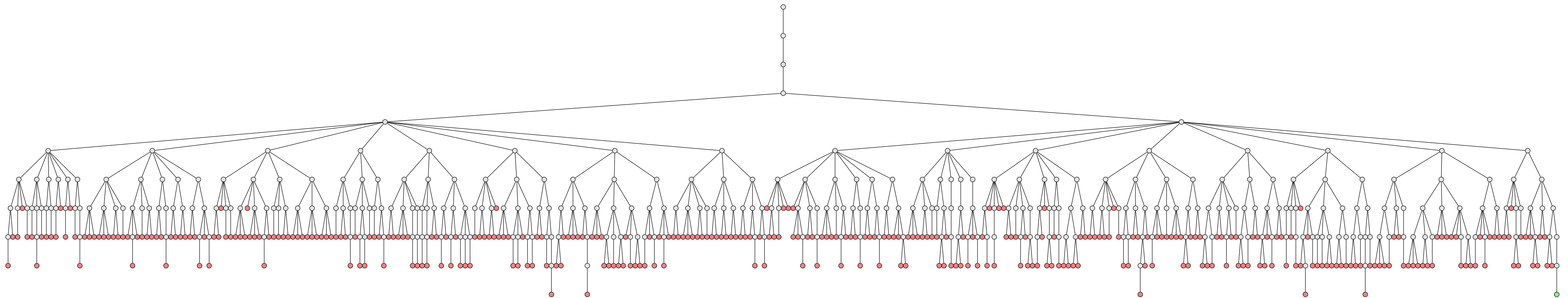


15-Queens Problem, first solution, static order

► Statistics:

- #choice: 687
- #fail: 344

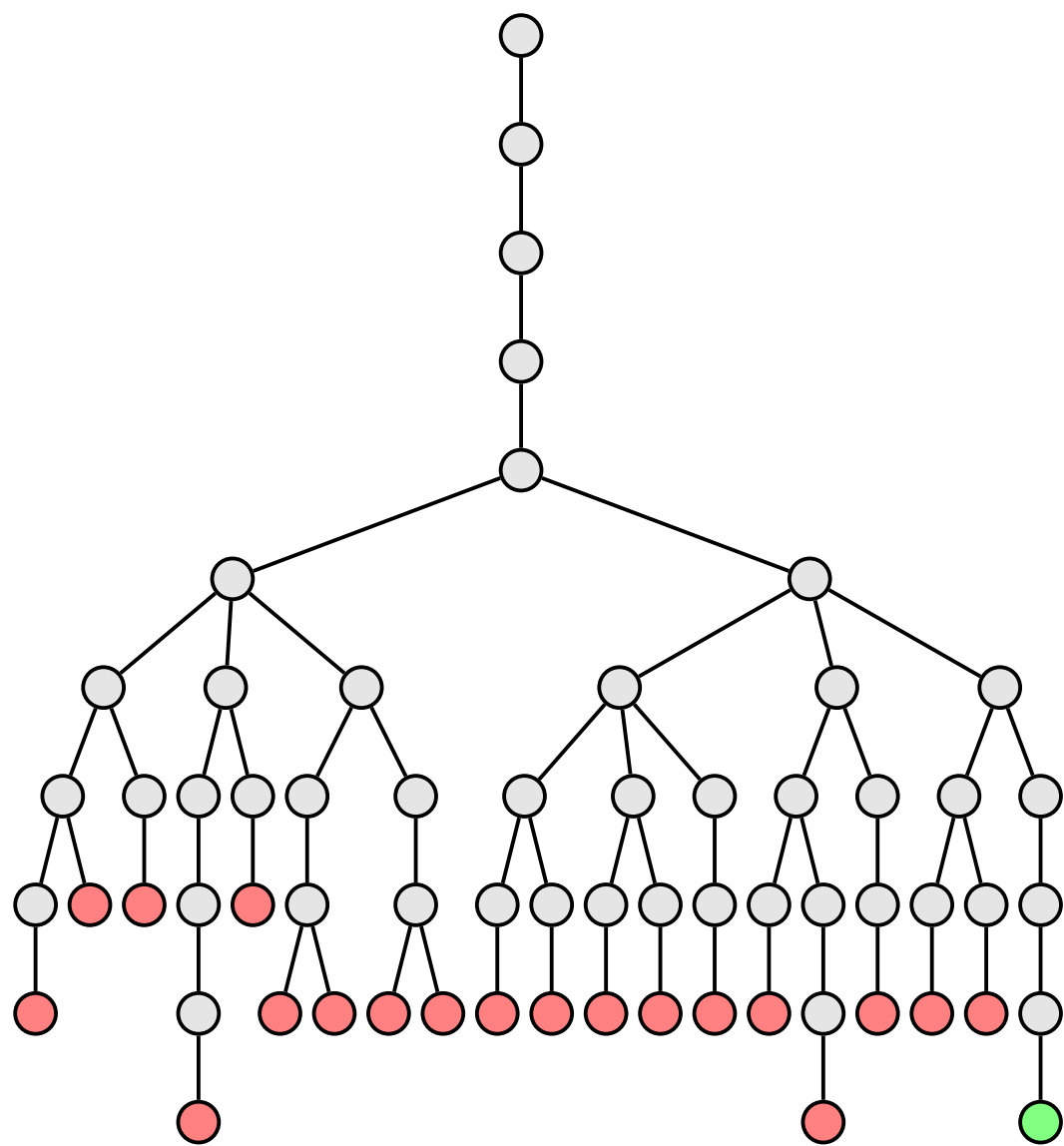
```
function SELECT-UNASSIGNED-VARIABLE(csp, assignment)
  for each var in csp.variables do
    if var not in assignment then
      return var
  return null # all variables are assigned
```



15-Queens Problem, first solution, first-fail order

► Statistics:

- #choice: 62
- #fail: 19



```
function SELECT-UNASSIGNED-VARIABLE(csp, assignment)
    bestVar ← null
    bestSize ← ∞

    for each var in csp.variables do
        if var not in assignment then
            domainSize ← SIZE(DOMAIN(var, csp, assignment))
            if domainSize < bestSize then
                bestSize ← domainSize
                bestVar ← var

    return bestVar
```

First-fail for variable selection:

Since all variables must eventually be fixed, if there are no solutions under a node (failure), then we prefer to detect this as soon as possible, so that not too much time is spent exploring the subtree under that node.

“To succeed, try first where you are most likely to fail.”

Robert M. Haralick and Gordon L. Elliott.

Increasing tree search efficiency for constraint satisfaction problems.

International Joint Conference on Artificial Intelligence (IJCAI), 1979.

- Can be implemented in various ways:
 - Min-Dom
 - Dom+Deg
 - Dom/Deg
 - Dom/Wdeg
 - Impact-based search
 - Activity-based search
 - Last-conflict search
 - Conflict-ordering search
 - ...

Min-Dom Heuristic

- ▶ Min-Dom: Select an unfixed variable with the smallest domain size.
- ▶ This heuristic was shown experimentally to minimize search-tree depth.
- ▶ It is quite intuitive that for $D(x_1) = \{1, 2\}$ and $D(x_2) = \{1, \dots, 100\}$ branching on x_1 first is likely to trigger more propagation.

```
int n = 8;
Solver cp = Factory.makeSolver(false);
IntVar[] q = Factory.makeIntArray(cp, n, n);
// constraints ...

DFSearch search = Factory.makeDfs(cp, () -> {
    IntVar qs = selectMin(q,
        qi -> qi.size() > 1,
        qi -> qi.size());
    if (qs == null) return EMPTY;
    else {
        int v = qs.min();
        return branch(() -> Factory.equal(qs, v),
            () -> Factory.notEqual(qs, v));
    }
});
```

Select an unfixed variable with the smallest domain size.

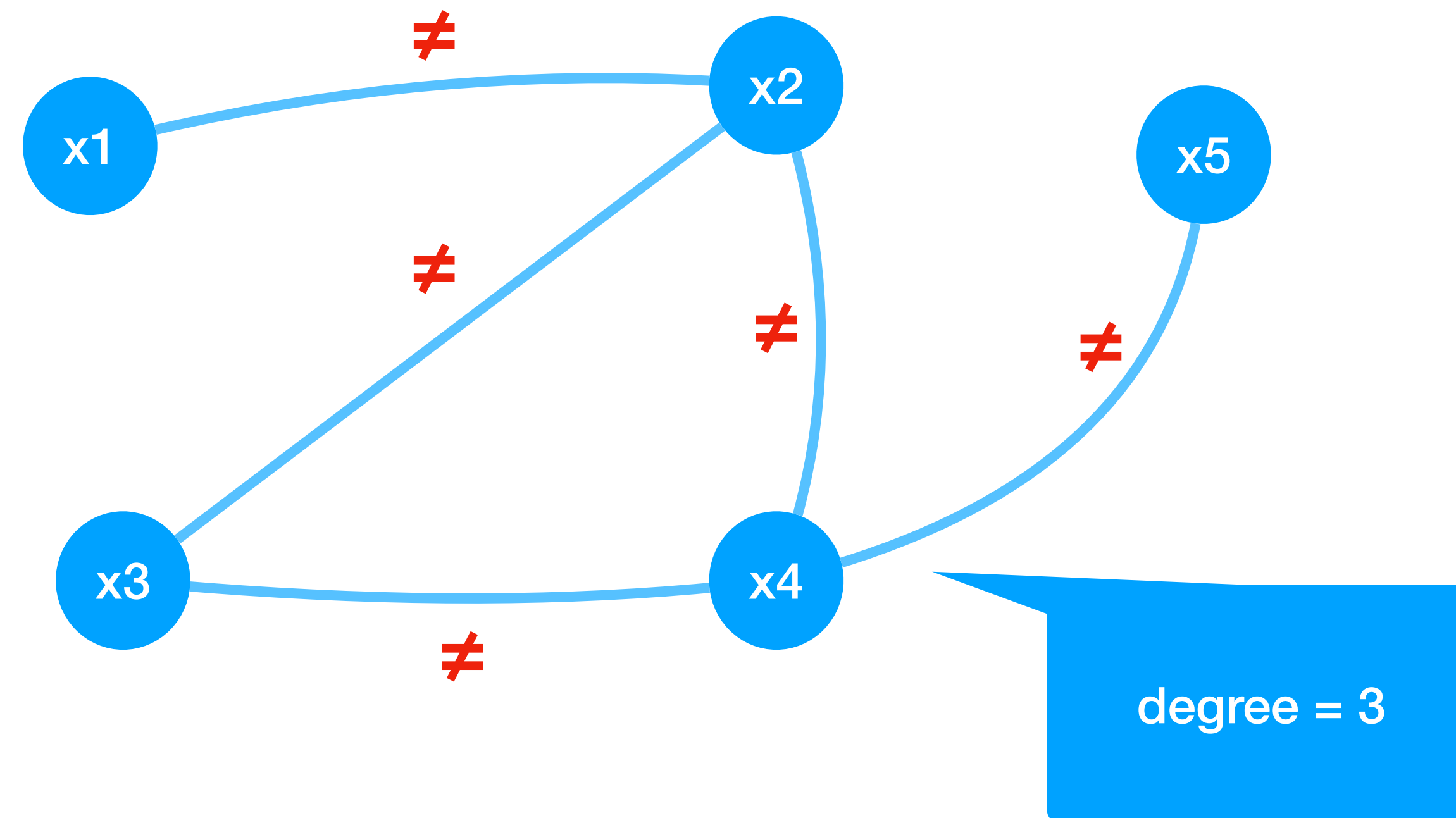
```
search.onSolution(() -> println("solution:" + Arrays.toString(q)));
SearchStatistics stats = search.solve();
```

Variable Selection

FF based on the degrees in the constraint graph

Degree-Based Heuristics

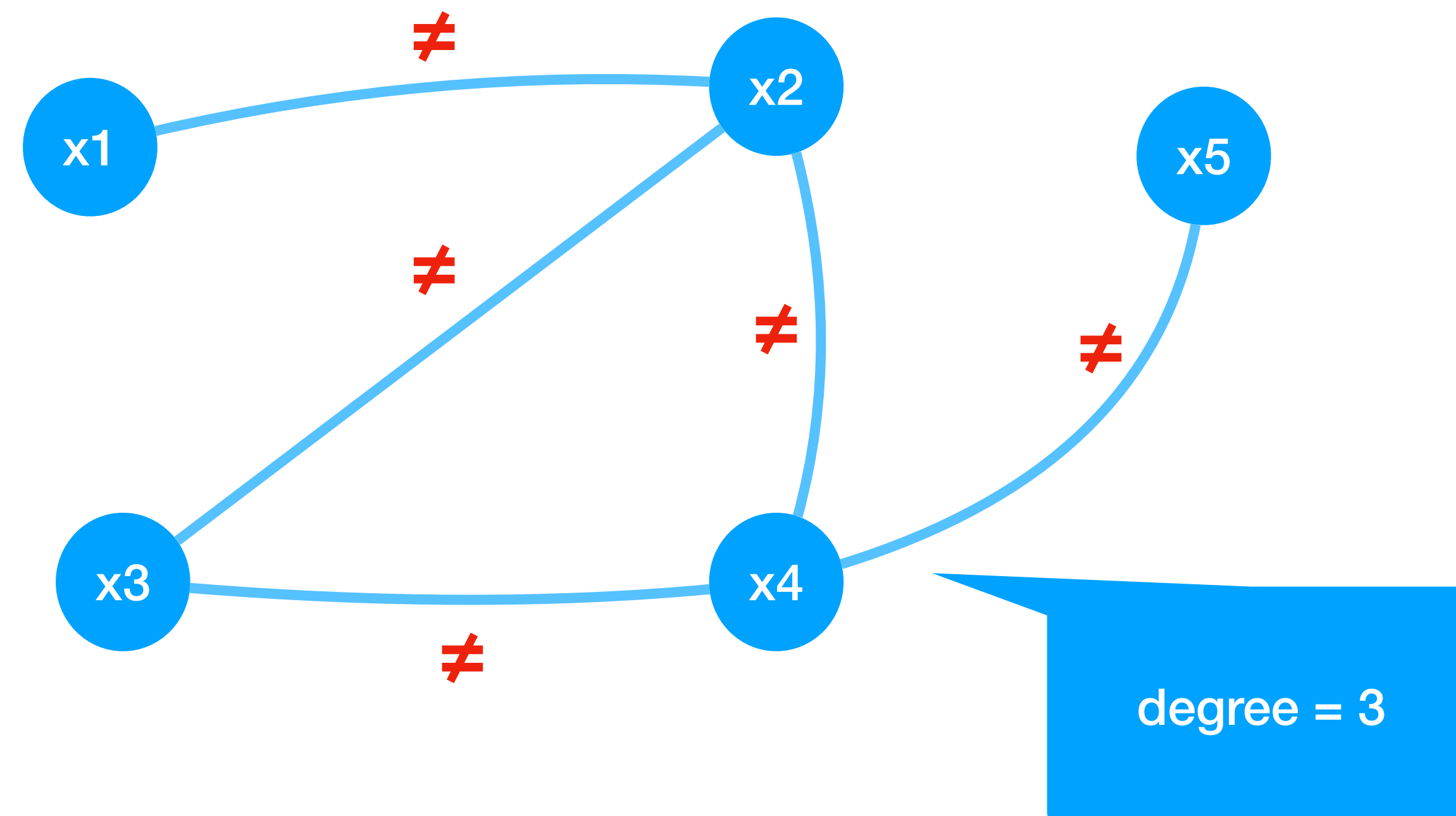
If a ***variable is involved in many constraints***, then it is likely that any filtering of the domain of this variable will trigger some filtering for other variables of those constraints.



Three variants ... H0, H1, H3

Blending Dom and Deg

- ▶ H0: Degree of a variable x = the number of constraints on x .
- ▶ H1: Dom+Deg:
 - Min-Dom, and break ties with the degree.
- ▶ H2: Dom/Deg:
 - Select an unfixed variable with the smallest ratio between domain size and degree.



H3: Weighted Degree (WDeg) Heuristic

- ▶ Idea: Introduce **learning** to find out what the “difficult” variables are.
- ▶ Each time a constraint fails, its **weight** is increased (by +1).
 - If a constraint has failed a lot in the past, it will probably fail again a lot in the future.
- ▶ **Weighted degree** of a variable x is
 - the sum of the weights of the constraints on x + at least one other unfixed variable.
- ▶ **Dom/WDeg**
 - Select the first unfixed variable with the smallest ratio $\frac{domainSize}{weighedDegree}$
- ▶ **Weakness**
 - A constraint that fails may not be the only guilty one
 - Cascade of propagations in the fixpoint algorithm!

Dom/WDeg Implementation

- ▶ Not shipped with MiniCP, but quite easy to implement.
- ▶ Pre-requisites
 - Add a method called **scope** to the class **AbstractConstraint**:
it returns the list of variables in the scope of the constraint.
 - ▶ Add a counter **nF** in **AbstractConstraint** to track how many failures it caused
 - ▶ Add a method **cstr** to **IntVarImpl** to get the set of constraints that use the variable
 - ▶ Add a method **dynamicDegree** to constraints that yields the number of free variables
 - ▶ Attach to the propagation an update of **nF** when exception raised
- ▶ Provide a method that *scores* a variable x according to failures in its network
 - ▶ It uses **$x.cstr()$** to retrieve all constraints mentioning **x**
 - ▶ It scores each constraint **c** with
 - ▶ $score(c) = c.nF * (c.dynamicDegree() - 1 > 0)$
 - ▶ It computes $score(x) = \frac{dom(x)}{\sum_{c \in cstr(x)} score(c)}$
- ▶ Your best variable x^* minimizes $score(x^*)$

Variable Selection

FF based on estimated filtering *impact*

Impact-Based Search (IBS)

- Idea: Take a branching decision that has the lowest estimated impact in terms of filtering.
- Preliminaries:

$$P = \langle X, D, C \rangle$$

CSP

$$S(P) = \prod_{x \in X} |D(x)|$$

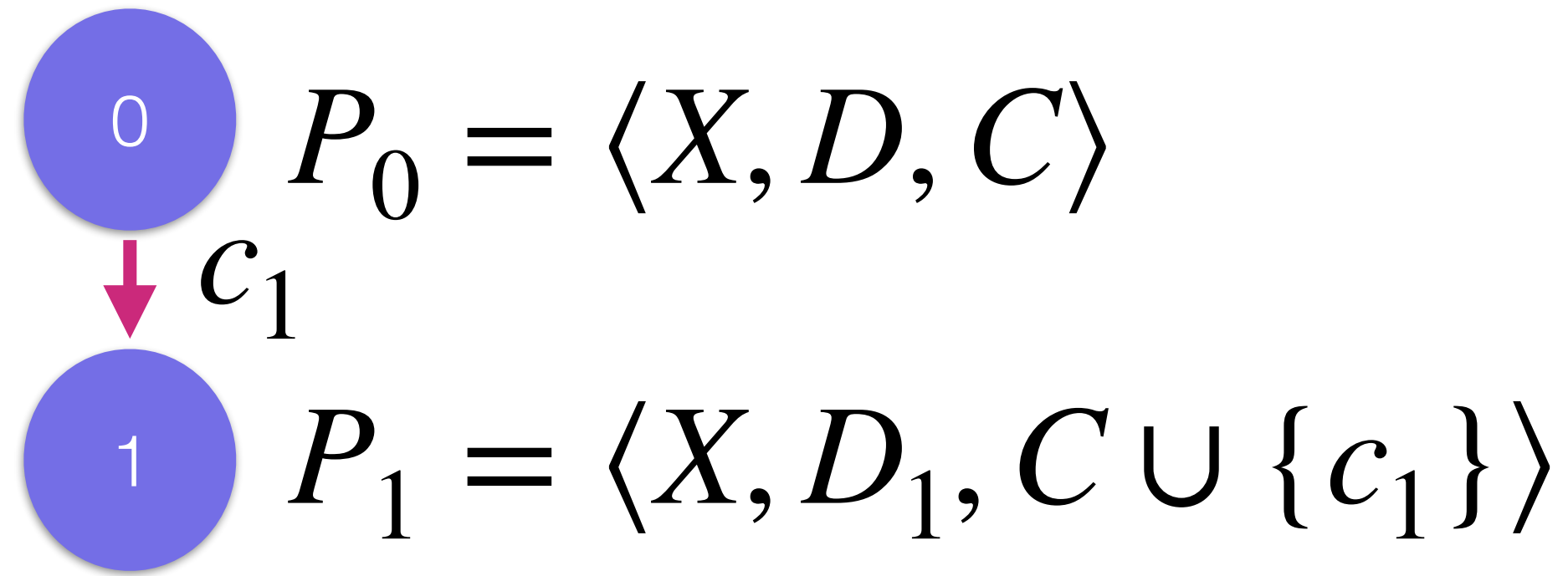
estimate of the size
of the search tree

Philippe Refalo.

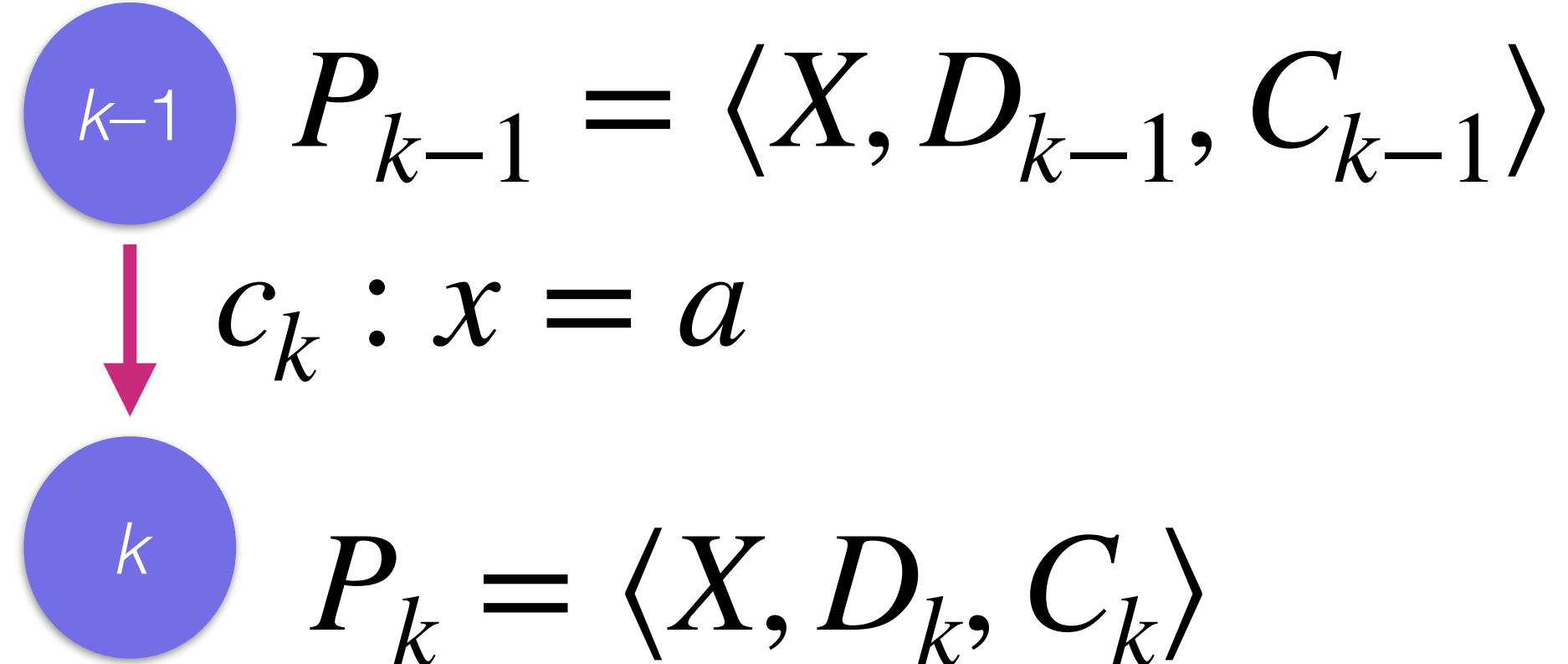
Impact-based search strategies for constraint programming.

International Conference on Principles and Practice of Constraint Programming (CP), 2004.

Impact-Based Search (IBS)



$$C_{k-1} = C \cup \{c_1, \dots, c_{k-1}\}$$



= 0 (no contraction)
= 1 (failure, full contraction)

Contraction of search space with
respect to parent node:

$$I^k(x = a) = 1 - \frac{S(P_k)}{S(P_{k-1})}$$

Impact-Based Search (IBS)

- We can estimate over a set $\mathcal{K}$ of search nodes the average impact of $x=a$:

$$\bar{I}(x = a) = \frac{1}{|\mathcal{K}|} \cdot \sum_{k \in \mathcal{K}} I^k(x = a)$$

- This estimate can also be updated at every node, with a forget factor, instead of being averaged:

$$\bar{I}_k(x = a) = (1 - \alpha) \cdot \bar{I}_{k-1}(x = a) + \alpha \cdot I^k(x = a)$$

Exponential moving average $\alpha \in [0,1]$:
the higher α is, the faster it will
discount older observations.

Impact-Based Search (IBS)

- Estimation of the size of the search tree when trying $x=a$:

$$S(P) \cdot (1 - \bar{I}(x = a))$$

= 0 (no contraction)
= 1 (failure, full contraction)

- Estimation of the size of the search tree when labeling the variable x :

$$\sum_{a \in D(x)} S(P) \cdot (1 - \bar{I}(x = a))$$

constant whatever the variable we branch on

- Impact** of a variable:

$$I(x) = \sum_{a \in D(x)} (1 - \bar{I}(x = a))$$

Variable heuristic:

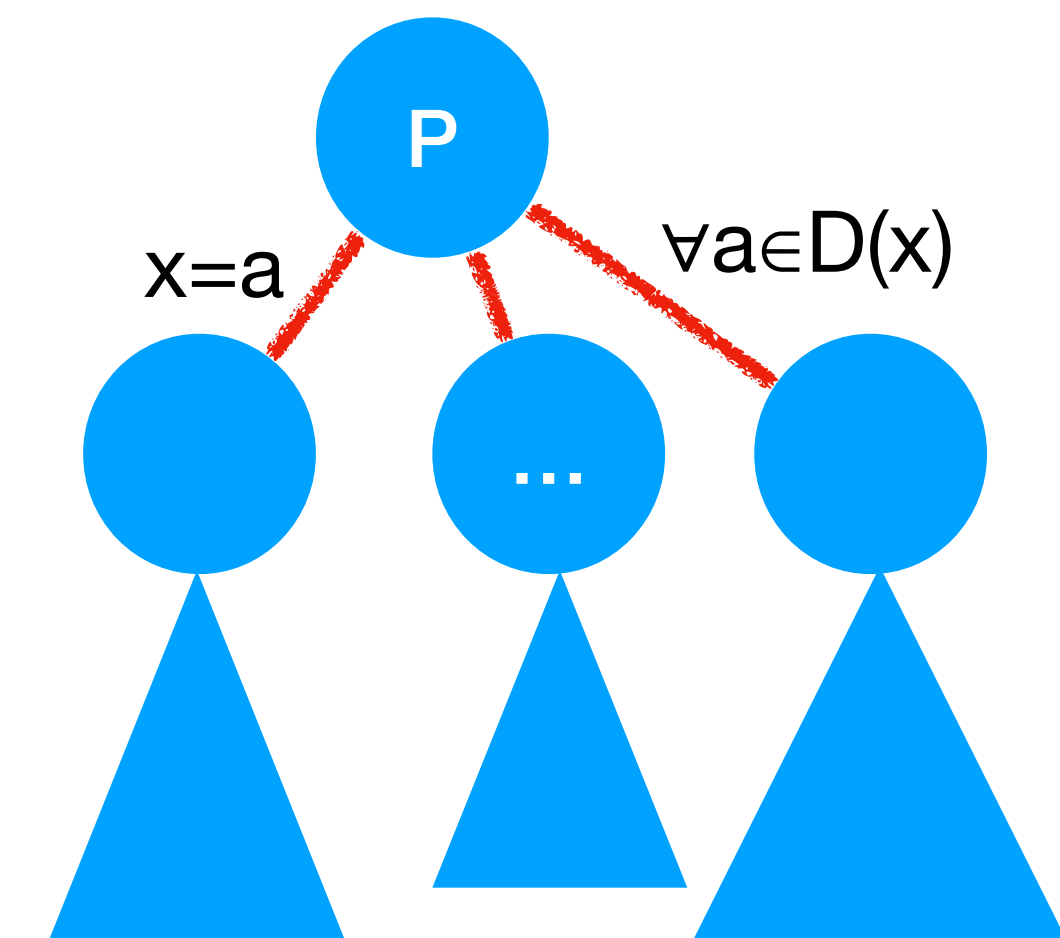
Select x with minimum

$$I(x)$$

Value heuristic:

Select a with minimum

$$\bar{I}(x = a)$$



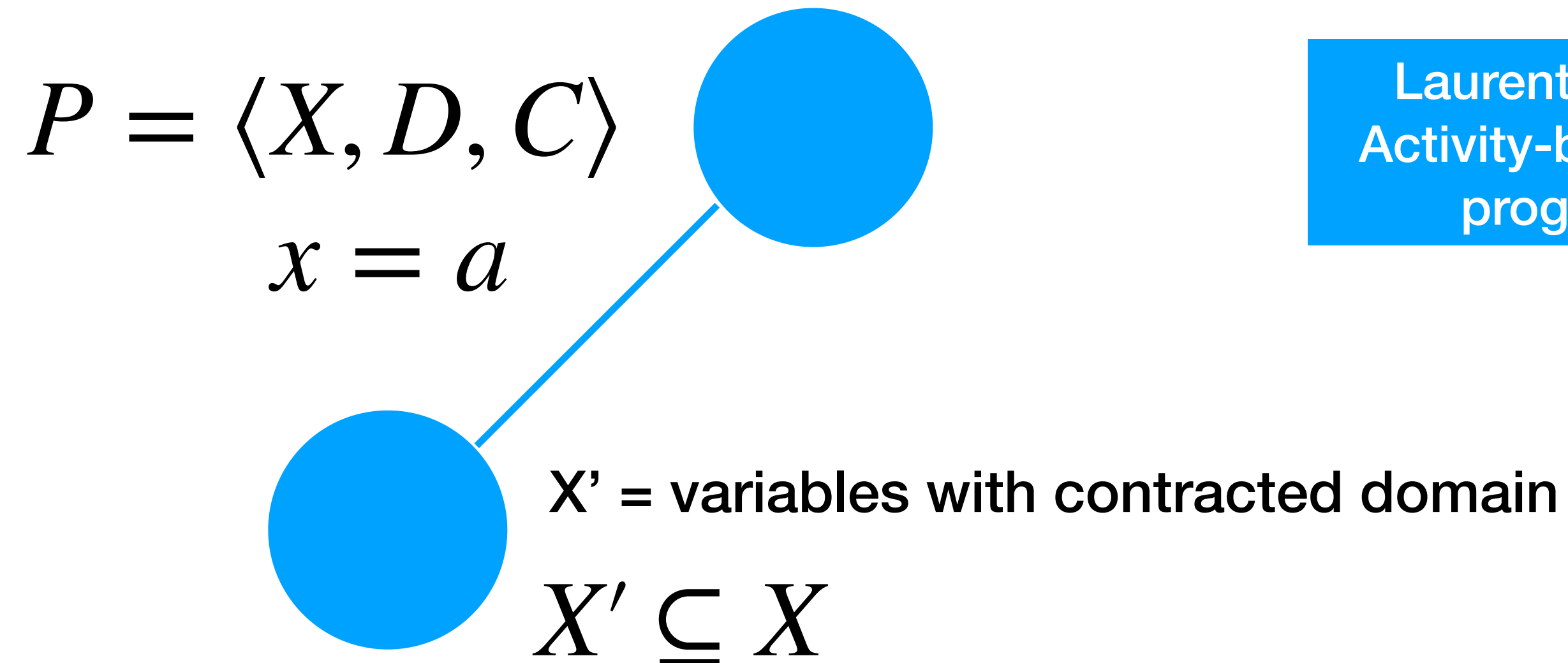
Variable Selection

FF based on estimated filtering *activity*

- ▶ Idea: Track how often a variable domain is contracted during search, as this allows some learning.
- ▶ We expect that a variable whose domain is often contracted is a good candidate to branch on early, in order to reduce the depth of the search tree.

Activity-Based Search (ABS)

Each time a search choice is executed, increment the counter $A(x)$, denoting the **activity** of x , of every variable x having a domain contraction:



Laurent Michel and Pascal Van Hentenryck.
Activity-based search for black-box constraint
programming solvers. *CPAIOR*, 2012.

decay factor in $[0..1]$

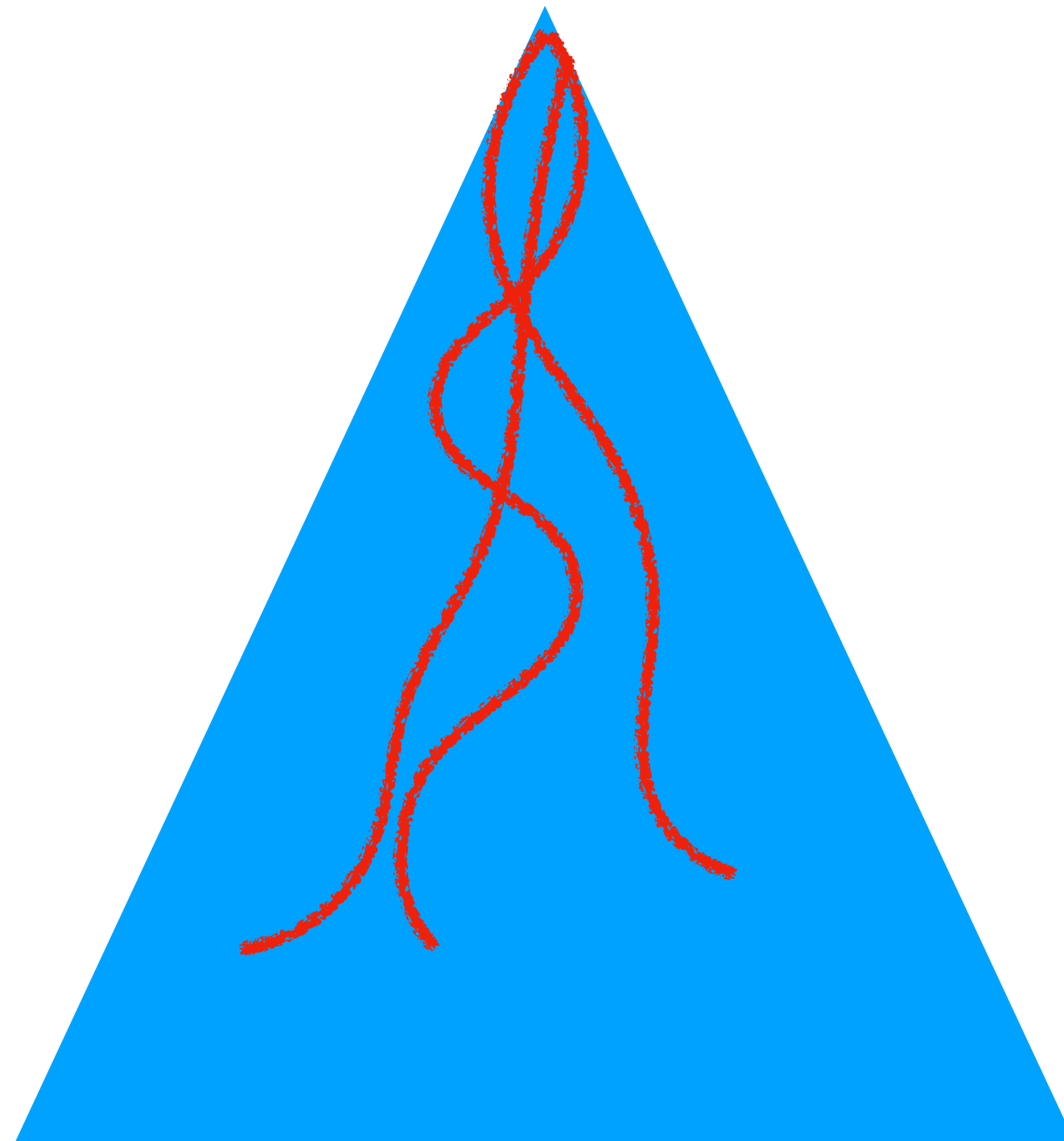
$\forall x \in X$ **where** $|D(x)| > 1 : A(x) := A(x) \cdot \gamma$

$\forall x \in X' : A(x) := A(x) + 1$

Variable Heuristic: $\frac{A(x)}{|D(x)|}$
Select x maximizing

Initialization of Learning Heuristics

Activity-based search and impact-based search are initialized with some random dives before starting a complete search:



Variable Selection

FF based on most recent conflicts

Last-Conflict Search

- ▶ Can be used in combination with another heuristic; let us call it **fallBackHeuristic**.
- ▶ Let **lastConflictVariable** store a reference, initialized to null, to a decision variable: this variable is the last one we branched on that led to failure of the fixpoint algorithm.

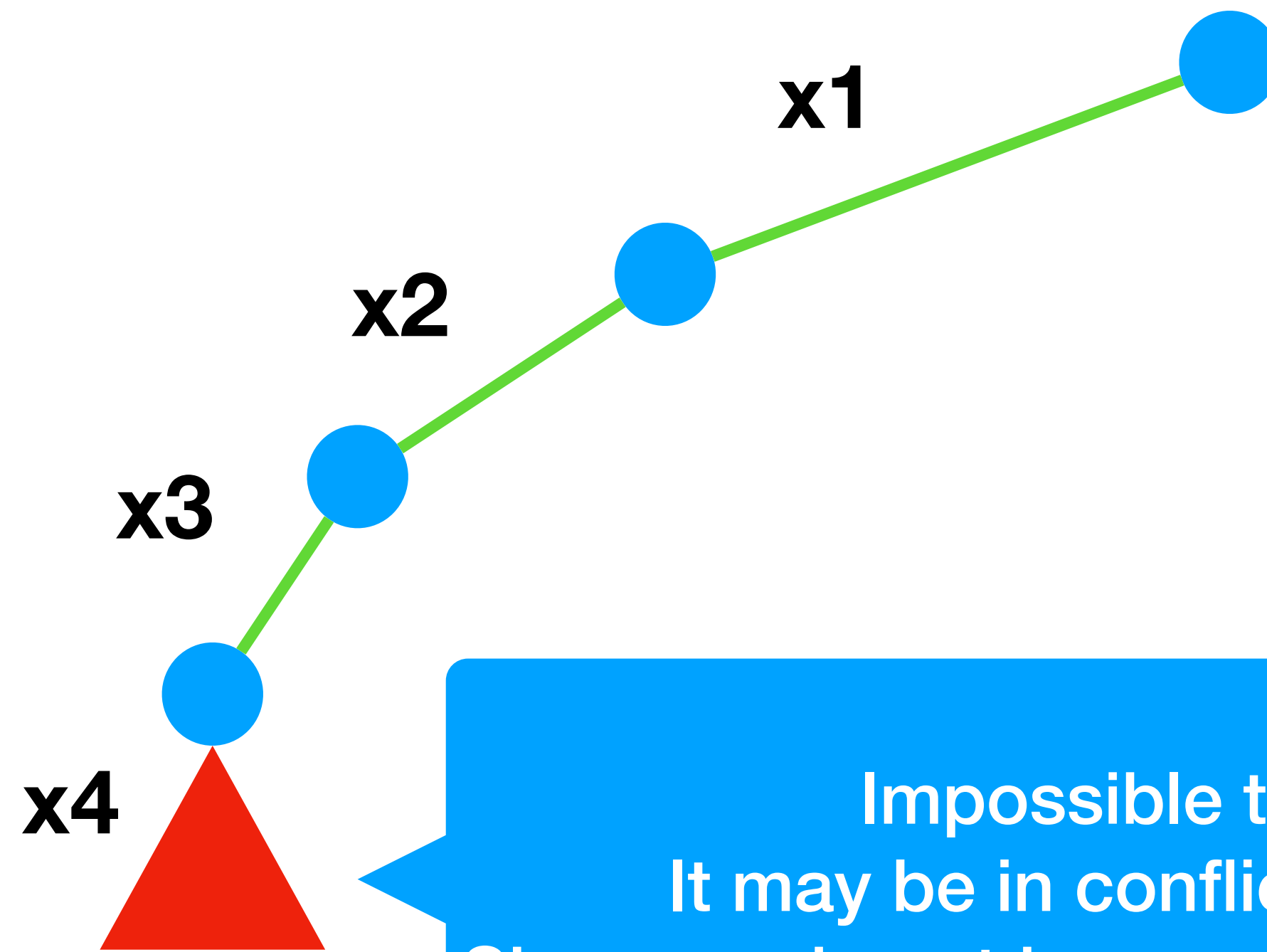
```
nextVarToBranchOn(X) {  
    if (lastConflictVariable == null) {  
        return fallBackHeuristic(X)  
    } else {  
        return lastConflictVariable  
    }  
}
```

This variable caused the most recent conflict, so we may legitimately believe that branching on it may cause a conflict again (since not that much has changed).

Christophe Lecoutre, Lakhdar Saïs, Sébastien Tabary, and Vincent Vidal.
Reasoning from last conflict(s) in constraint programming.
Artificial Intelligence, 2009.

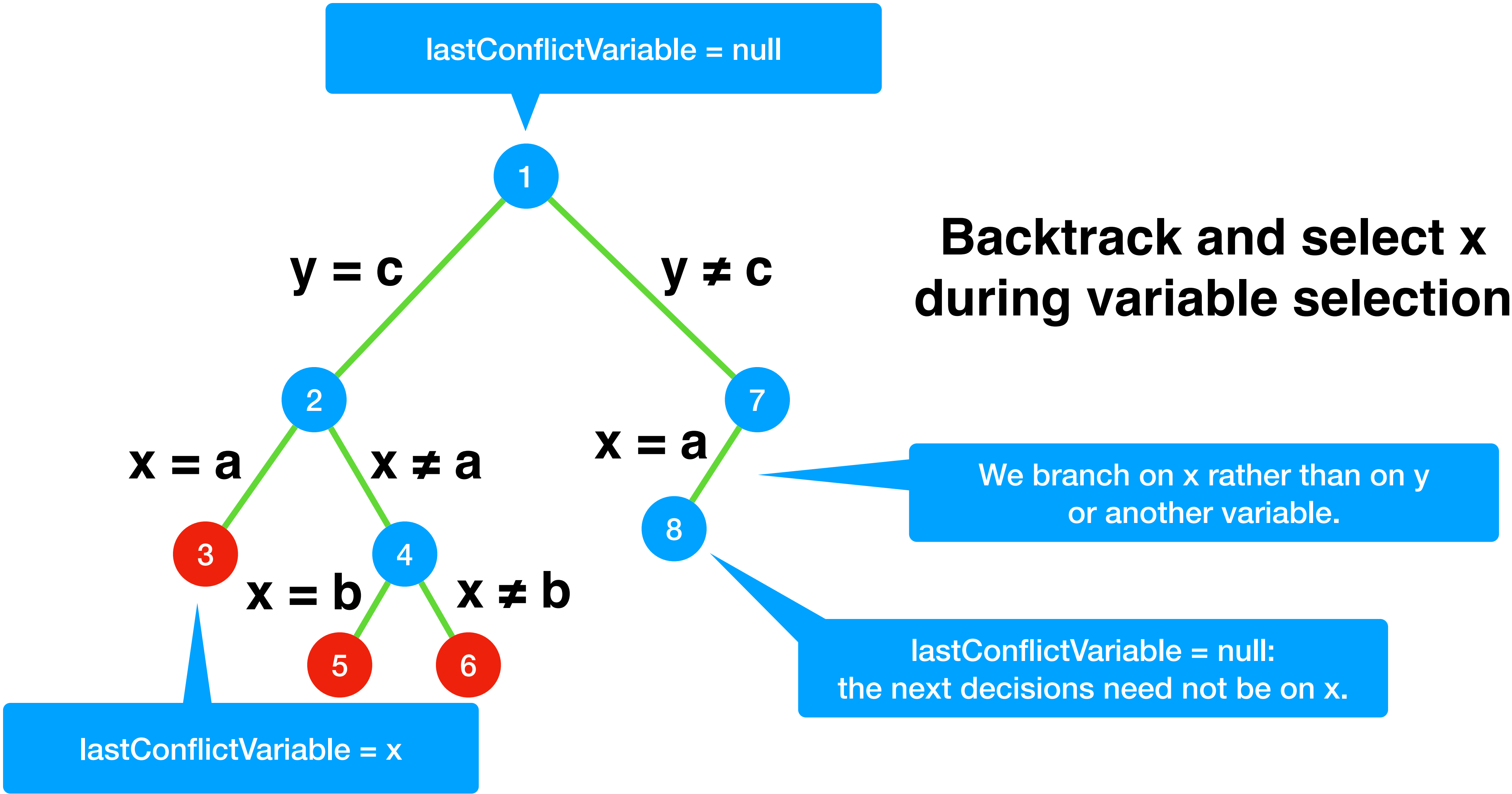
Last-Conflict Search

When branching on a variable leads to a failure,
always branch on this variable first, until it is successfully fixed.



Impossible to fix x_4 to some value. Why?
It may be in conflict with the choices on x_3 , x_2 , or x_1 .
Since we do not know, we should always try to branch on x_4 first,
also after backtracking, as long as this conflict persists.

Last-Conflict Search: Example



Conflict-Ordering Search (COS)

- Generalization and extension of last-conflict search:
 - Each branching decision is timestamped with a shared counter, which is incremented at each failed visited node of the search tree.
 - Each variable has a **timestamp**: it is the timestamp of the most recent node where a failure occurred when branching on this variable, and null if it caused no failure so far.
 - **COS**: Select the unfixed variable with the largest timestamp. If no unfixed variable has a failure timestamp yet, then use a fallback heuristic.

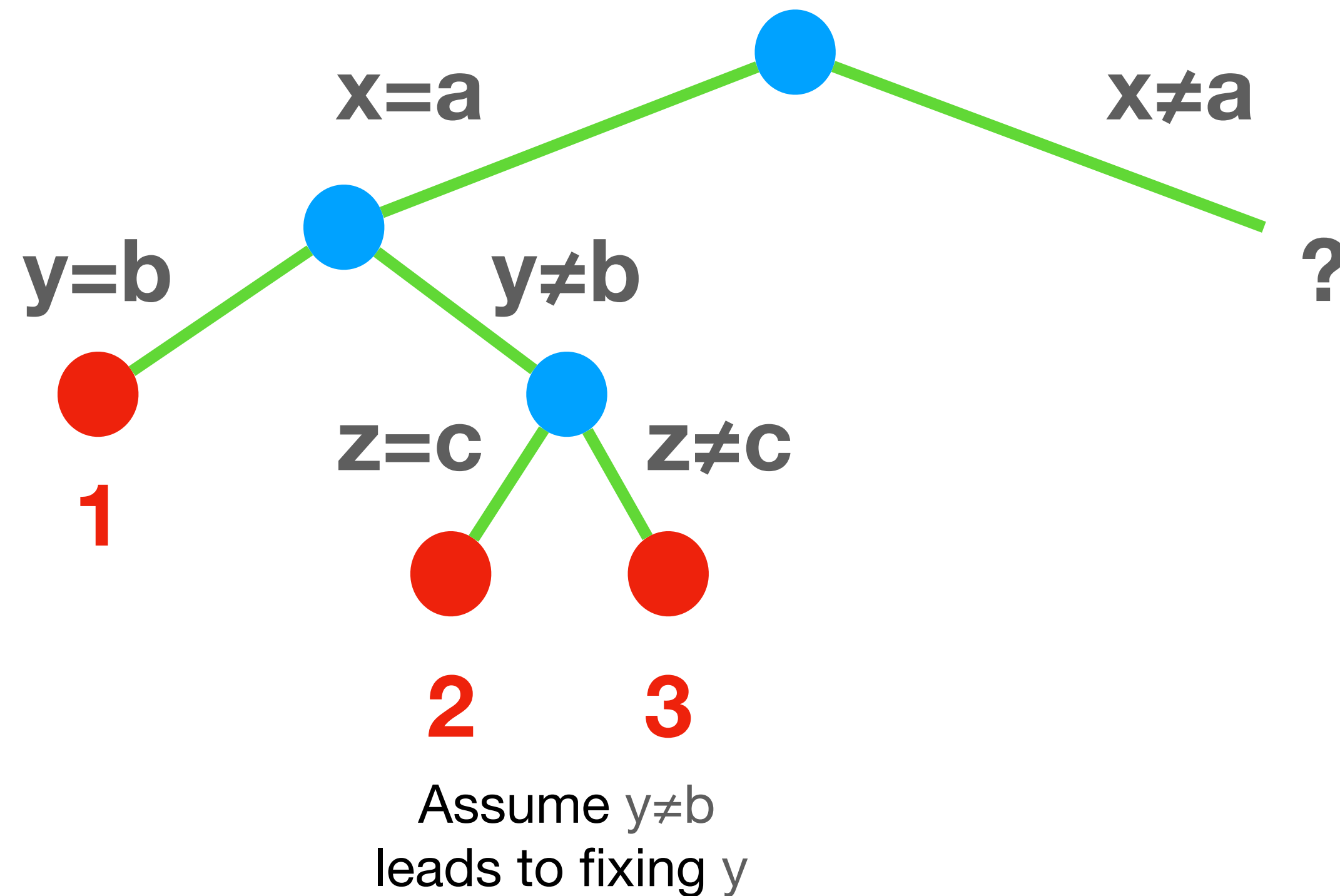
Steven Gay, Renaud Hartert, Christophe Lecoutre, and Pierre Schaus.

Conflict ordering search for scheduling problems.

International Conference on Principles and Practice of Constraint Programming, 2015.

Conflict-Ordering Search (COS)

- ▶ A **timestamp** is associated with each variable and is increased (to the node timestamp) at each failure it causes.
- ▶ **COS**: Select the unfixed variable with the largest timestamp.



Variable	x	y	z
Time-stamp	null	1	3

Conflict-Ordering Search (COS)

Algorithm 1: $COS(P = (\mathcal{X}, \mathcal{C}): \text{CSP})$

Output: true iff P is satisfiable

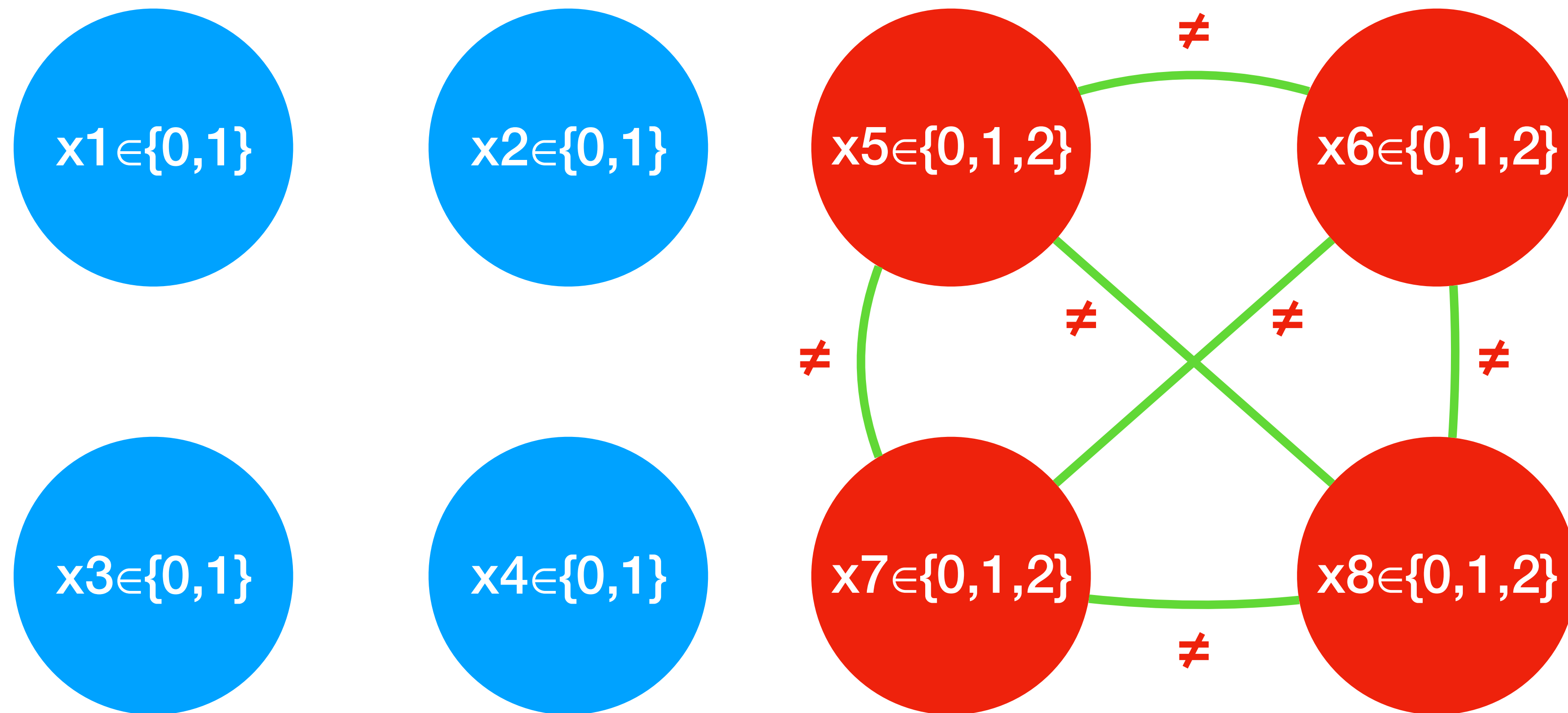
```
1  $P \leftarrow \phi(P)$ 
2 if  $P = \perp$  then
3   if  $\text{lastVar} \neq \text{null}$  then
4      $\text{nConflicts} \leftarrow \text{nConflicts} + 1$ 
5      $\text{stamp}[\text{lastVar}] \leftarrow \text{nConflicts}$ 
6   return false
7 if  $\forall x \in \mathcal{X}, |\text{dom}(x)| = 1$  then
8   return true
9  $\text{failed} \leftarrow \{x \in \mathcal{X} : \text{stamp}[x] > 0 \wedge |\text{dom}(x)| > 1\}$ 
10 if  $\text{failed} = \emptyset$  then
11    $\text{lastVar} \leftarrow \text{varHeuristic.select}()$ 
12 else
13    $\text{lastVar} \leftarrow \text{argmax}_{x \in \text{failed}} \{\text{stamp}[x]\}$ 
14  $v \leftarrow \text{valHeuristic}[\text{lastVar}].\text{select}()$ 
15 return  $COS(P|_{\text{lastVar} \leq v}) \vee COS(P|_{\text{lastVar} > v})$ 
```

Failure: update the timestamp.

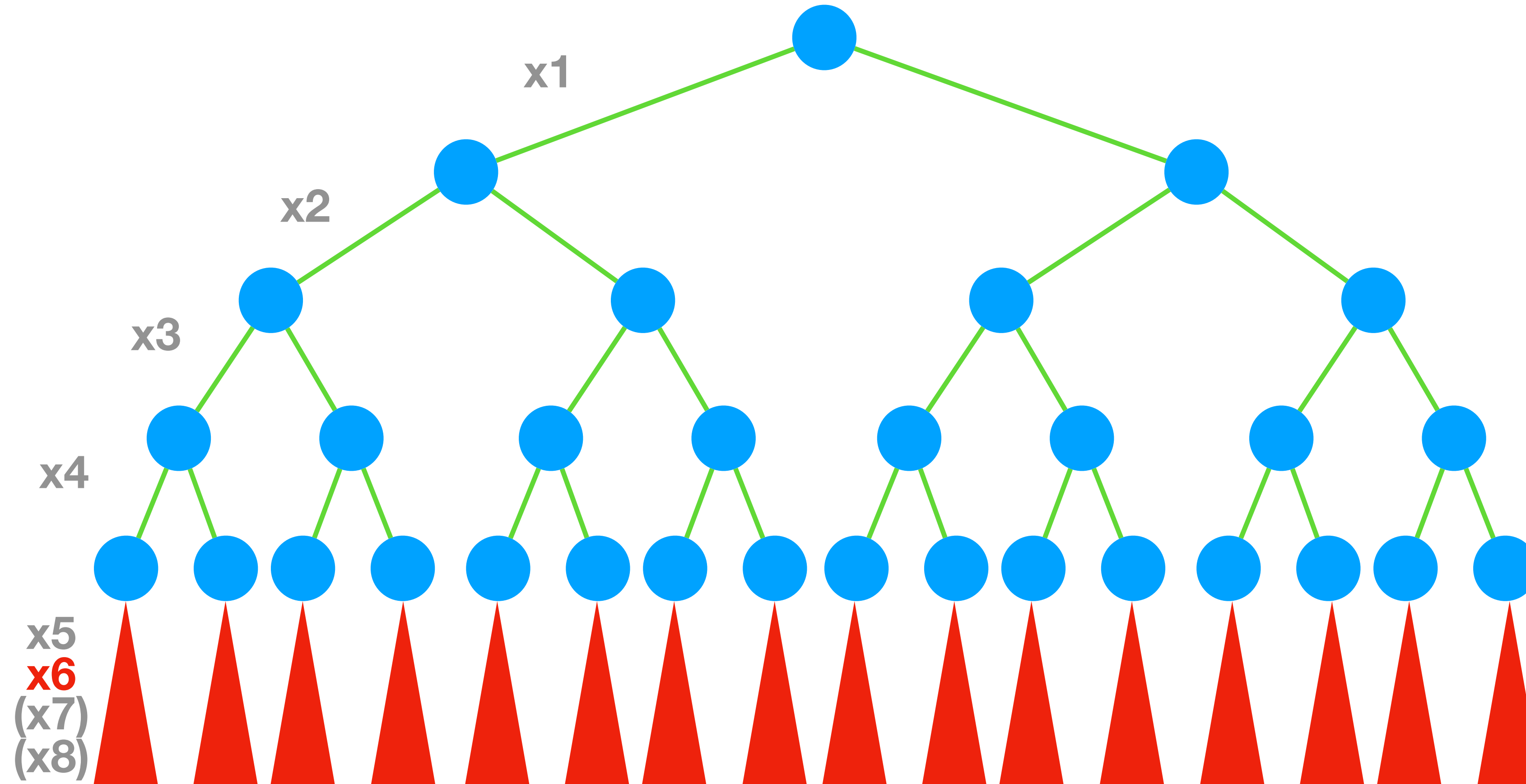
Failure: select the variable with the largest timestamp.

Example

Consider the CSP below, whose infeasibility is not detected by the fixpoint algorithm because of the AllDifferent decomposition. Assume we branch in the order $x_1, \dots, x_8$.

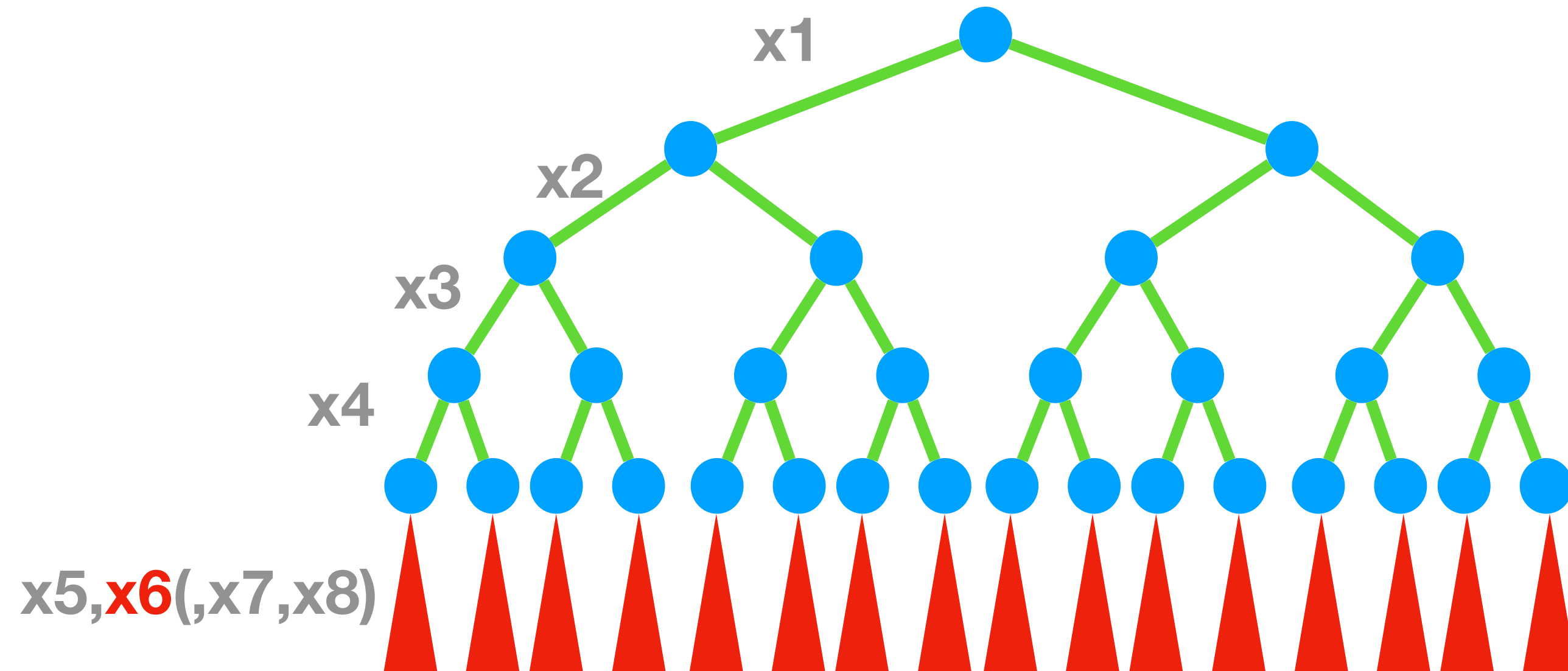


Example: Search Tree

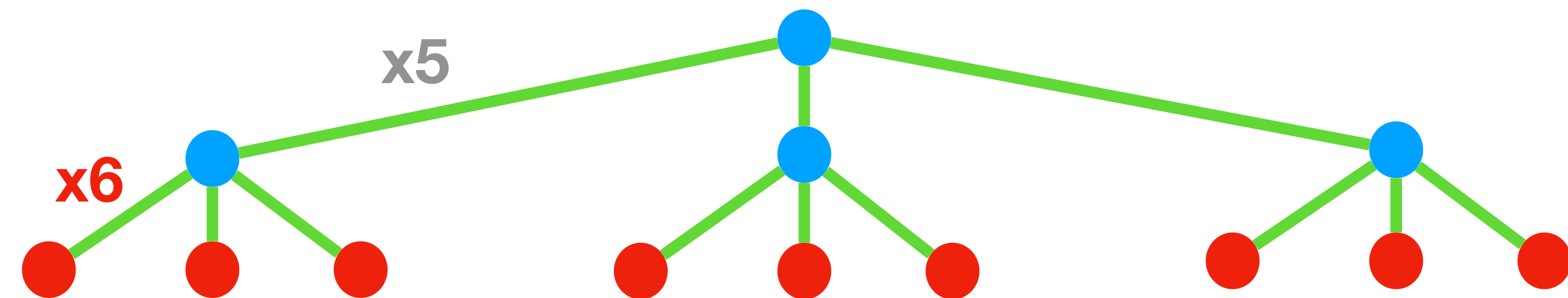


The problem is that there is a subset of variables, $\{x5, x6, x7, x8\}$, causing the conflict. Ideally we should detect this earlier in the search tree.

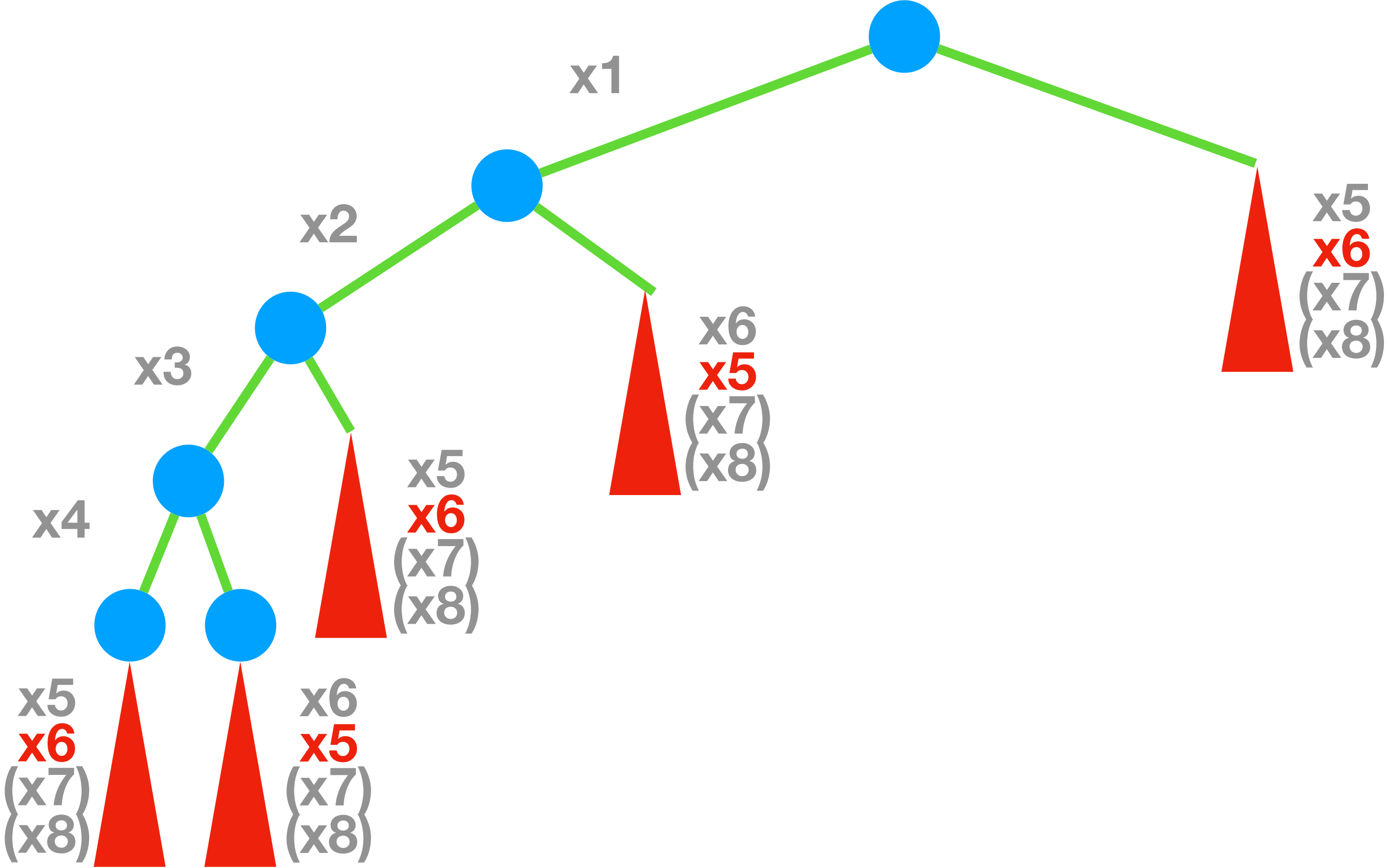
Example: Search Trees with Two Static Orders



Search tree obtained with the static order $[x_5, x_6, x_7, x_8, x_1, x_2, x_3, x_4]$: ideally, a heuristic learns this order.



Example: Search Tree with COS



Pick/Dom Variable Heuristic

Select the next variable with the smallest ration $\frac{pick(x)}{|D_x|}$

$pick(x)$ is an accumulated measure representing how much/how frequently a variable domain was reduced during the fix-point computation *when a conflict occurs (to me that is the main difference with activity-based search).*

Audemard, G., Lecoutre, C., & Prud'Homme, C. Guiding Backtrack Search by Tracking Variables During Constraint Propagation. CP 2023

Pick/Dom Heuristic: Instrumenting the fix-point algo

Data: The CSP $\langle X, \mathcal{D}^0, C \rangle$

Result: The greatest fixpoint domain

$Q \leftarrow C$

$\mathcal{D} \leftarrow \mathcal{D}^0$

$\delta(x) \leftarrow 0 \ \forall x \in X$

while $|Q| > 0$ **do**

$c = \text{dequeue}(Q)$

$\mathcal{D}' \leftarrow \mathcal{F}_c(\mathcal{D})$

$V \leftarrow \{x \in \text{Vars}(c) : \mathcal{D}'(x) \neq \mathcal{D}(x)\}$

if $|V| > 0$ **then**

if $\exists x \in V : \mathcal{D}'(x) = \emptyset$ **then**

$\text{pick}(x) \leftarrow \text{pick}(x) + \delta(x) \ , \ \forall x \in V$
 return failure

end

$\delta(x) \leftarrow \delta(x) + 1 \ \forall x \in V$

$Q \leftarrow Q \cup \{c : |\text{Vars}(c) \cap V| > 0\}$

$\text{enqueue}(Q, \{c' : |\text{Vars}(c') \cap V| > 0\})$

end

$\mathcal{D} \leftarrow \mathcal{D}'$

end

Accumulates the number of times the domain of x has changed during the fix-point

If there is a failure during the fix-point, we add the counter attached to each variable that counted the number of times the domain of the variable was shrunk (there is a variant that counts the number values removed).

Failure Rate Based Based Search

- ▶ $FRB(A)/dom$ selects in priority the variables that most often lead to a conflict when assigning them.
- ▶ Two counters:
 - $\#Conflicts(x)$ is the number of times a failure has been observed by propagating an assignment involving x
 - $\#Assigns(x)$ is number of times the variable x was assigned

Then $FR(x) = \frac{\#Conflicts(x)}{\#Assigns(x)}$ is the failure rate.

Select next variable with the the highest $FRB(x) = \frac{FR(x)}{|D_x|}$

Higher the failure rate & smaller domain = First-fail principle

Failure Rate Based Search (1/2)

- Improved Idea: Bias the selection to variables with more recent failures
- $\#TotalFailures$ = total failure since the beginning
- $LastFailure(x)$ is the value of $\#TotalFailures$ when the last failure caused by x occurred.

$$A(x) = \frac{1}{\#TotalFailures - LastFailure(x) + 1}$$

- $0 < A(x) \leq 1$.
 - $A(x) = 1$ then the variable x caused the more recent failure.
 - $A(x) < 1$ it has been a while since x didn't cause a failure (the smaller the longer).

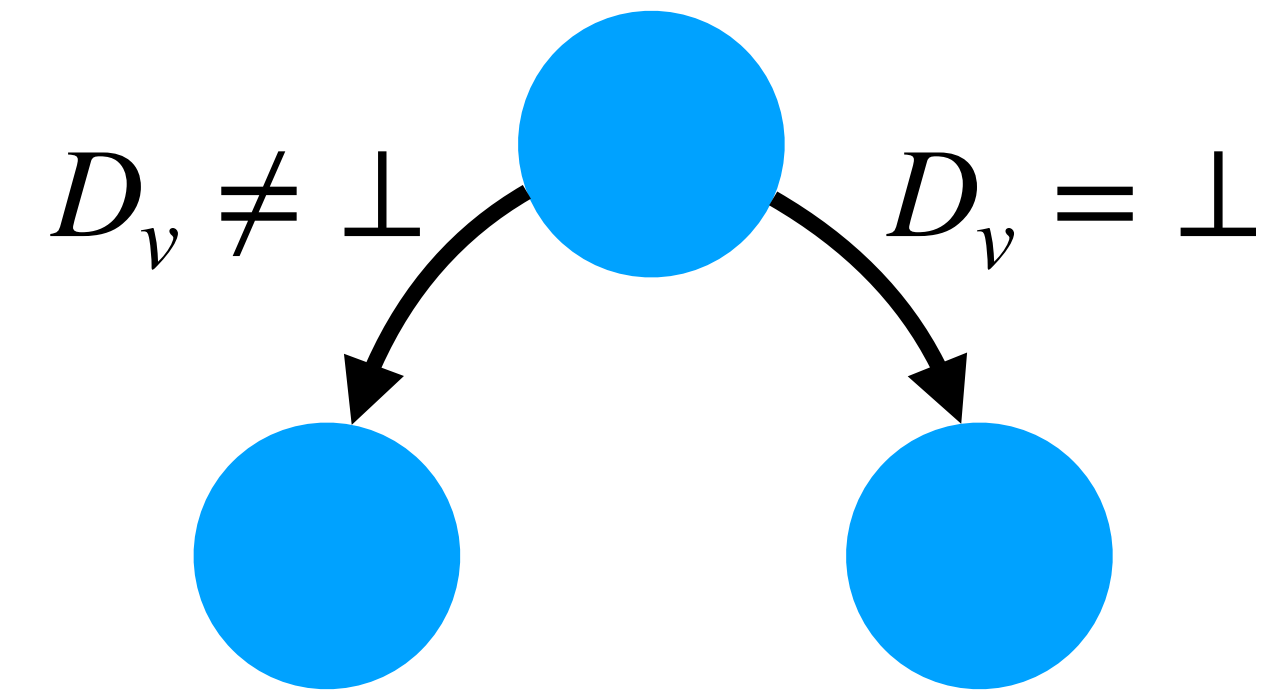
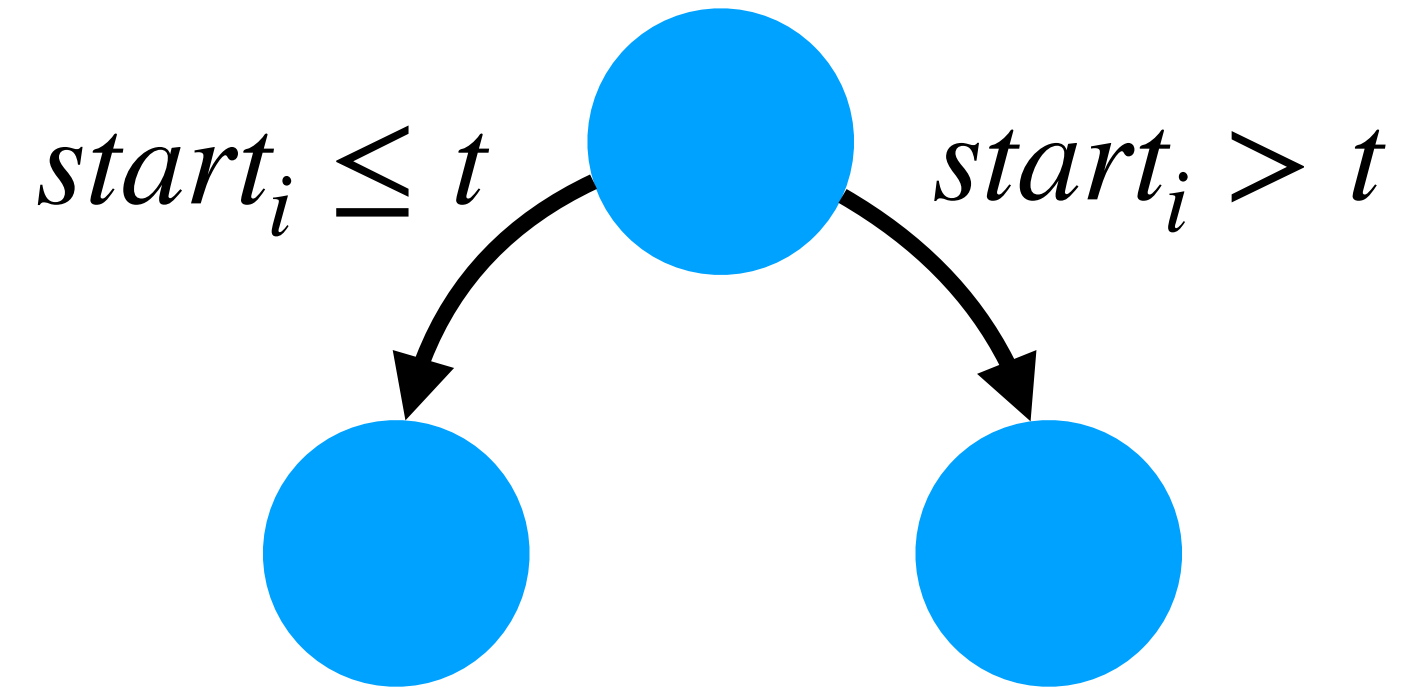
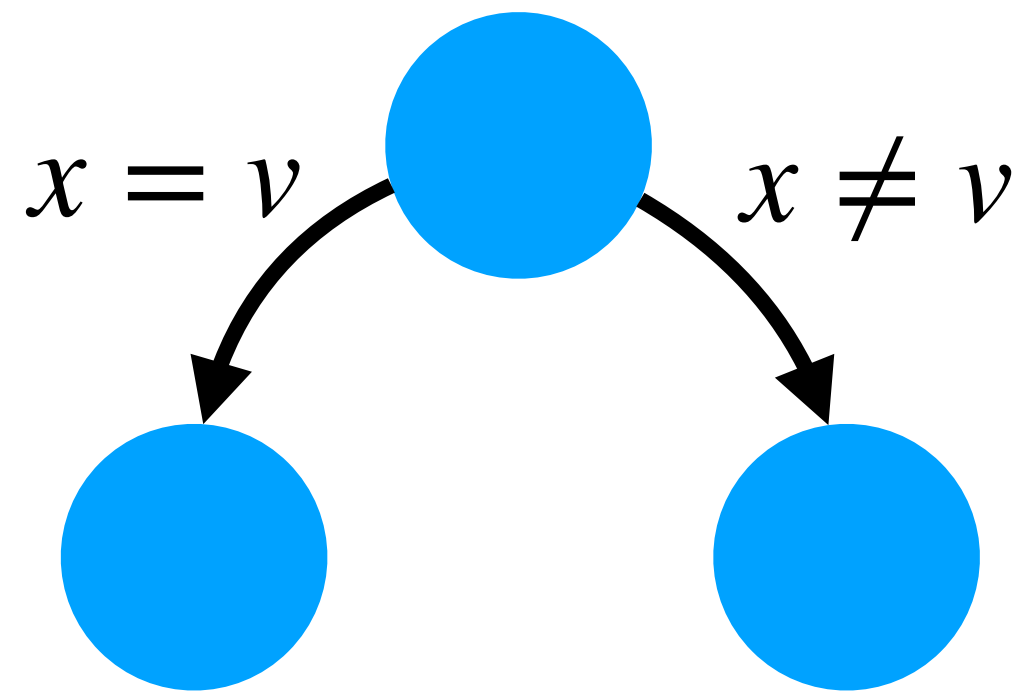
Select next variable with the highest $FRBA(x) = \frac{FR(x) + A(x)}{|D_x|}$

Failure Directed Search

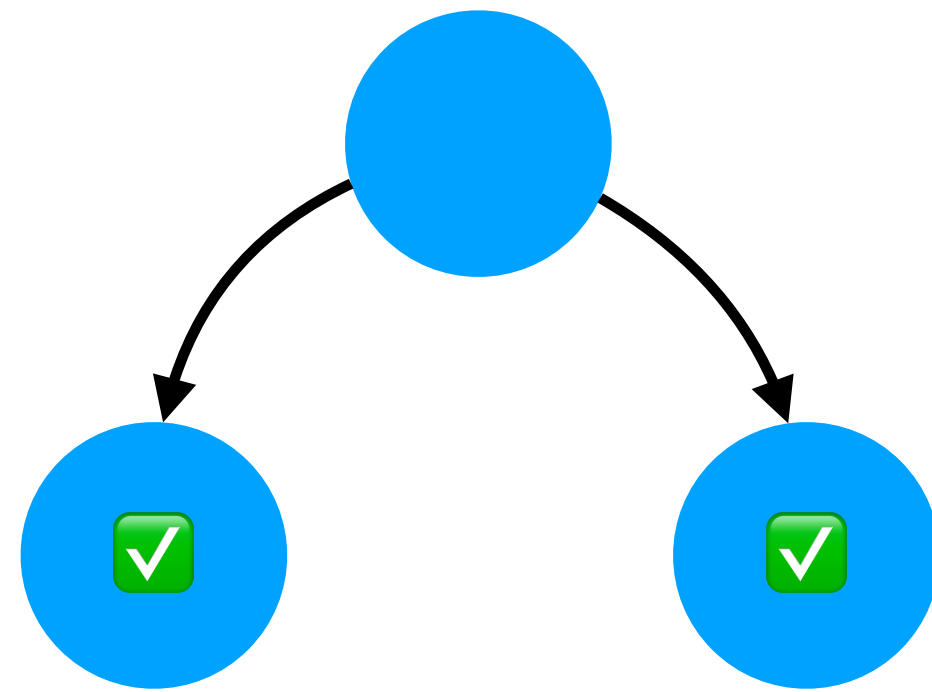
A search aimed at proving optimality

- ▶ You believe there is no better solution, you want to prove there is not more solution
- ▶ Instead of looking for a feasible solution, we try to close the branches as early as possible.
- ▶ Failure directed search was conceived and successful for proving optimality of scheduling problems (Jobshop, RCPSP, etc)

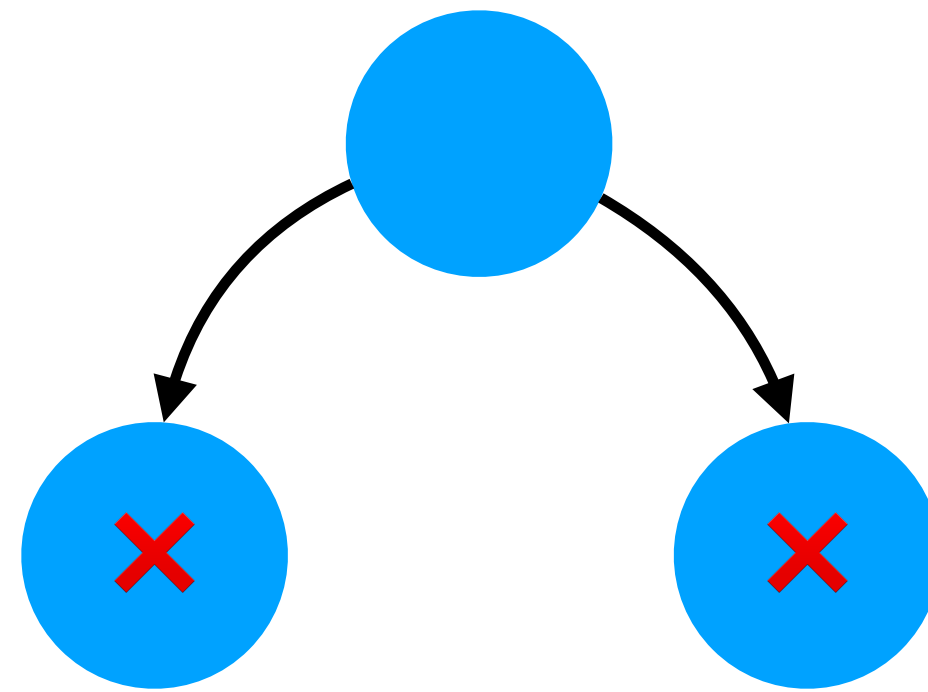
- Binary choice **c**, examples:



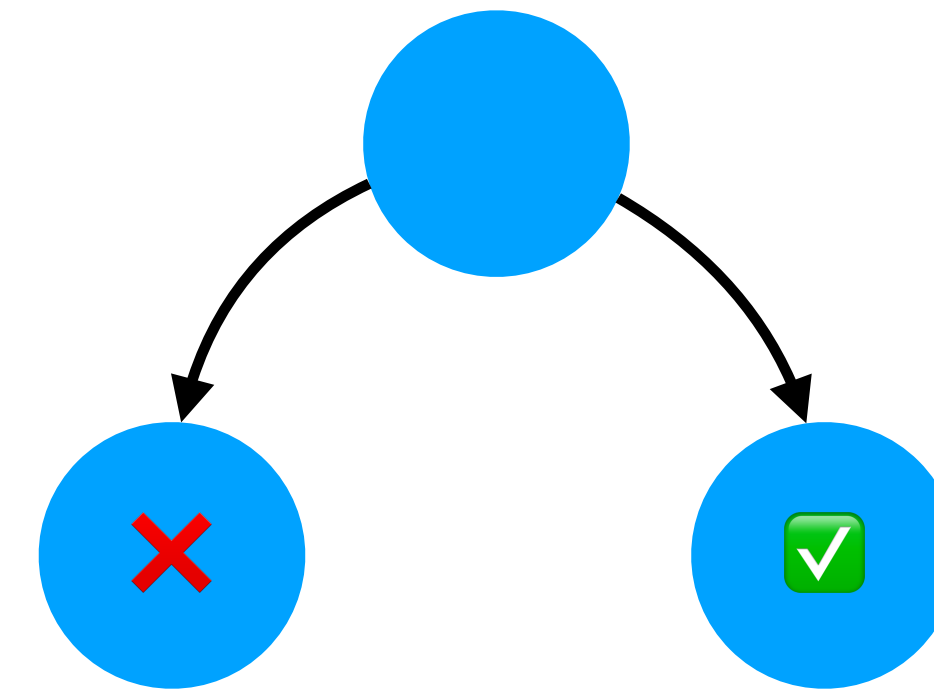
- Two branches for a choice c^+ , c^- , which one is left or right is fixed. All the possible choices are generated apriori
- Each choice has a $rating[c] = rating[c^+] + rating[c^-]$, the lower the rating the higher is the priority for selecting this choice



Zero fail



Two fails



One fail

Rating update

$$R = \frac{|D'_{x_1}| \times \dots \times |D'_{x_n}|}{|D_{x_1}| \times \dots \times |D_{x_n}|} \text{ reduction of search space a decision choice.}$$

$$fail = 0 \leq R \leq 1 = nochange$$

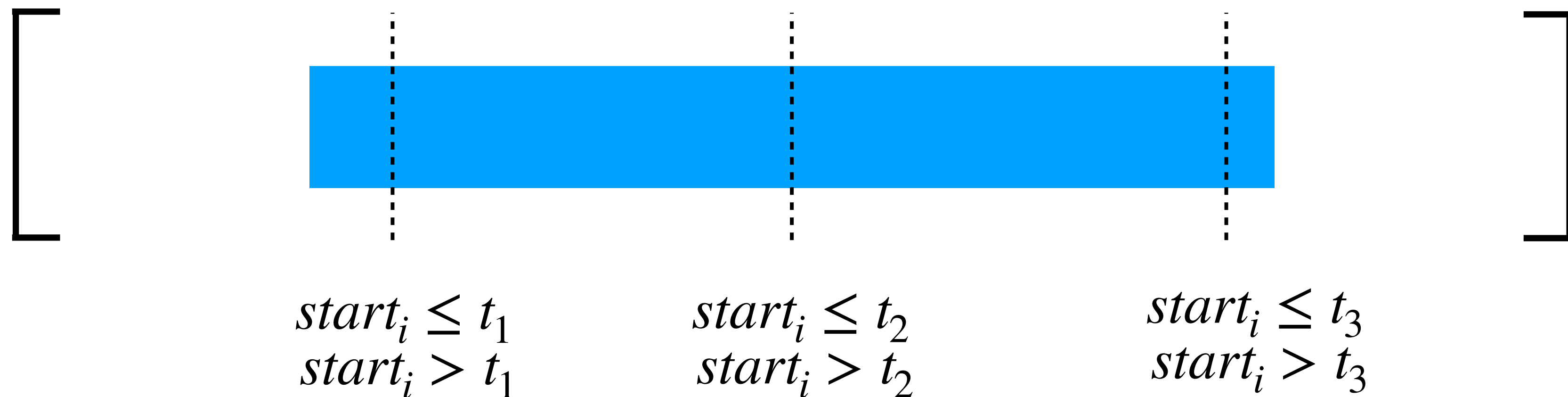
$$localRating := \begin{cases} 0, & \text{if the branch fails immediately} \\ 1 + R, & \text{otherwise.} \end{cases}$$

$$rating[c^{+/-}] = \alpha \cdot rating[c^{+/-}] + (1 - \alpha) \cdot \frac{localRating}{avgRating[d]}$$

0.9

Avg rating at depth d

- ▶ Failure-Directed requires that all the possible choices have been generated in advance (there might be a lot, especially if many points, like in scheduling).
- ▶ Idea: only generate a few of them to generate mandatory parts (good for propagation). If the search along a branch has exhausted the choice points, continue with a traditional DFS.



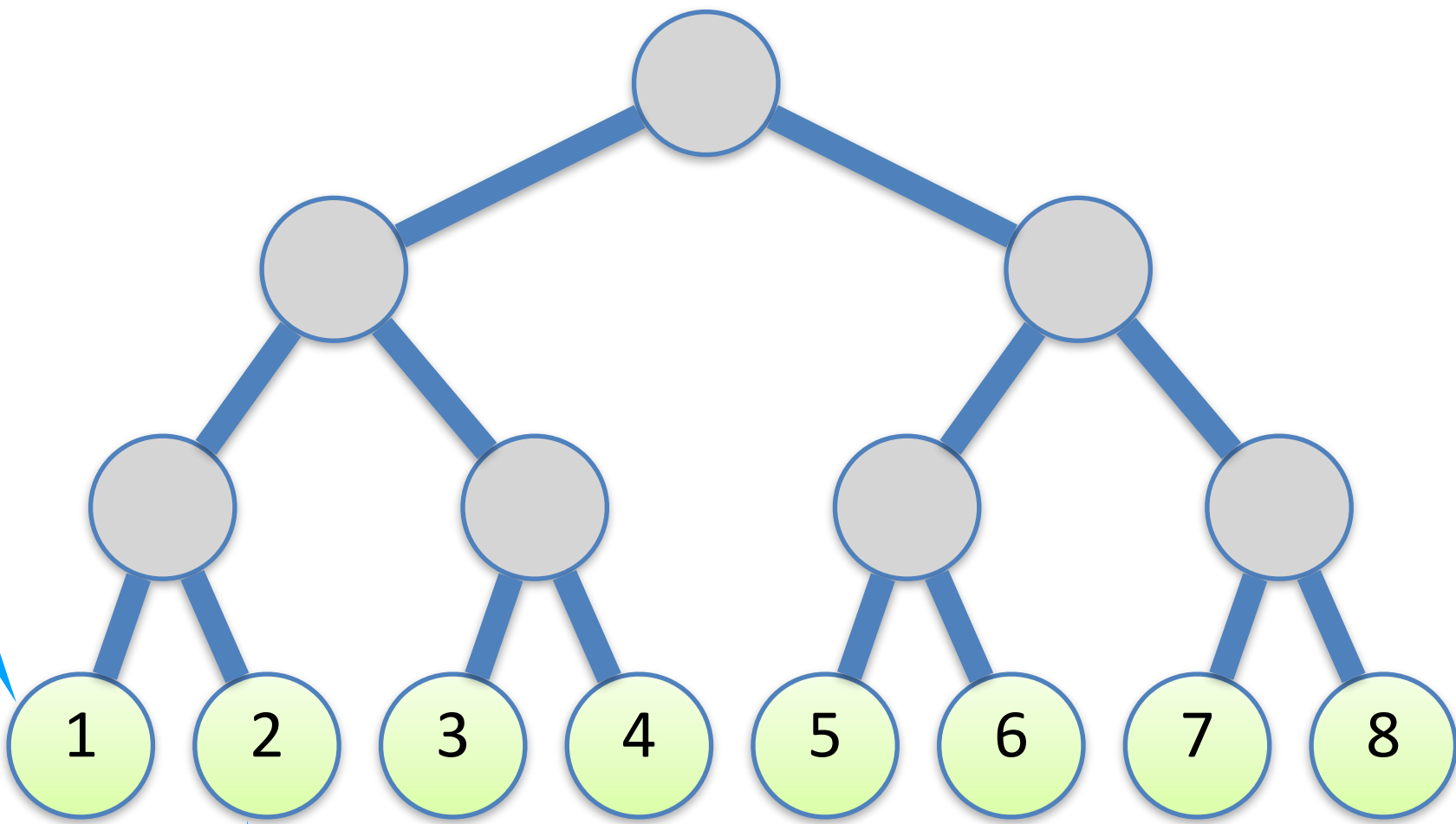
Discrepancy Search

- Very useful if you trust your value heuristic
- Search combinator $\perp$ to the other techniques

Question: Assume I have a good heuristic



Then this solution
should be really good.



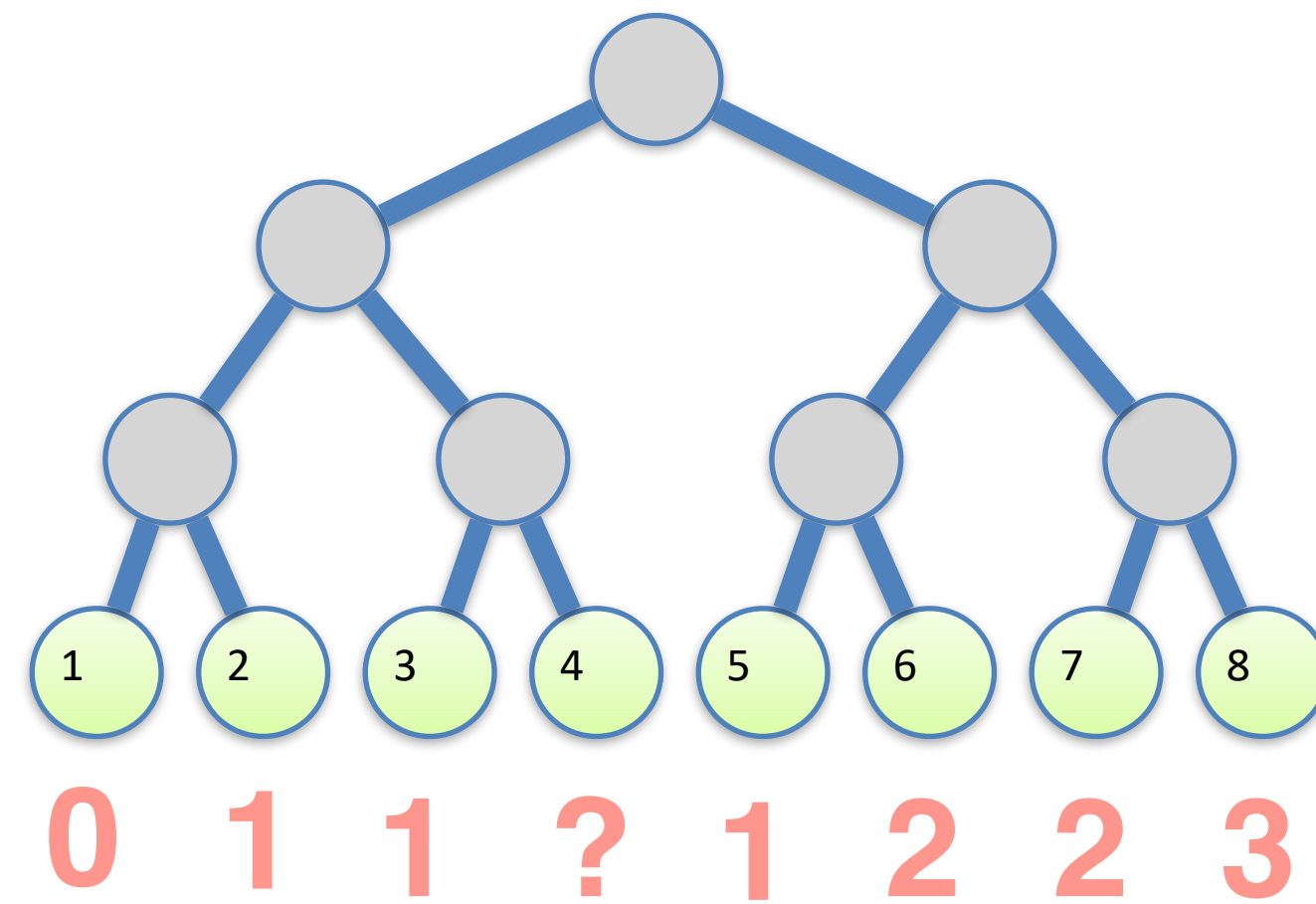
But is this one
supposed to be
better than...

...this one?

Classify solutions according to
the mistakes of the heuristic!

Discrepancy-Based Search

- **Discrepancy** = the number of right decisions.



Given that:

- I trust my heuristic
- Wrong decisions generally occur at early stages

We should visit solutions by increasing discrepancy.

Combinator: Iterative Discrepancy

```
Objective obj = cp.minimize(totCost);

for (int dL = 0; dL < x.length; dL++) {
    DFSearch dfs = makeDfs(cp, limitedDiscrepancy(firstFail(x),dL));
    dfs.optimize(obj);
}
```

limitedDiscrepancy wraps the search:
it is called a *search combinator*.

Implementation = Exercise

```
public class LimitedDiscrepancyBranching implements Supplier<Procedure[]> {

    private int curD;
    private final int maxD;
    private final Supplier<Procedure[]> bs;

    public LimitedDiscrepancyBranching(Supplier<Procedure[]> branching,
                                       int maxDiscrepancy) {
        if (maxDiscrepancy < 0)
            throw new IllegalArgumentException("max discrepancy should be >= 0");
        this.bs = branching;
        this.maxD = maxDiscrepancy;
    }

    @Override
    public Procedure[] get() {
        // TODO
        throw new NotImplementedException();
    }
}
```

- Eliminate alternatives that would exceed maxD
- Wrap each alternative (closure) such that the call method of the wrapped alternatives:
 - Augments curD depending on its position
 - +0 for alts[0], ..., +i for alts[i]

Restarting

How to take advantage of anything learned...

Restarting Idea

► Two observations

1. Early mistakes (at shallow depths in the tree) are very costly

- You need to backtrack all the way up to revise those.

2. When the variable / value selection **has a learning component...**

- Its behavior would be different from the start with the learned signal in tow!

► Corollary

- Give yourself a chance and start *all over again*
- With the wiser heuristic
- And a restarting schedule with a limit (e.g. on # of failures) such as Luby

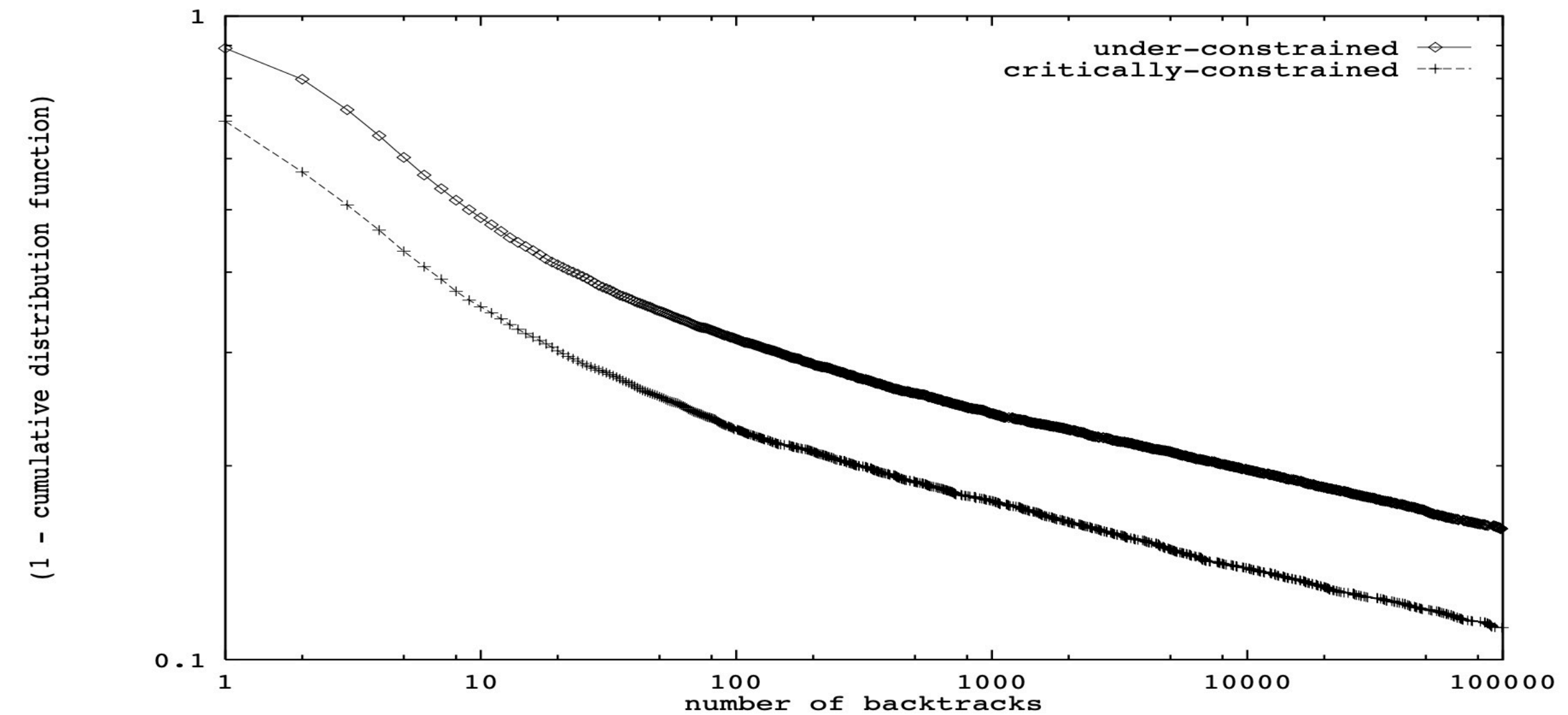
$$S_{F_{lim}} = (1, 1, 2, \textcolor{red}{1}, \textcolor{red}{1}, \textcolor{red}{2}, \textcolor{red}{4}, \textcolor{green}{1}, \textcolor{green}{1}, \textcolor{green}{2}, \textcolor{green}{1}, \textcolor{green}{1}, \textcolor{green}{2}, \textcolor{green}{4}, \textcolor{green}{1}, \textcolor{green}{1}, \textcolor{green}{2}, \textcolor{green}{4}, \textcolor{green}{8}, \textcolor{blue}{1}, \dots)$$

Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. Inf. Process. Lett. 47(4) (1993) 173–180

- ▶ If
 - You do not have learning
 - But you have randomization in the heuristic(s)
- ▶ That works too!

Heavy Tails

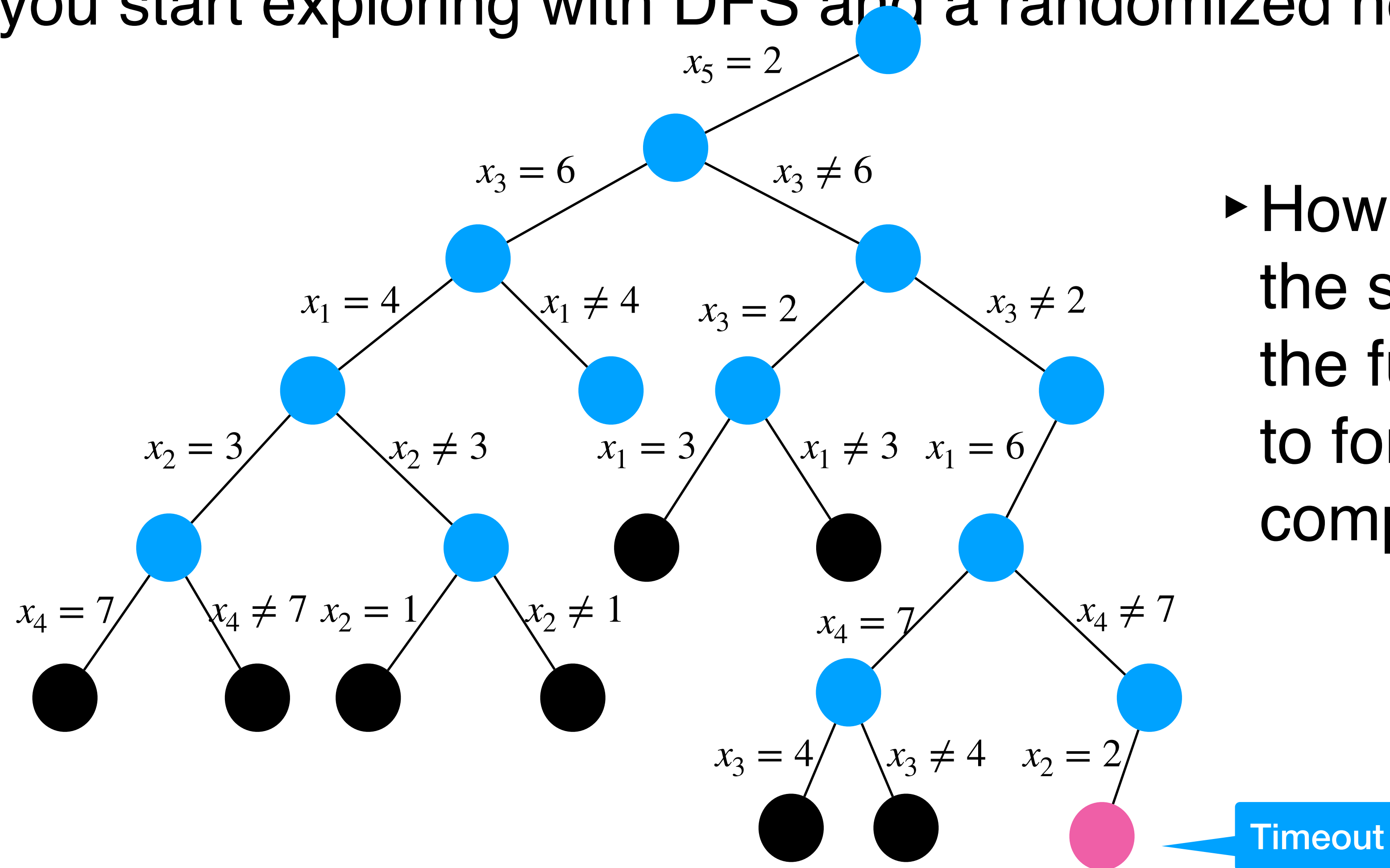
- What is that?
 - Distribution of **runtimes** of a search process
 - The distribution has “infinite” moments (mean, variance, etc.)
 - Implication
 - Some runs are *very long*
 - Some runs can be quite short
 - Take advantage with restarts when the search is too long



Heavy-tailed distributions in combinatorial search
Principles and Practice of Constraint Programming-CP97, 1997, Volume 1330
Carla P. Gomes, Bart Selman, Nuno Crato

Learning from restart, possible ?

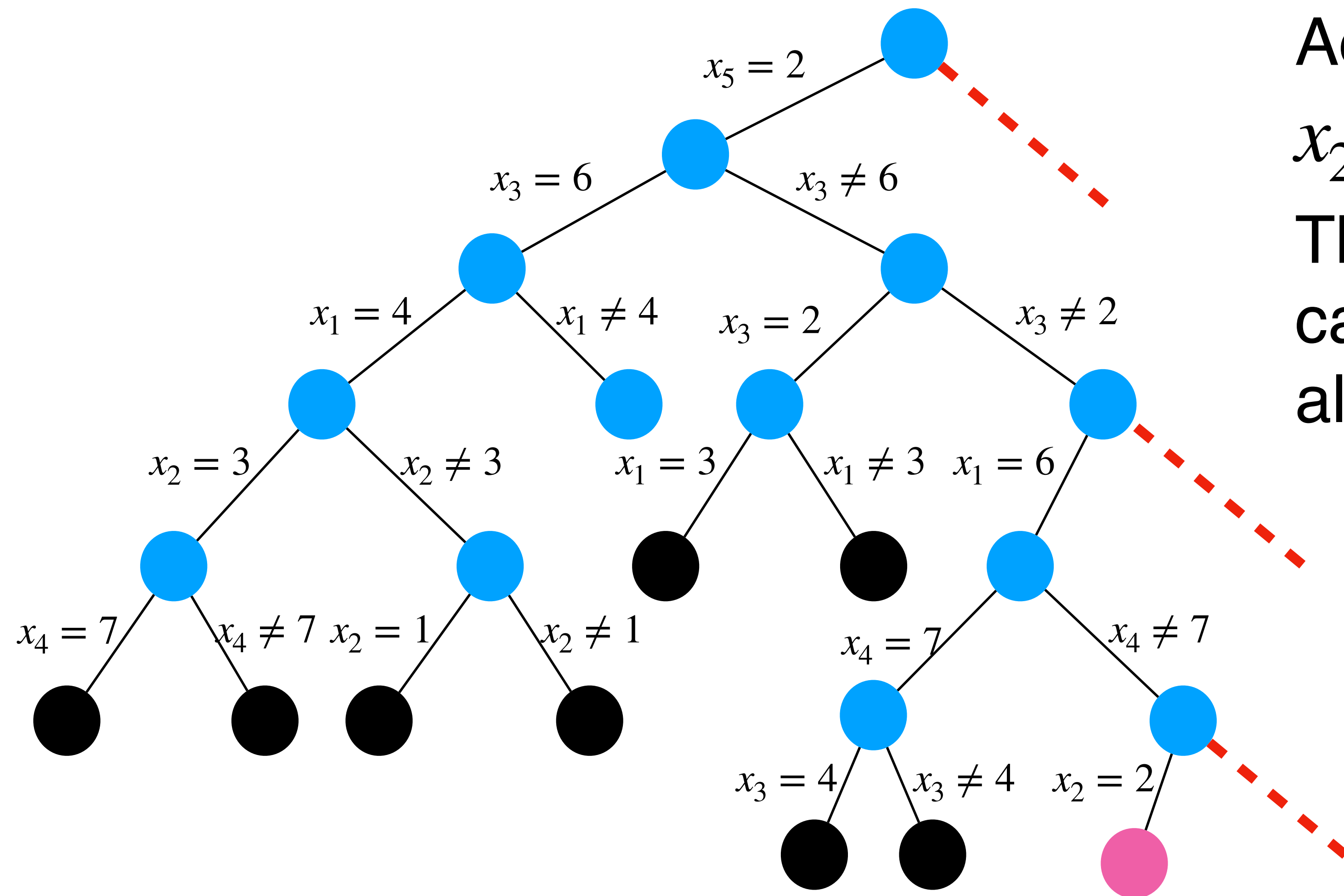
- Imagine that you have set a time-out (or a limit on the number of backtracks) and you start exploring with DFS and a randomized heuristic.



- How to avoid re-exploring the same search space in the future ? Are we doomed to forget the past completely ?

No goods

- Are we doomed to forget the past completely ? **NO!**



Add the constraint

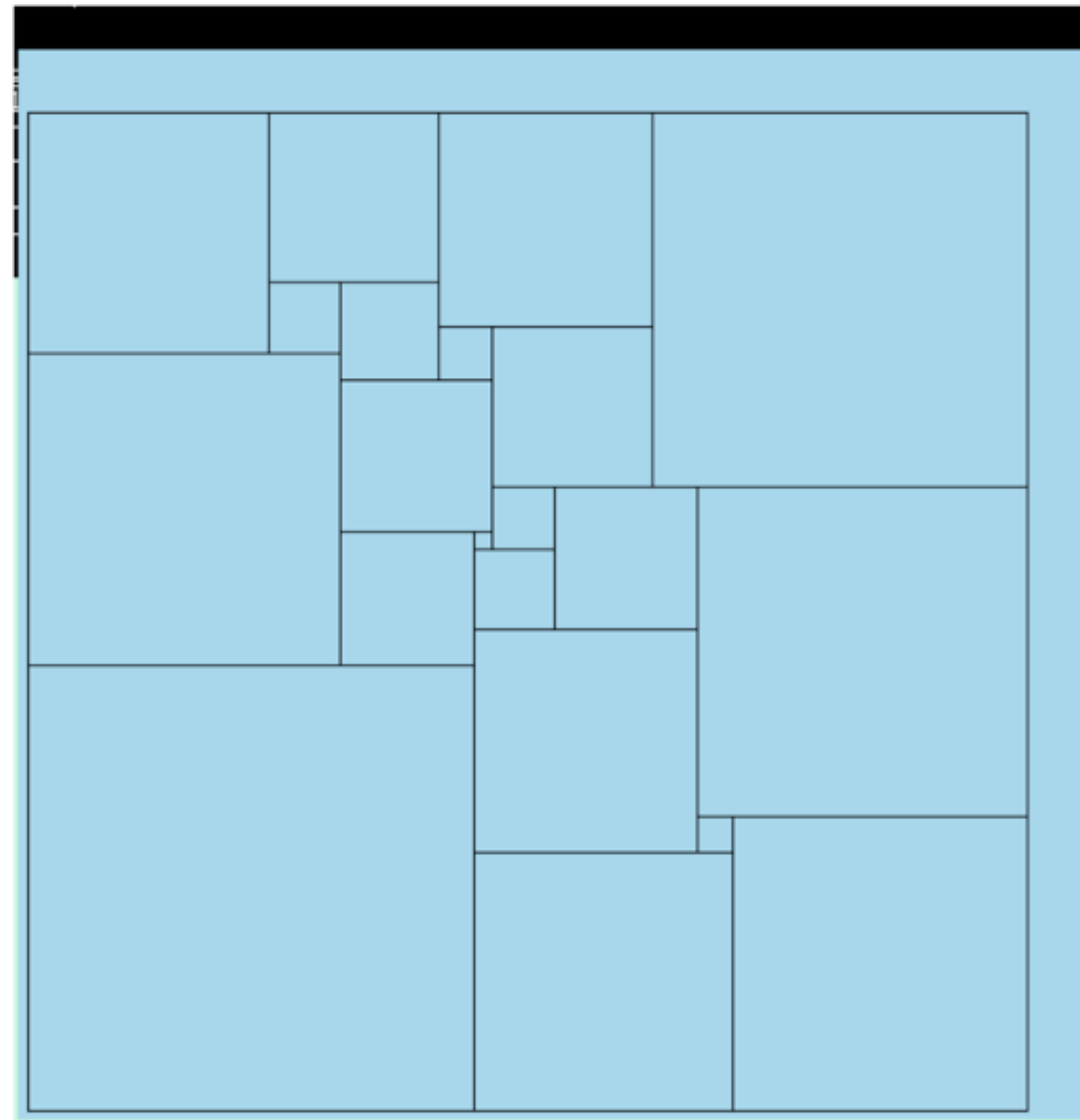
$$x_2 \neq 2 \vee x_1 \neq 6 \vee x_5 \neq 2$$

This is called a No-good. We can accumulate no-goods along the restarts

Perfect Square Packing

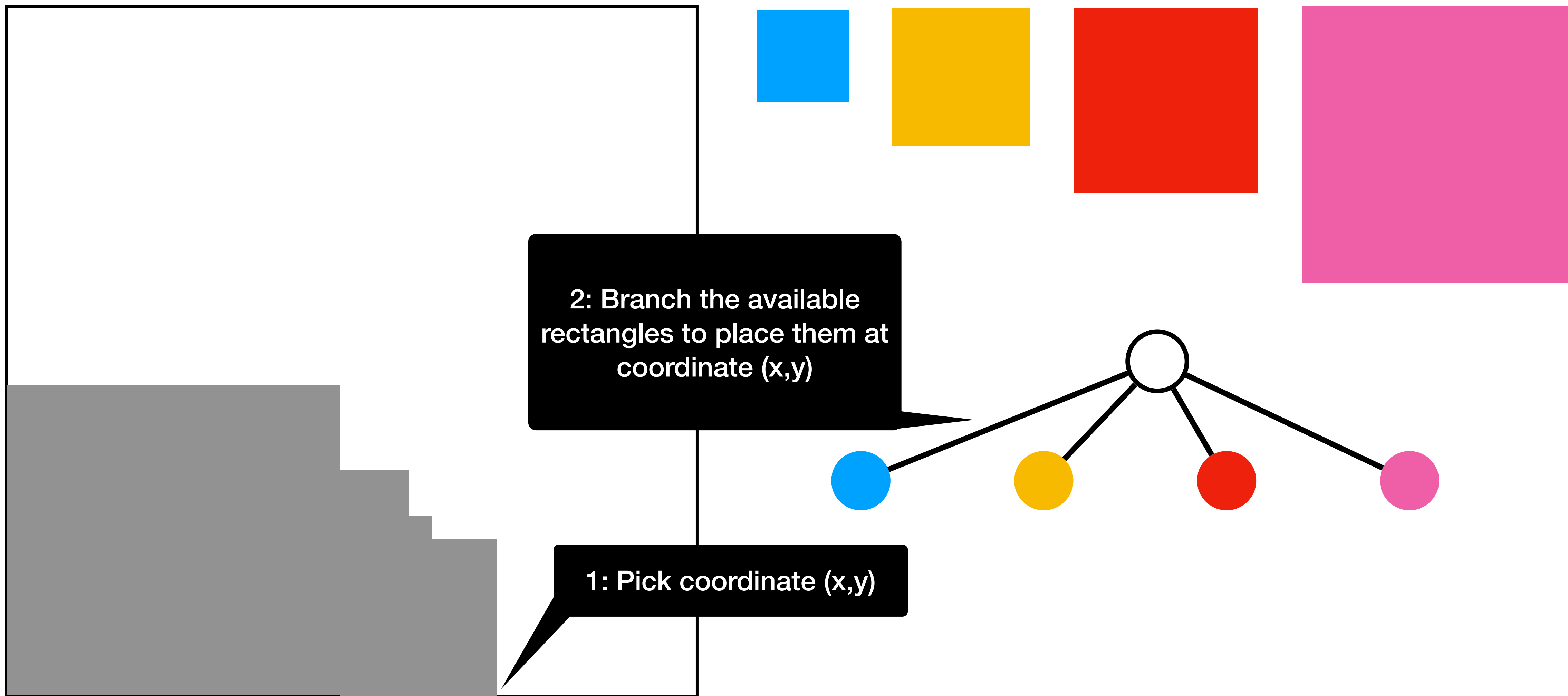
Place squares of different sizes into a big squares such that they do not overlap

```
int s = 112;
int [] side = new int[] {50, 42, 37, 35, 33, 29, 27, 25, 24, 19, 18, 17, 16, 15, 11, 9, 8, 7, 6, 4, 2};
```



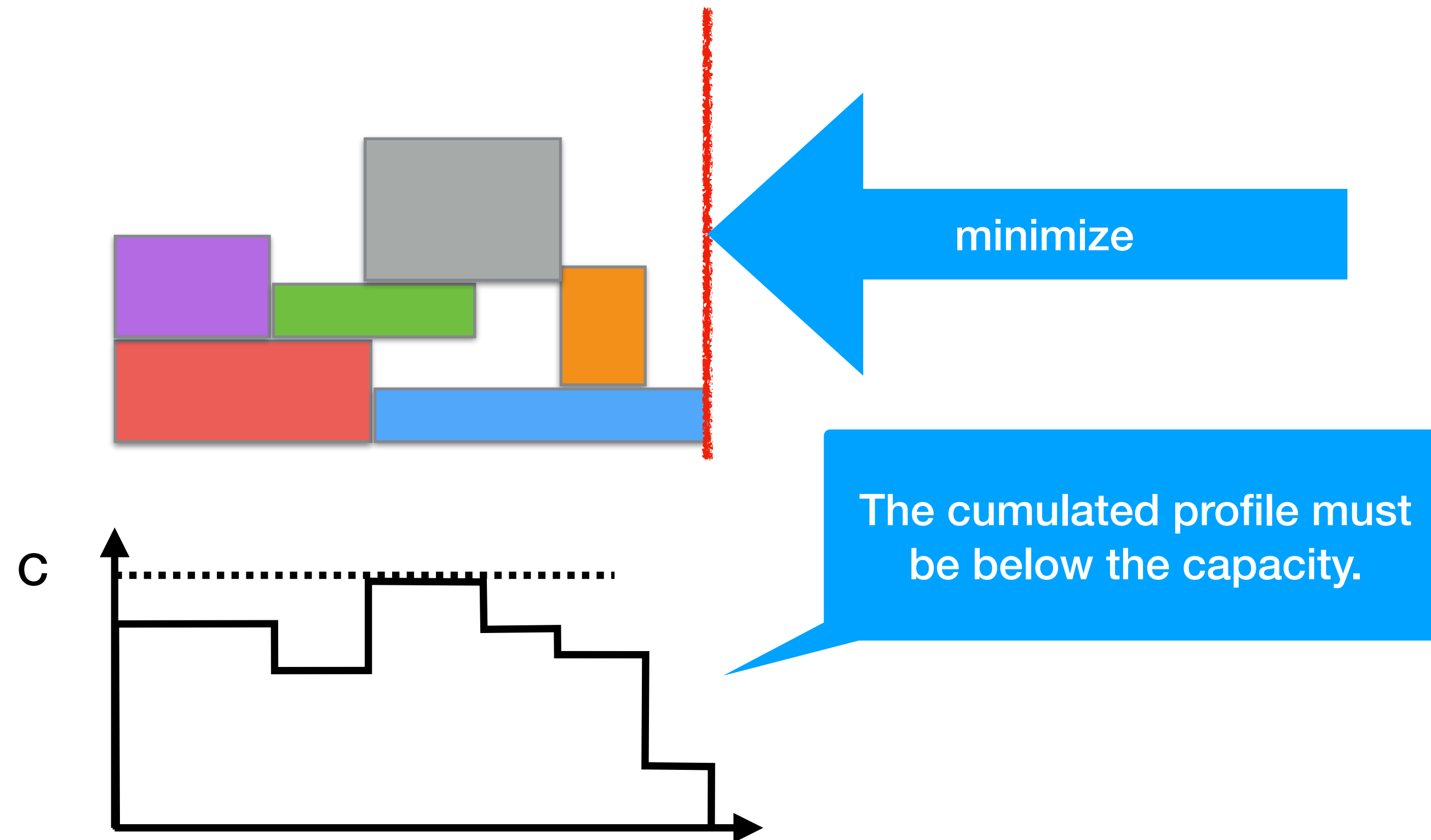
21 rectangles

Perfect Square Packing



Resource Constrained Scheduling Problem

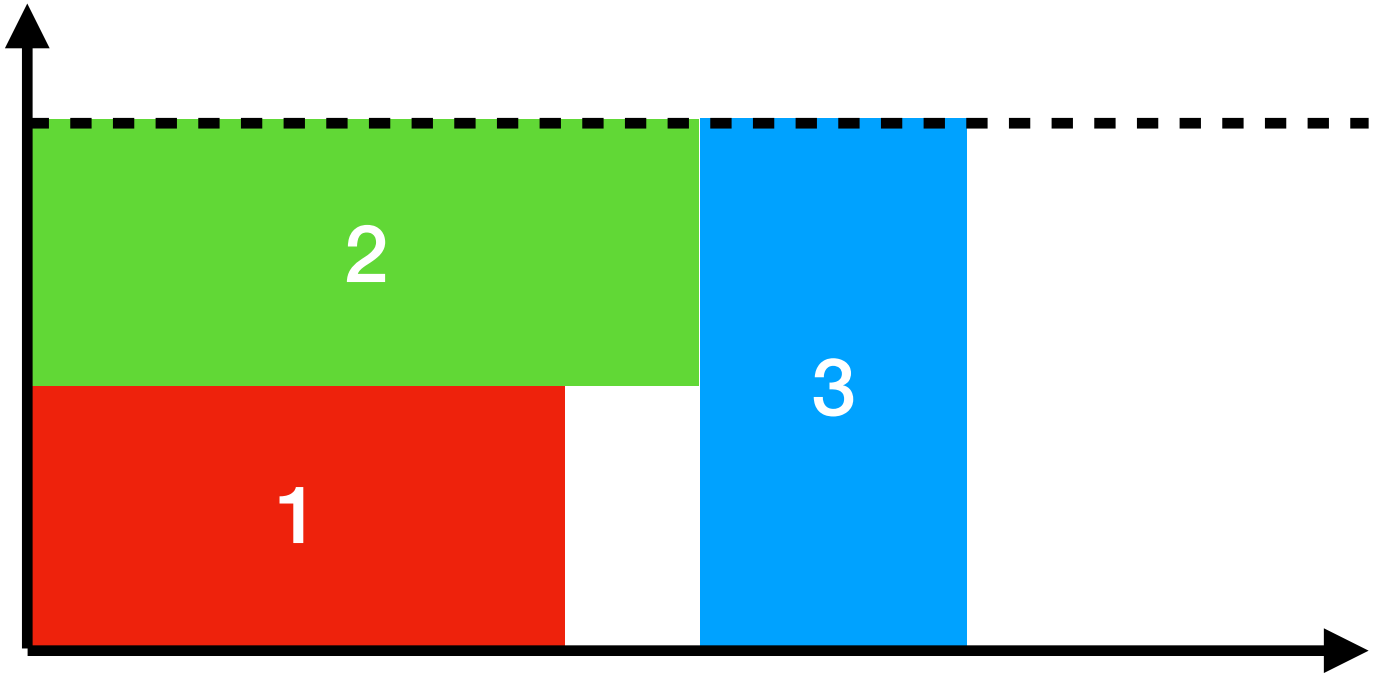
- Color = resource (or: machine), with capacity 1.
- Precedence constraints (denoted $\ll$) on the activities of a job.



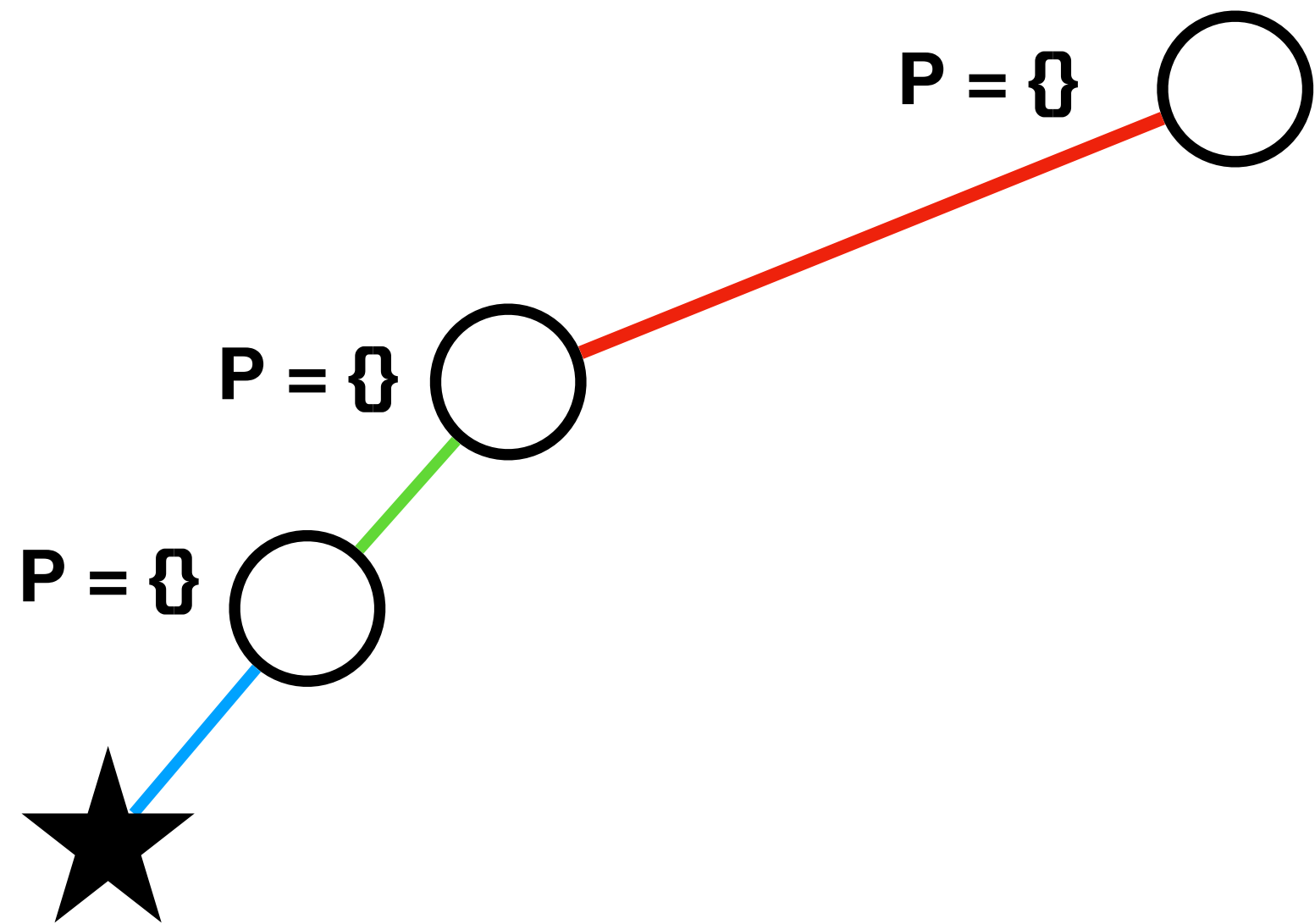
- ▶ Pick the unfixed task i , that is not postponed, with smallest possible start time (est_i)
- ▶ On the left branch: Assign the start of the task to est_i
- ▶ On the right branch: Set the task *postponed*. It will remain so as long as its est_i does not change
- ▶ If we detect that a task would be postponed forever, or if all the unfixed tasks are postponed, then **fail** and backtrack

⚠ complete search for makespan minimization, and simple constraints, like precedences (not if you have $start(A) - start(B) \leq 3$).

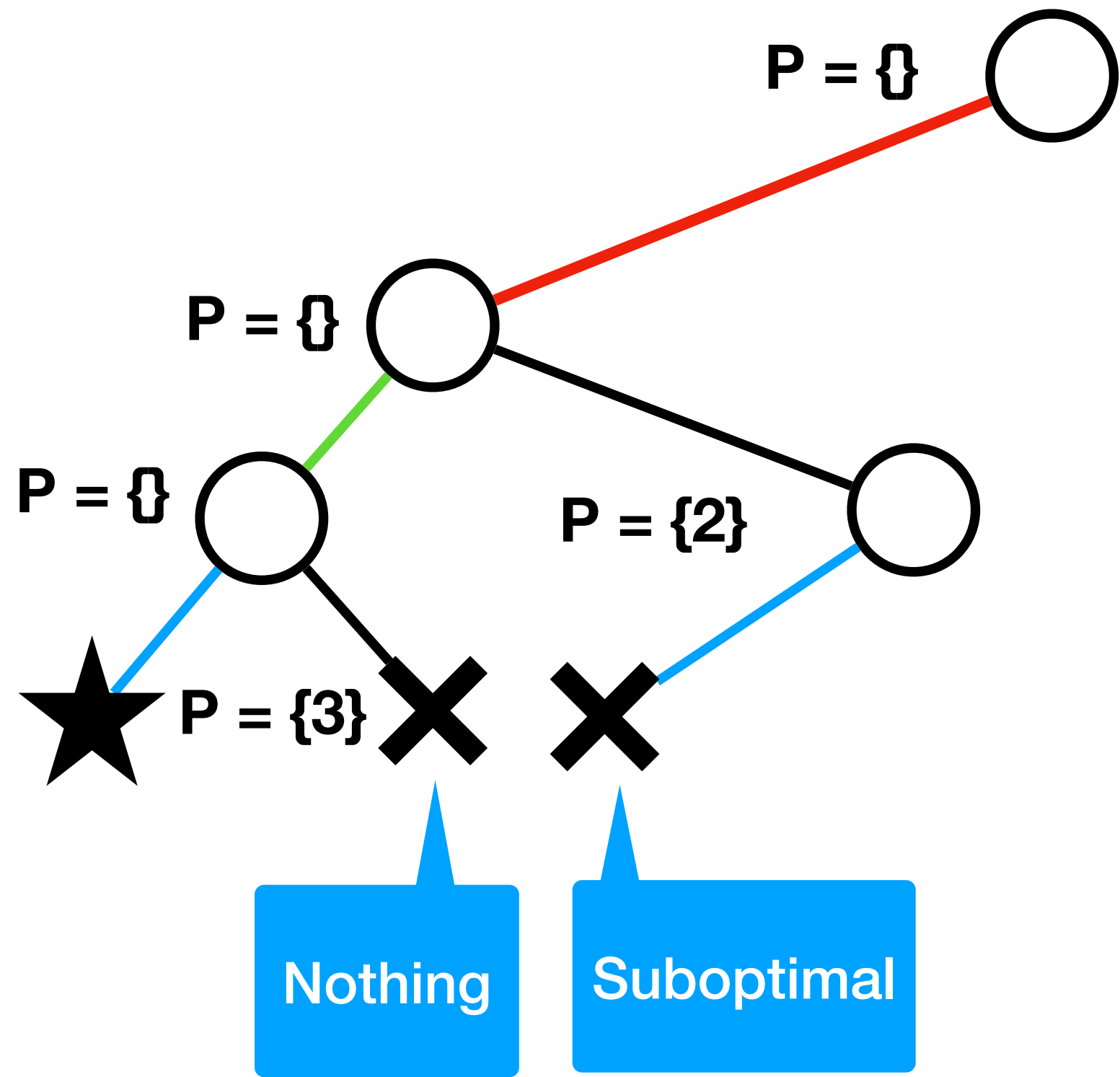
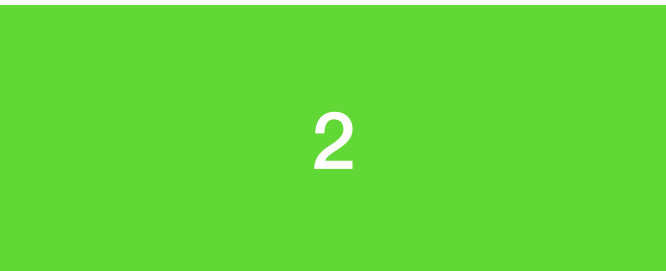
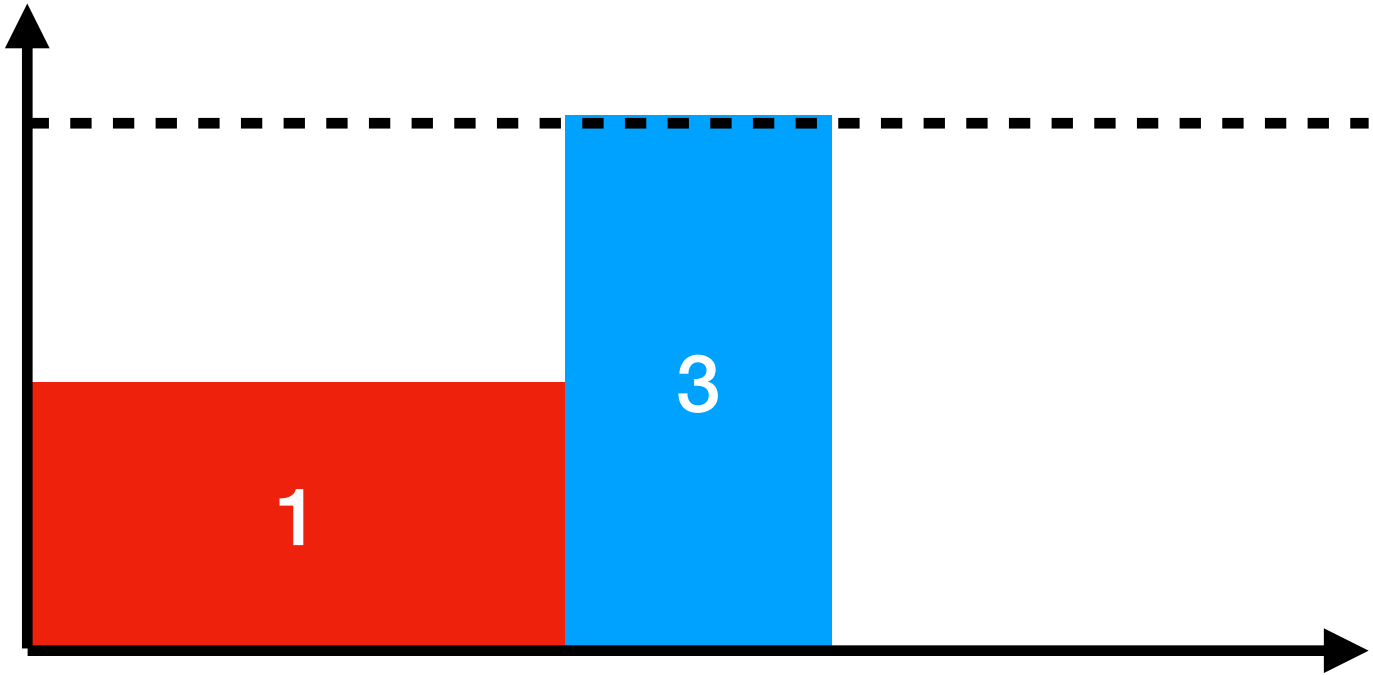
Le Pape, C., Couronne, P., Vergamini, D., & Gosselin, V. (1995). Time-versus-capacity compromises in project scheduling.

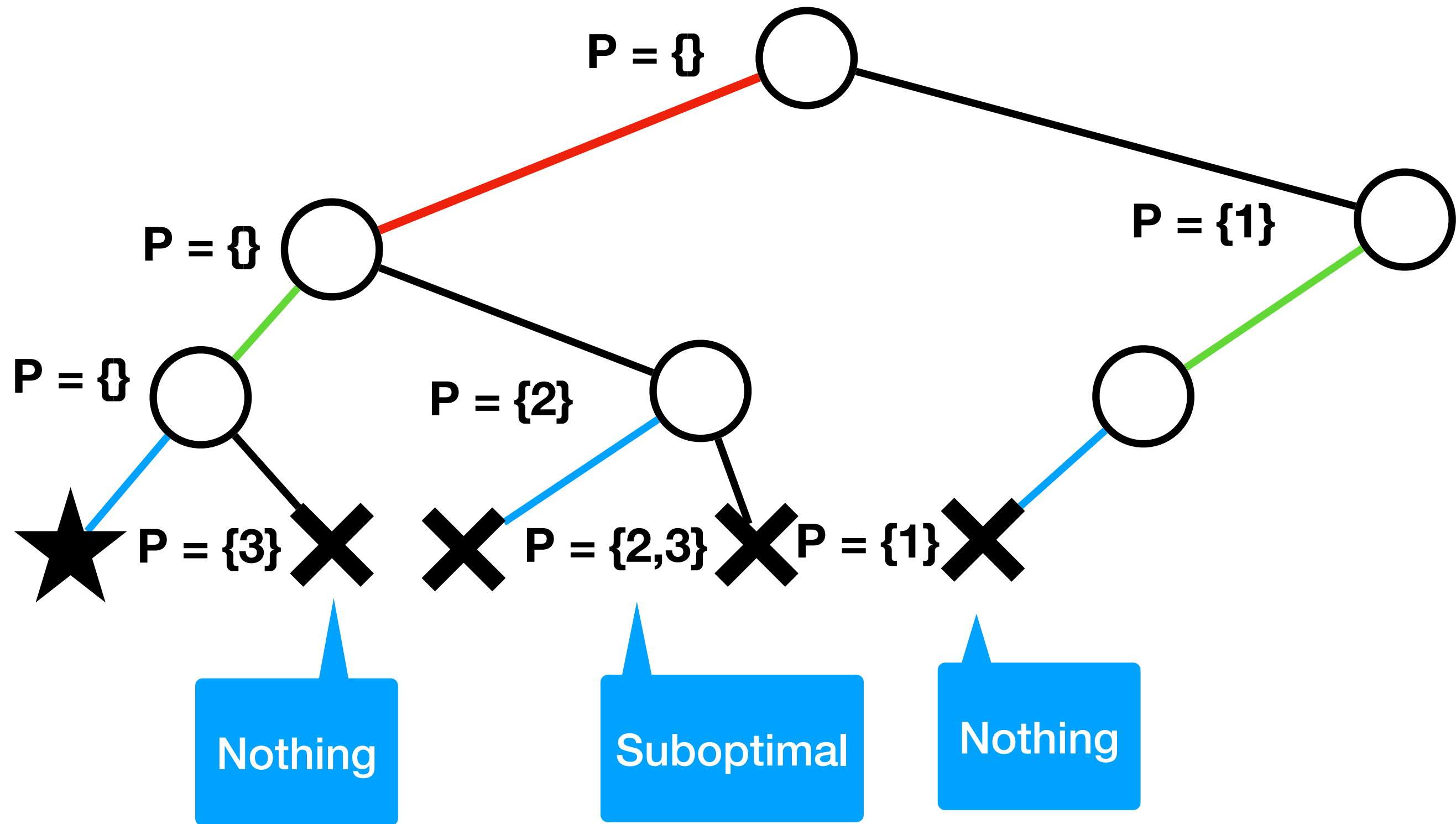
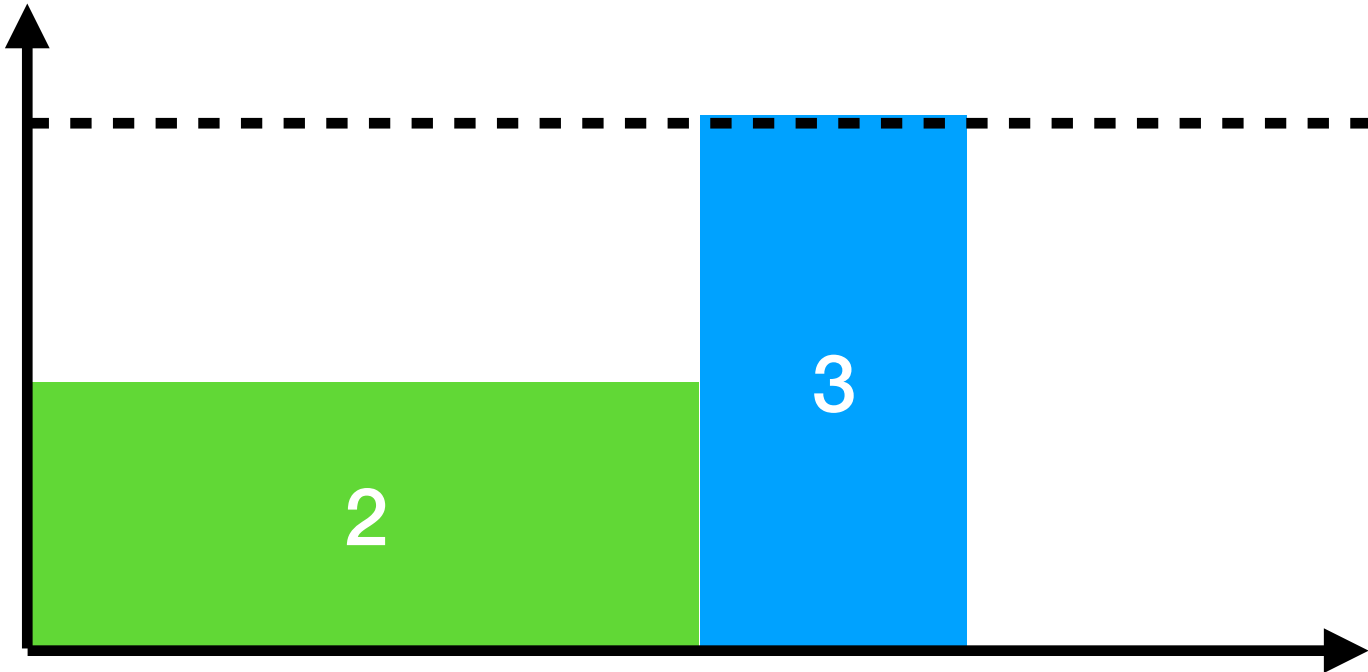


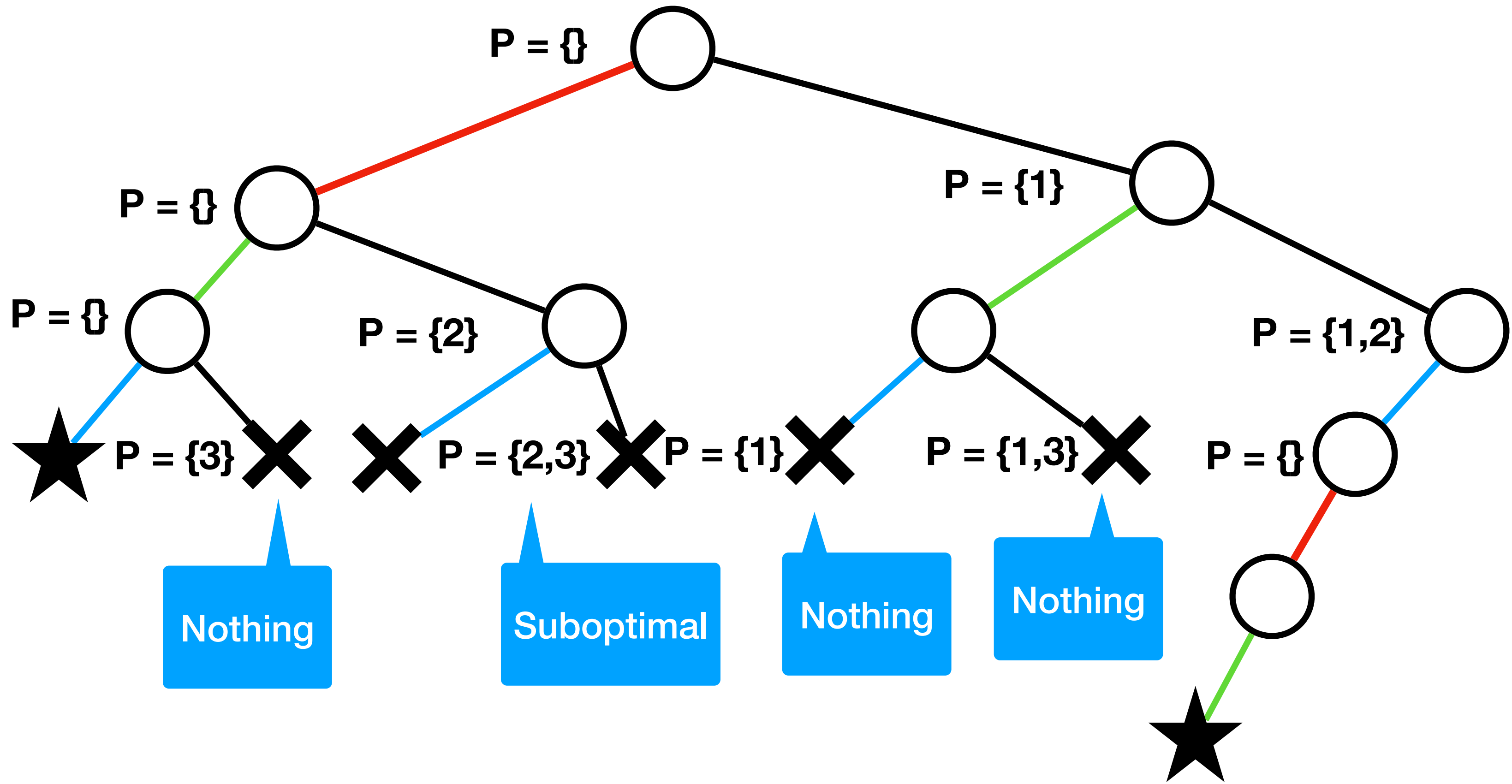
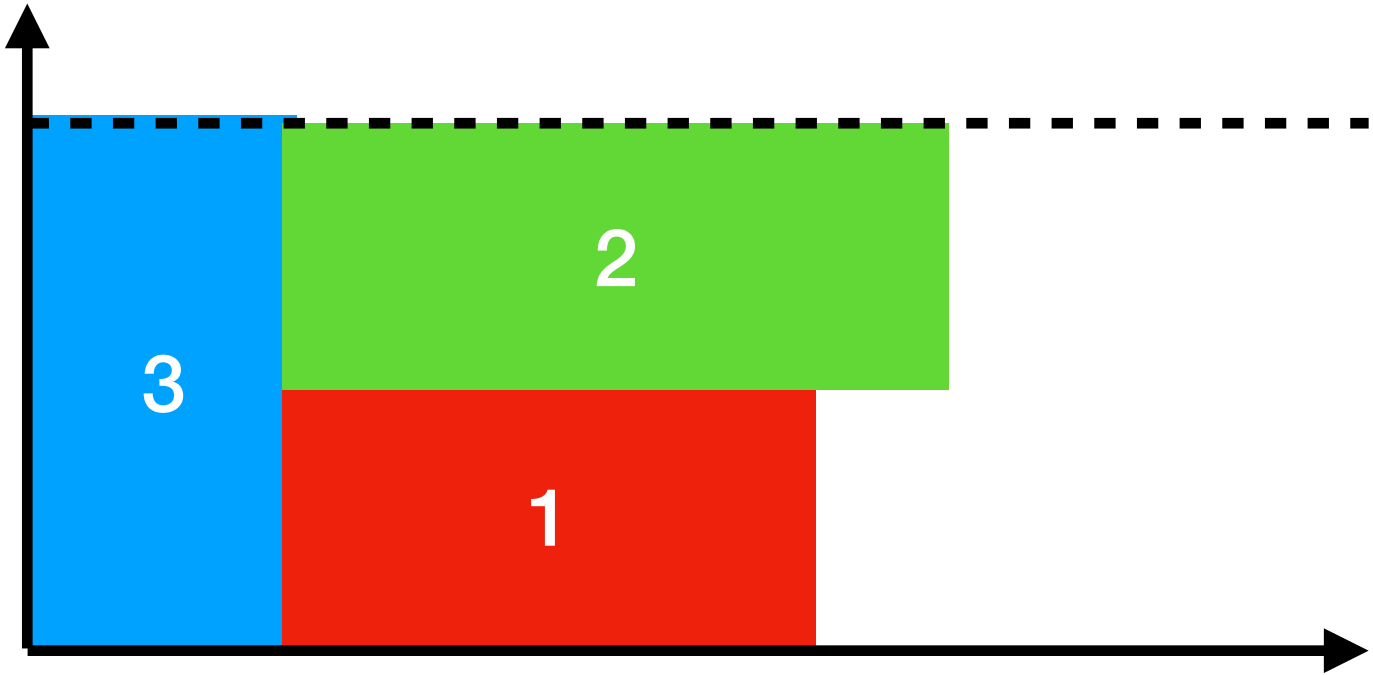
P = set of postponed tasks



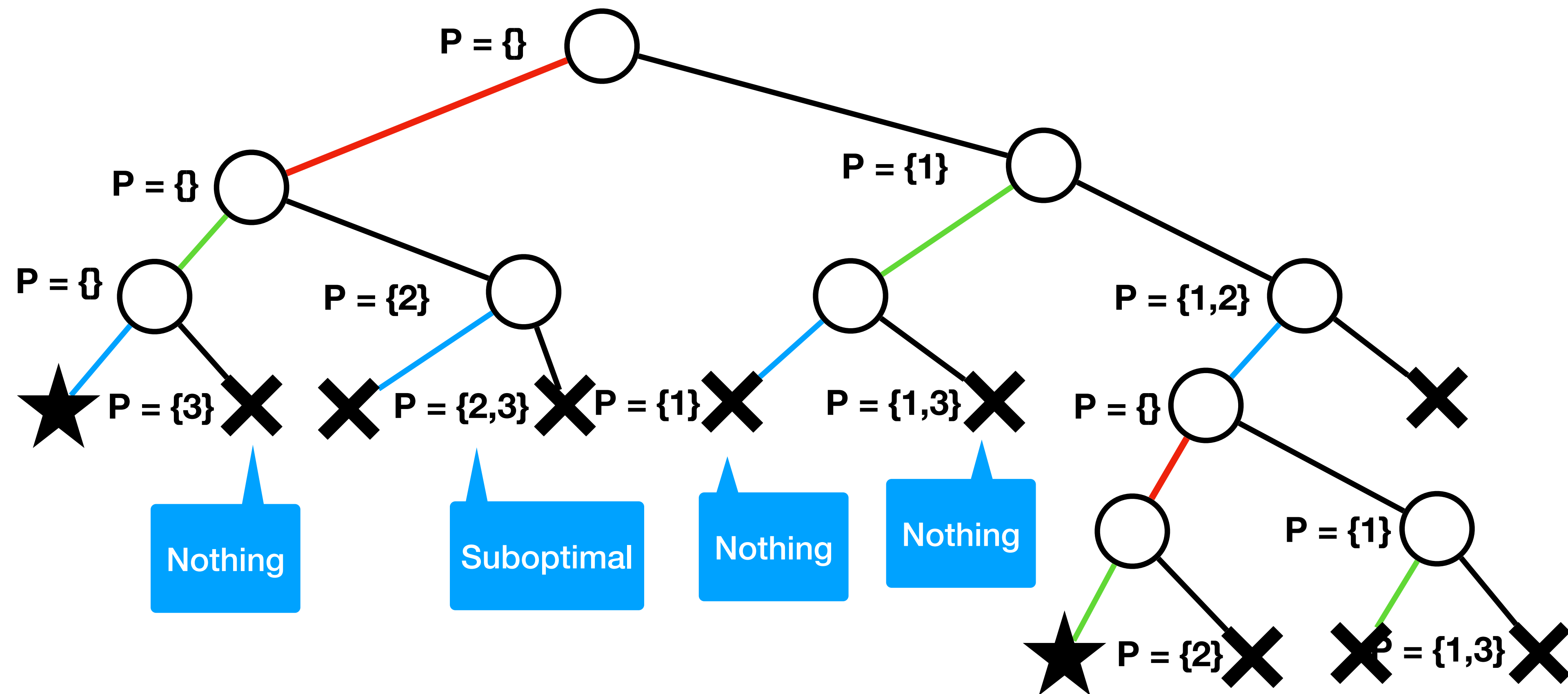
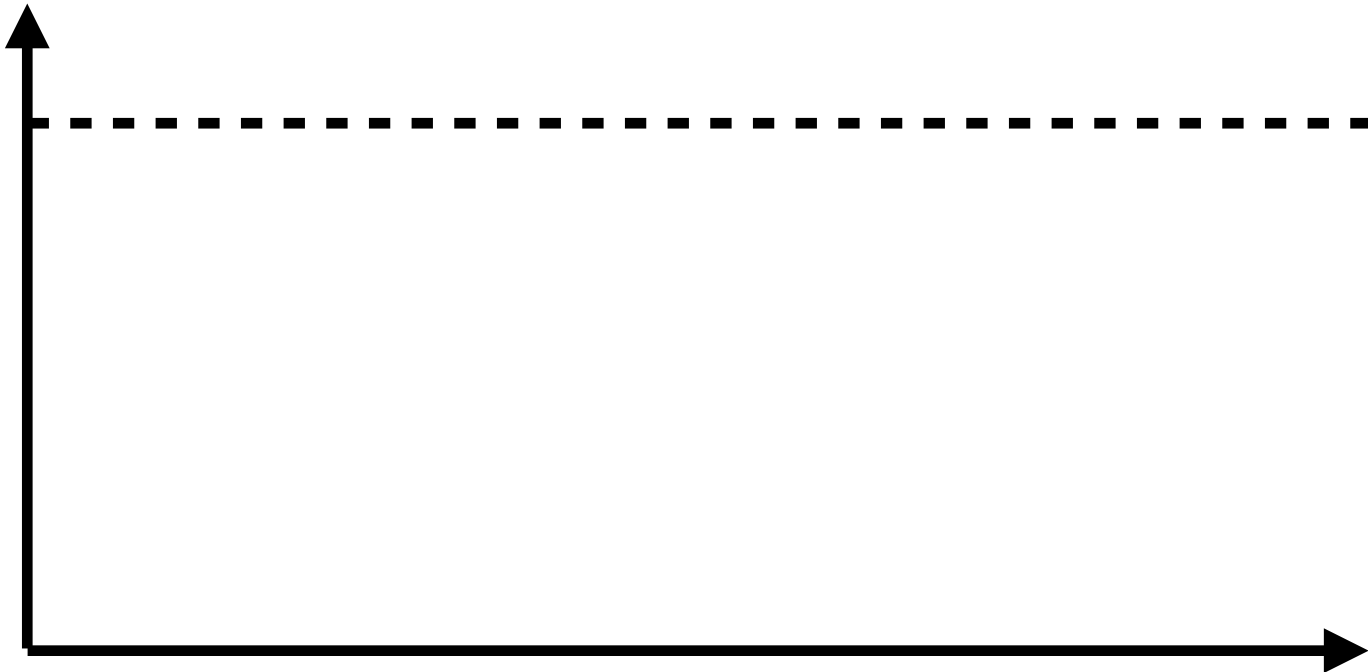
Le Pape, C., Couronne, P., Vergamini, D., & Gosselin, V. (1995). Time-versus-capacity compromises in project scheduling.



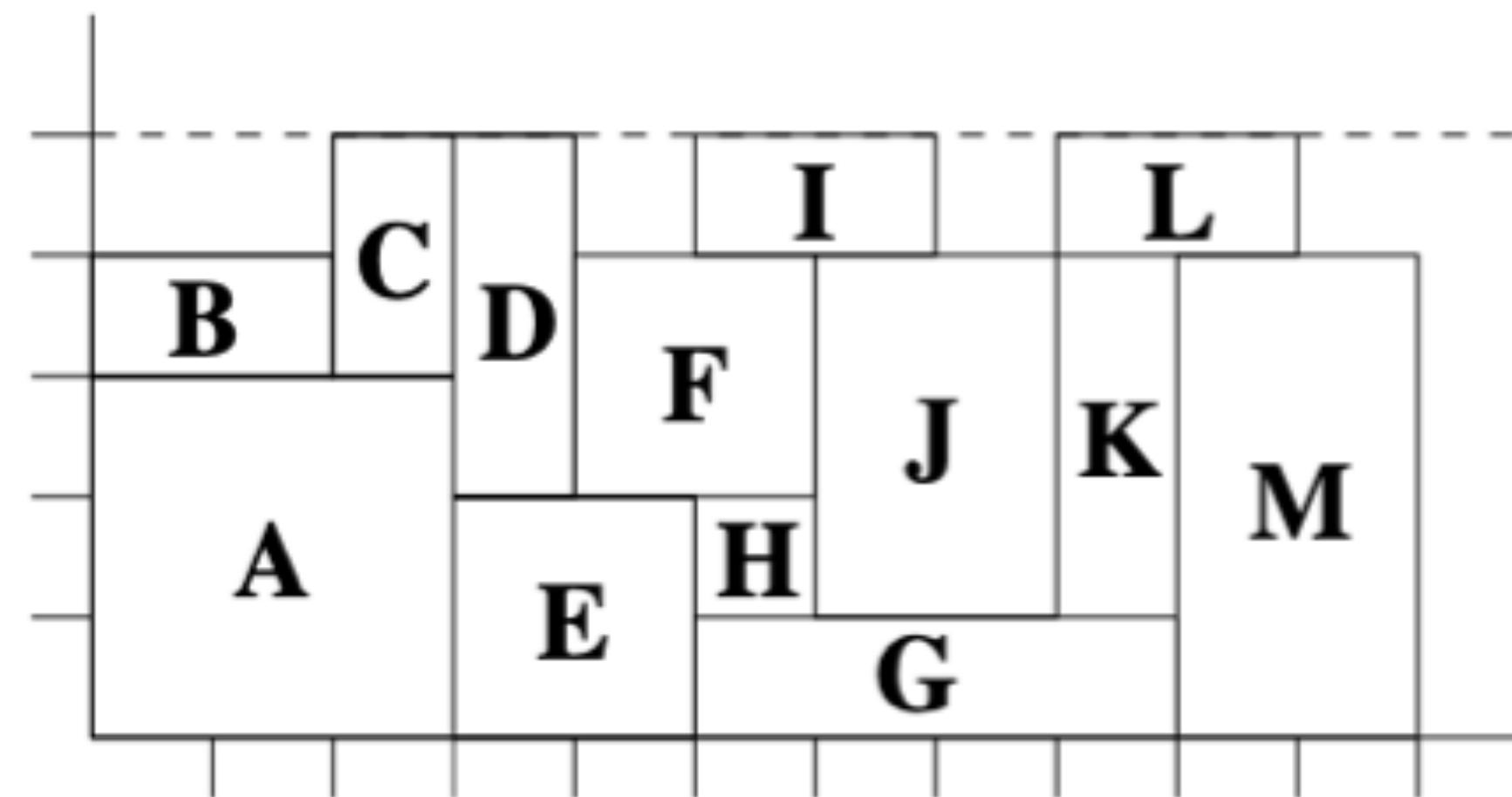




SetTimes



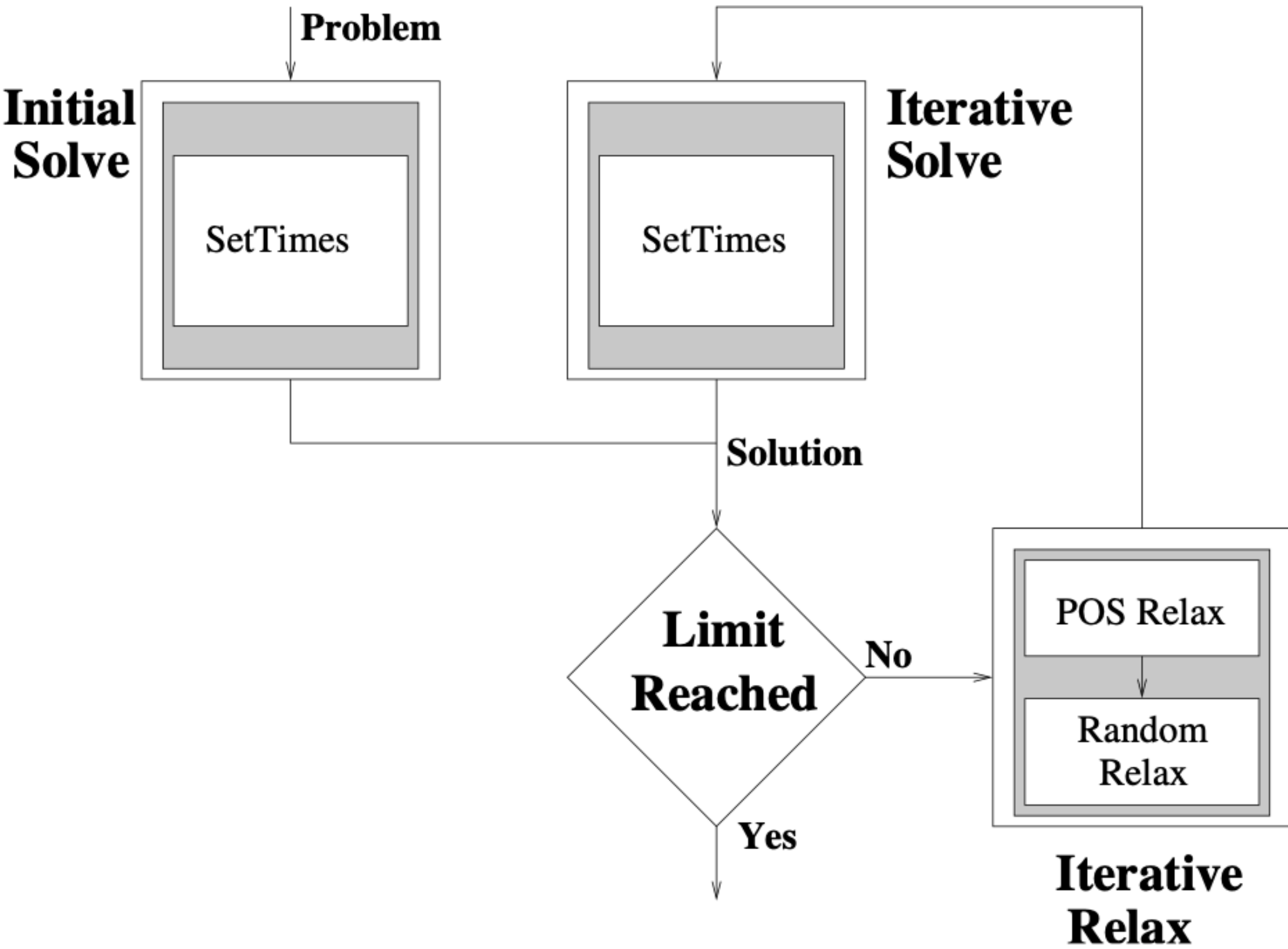
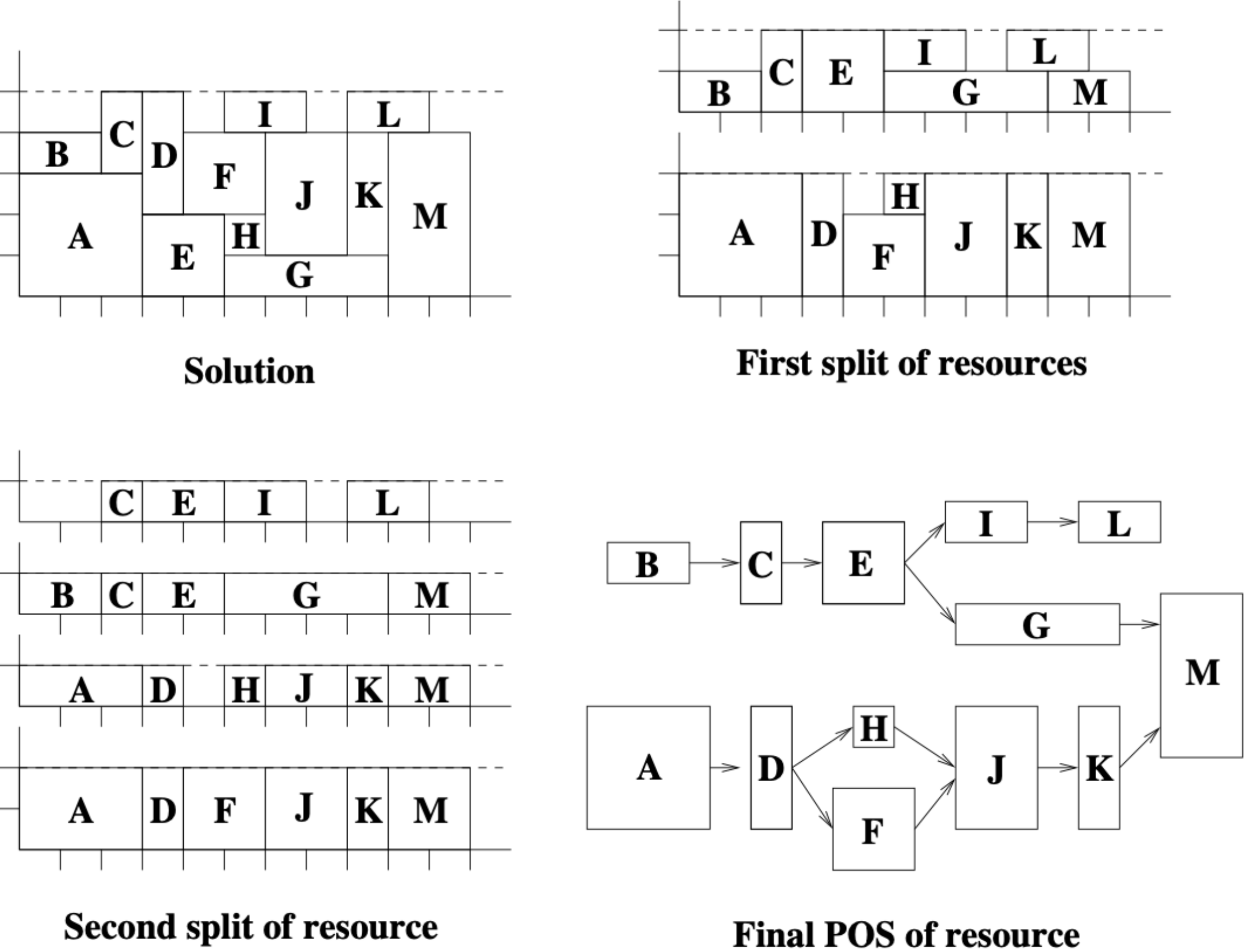
- ▶ If you relax some start-times while keeping fixed other start-times, you don't have much chance to improve the makespan at next iteration
- ▶ Idea: relax all start times, keep some precedence constraints (keep relative order constraints from the the best solution)



Solution

Godard, D., Laborie, P., & Nuijten, W. Randomized Large Neighborhood Search for Cumulative Scheduling. In *ICAPS* 2005

LNS for Scheduling: Partial Order Schedule



Godard, D., Laborie, P., & Nuijten, W. Randomized Large Neighborhood Search for Cumulative Scheduling. In *ICAPS* 2005

Parallelization in CP

You have more than one processor, what can you do ?



- ▶ Portfolio: use several heuristics (or randomized search)
 - For COP: communicate the bounds when a new best solution is found
 - If you use restart, you can share the No-Goods (not always easy)
- ▶ If you want to stay complete two strategies:
 - Work-stealing (quite a lot of communication):
 - Search Procedures and Parallelism in Constraint Programming. Perron, L. CP1999
 - Michel, L., See, A., & Van Hentenryck, P. Parallelizing constraint programs transparently. CP2007
 - Chu, G., Schulte, C., & Stuckey, P. J. Confidence-based work stealing in parallel constraint programming. CP2009
 - Embarrassingly parallel search (try to keep communication very low):
 - Régim, J. C., Rezgui, M., & Malapert, A. Embarrassingly parallel search. CP2013

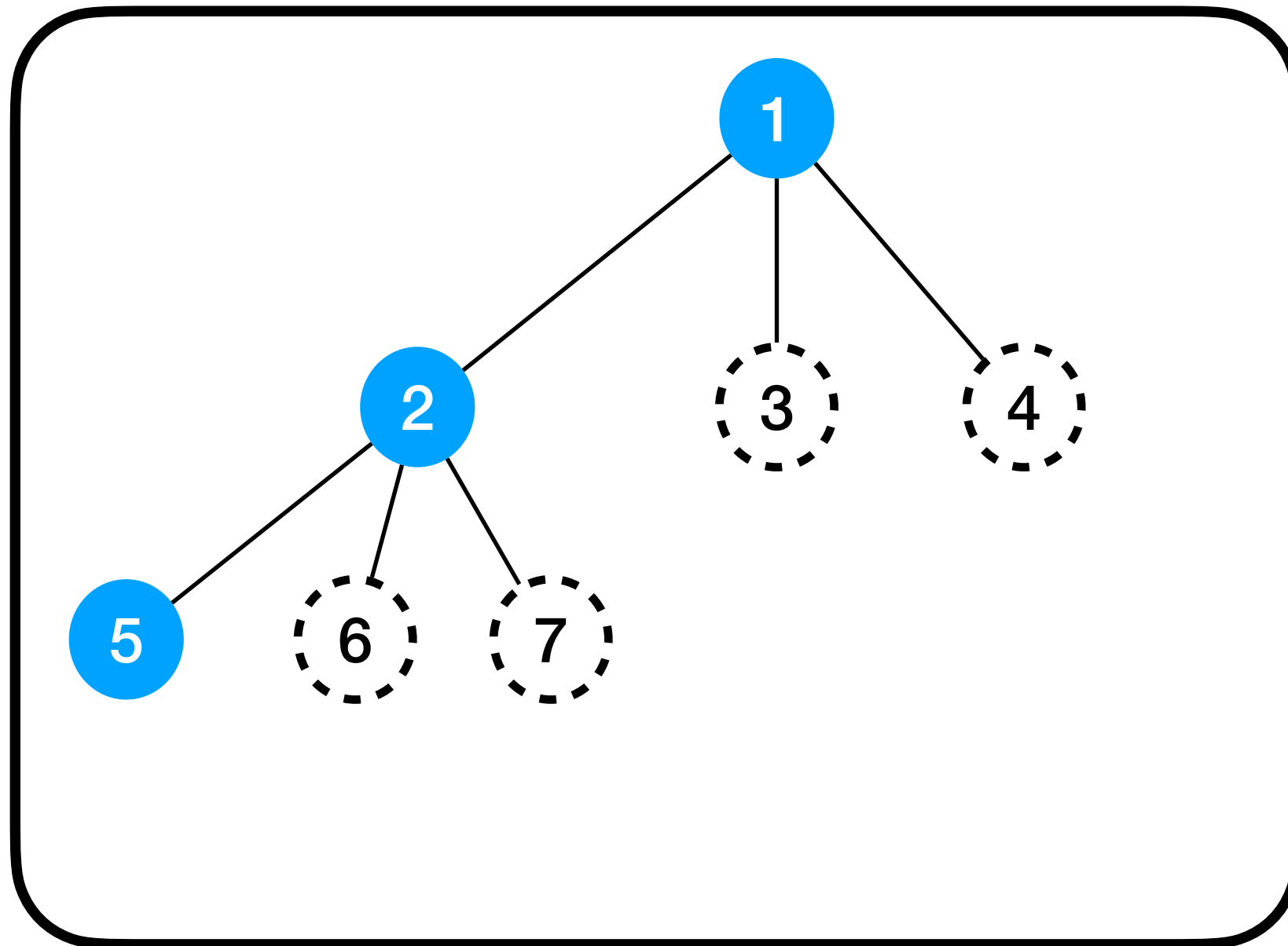
Work Stealing

- If a worker is idle, it steals sub-problems from another active worker.

Busy worker1

Idle worker2

Idle worker2

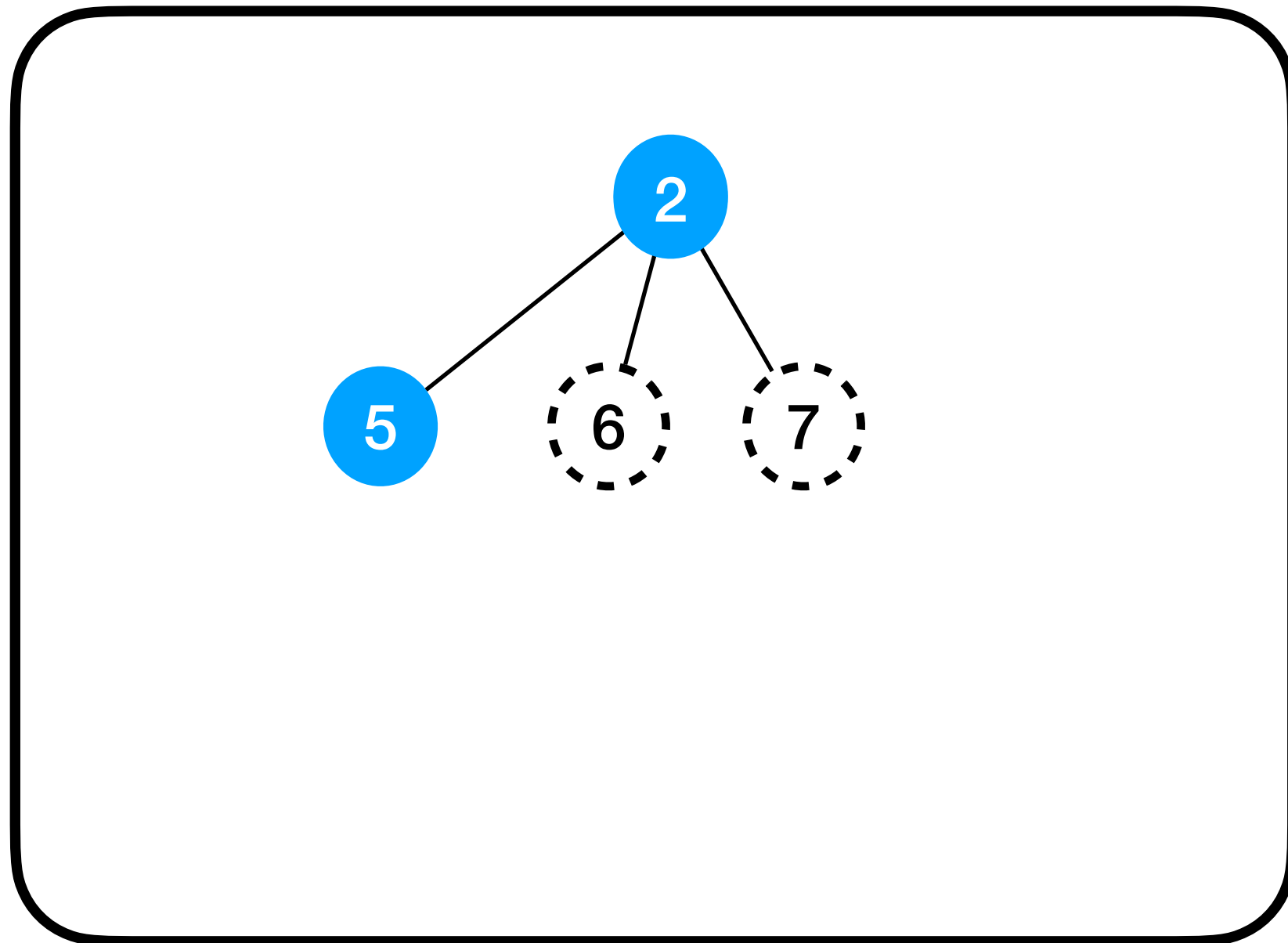


Michel, L., See, A., & Van Hentenryck, P. Parallelizing constraint programs transparently. CP2007

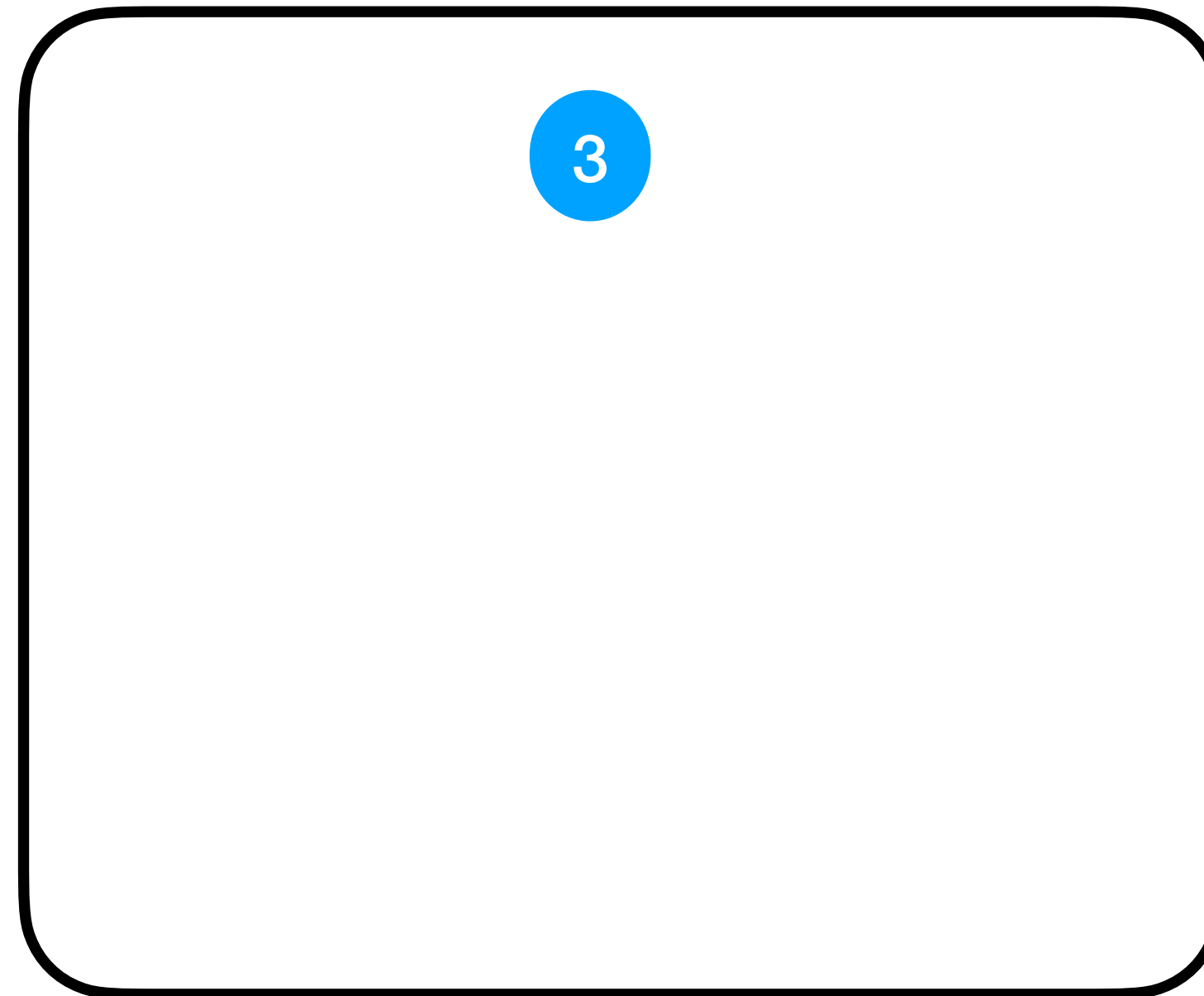
Work Stealing

- If a worker is idle, it steals sub-problems from another active worker.

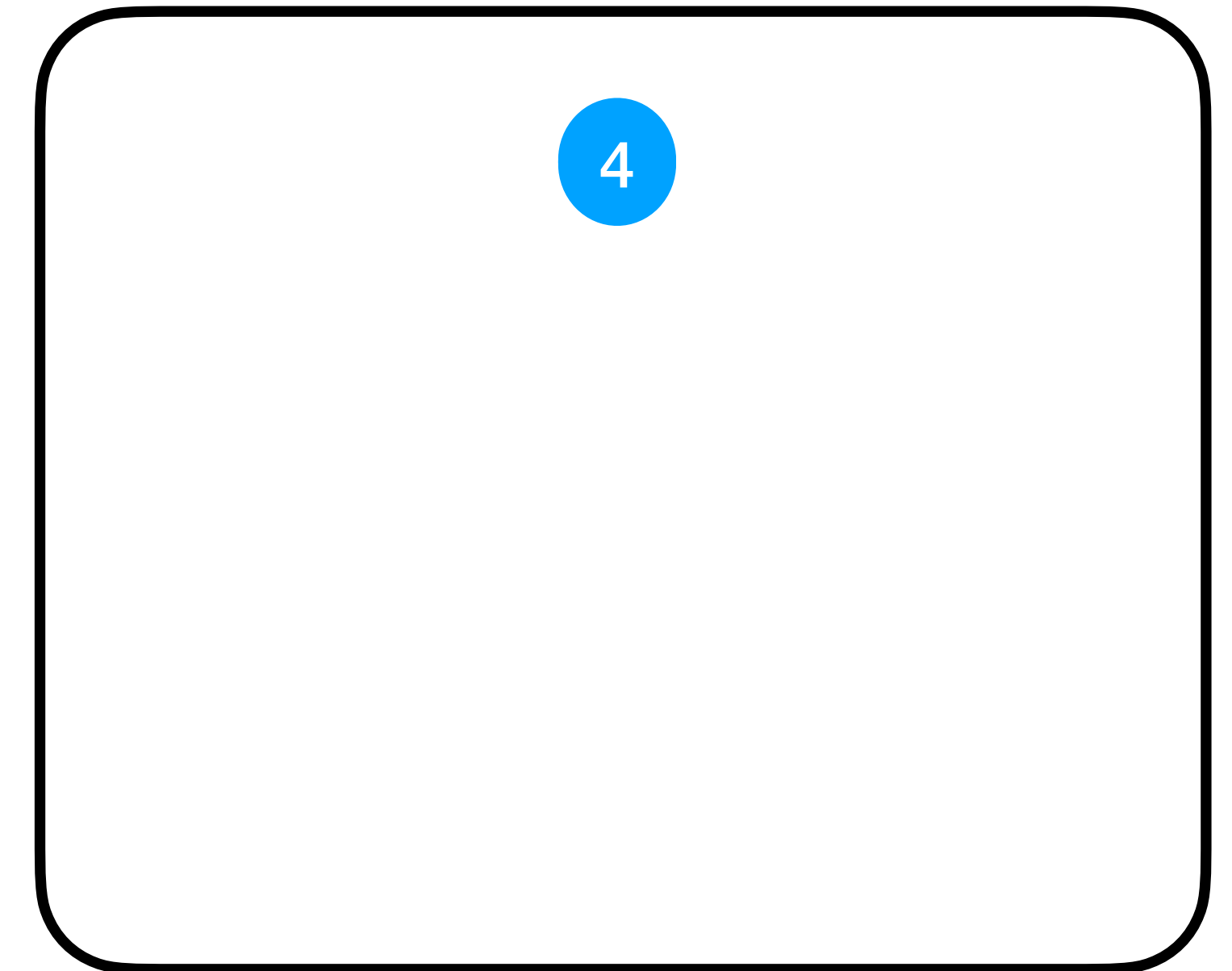
Busy worker1



Idle worker2



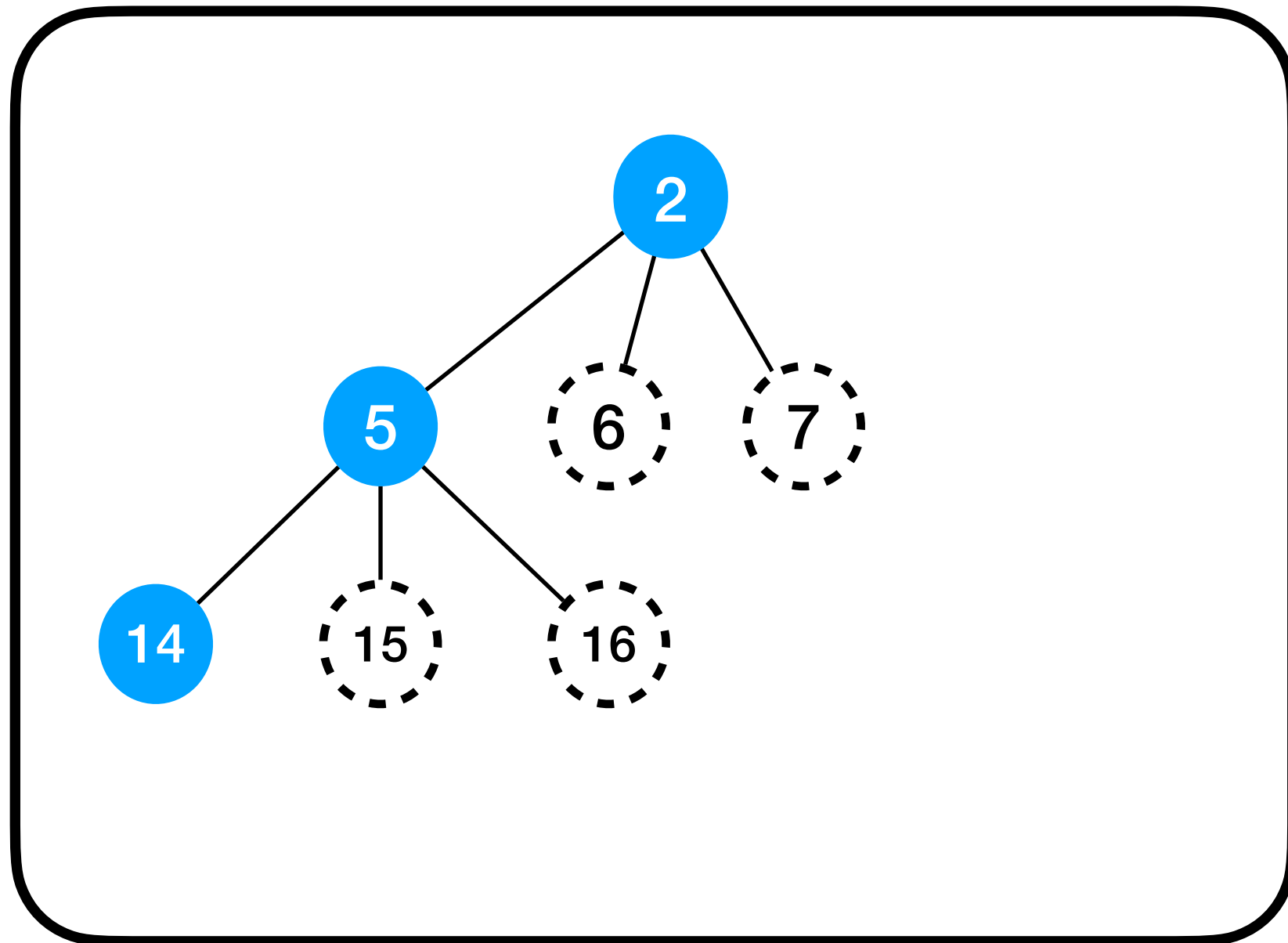
Idle worker2



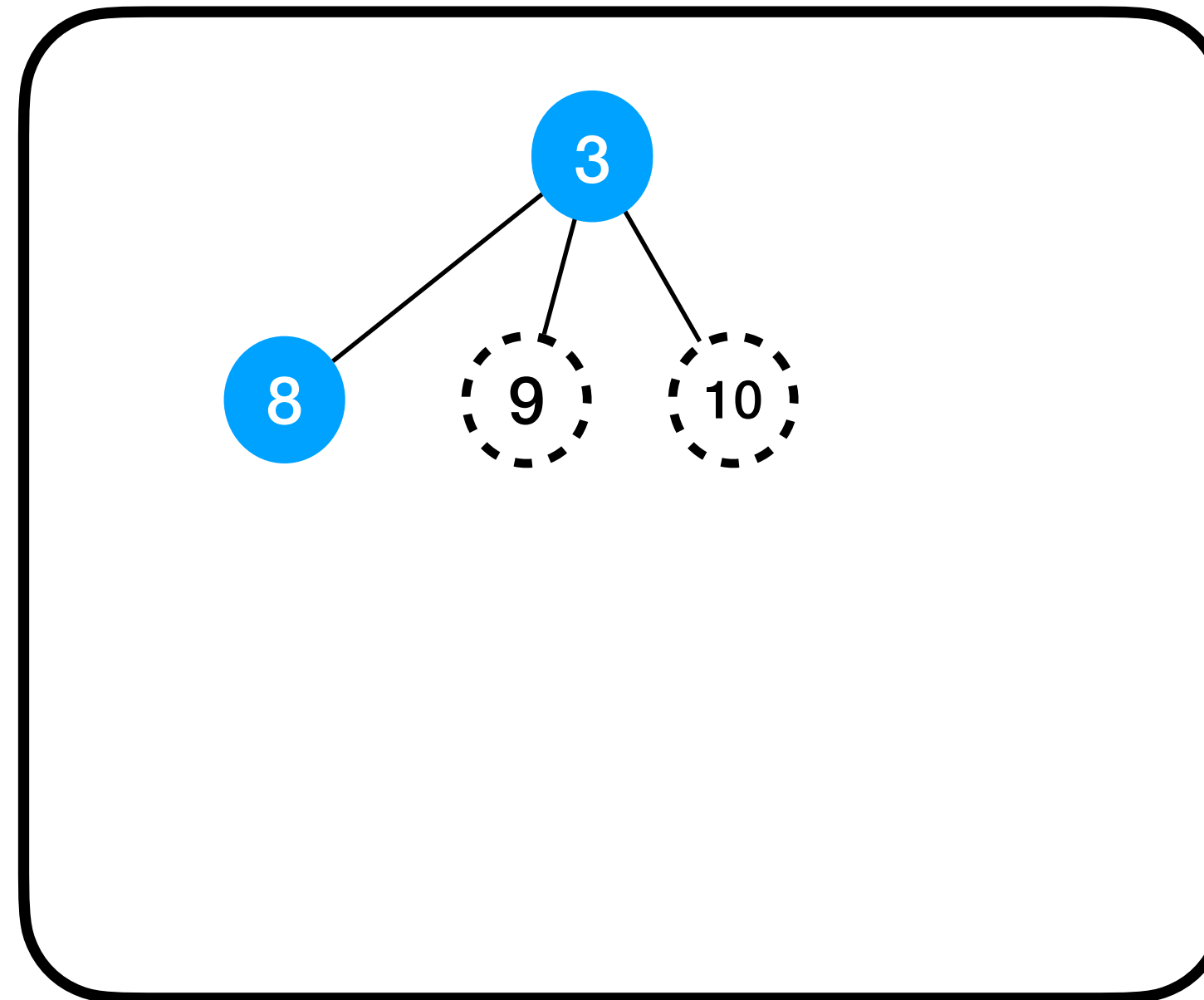
Work Stealing

- If a worker is idle, it steals sub-problems from another active worker.

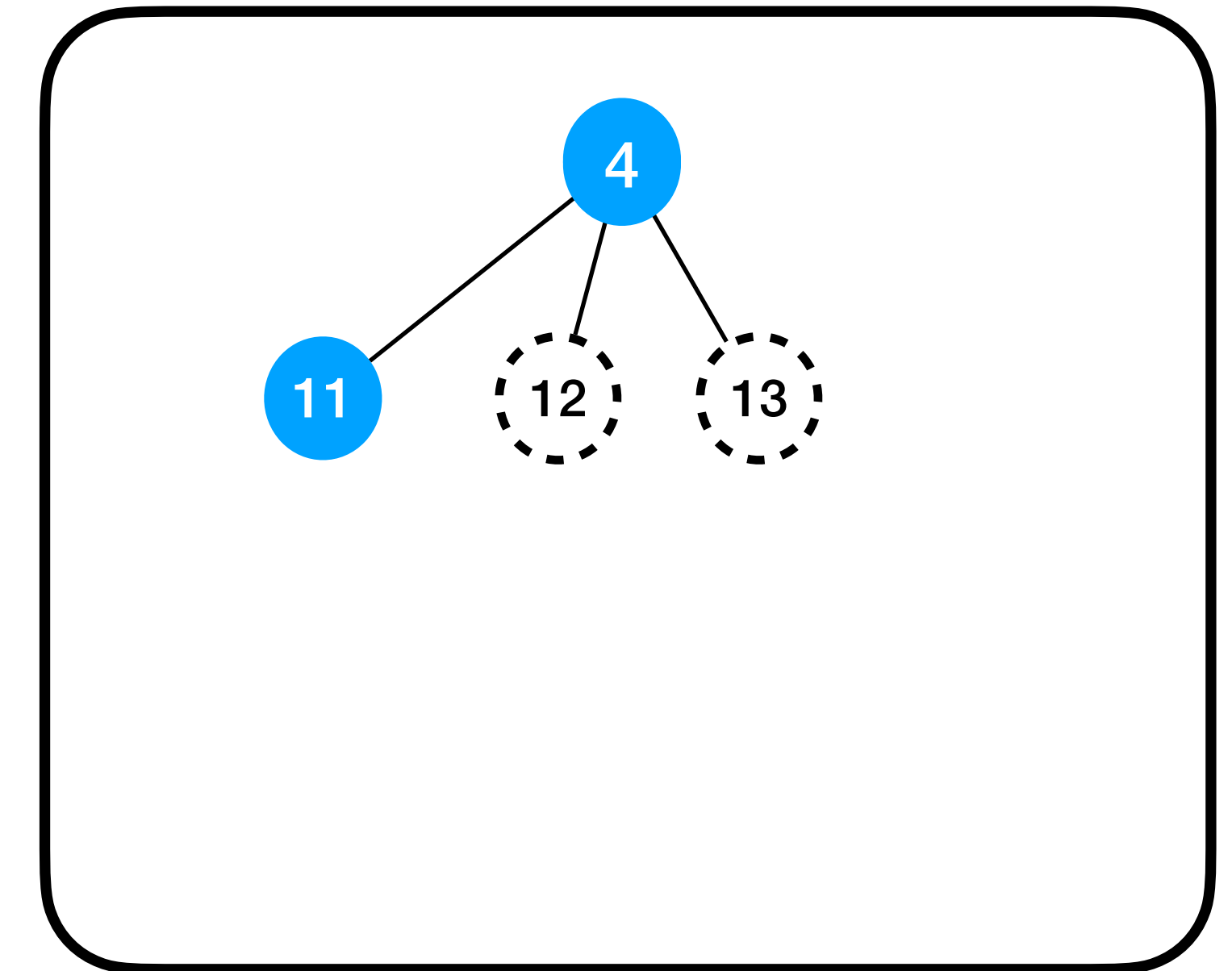
Busy worker1



Idle worker2

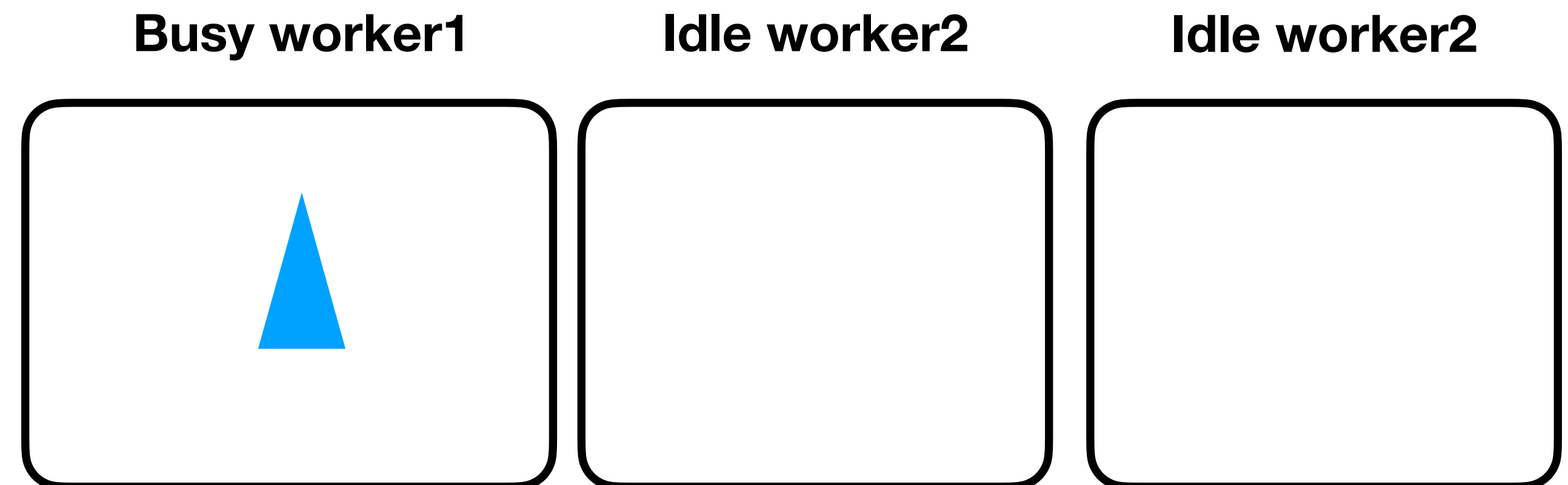
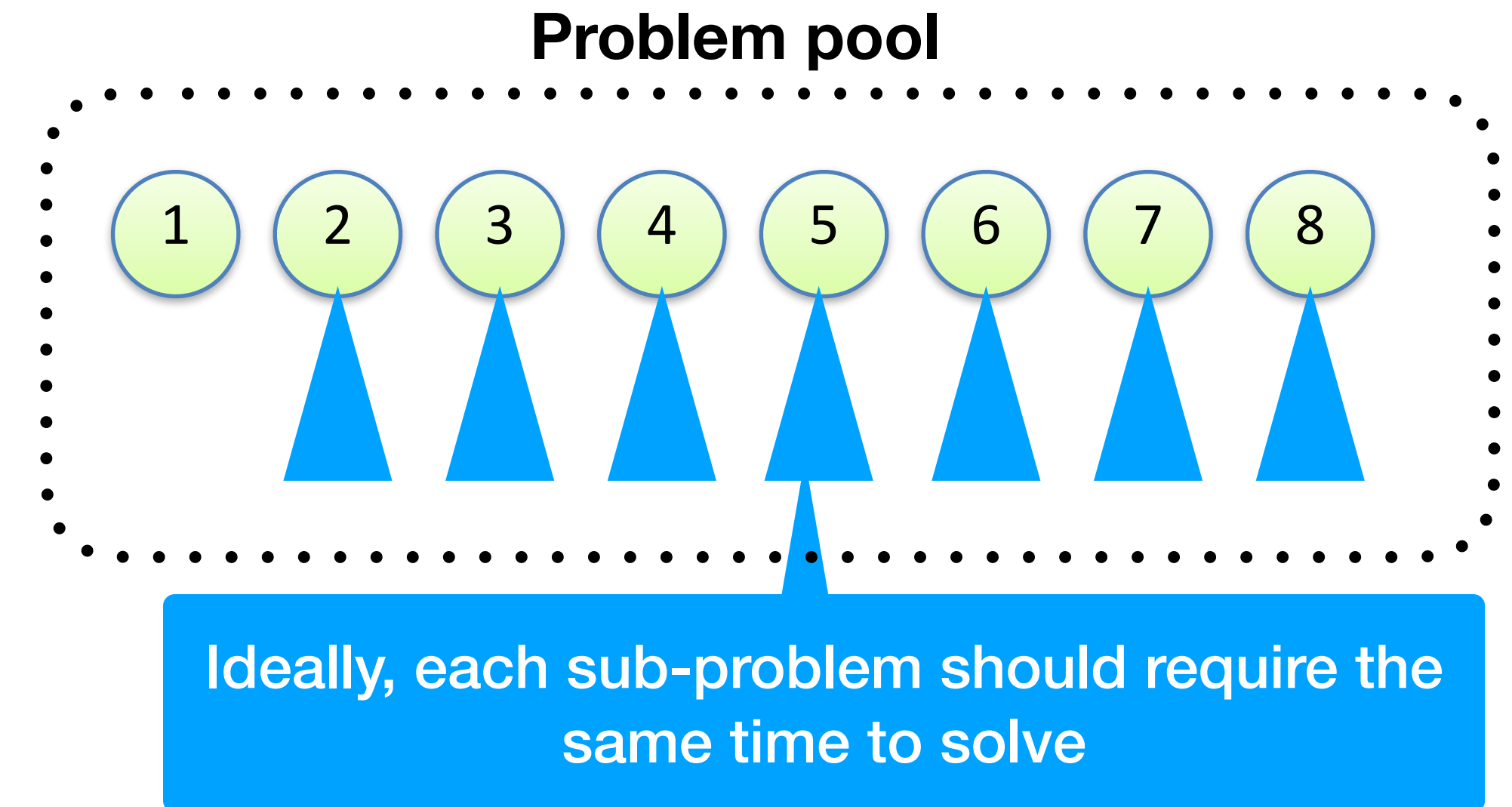
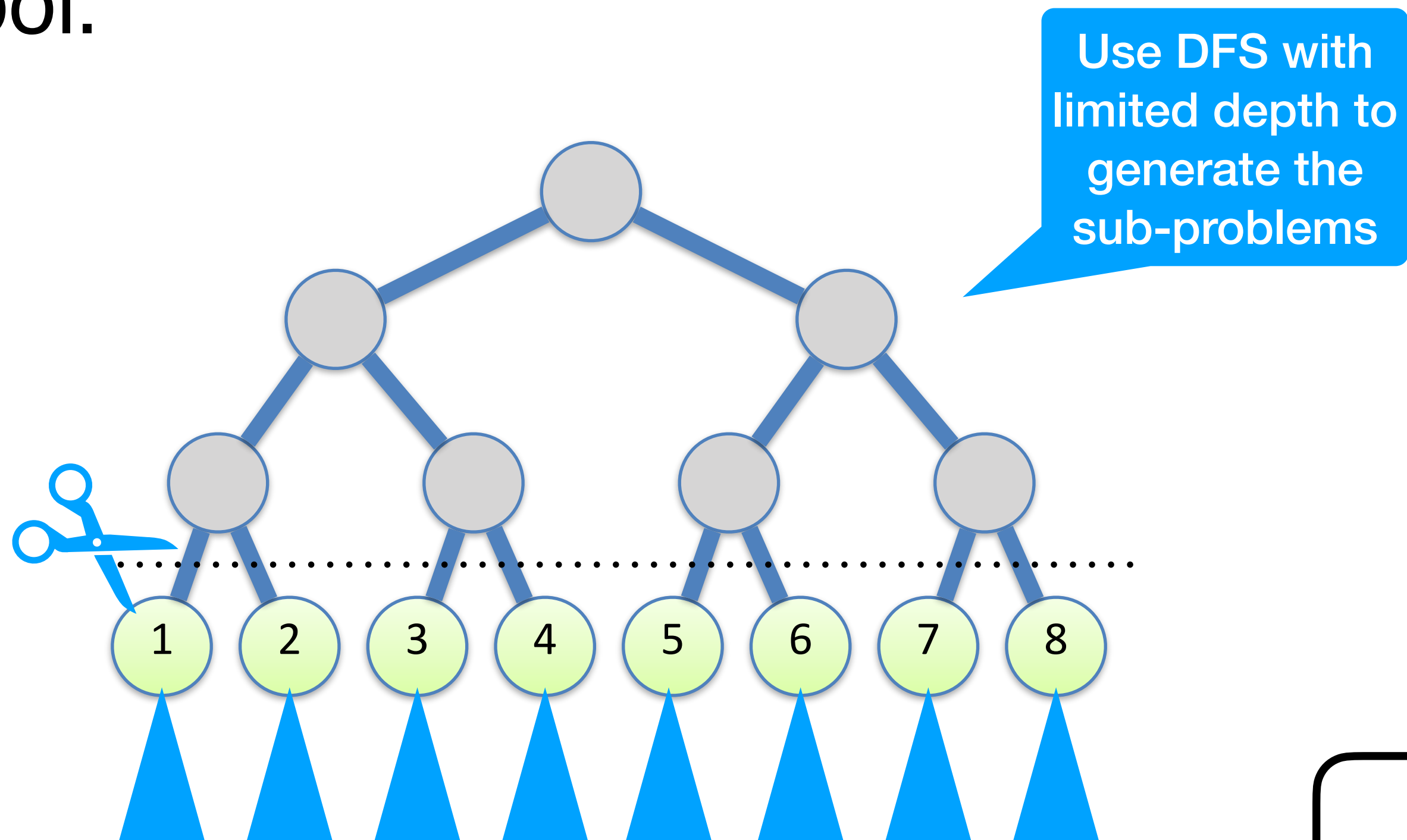


Idle worker2



Embarrassingly parallel search

- Split the initial problem into a huge number of small sub-problems, add them to a pool so that whenever a worker is idle, it can just take the next one in the pool.



- Régim, J. C., Rezgui, M., & Malapert, A. Embarrassingly parallel search. CP2013

Search Wrap up

- ▶ Value selection heuristic:
 - Bound-Impact Value Selector (BIVS)
 - Phase saving
- ▶ Variable selection using the first-fail heuristic:
 - Degree and Weighted Degree
 - Impact-Based Search (IBS):
 - Choose the variable and value that have a strong impact in terms of filtering.
 - The impacts are estimated and learned during the search.
 - Activity-Based Search (ABS):
 - Choose the variable whose domain is often filtered: activity.
 - The activity is learned during the search.
 - Last-Conflict Search (LC):
 - Choose the variable that led to the latest conflict, if any.
 - Conflict-Ordering Search (COS):
 - Choose the variable with the largest timestamp.
- Discrepancy-Based Search Combinator