Constraint Programming

Search

Module Overview

- Motivation
- Heuristics
  - Value Selection Heuristics
  - Variable Selection Heuristics
    - First Fail
    - Degree
    - Impact
    - Activity
    - Most Recent Conflicts
- Strategies (a.k.a. Metaheuristics)
  - Discrepancy Search
  - LNS and Restarts

Motivation

Black-Box Search: The Vision

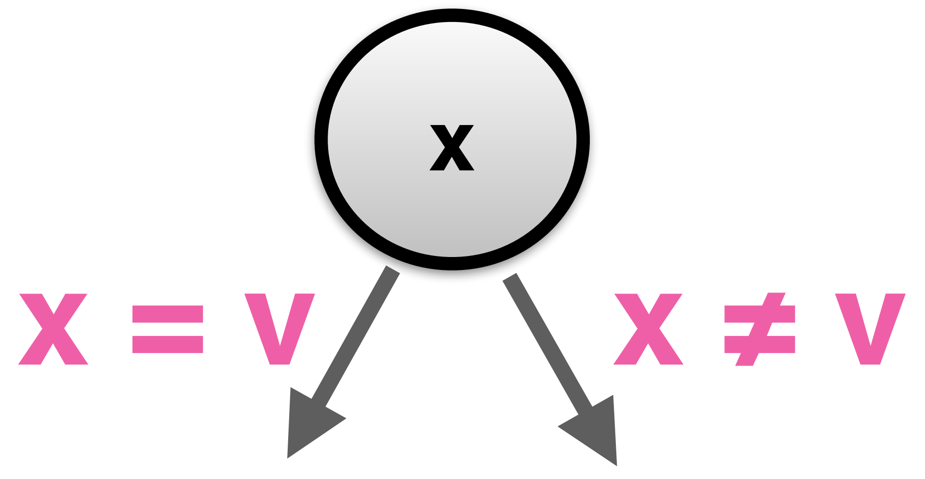
- Constraint Program = Model + Search

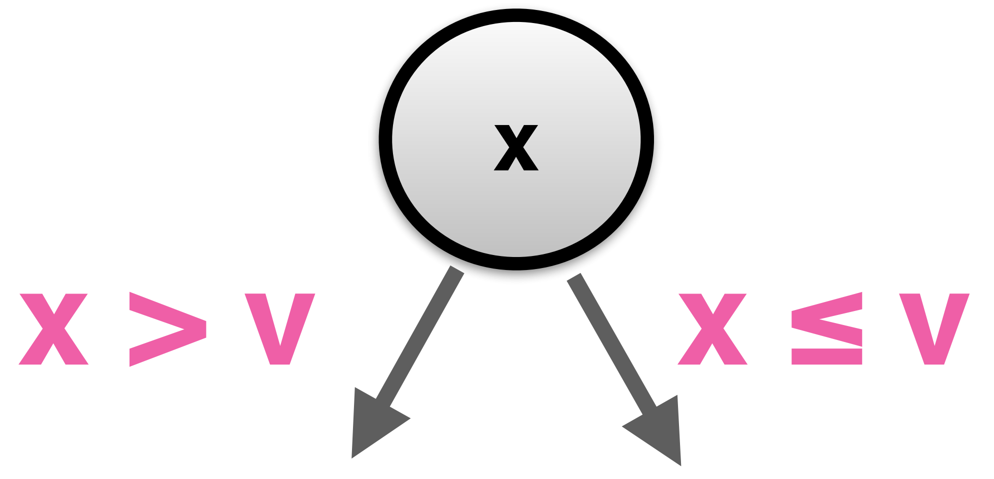
The vision is that we should
not have to worry too much about the search.
A CP solver should be smart enough to learn
or select a good search for the model.

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

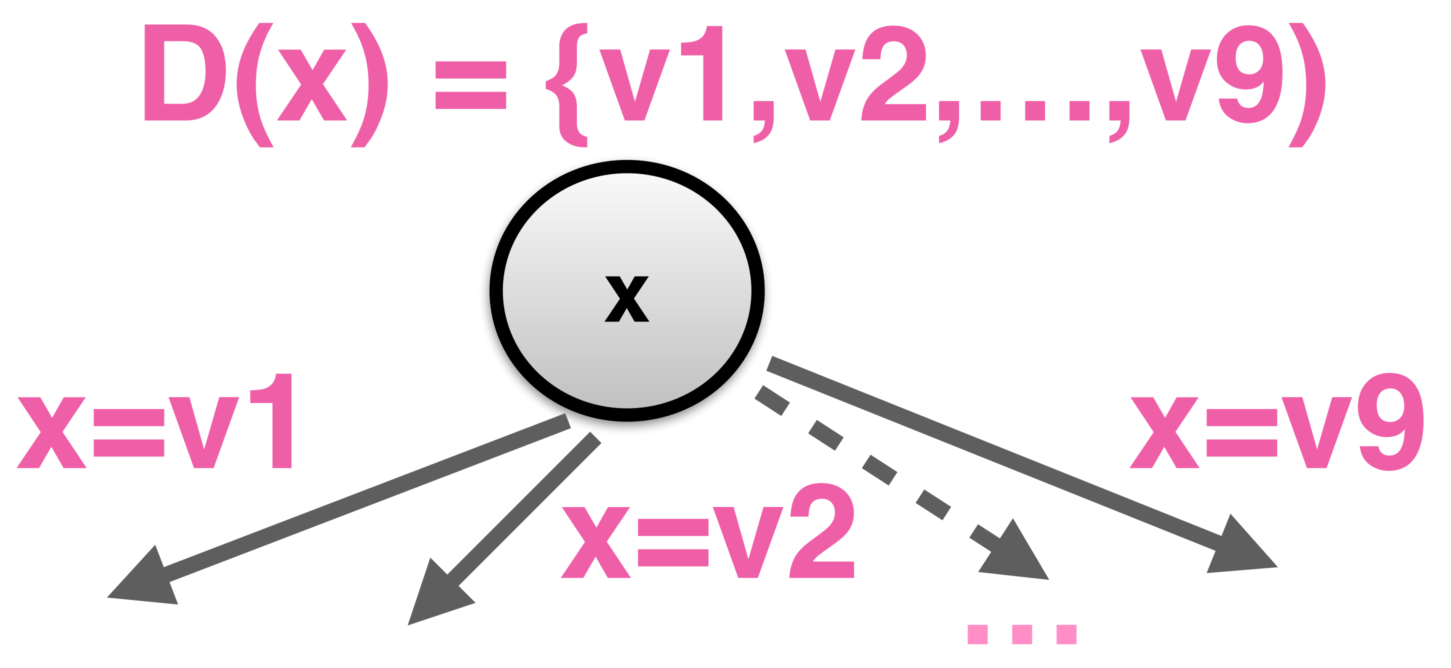
Refresher: Branching = 2-Step Choice

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





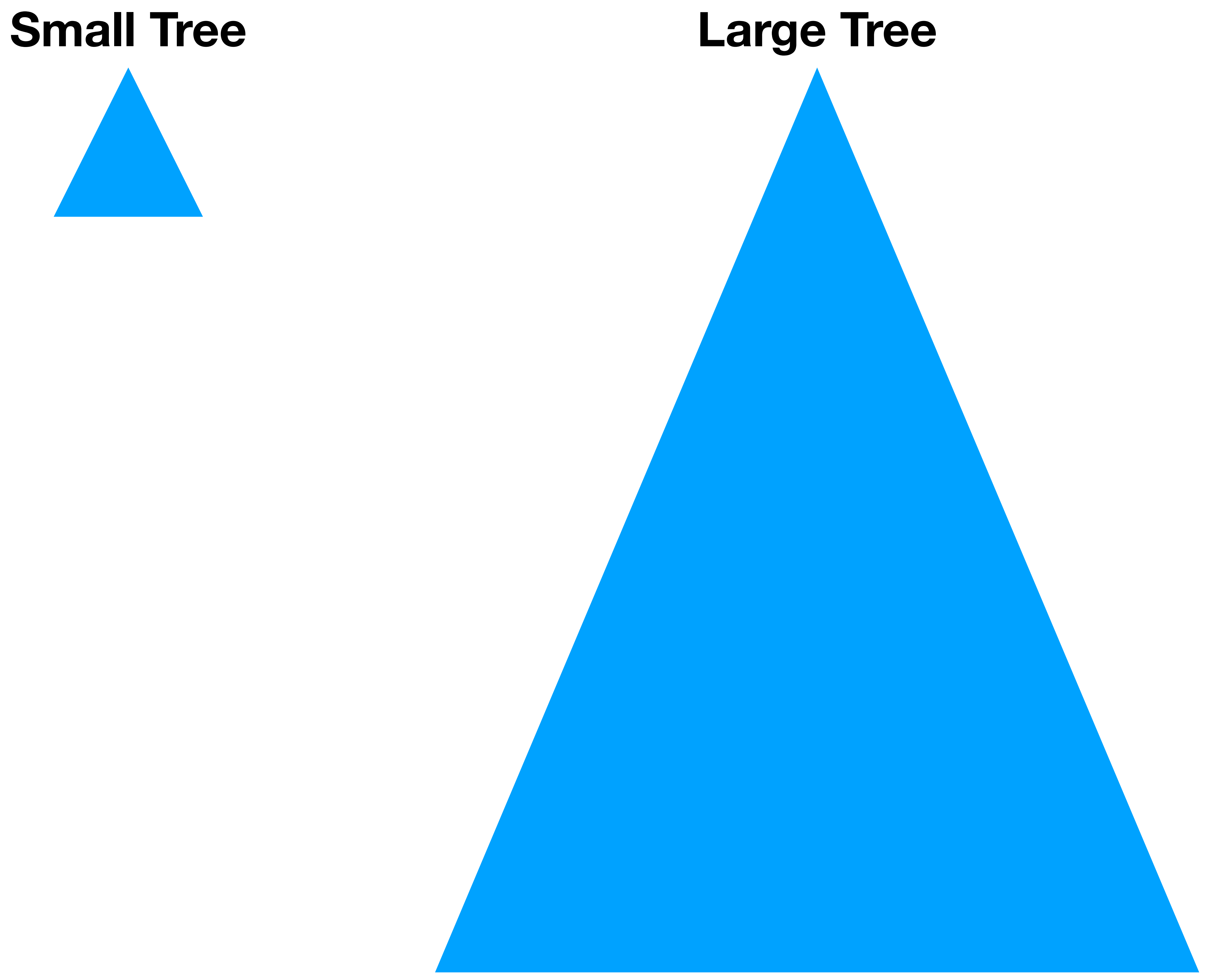
Branching does not need to be binary:



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

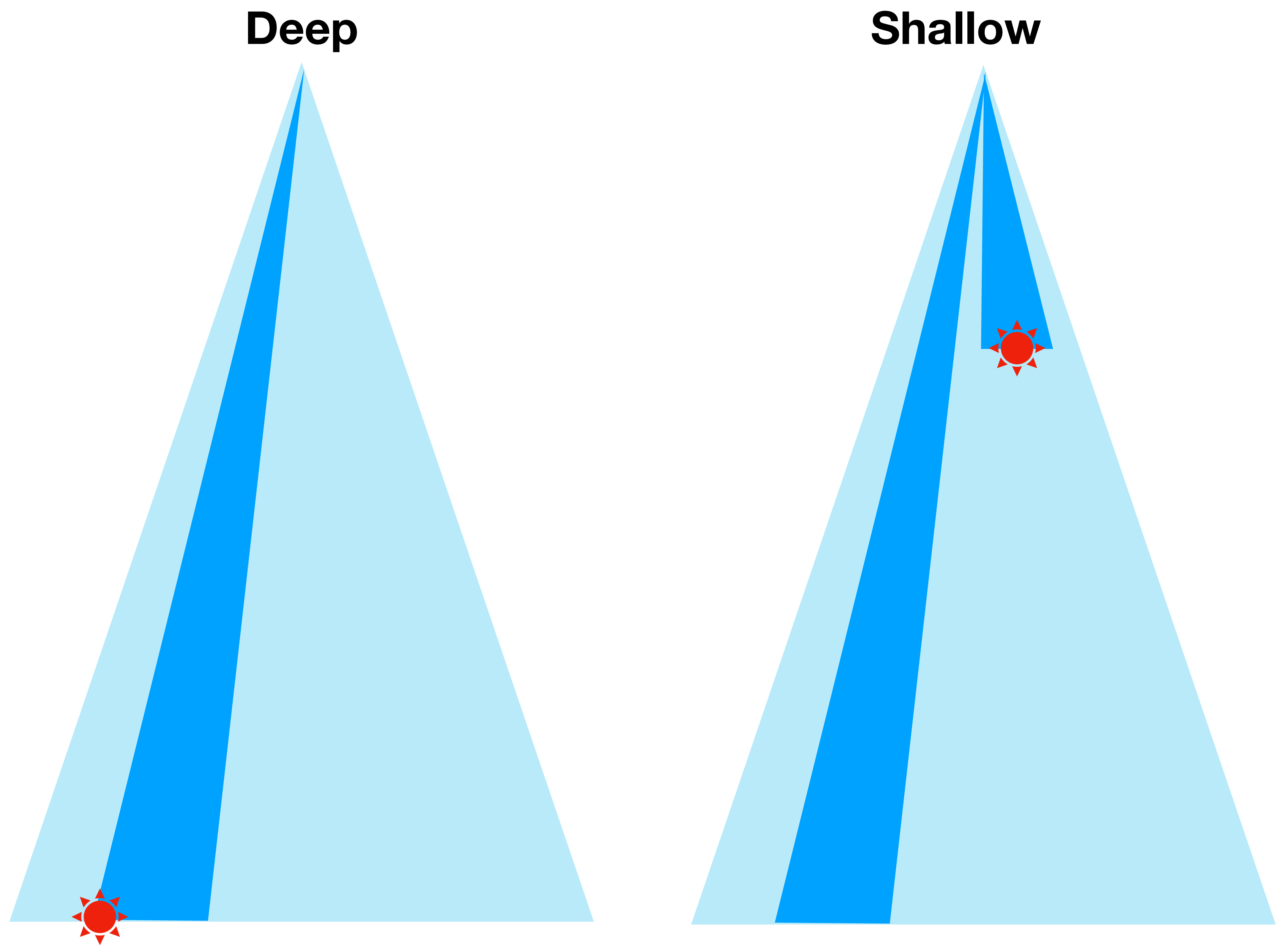
Searching…

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



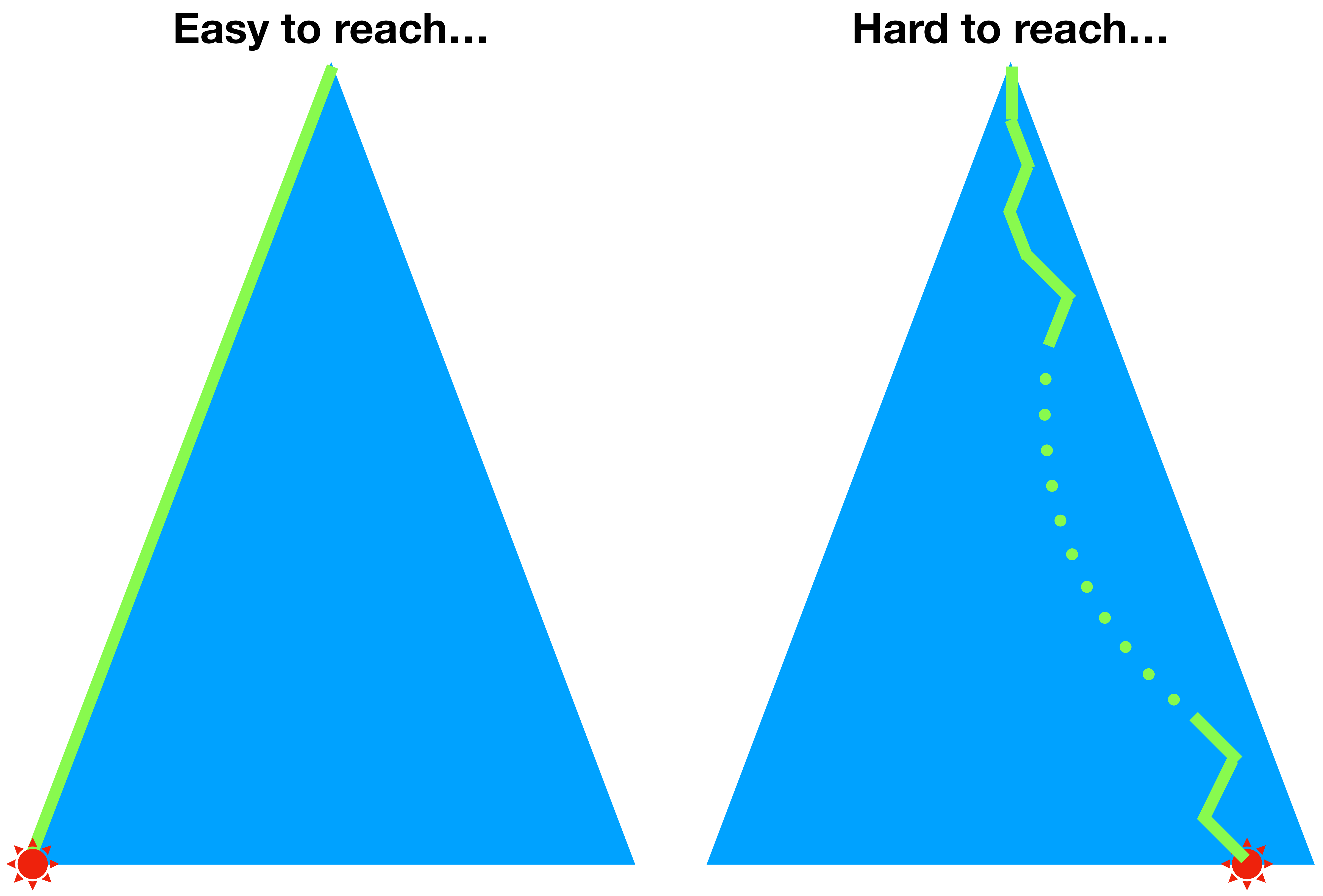
Searching…

- 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



Bottomline

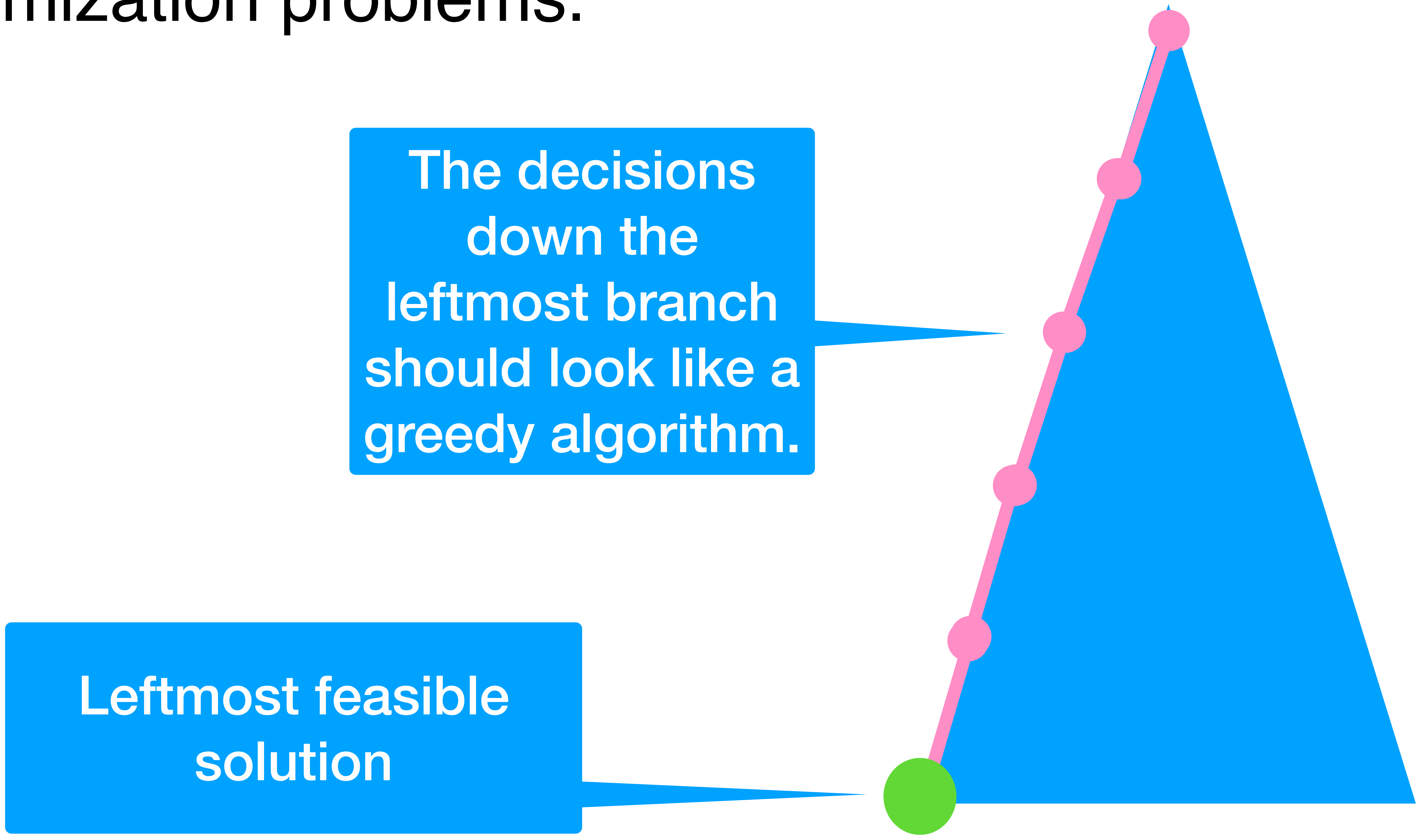
- Variable selection has an impact
  - on the size of the tree (because of propagation)
  - on the quality of solutions to a COP when timing out
  - on the depth where (good-quality) solutions are
- Value selection has an impact
  - on the size of the tree
  - on the quality of solutions to a COP when 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

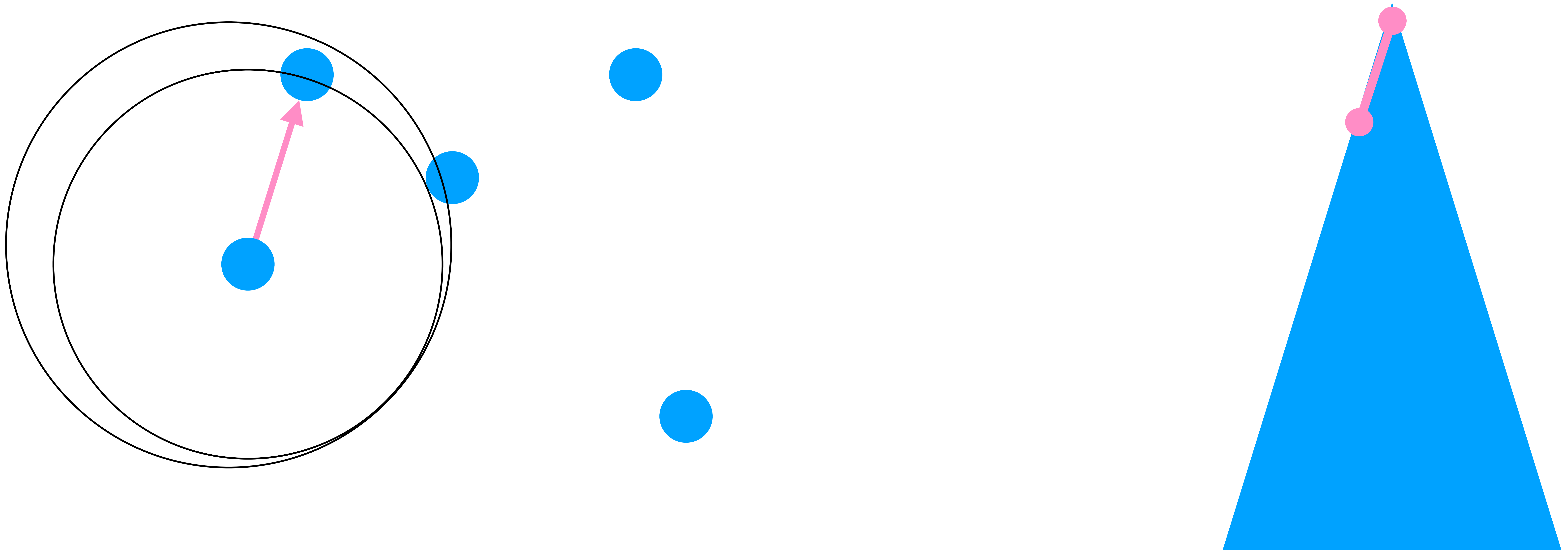
Value Selection for Optimization Problems

- 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.

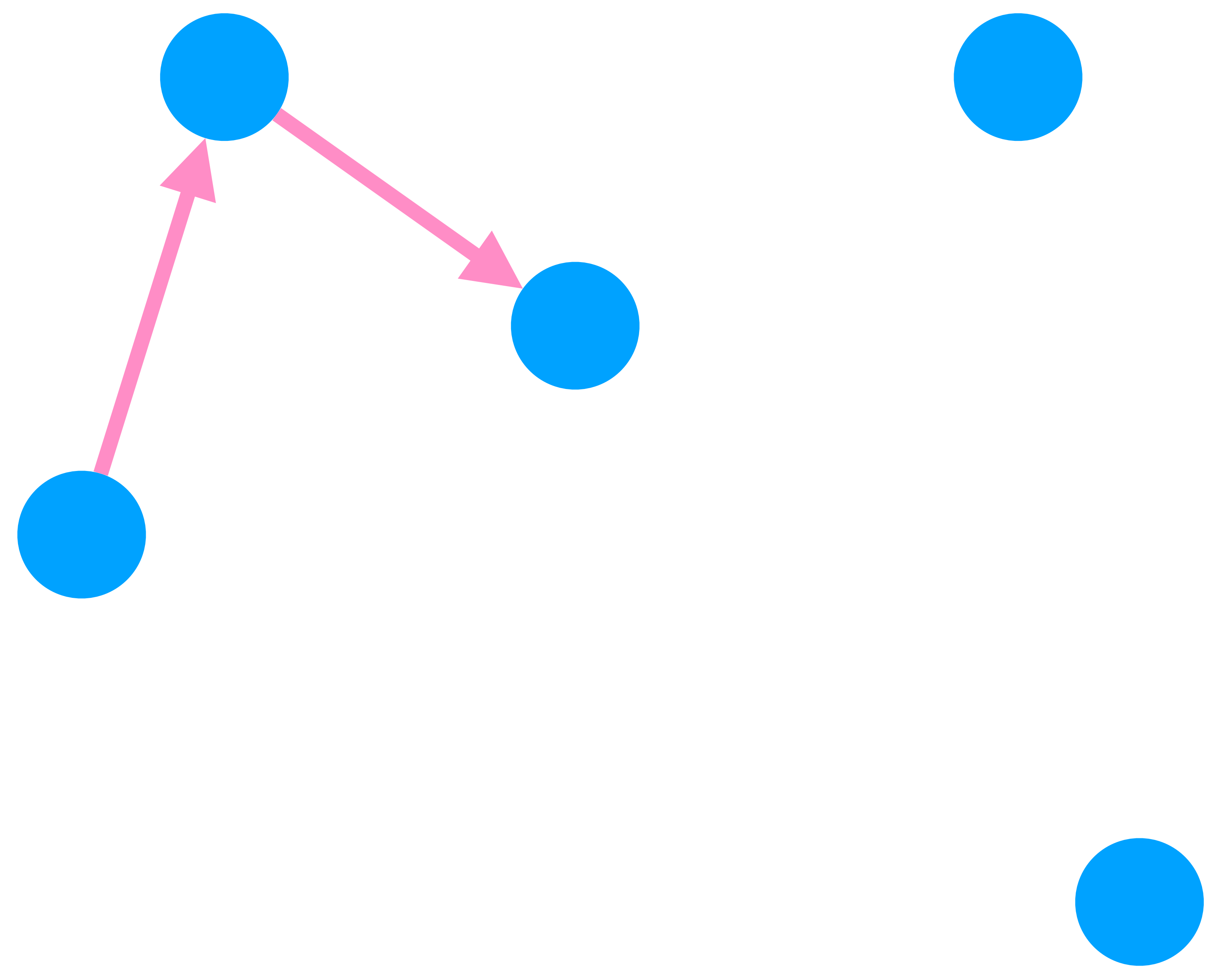

The decisions down the leftmost branch should look like a greedy algorithm.
Leftmost feasible solution

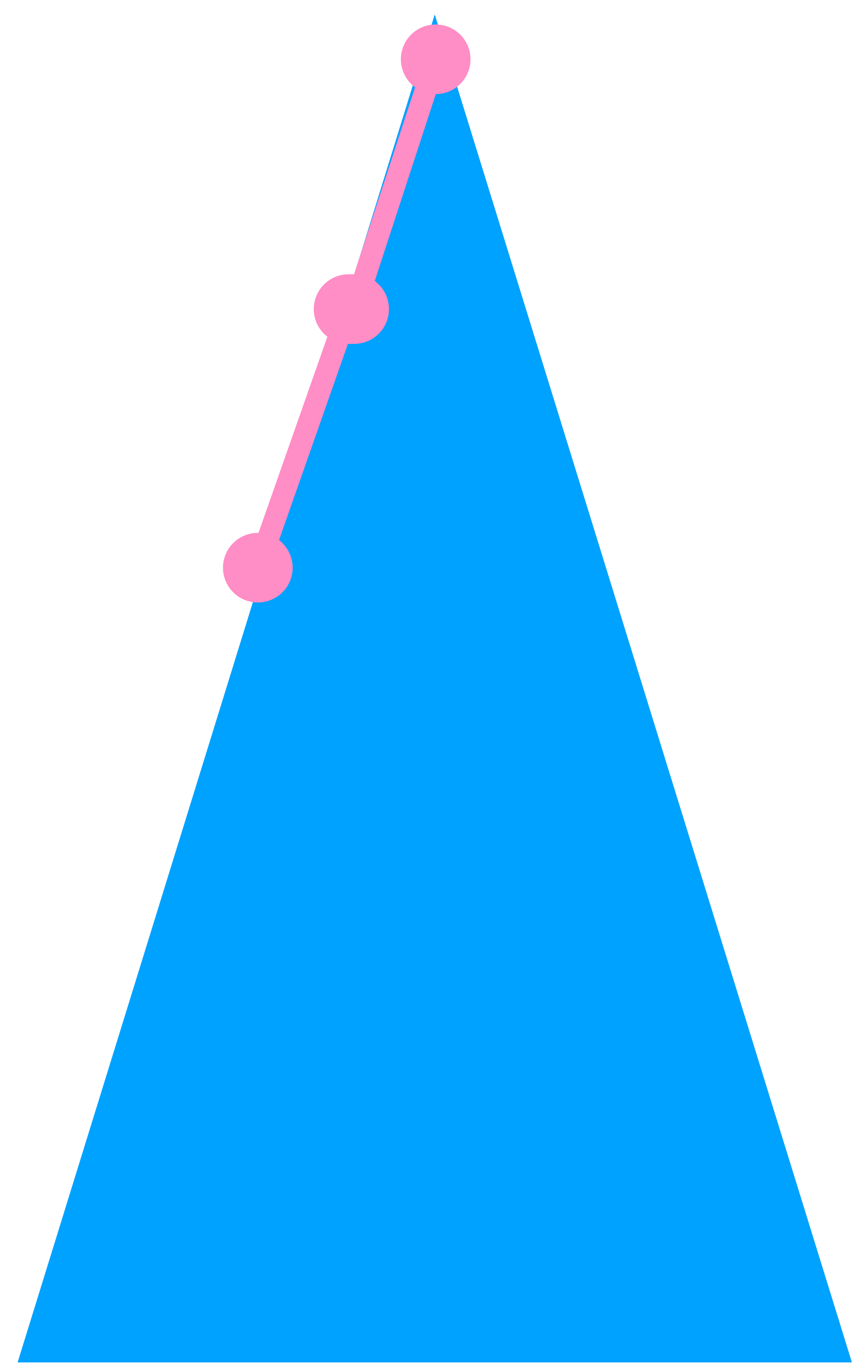
Example: Traveling Salesperson Problem

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

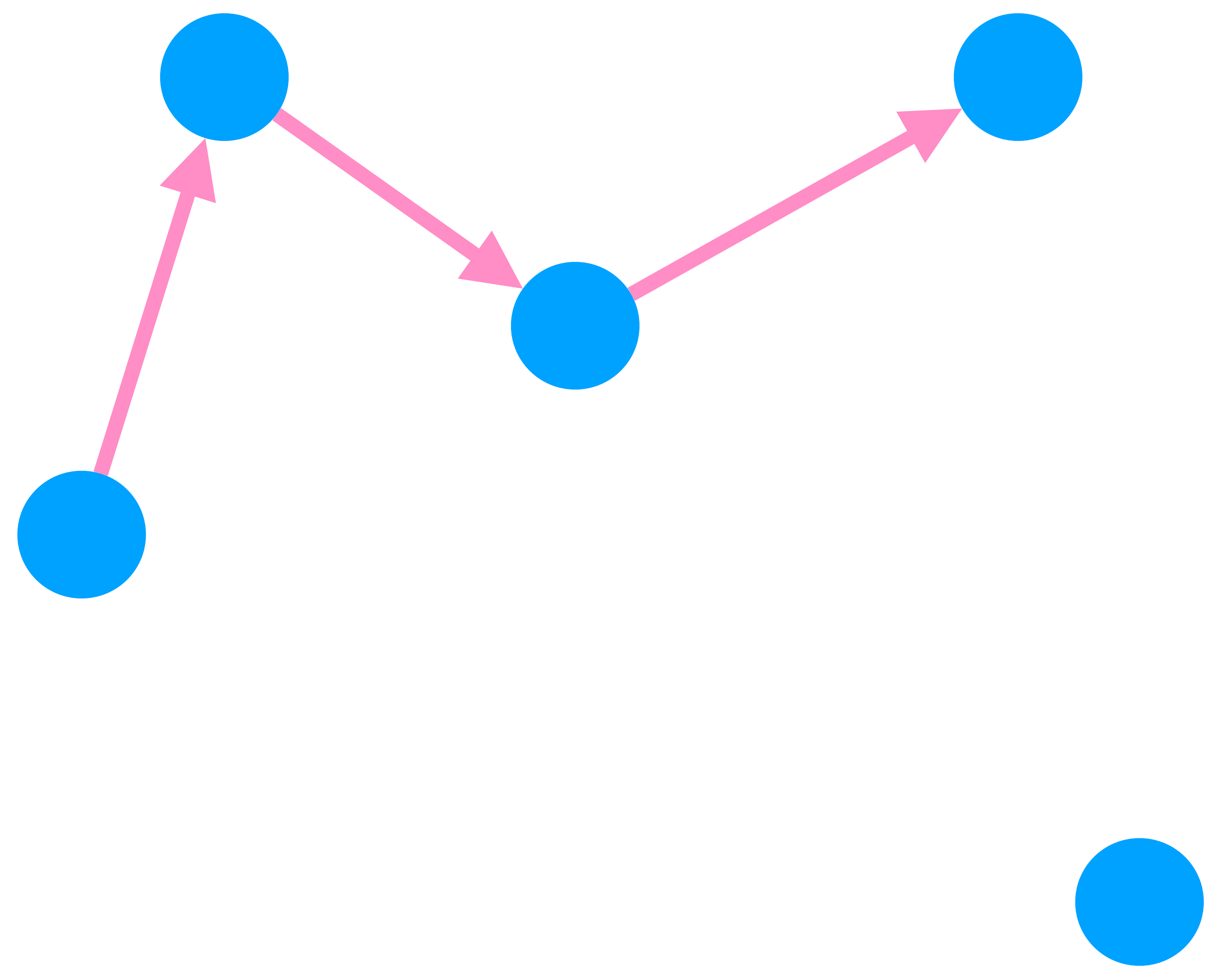


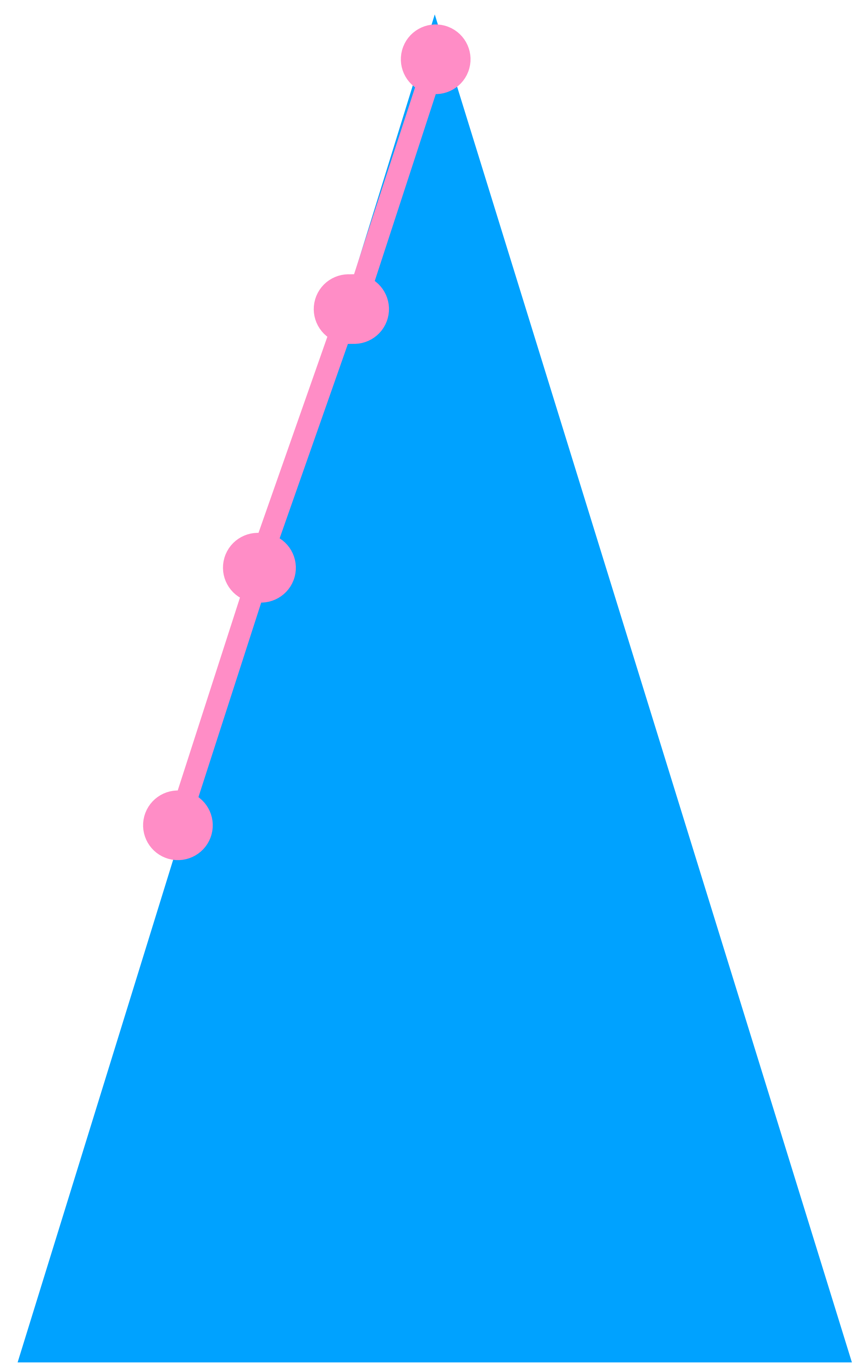
Example: Traveling Salesperson Problem



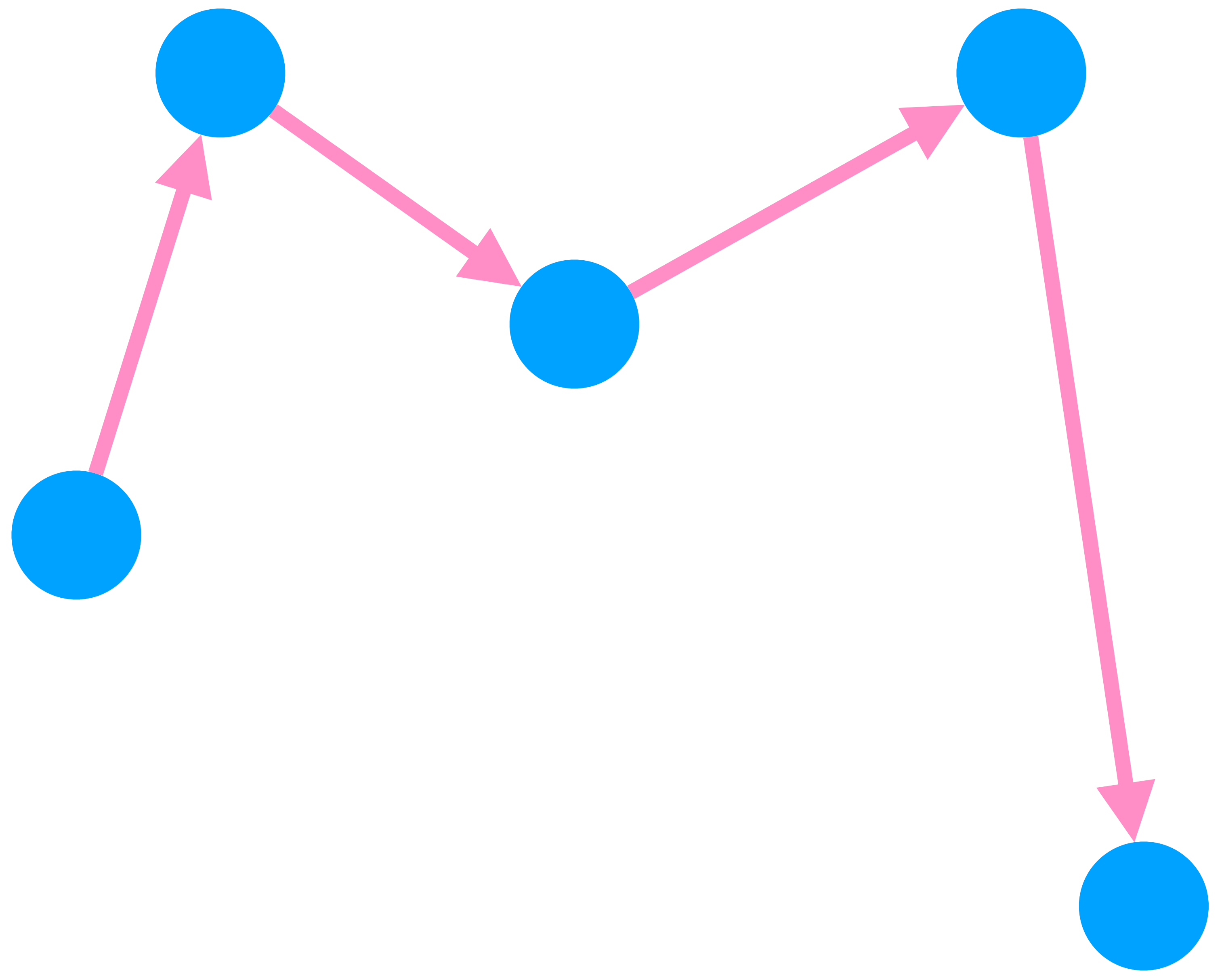


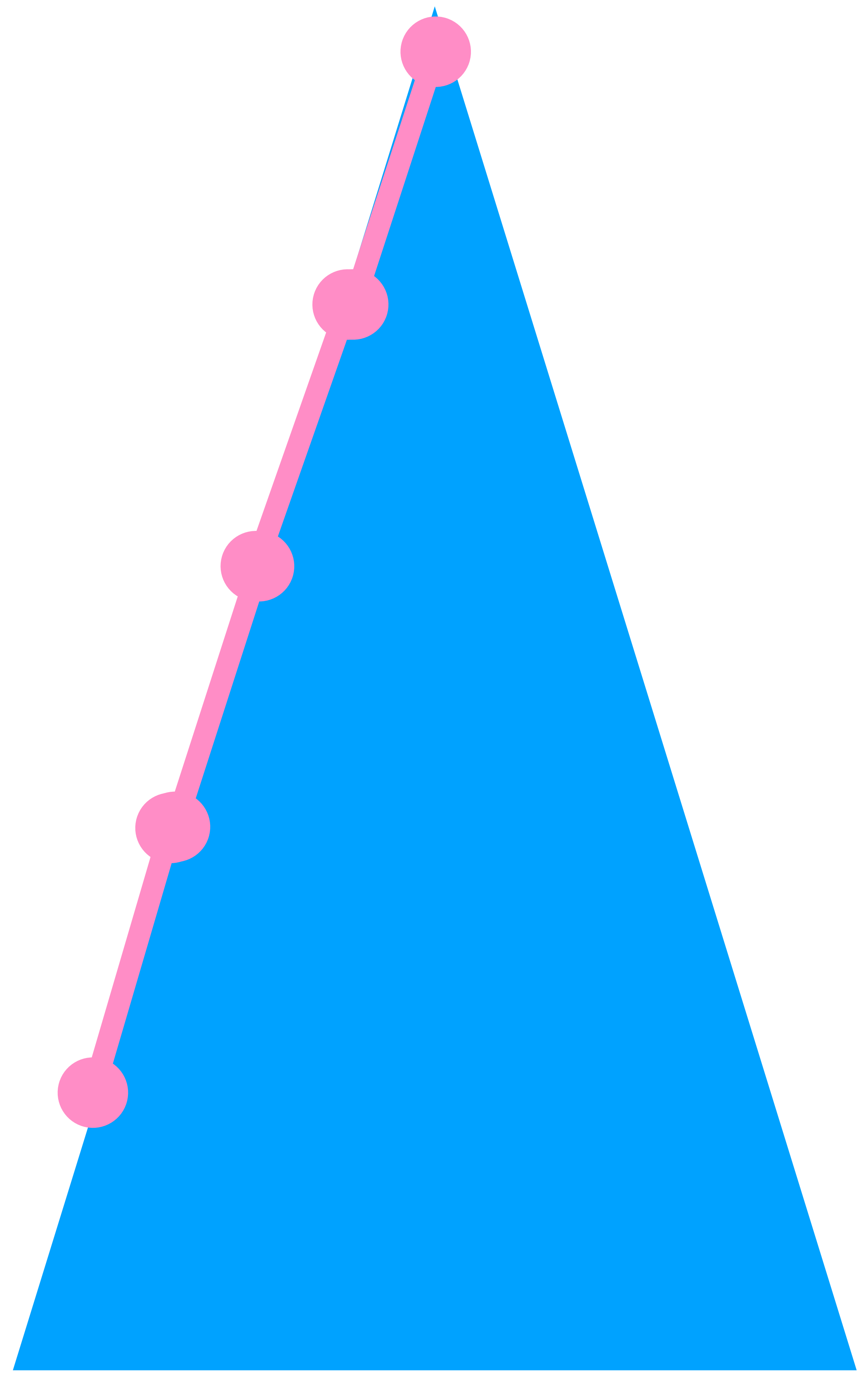
Example: Traveling Salesperson Problem





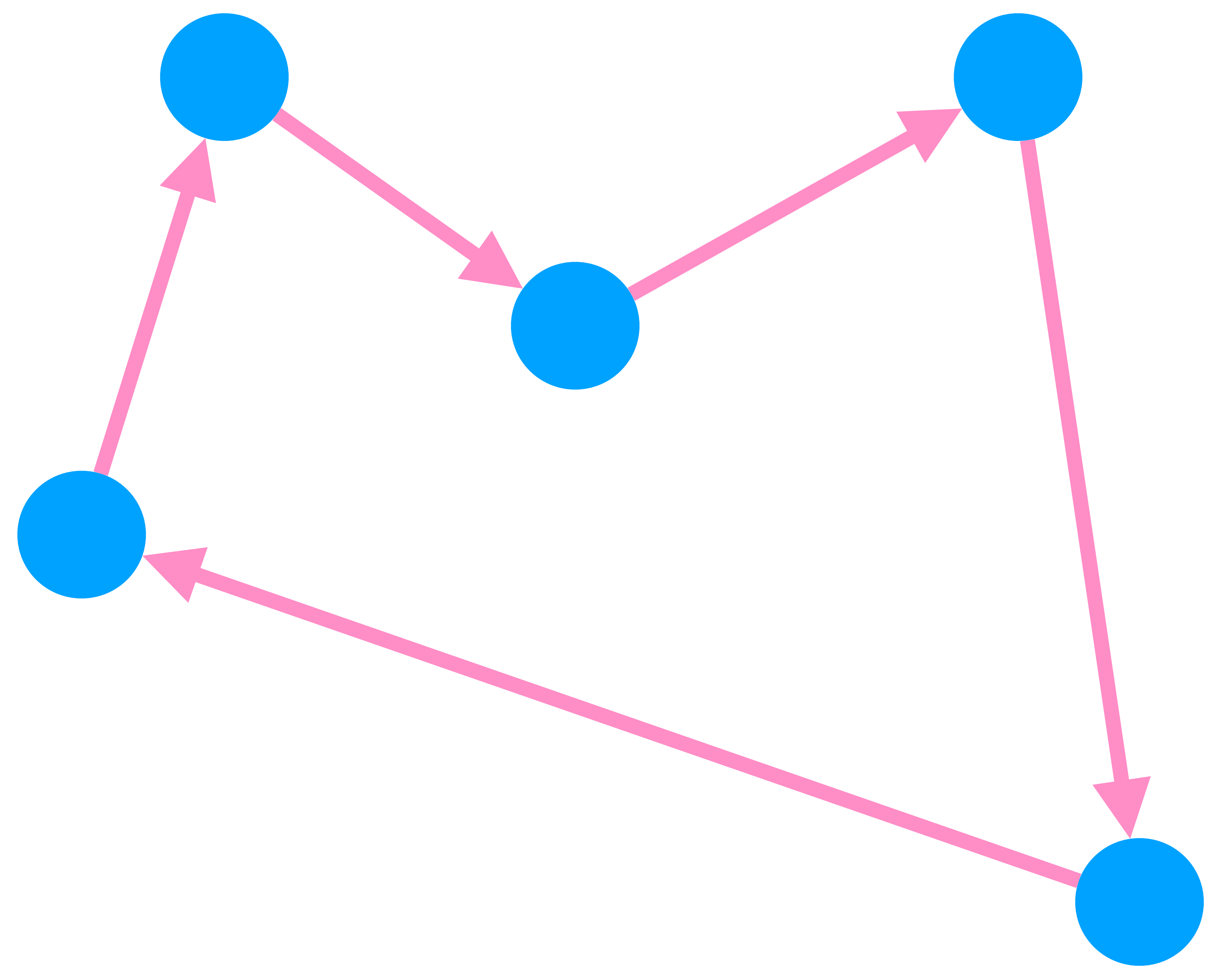
Example: Traveling Salesperson Problem

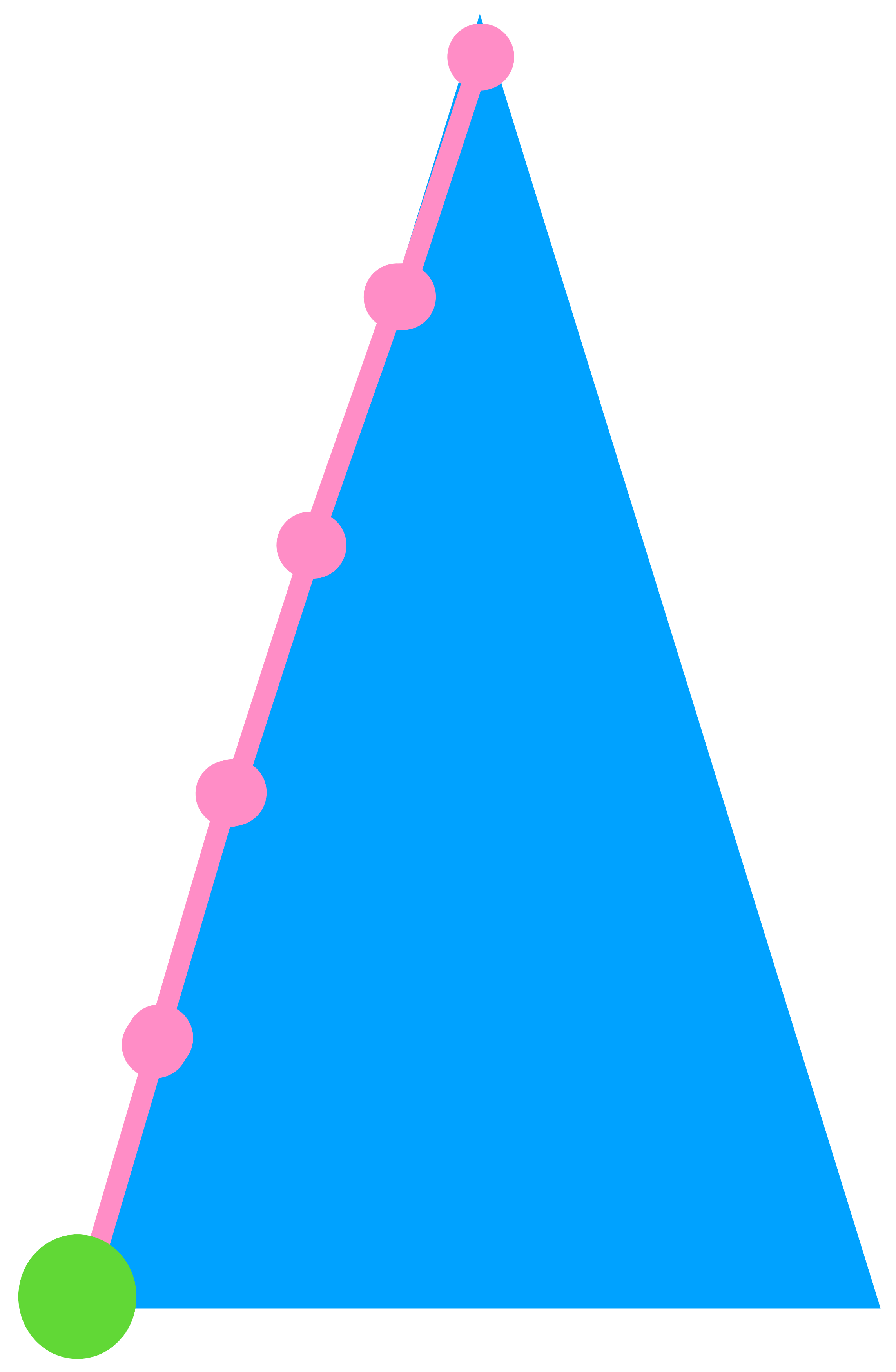




Example: Traveling Salesperson Problem

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





Making the First Solution Good!

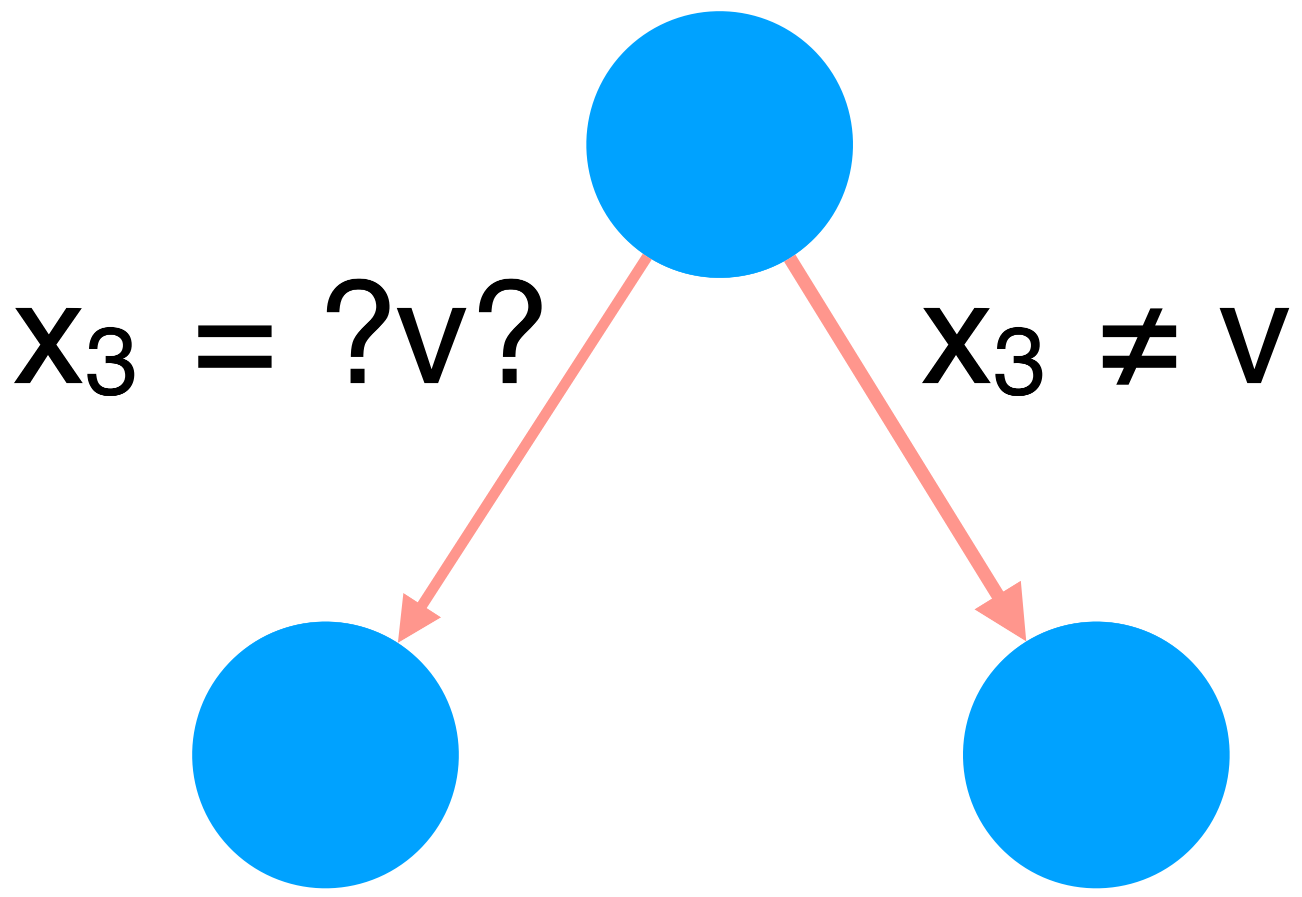
Branching = 2 steps (in general: not necessarily in this order):

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 $O = f(x_1,x_2,\ldots,x_{10})$ 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 {$v_1,v_2,v_3,v_4$}
        sm.saveState()
        cp.post($x_3$ = v)
        if min(O) < bestObj:
                best = v
                bestObj = min(O)
        sm.restoreState()
branch {cp.post($x_3$ = best)}
          {cp.post($x_3$ != 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.

One can easily implement a default search doing phase saving.

```
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)));
        }
    };
}
```

```
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

Branching

Reminder:

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

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

First-Fail Principle

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.

First-Fail Principle for Variable Selection

- 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(x1) = {1,2} and D(x2) = {1,…,100} branching on x1 first is likely to trigger more propagation.

Min-Dom Heuristic

```
 int n = 8;
Solver cp = Factory.makeSolver(false);
IntVar[] q = Factory.makeIntVarArray(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));
    }
});


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

Select an unfixed variable with the smallest domain size.

Variable Selection

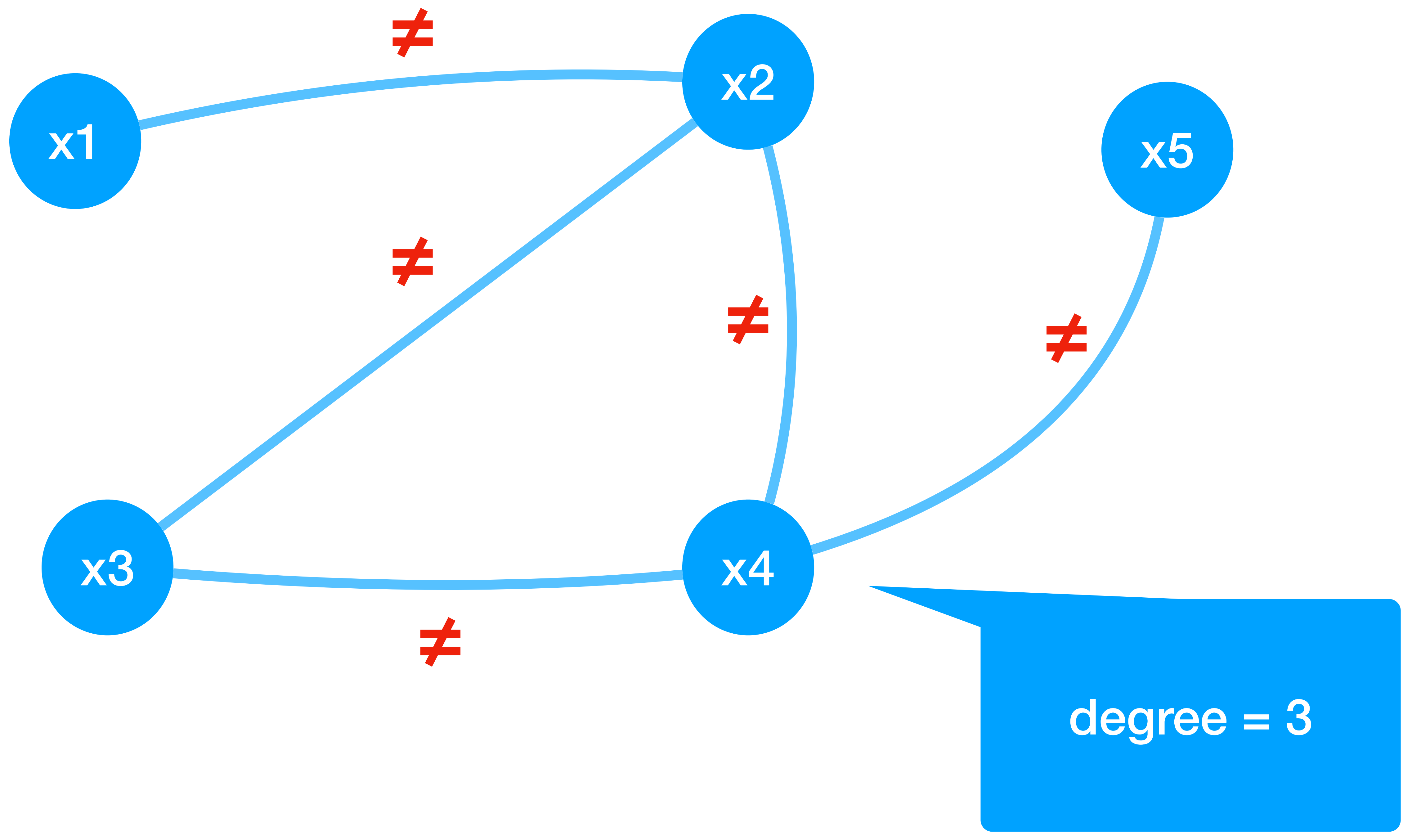
FF based on the degrees in the constraint graph

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.

Dom+Deg and Dom/Deg

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



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).
  - The idea is that if a constraint has failed a lot in the past, then it will probably continue this trend in the future.
- Weighted degree of a variable x = the sum of the weights of the constraints on x and at least one other unfixed variable.
- Dom/Wdeg: Select an unfixed variable with the smallest ratio between domain size and weighted degree.
- Weakness: A constraint that fails may not be the only guilty one for the failure (cascade of propagations in the fixpoint algorithm).

Dom/Wdeg Implementation

- Not shipped with MiniCP, but quite easy to implement.
- Each constraint should be aware of the variables in its scope.
- Add a method called `scope` to the class `Constraint`:
  it returns the list of variables in the scope of the constraint.

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)

$$\begin{tikzcd}
0 \arrow[d, "c_1"] & P_0 = \langle X, D, C \rangle \\
1 \arrow[d, dotted] & P_1 = \langle X, D_1, C \cup \{c_1\} \rangle \\
k{-}1 \arrow[d, "c_k : x = a"] & P_{k-1} = \langle X, D_{k-1}, C_{k-1} \rangle \\
k & P_k = \langle X, D_k, C_k \rangle
\end{tikzcd}$$

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

= 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 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)

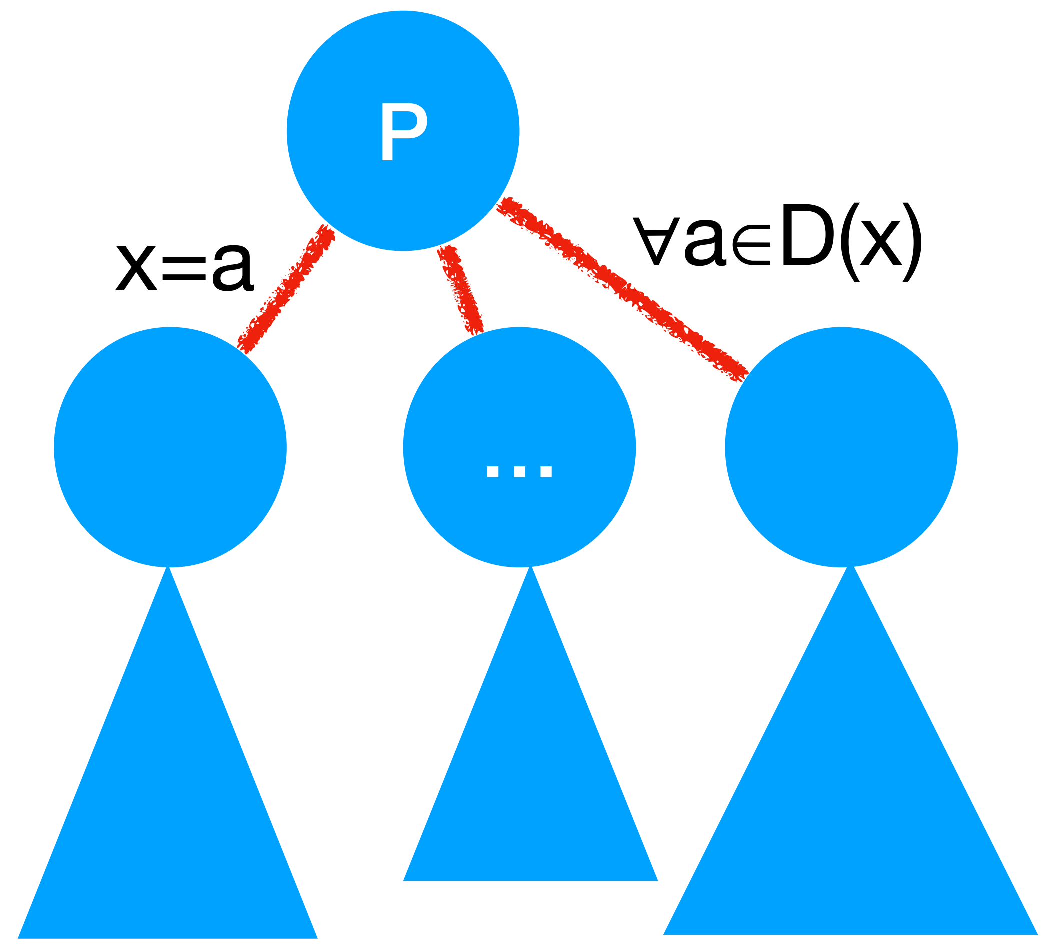
- Estimation of the size of the search tree when labeling the variable x:

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

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*

Activity-Based Search (ABS)

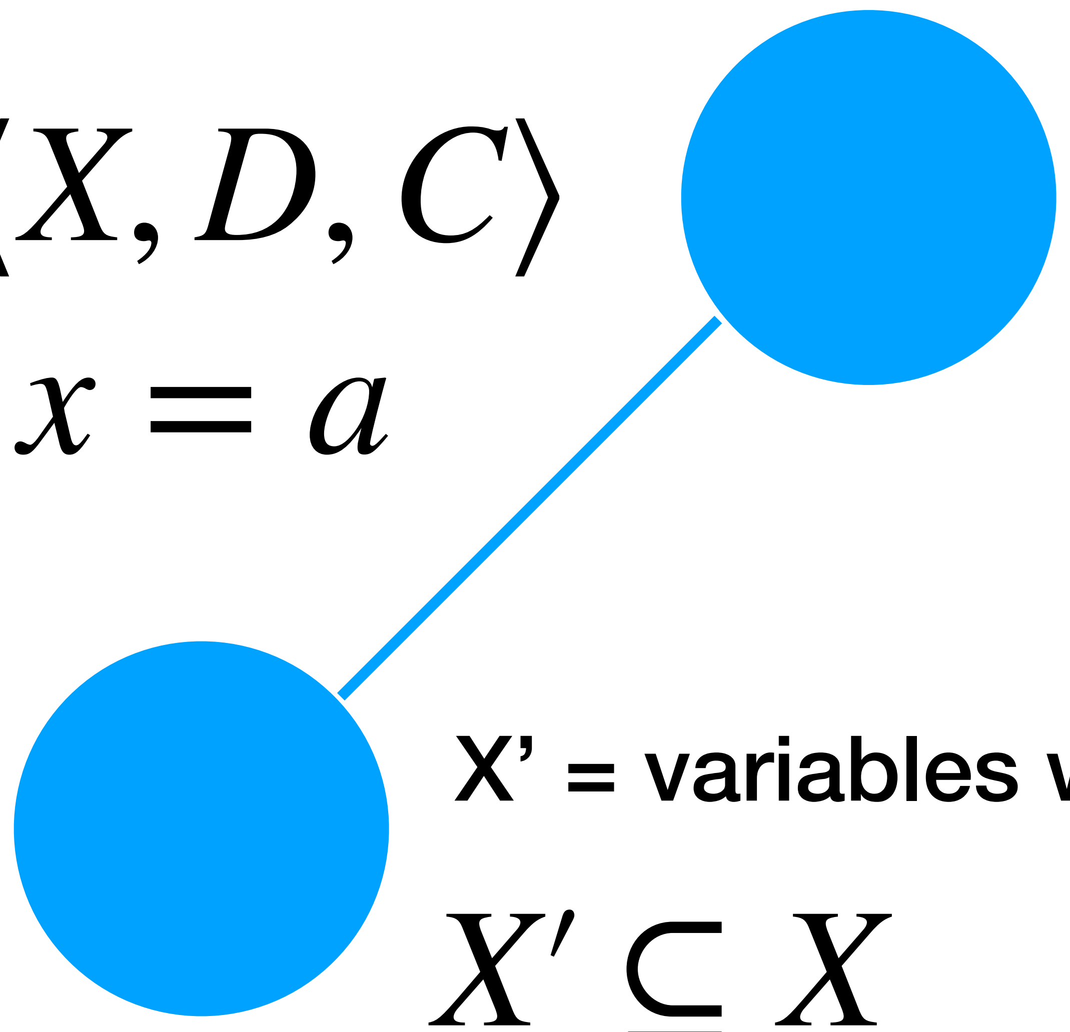
- Idea: Track how often a variable domain is contracted during search, as this allows some learning.
- We expect that a variable whose domain is often contracted is a good candidate to branch on early, in order to reduce the depth of the search tree.

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:

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

$$x = a$$



X' = variables with contracted domain

$$X' \subseteq X$$

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

decay factor in [0..1]

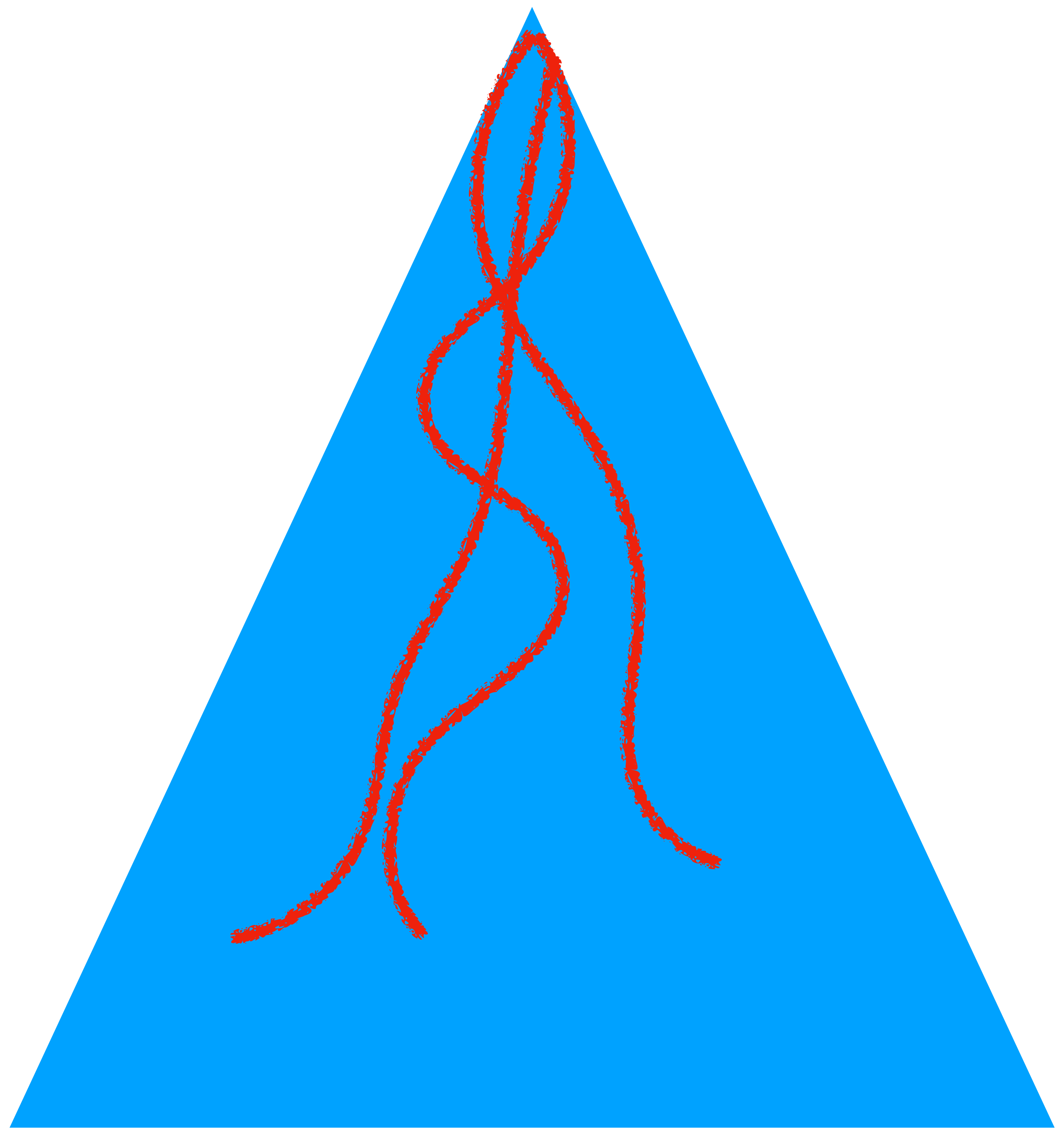
$$\forall x \in X \textbf{ where } |D(x)| > 1 : A(x) := A(x) \cdot \gamma$$

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

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

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.

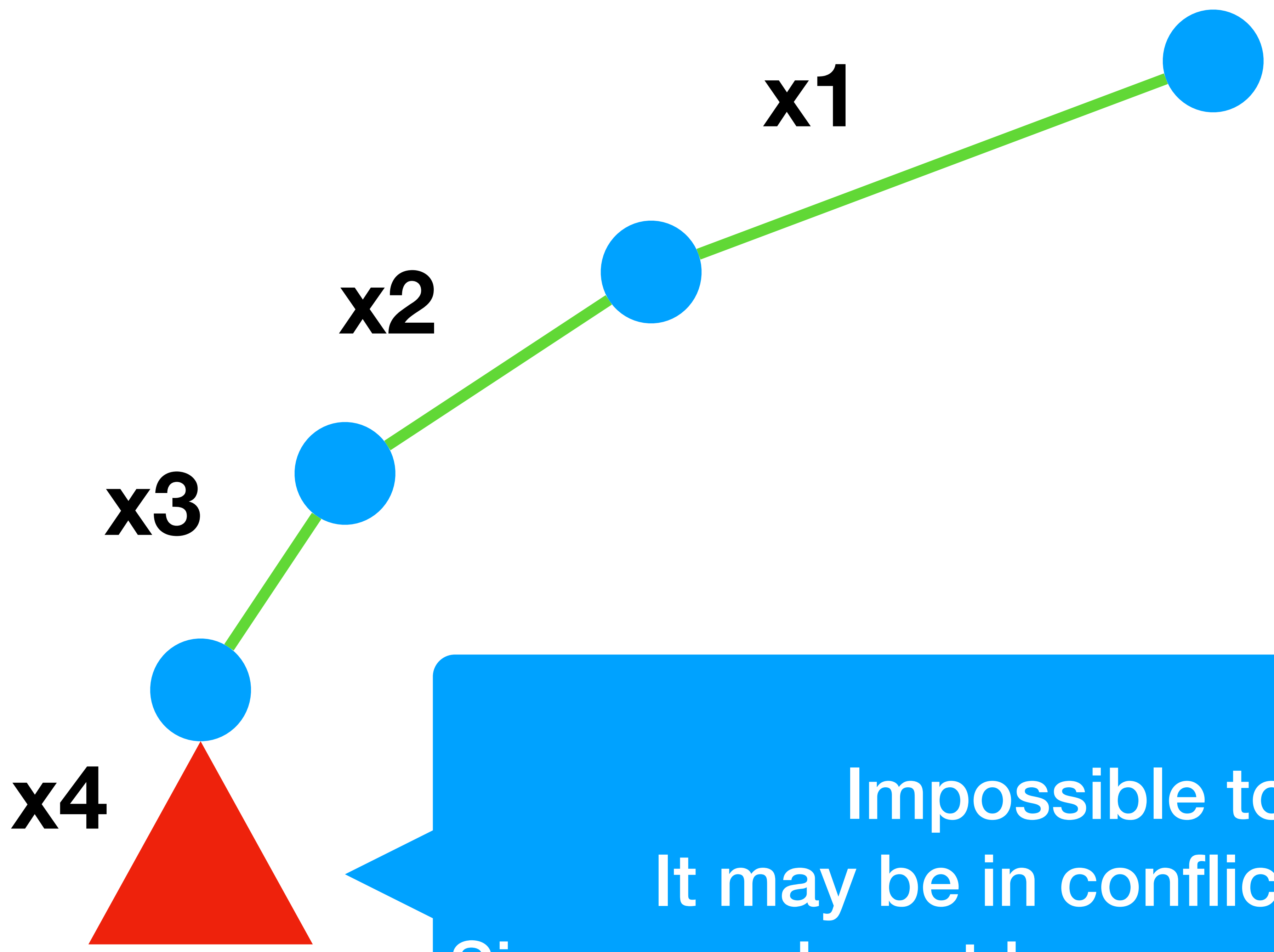
```
nextVarToBranchOn(X) {
   if (lastConflictVariable == null) {
      return fallBackHeuristic(X)
   } else {
      return lastConflictVariable
   }
}
```

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.

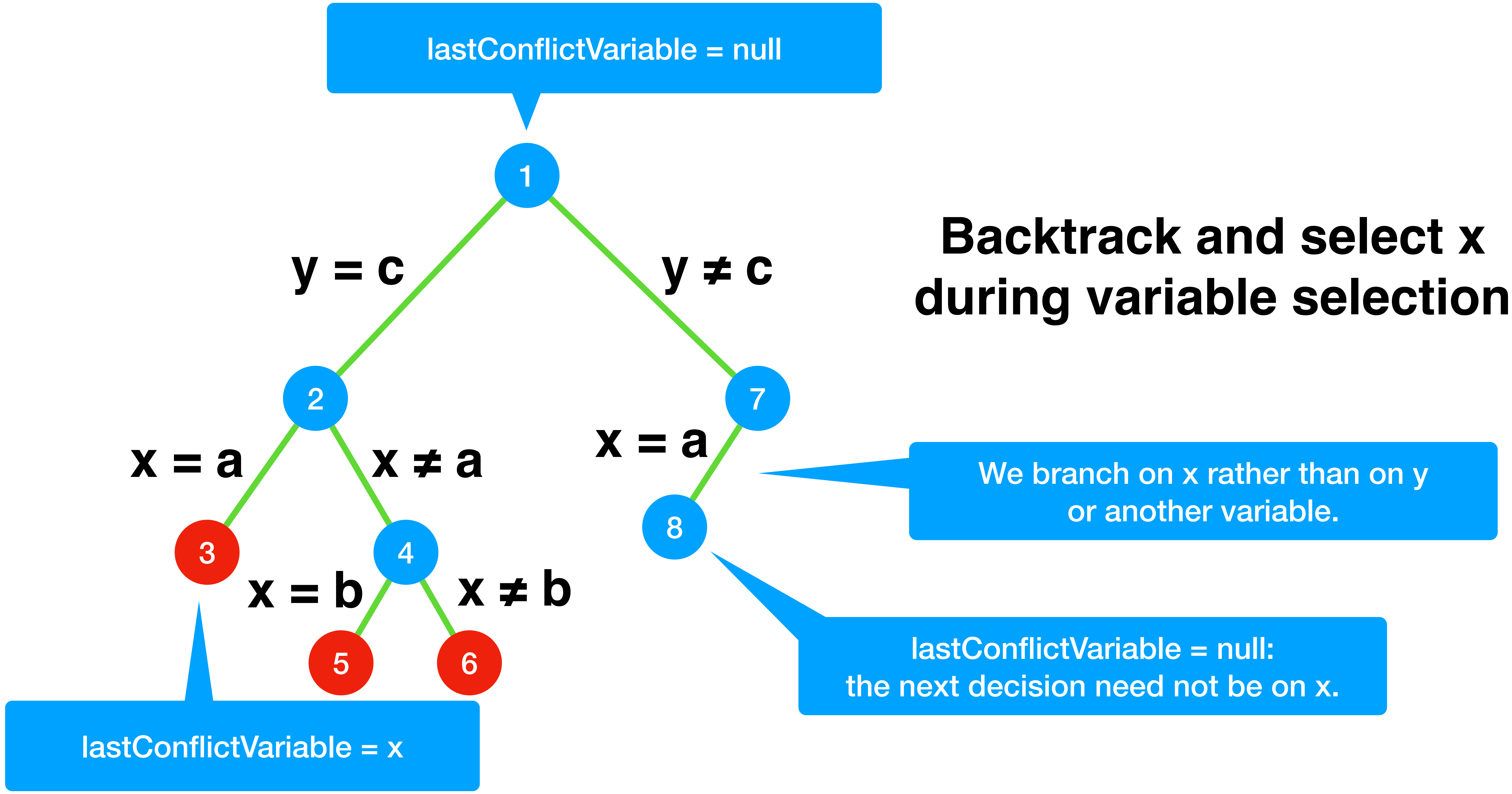
Last-Conflict Search

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



Impossible to fix x4 to some value. Why?
It may be in conflict with the choices on x3, x2, or x1.
Since we do not know, we should always try to branch on x4 first,
also after backtracking, as long as this conflict persists.

Last-Conflict Search: Example



lastConflictVariable = null
1
y = c
y ≠ c
2
7
x = a
x ≠ a
x = a
3
4
8
x = b
x ≠ b
5
6
lastConflictVariable = x
We branch on x rather than on y or another variable.
lastConflictVariable = null: the next decision need not be on x.

Backtrack and select x during variable selection

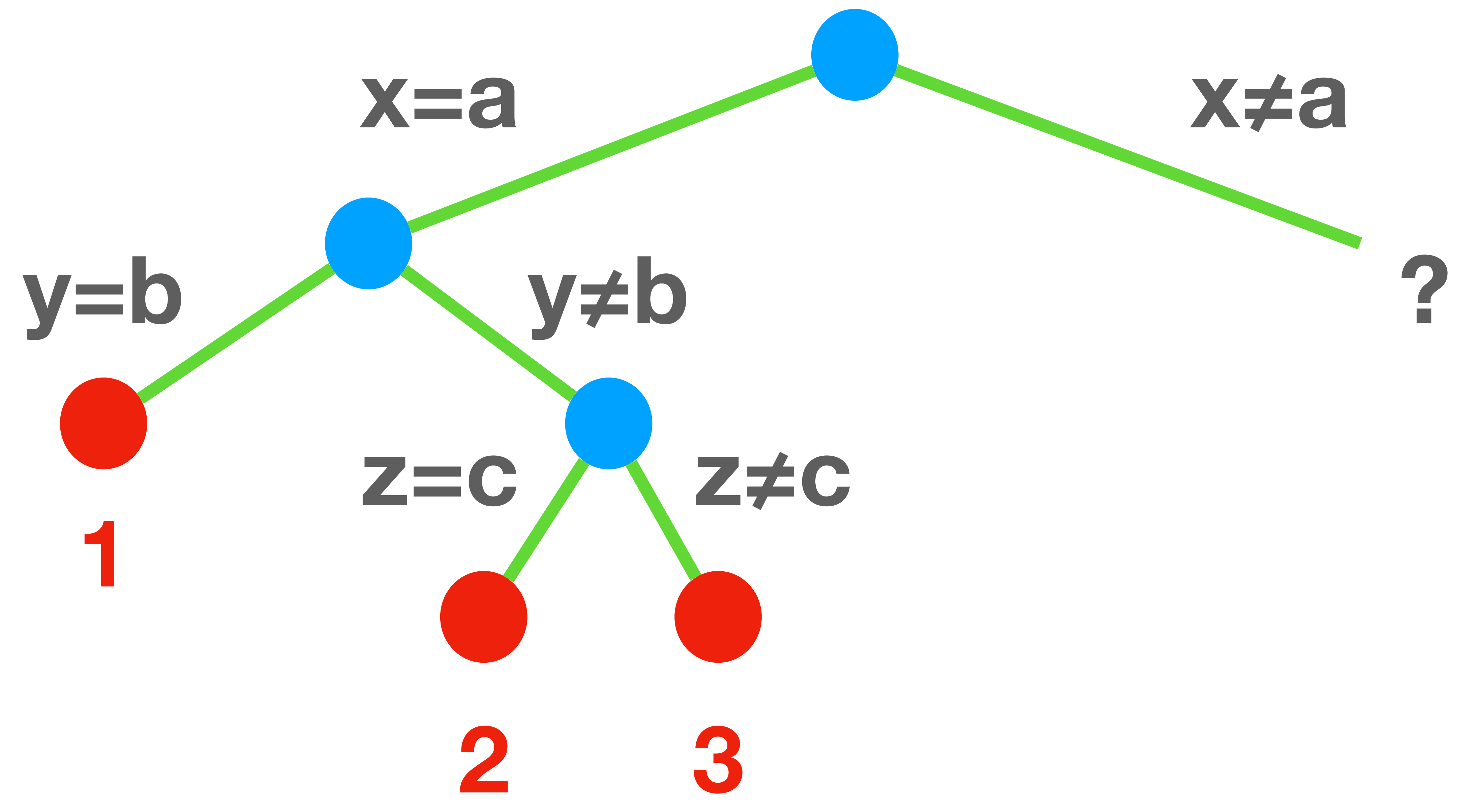
Conflict-Ordering Search (COS)

- 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.



Assume y≠b
leads to fixing y

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

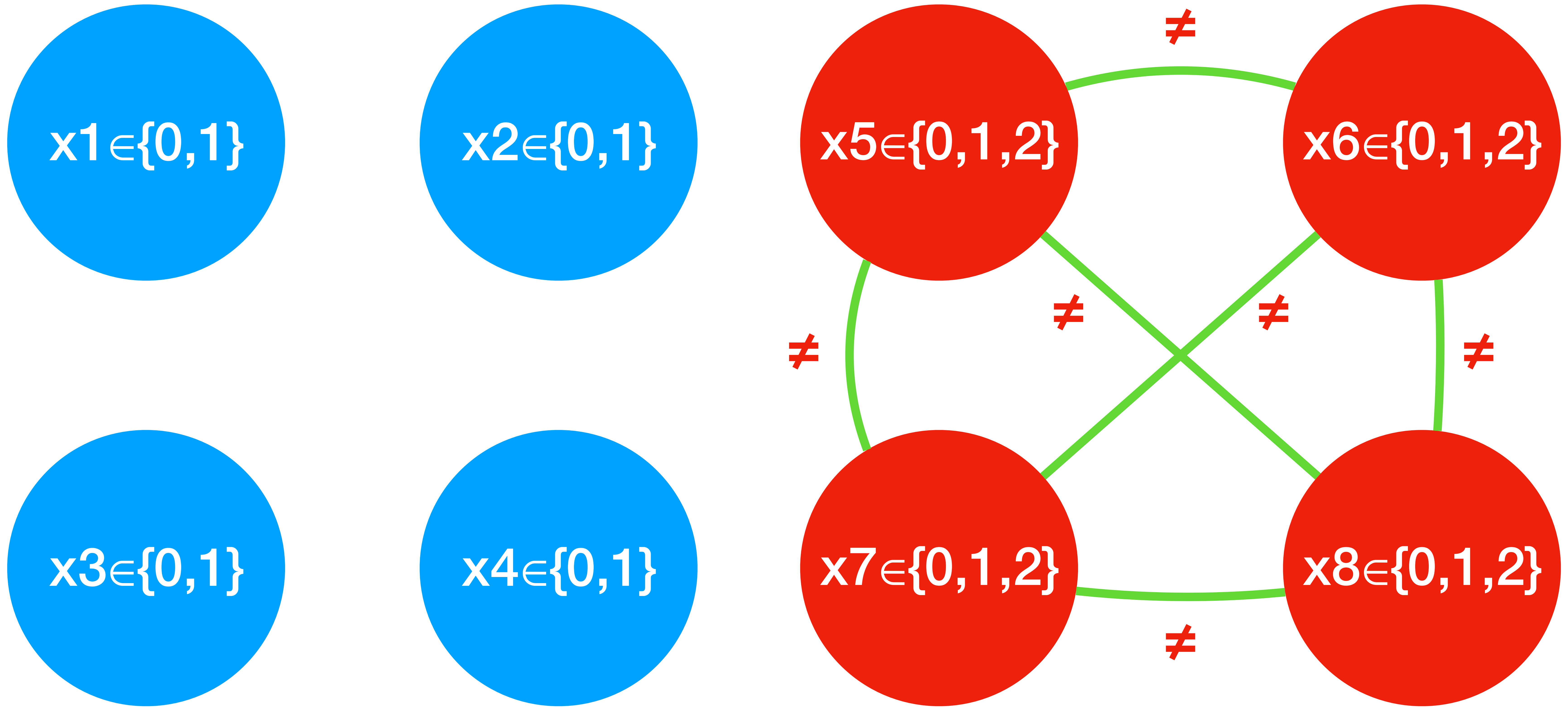
```
1  $P \leftarrow \phi(P)$
2  if $P = \perp$ then
3      if $\texttt{lastVar} \neq null$ then
4          $\texttt{nConflicts} \leftarrow \texttt{nConflicts} + 1$
5          $\texttt{stamp}[\texttt{lastVar}] \leftarrow \texttt{nConflicts}$
6      return false
7  if $\forall x \in \mathcal{X}, |dom(x)|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

Failure: update the timestamp.

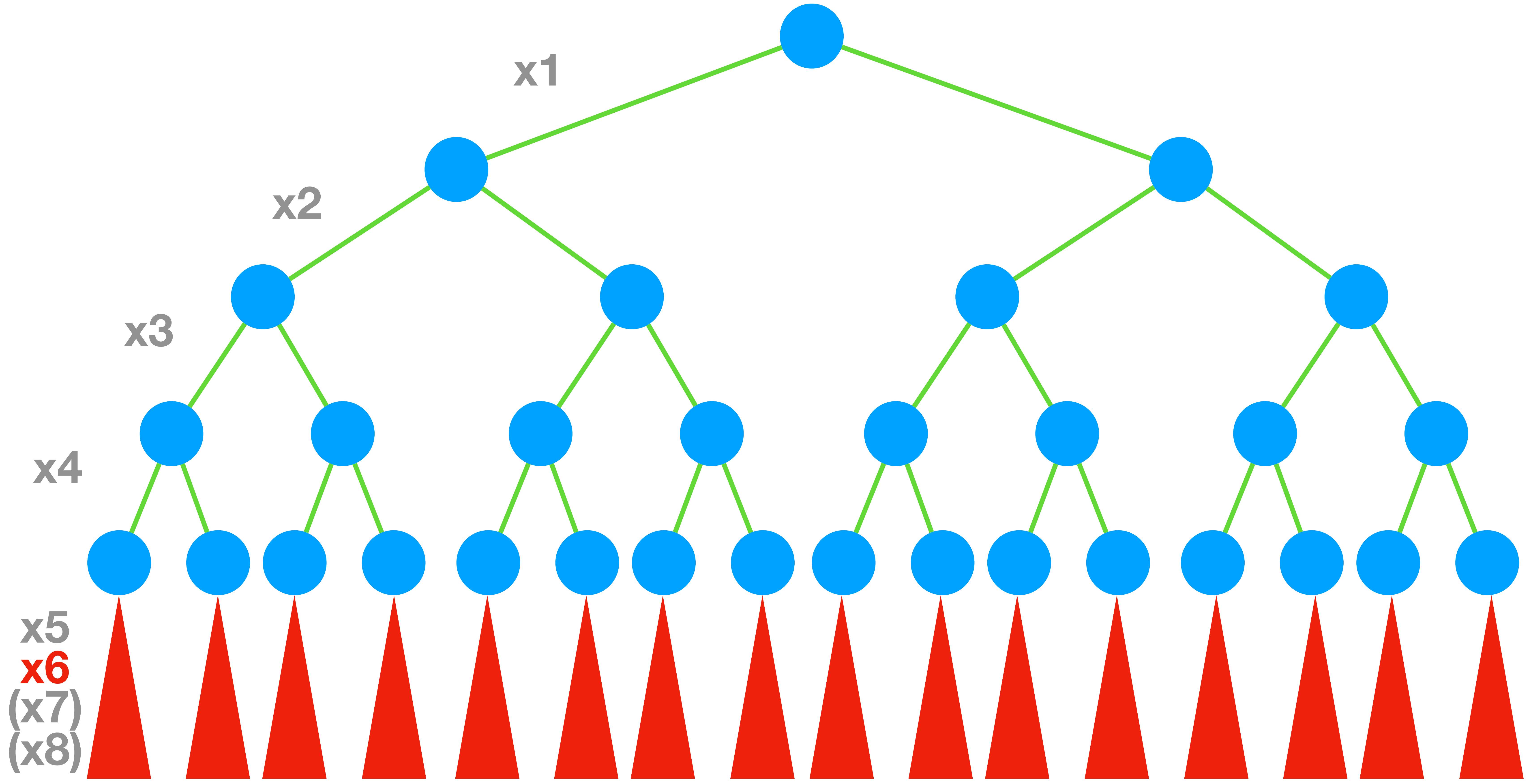
Failure: select the variable with the largest timestamp.

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 x1, …, x8.

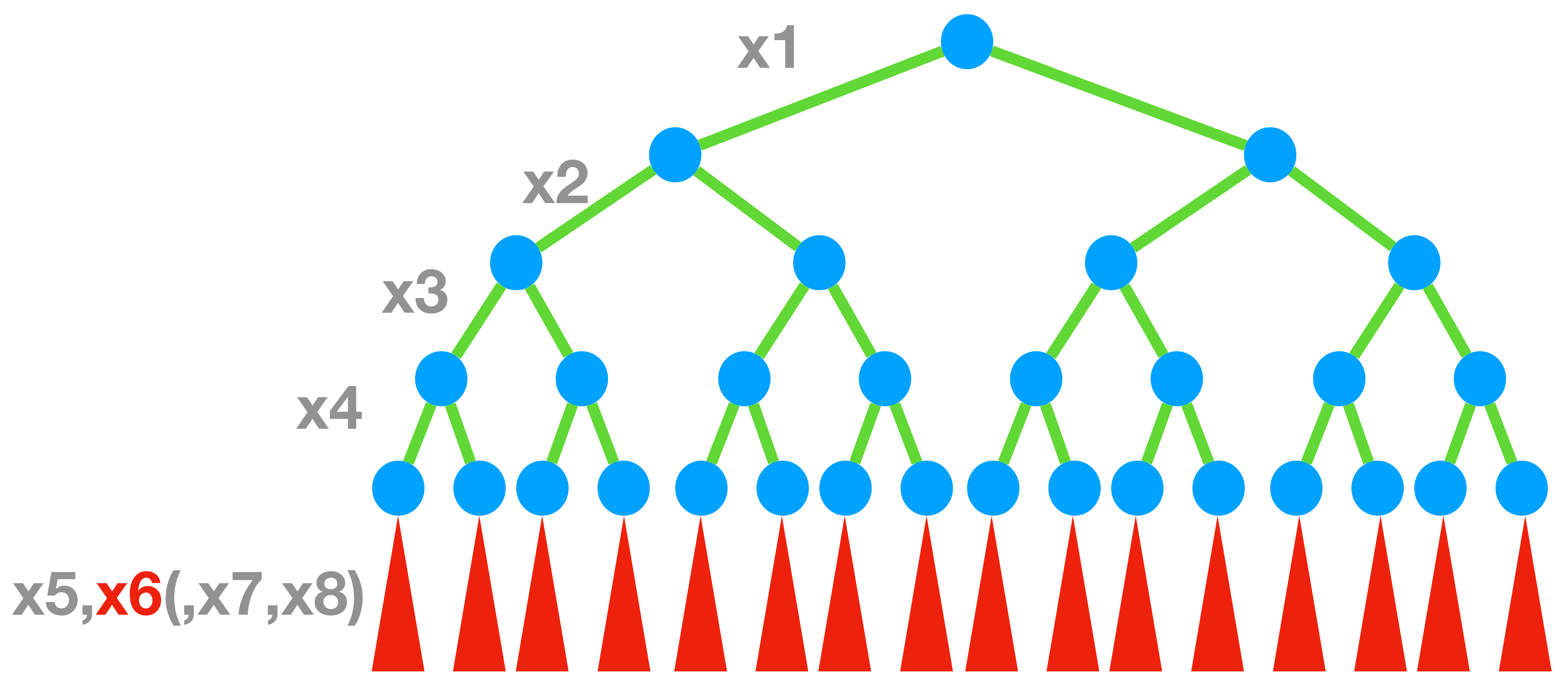


Example: Search Tree



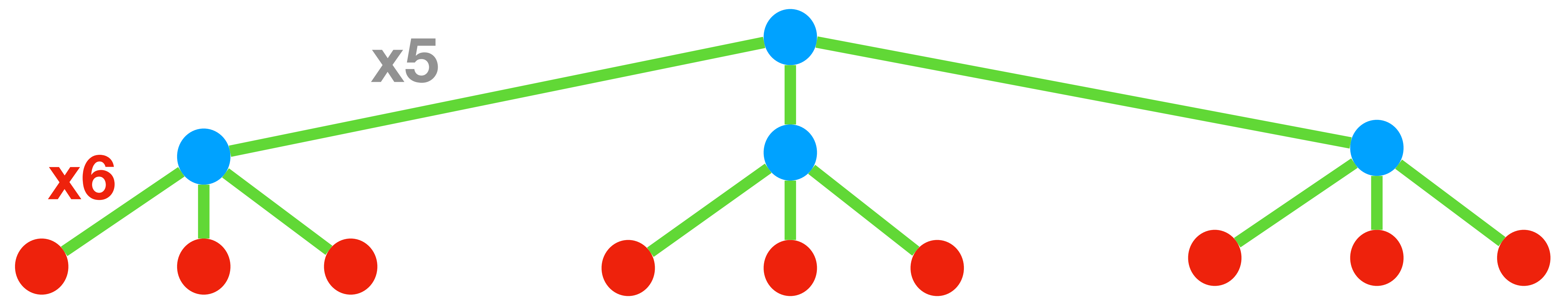
The problem is that there is a subset of variables, {x5,x6,x7,x8}, causing the conflict. Ideally we should detect this earlier in the search tree.

Example: Search Trees with Two Static Orders

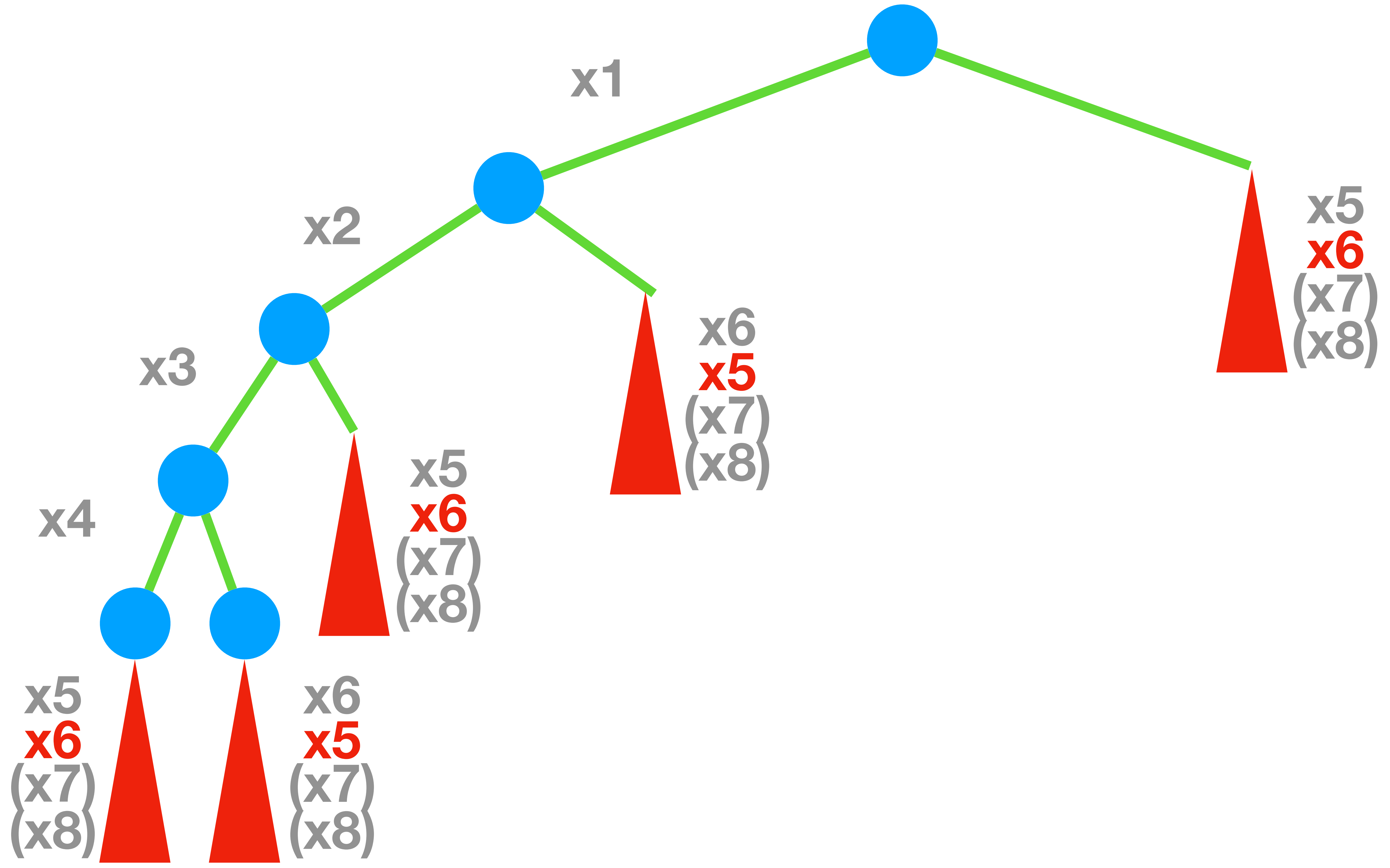


Search tree obtained with the static order [x1,x2,x3,x4,x5,x6,x7,x8].

Search tree obtained with the static order [x5,x6,x7,x8,x1,x2,x3,x4]: ideally, a heuristic learns this order.



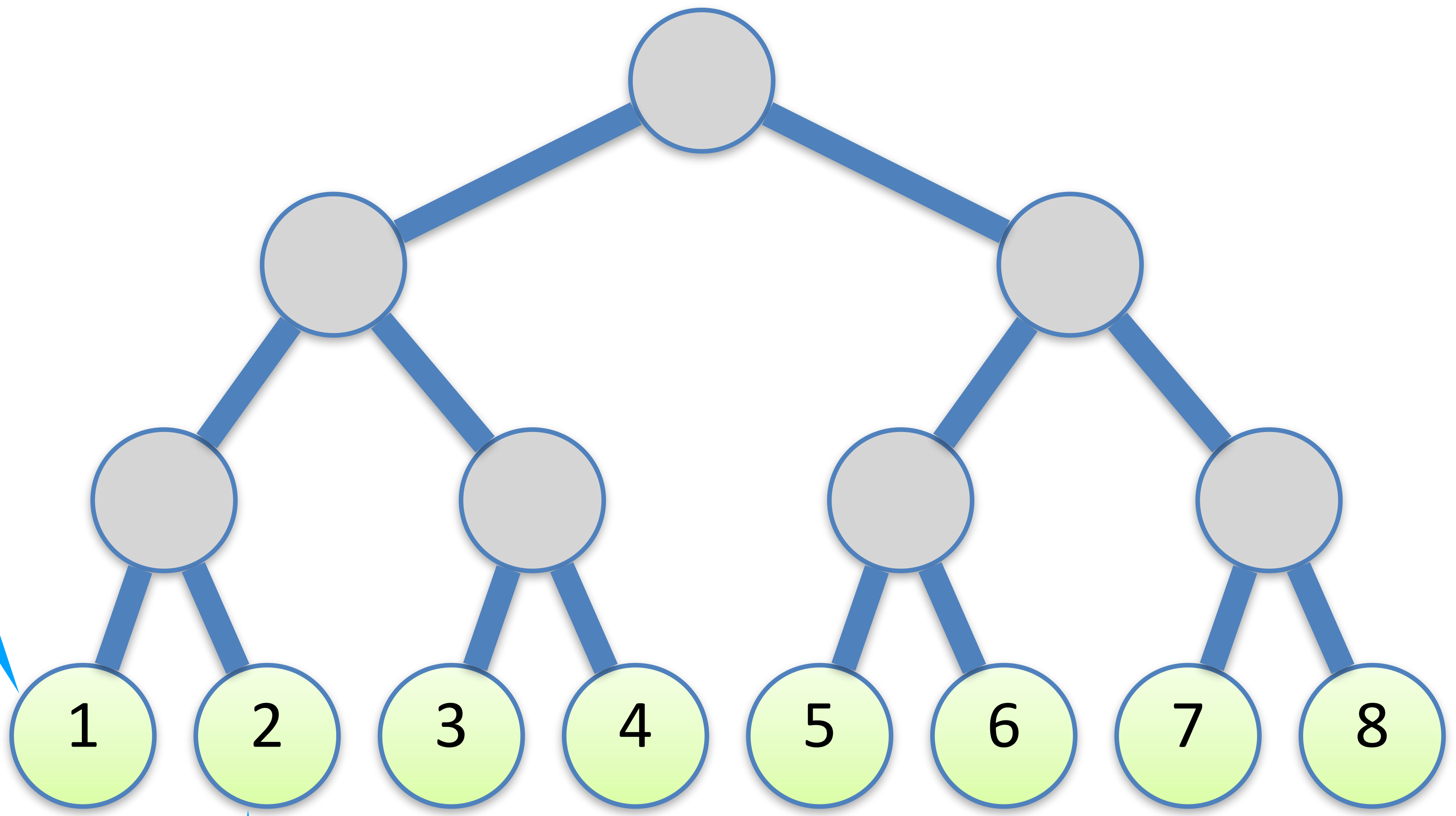
Example: Search Tree with COS



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.

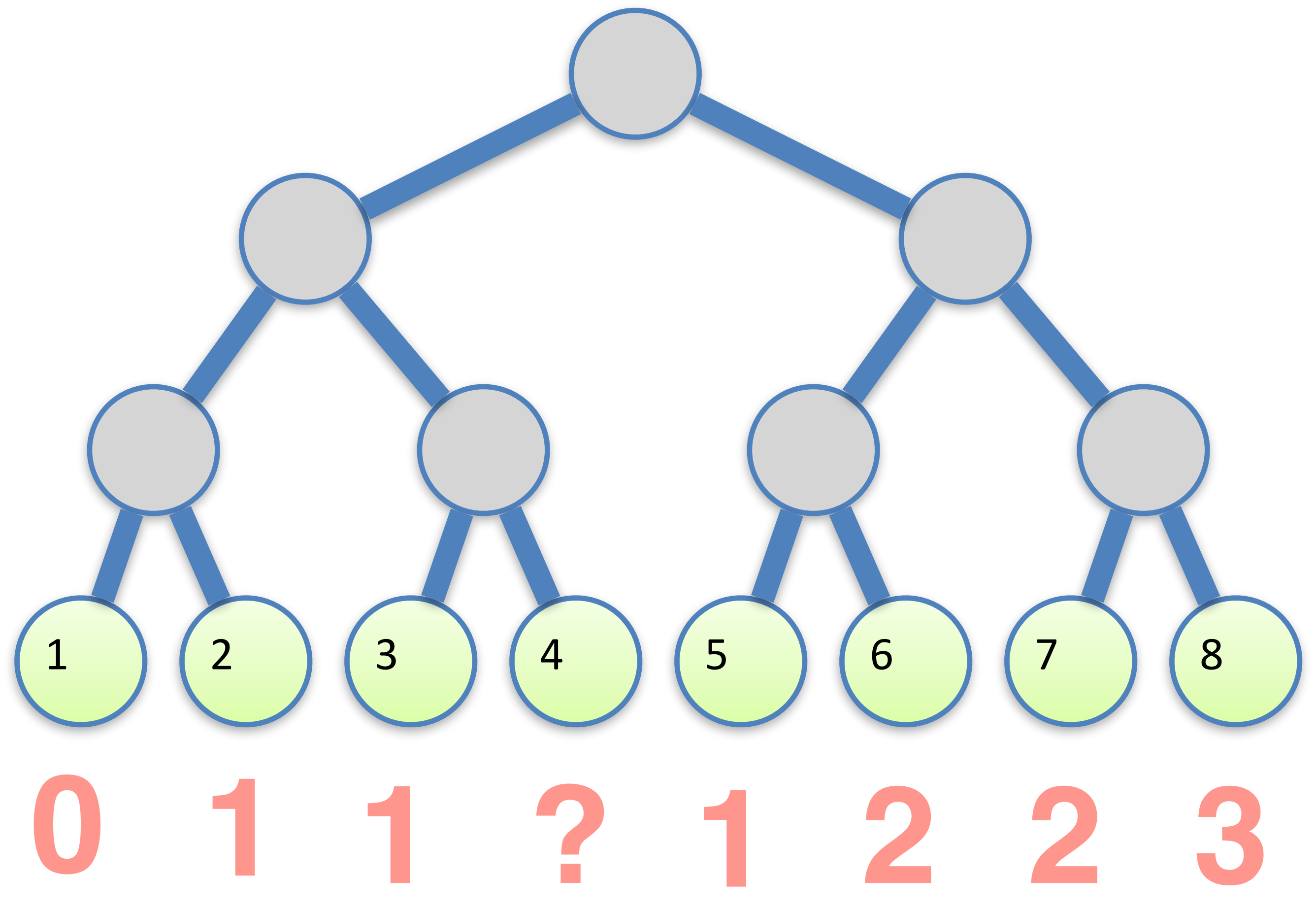
But is this one supposed to be better than…

…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

```java
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]

TBD with Laurent

-LNS, Restarts and Heavy Tails for feasibility

Search Wrap up

Search: Summary

- 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