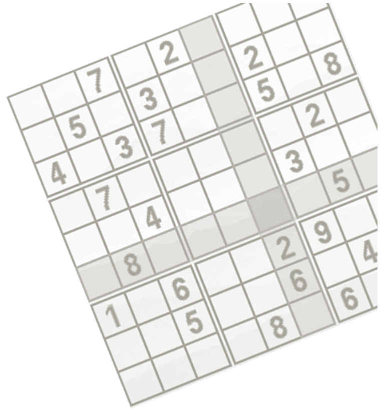
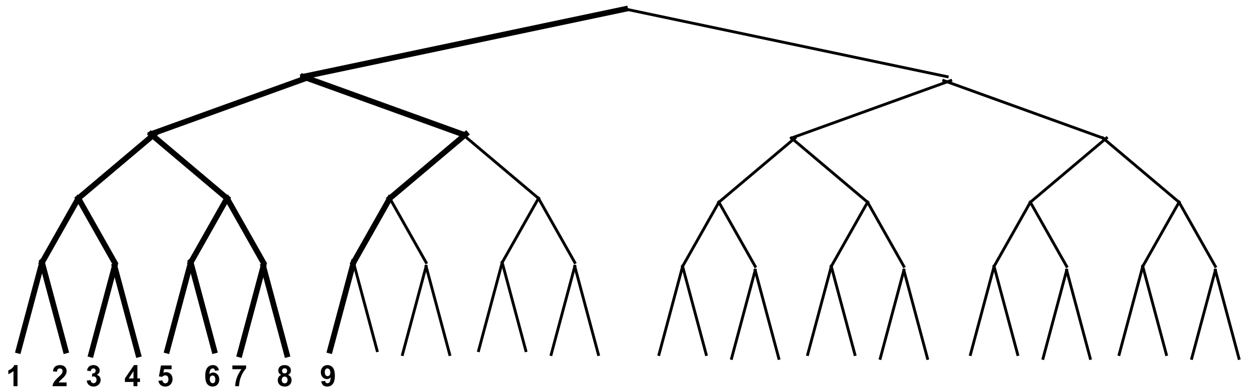


# Constraint Programming

**Roman Barták**
Department of Theoretical Computer Science and Mathematical Logic

**Search and Optimization Techniques**

Let us go back to foundations: DFS = Depth First Search




**Observation:**

Real-life problems have huge states spaces that cannot be fully explored.

**We can explore just part of the state space!**

↳ **incomplete tree search**

**Do not explore the state space completely.**

- Do not guarantee proving that there is no solution and they do not guarantee finding a solution (completeness)
  - sometimes completeness can be guaranteed with time-complexity trade-off

For many problems, incomplete techniques **find solutions faster.**

Frequently **based on a complete search algorithm** such as DFS.

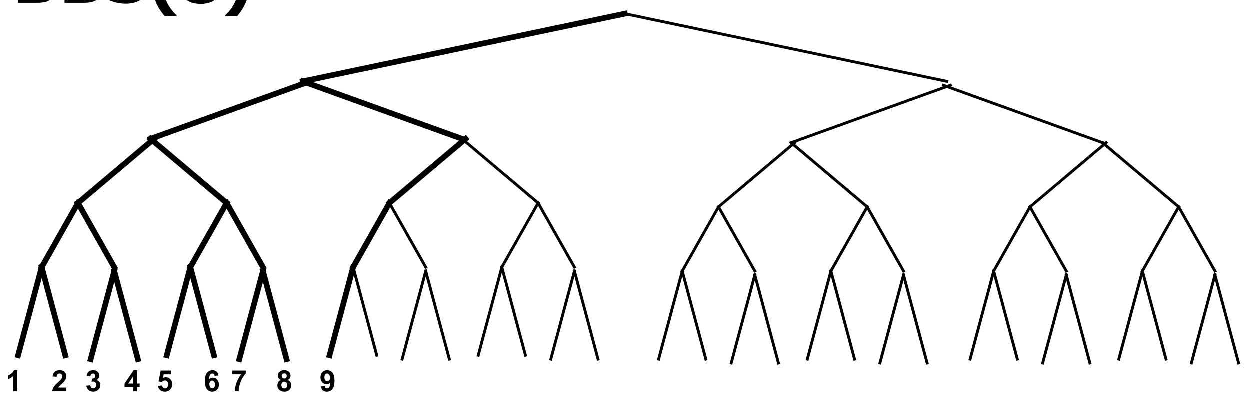
- **Cutoff**
  - after exploiting allocated resources (time, backtracks, credit, …)
  - may be global (for the whole search tree) or local (for a given sub-tree or a search node)
- **Restart**
  - with different parameters (for example with more resources)
  - learning can be used before the next iteration

**Limited number of backtracks** (cutoff)

- backtracking is counted from the point with other alternatives
- “limited number of leafs”

After failure, **increase the limit by one** (restart).

**Example: BBS(8)**




**Implementation:**

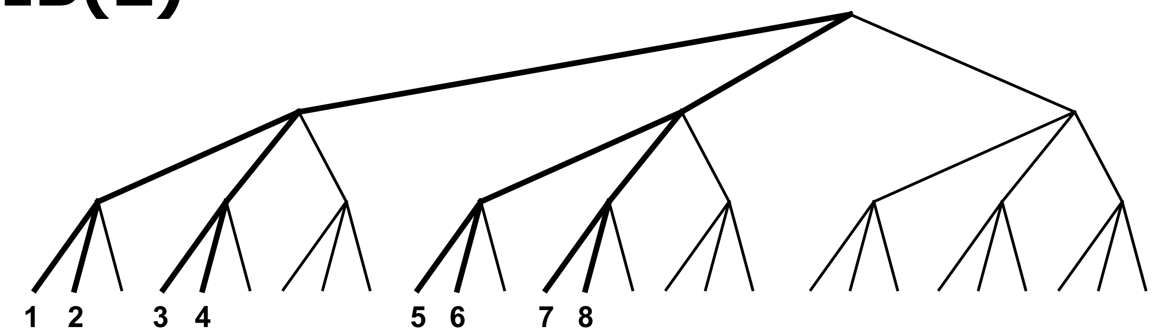
- count the number of backtracks (failures)
- stop after exceeding the limit

**Limited number of alternatives** (width) for each node (cutoff).

- try a given number of alternatives for each node
- Beware, this is still exponential!

After failure, **increase the width by one** (restart).

**Example: IB(2)**




**Implementation:**

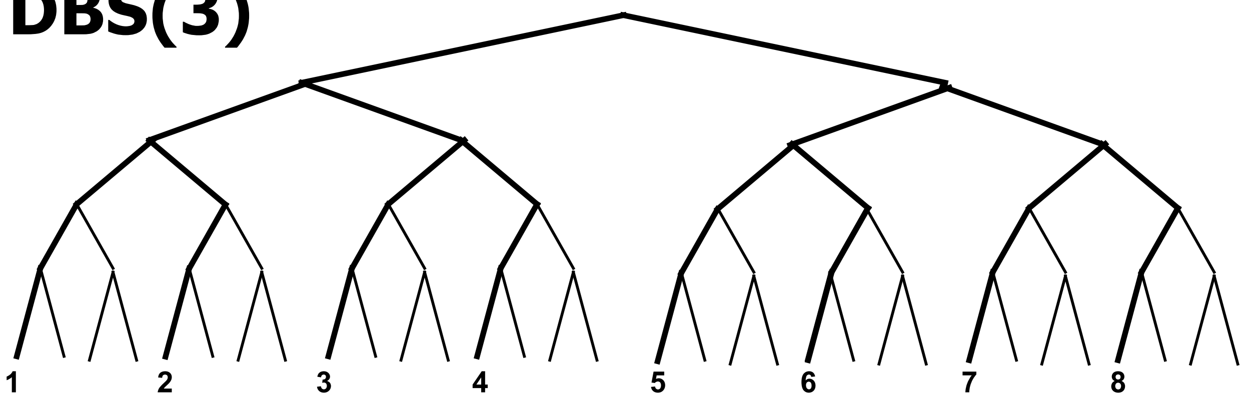
- restrict the number of tries in nodes (inner for-loop)

**Limited depth** for tree search (cutoff).

- till given depth, all alternatives are tried
- after exceeding the depth, another incomplete search can be used

After failure, **increase the depth by one** (restart).

**Example: DBS(3)**




**Implementation:**

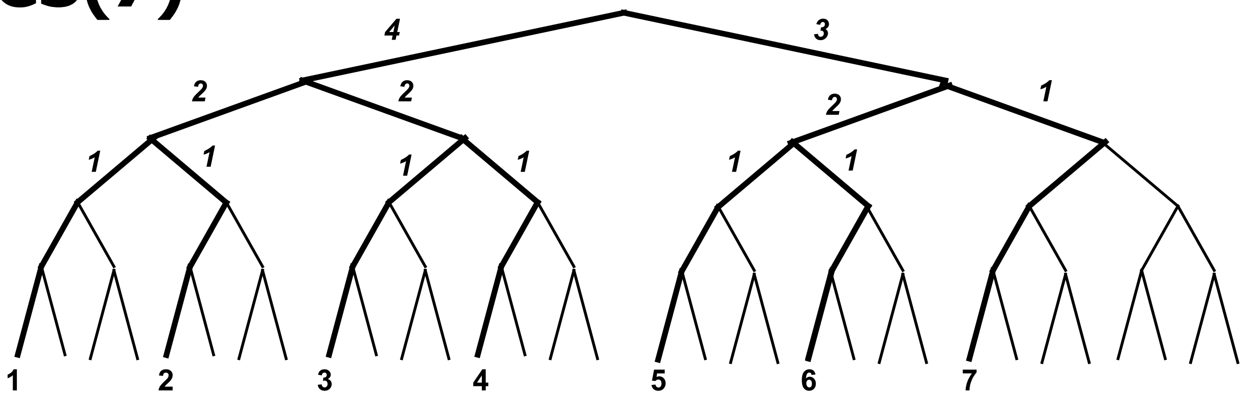
- keep the number of instantiated variables
- if the number is larger than a given limit, try just one alternative – BBS(0)

**Limited credit** (the number of backtracks) for search (cutoff).

- credit is split among available alternatives
- unit credit means no alternatives (values)

After failure, **increase credit by one** (restart).

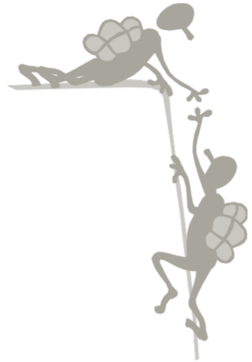
**Example: CS(7)**




**Implementation:**

- in each node the (non-unit) credit is uniformly split among alternative sub-trees
- for a unit credit, just one alternative is tried

# Search and heuristics



**When solving real-life problems we frequently have some experience with “manual” solving of the problem.**

**Heuristics** – a guide where to go

- they recommend a value for assignment (value ordering)
- frequently lead to a solution

**But what to do when the heuristic is wrong?**

- DFS takes care about the end of branches (leafs of tree)
- it repairs latest failures of the heuristic rather than earlier failures
- so it assumes that heuristic was right at the beginning of search

**Observation1:**

The number of wrong heuristic decisions is **low**.

**Observation2:**

Heuristics are usually **less reliable at the beginning** of search than at its end (more information and fewer choices are available there).

**How to make search more efficient?**

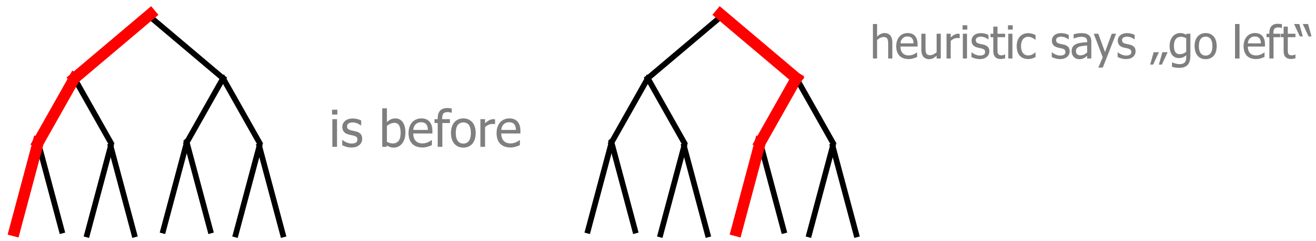
– Backtracking is "blind" with respect to heuristics.

**Discrepancy = violation of heuristic (different value is used)**

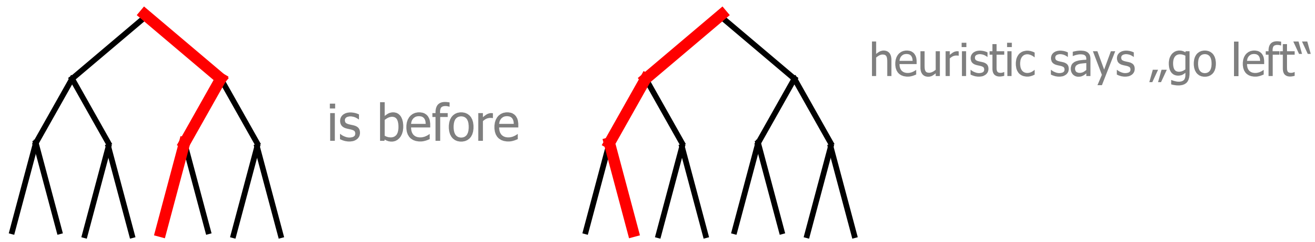
Core principles of discrepancy search:

– we change the order of branches based on discrepancies

– explore first the branches with **less discrepancies**




– explore first the branches with **earlier discrepancies**



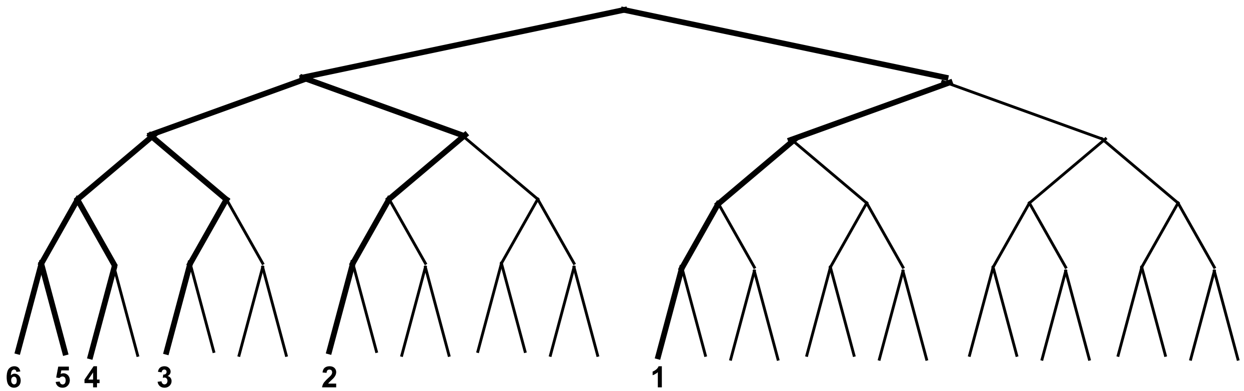
**Limited number of discrepancies** (cutoff)

- branches with less discrepancies are explored first

After failure **increase the number of allowed discrepancies** by one (restart).

- first, follow the heuristic
- then explore paths with at most one discrepancy

**Example: LDS(1),** heuristic suggests going to left




A note for **non-binary domains**:

- non-heuristic values are assumed as **one discrepancy** (here)
- each other non-heuristic value means **increase of the number of discrepancies** (e.g. third value = two discrepancies)

# Algorithm LDS

```
procedure LDS-PROBE(Unlabelled,Labelled,Constraints,D)
    if Unlabelled = {} then return Labelled
    select X in Unlabelled
    Values_X ← D_X - {values inconsistent with Labelled using Constraints}
    if Values_X = {} then return fail
    else select HV in Values_X using heuristic
        if D>0 then
            for each value V from Values_X-{HV} do
                R ← LDS-PROBE(Unlabelled-{X}, Labelled ∪ {X/V}, Constraints, D-1)
                if R≠ fail then return R
            end for
        end if
        return LDS-PROBE(Unlabelled-{X}, Labelled∪{X/HV}, Constraints, D)
    end if
end LDS-PROBE

procedure LDS(Variables,Constraints)
    for D=0 to |Variables| do    % D determines the allowed number of discrepancies
        R ← LDS-PROBE(Variables,{},Constraints,D)
        if R≠ fail then return R
    end for
    return fail
end LDS
```

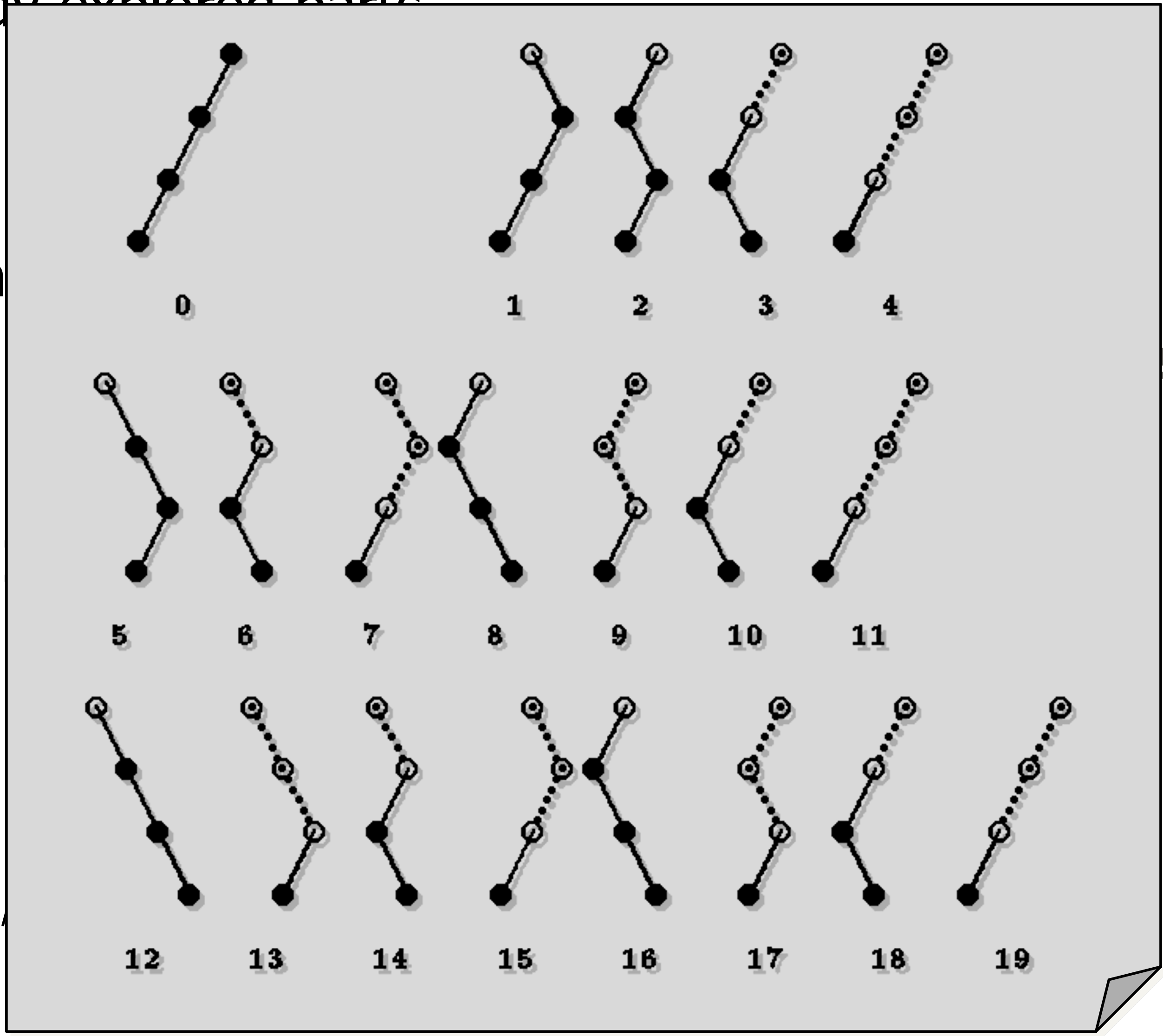
# Improved LDS

In each iteration LDS **explores branches from the previous iteration**, i.e., it repeats already done computation and returns to alread

↳ **ILDS:**

- **a given**
  - "bran
- After fail **s by one** (restart)

**Example:**



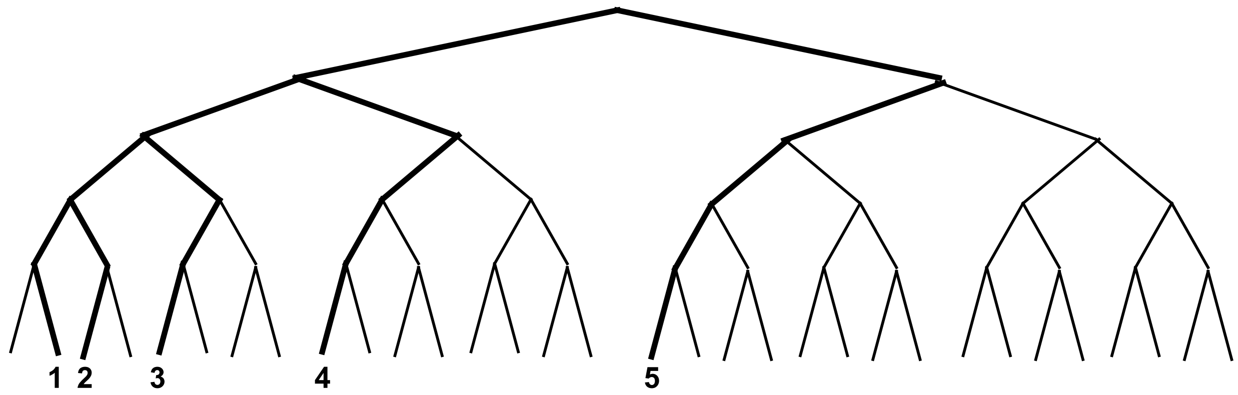
# Improved LDS

In each iteration LDS **explores branches from the previous iteration**, i.e., it repeats already done computation and returns to already explored parts.

↳ **ILDS:**

- **a given number of discrepancies** (cutoff)
  - "branches with later discrepancies explored first""
- After failure **increase the number of discrepancies by one** (restart)

**Example: ILDS(1),** heuristic suggests going to left

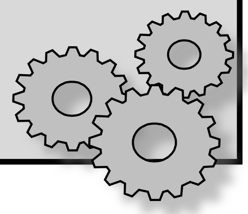


# Algorithm ILDS

```
procedure ILDS-PROBE(Unlabelled, Labelled, Constraints, D)
    if Unlabelled = {} then return Labelled
    select X in Unlabelled
    Values_X ← D_X - {values inconsistent with Labelled using Constraints}
    if Values_X = {} then return fail
    else select HV in Values_X using heuristic
        if D<|Unlabelled| then
            R ← ILDS-PROBE(Unlabelled-{X}, Labelled∪{X/HV}, Constraints, D)
            if R≠ fail then return R
        if D>0 then
          for each value V from Values_X - {HV} do
            R ← ILDS-PROBE(Unlabelled-{X}, Labelled∪{X/V}, Constraints, D-1)
            if R≠ fail then return R
          end for
    end if
end ILDS-PROBE

procedure ILDS(Variables,Constraints)
    for D=0 to |Variables| do    % D determines the allowed number of discrepancies
        R ← ILDS-PROBE(Variables,{},Constraints,D)
        if R≠ fail then return R
    end for
    return fail
end LDS
```

***Difference from LDS***



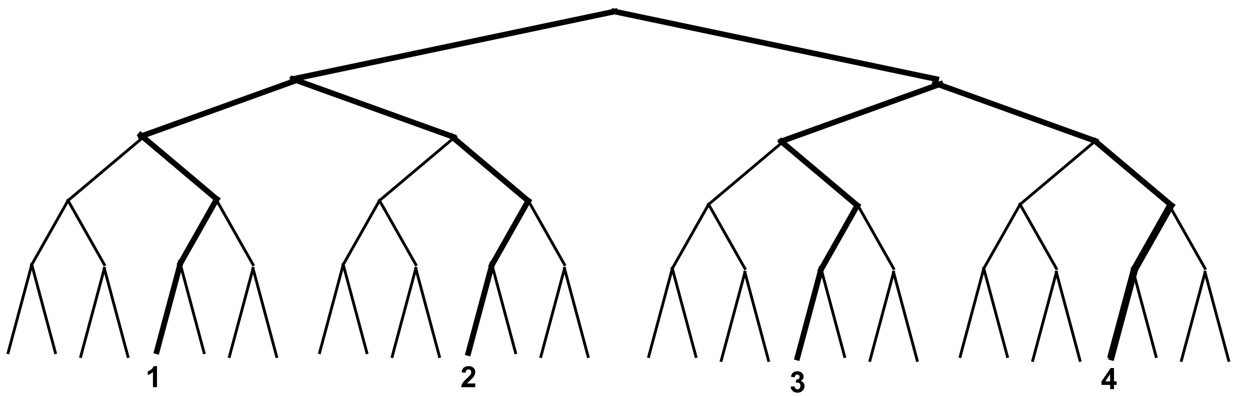
# *Depth-Bounded Discrepancy Search*

**ILDS explores branches with later discrepancies first.**

**↳ DDS:**

- **Discrepancies allowed to some depth** (cutoff)
  - at the limit depth, there must be discrepancy (so no branches from previous iterations are re-visited)
  - depth limit also restricts the number of discrepancies
  - branches with earlier discrepancies are tried first
- After failure **increase the depth limit by one** (restart)

**Example: DDS(3),** heuristic suggests going to left



So far we looked for any solution satisfying the constraints.

Frequently, we need to find an optimal solution, where solution quality is defined by some objective function.

**Definition:**

- **Constraint Satisfaction Optimisation Problem** (CSOP) consists of a CSP P and an objective function *f* mapping solutions of P to real numbers.
- A **solution to a CSOP** is a solution to P minimizing / maximizing the value of *f*.
- When solving CSOPs we need methods that can provide more than one solution.

The method **branch-and-bound** is a frequently used optimisation technique based on pruning branches where there is no optimal solution.

It uses a **heuristic function** h that estimates the value of objective function f.

- admissible heuristic for minimization satisfies $h(x) \leq f(x)$
  [for maximization $f(x) \leq h(x)$]
- heuristic closer to f is better

We stop exploring the search branch when:

- there is **no solution** in the sub-tree
- there is **no optimal solution** in the sub-tree
  - Bound $\leq h(x)$, where Bound is the maximal value of f for an acceptable solution

**How to obtain the Bound?**

- for example the value of the solution found so far

# Branch and bound for constrained optimization

**Objective function is encoded in a constraint**

we “optimize” the value v, where v = f(x)

- the first solution is found using no bound on v
- the next solutions must be better than the last solution found (v < Bound)
- repeat until no feasible solution is found

**Algorithm Branch & Bound**

```
procedure BB-Min(Variables, V, Constraints)
    Bound ← sup
    NewSolution ← fail
    repeat
        Solution ← NewSolution
        NewSolution ← Solve(Variables,Constraints ∪ {V<Bound})
        Bound ← value of V in NewSolution (if any)
    until NewSolution = fail
    return Solution
end BB-Min
```

## Branch and bound: notes

- Heuristic h is hidden in the **propagation of constraint** v = f(x).
- Efficiency of search depends on:
  - **good heuristic** (good propagation through the objective constraint)
  - **good solution found early**
    using an initial bound may help
- We can find the optimal solution fast
  - but the **proof of optimality takes time** (explore the rest of search tree)
- Frequently, we do not need optimal solution, good solution is enough
  - BB can stop after finding a good enough solution
- BB can be speeded up by using using both upper and lower bounds

```
repeat
    TempBound ← (UBound+LBound) /  2
    NewSolution ← Solve(Variables,Constraints ∪ {V≤TempBound})
    if NewSolution=fail then
        LBound ← TempBound+1
    else
        UBound ← TempBound
until LBound = UBound
```

Department of Theoretical Computer Science and Mathematical Logic
bartak@ktiml.mff.cuni.cz