OptalCP

Constraint Programming with Parallel Search and Reinforcement Learning-Based Acceleration

Petr Vilím · ScheduleOpt

Vilém Heinz · Czech Technical University in Prague

Scheduling Seminar · schedulingseminar.com

What is OptalCP?

- **Constraint Programming solver for scheduling problems.**
- From the outside similar to *IBM ILOG CP Optimizer*.
  - Similar modeling language and concepts.
  - Interval variables, sequences, cumulative resources.
- From the inside, completely different.
  - Modern architecture, designed for **parallel search**.
  - Written in **C++20**, APIs in **TypeScript**/JavaScript and **Python**.

Today's focus: How does the solver work inside?

What makes OptalCP Different?

I’ve built CP solvers before. **Now I’m free to rethink EVERYTHING.**
In particular the **internals**.

Architecture:

- **Built for speed** from the ground up.
- True parallelism.
- Heterogeneous workers.
- External heuristic hybridization.

Modeling & API:

- Native **Python** and **TypeScript** APIs.
- **Async** event-driven solving.
- Integers with **optional presence**.
- New modeling constructs.

> It's not "just faster" — it's a different architecture that enables new capabilities.

What makes OptalCP Different?

I’ve built CP solvers before. **Now I’m free to rethink EVERYTHING.**
In particular the **internals**.

Architecture:

- **Built for speed** from the ground up.
- True parallelism.
- Heterogeneous workers.
- External heuristic hybridization.

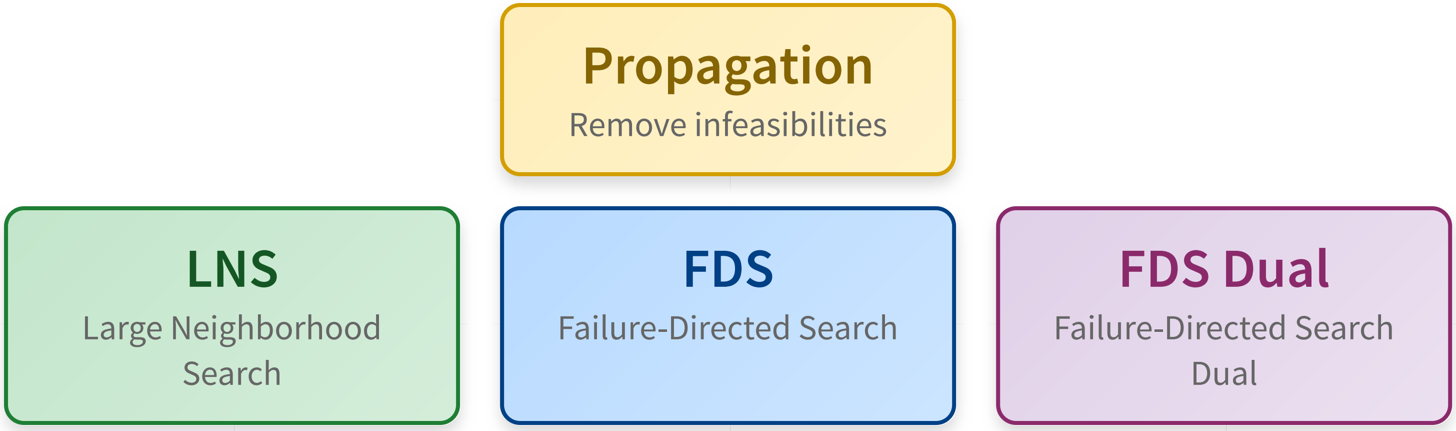
Modeling & API:

- Native **Python** and **TypeScript** APIs.
- **Async** event-driven solving.
- Integers with **optional presence**.
- New modeling constructs.

> It's not "just faster" — it's a different architecture that enables new capabilities.

> For **academic licenses**, send me your GitHub username.

Inside the Solver



Propagation

Remove infeasibilities

LNS

Large Neighborhood Search

FDS

Failure-Directed Search

FDS Dual

Failure-Directed Search Dual

Every algorithm has **strengths** and **weaknesses**.

Propagation

Propagation

Always in the Party

Role: Support

Action: Remove impossible values

Produces: Smaller domains

✓ Prunes domains | Detects infeasibility

✗ Can't solve alone

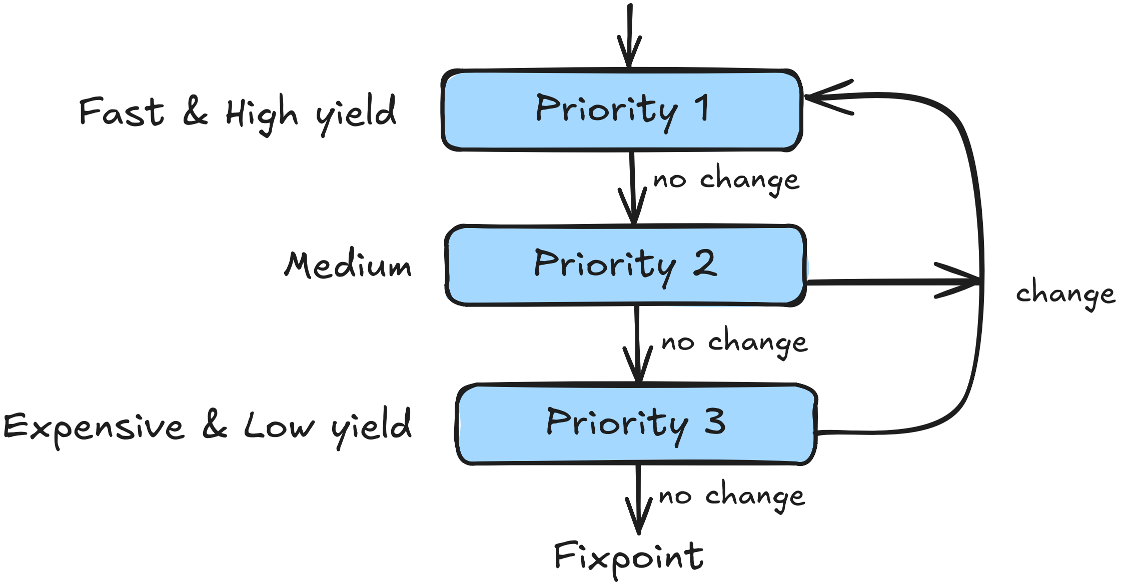
Propagation Algorithms for Scheduling

Algorithms for resource constraints in OptalCP:

- Detectable Precedences
- Edge-finding
- Not-first / Not-last
- Timetabling
- Timetable Edge-Finding

Propagation

In every search node, **propagation** removes impossible values until fixpoint or infeasibility.



Large Neighborhood Search (LNS)

LNS

Large Neighborhood Search

Type: Local search with CP repair

Assumes: Better solution exists nearby

Strategy: Destroy part, repair with CP

Produces: Better solutions

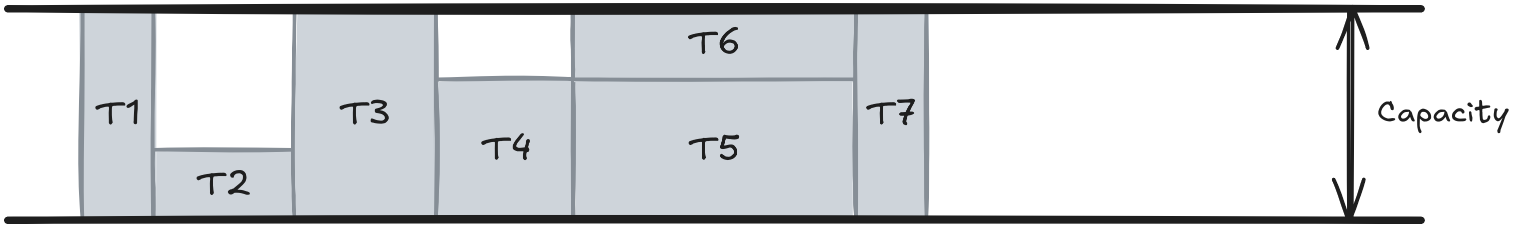
✓ Fast solutions · Exploits structure

✗ Requires initial solution

✗ Local optima · No optimality proof

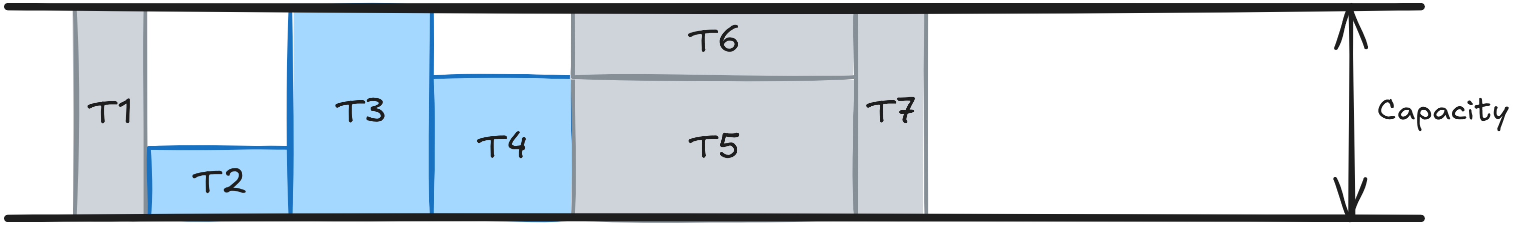
How *Standard* LNS Works

Suppose the following solution:



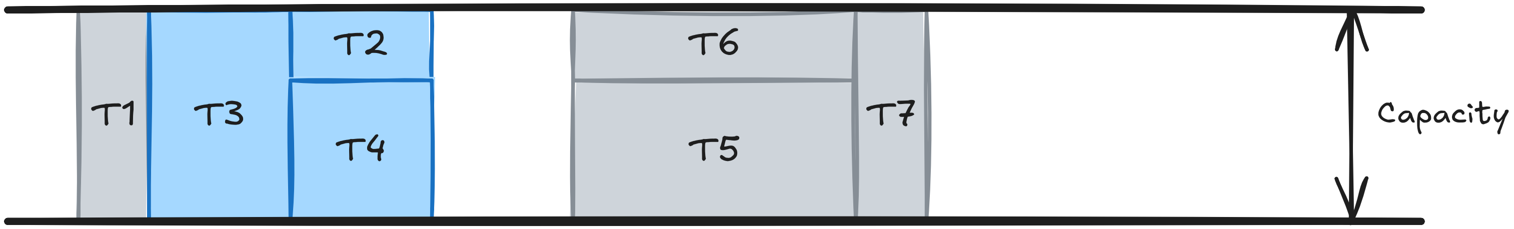
How *Standard* LNS Works

We relax part of it:



How *Standard* LNS Works

We solve the relaxed problem:



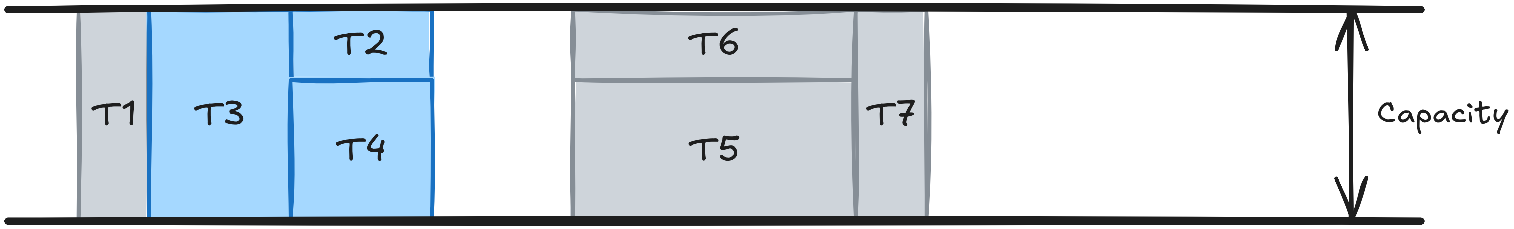
The problem: Standard LNS fixes the ***values*** of non-relaxed variables.

Since these are ***times***, we can't improve the makespan unless we relax more variables.

> Keeping variable **values** fixed is too restrictive.

How *Standard* LNS Works

We solve the relaxed problem:



The problem: Standard LNS fixes the ***values*** of non-relaxed variables.

Since these are ***times***, we can’t improve the makespan unless we relax more variables.

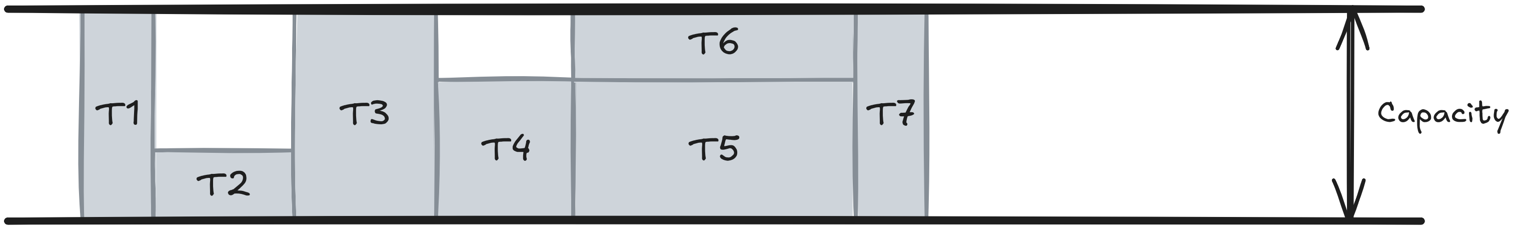
Keeping variable **values** fixed is too restrictive.

Idea: Modify relaxation to capture the ***structure*** of the solution instead.

Philippe Laborie, Daniel Godard:
Self-adapting large neighborhood search: Application to single-mode scheduling problems

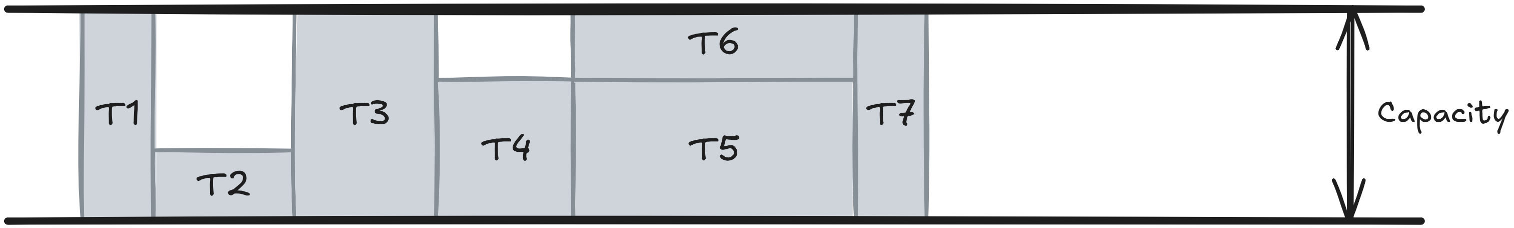
Partial Order Schedule (POS)

Suppose we have the following solution:



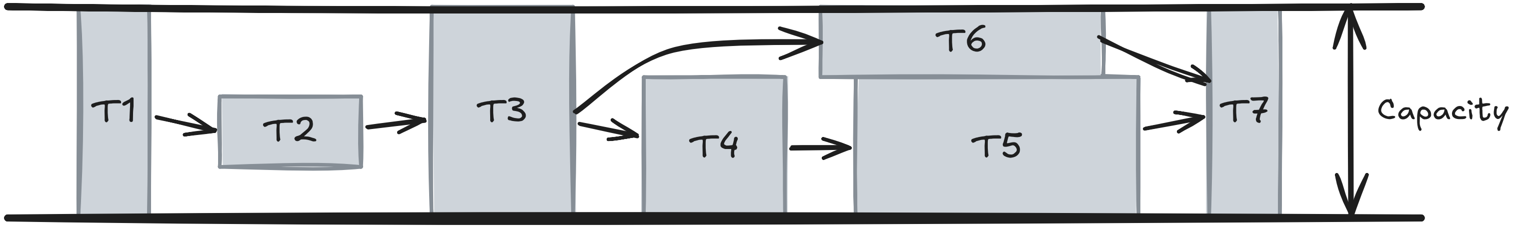
Partial Order Schedule (POS)

Suppose we have the following solution:

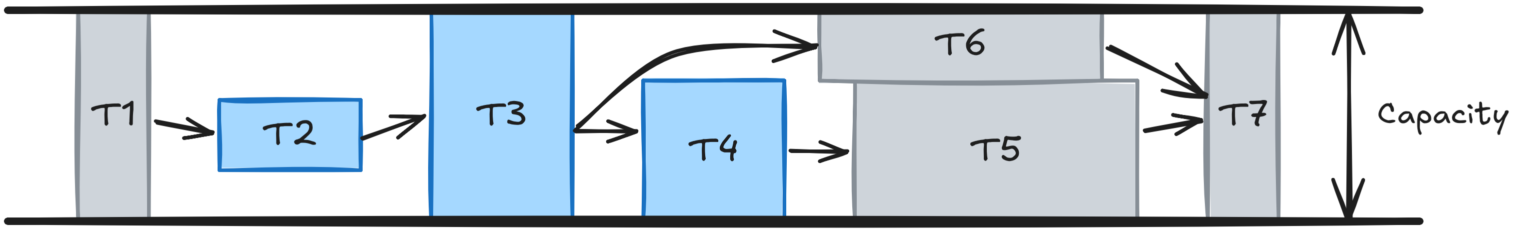


POS = **structure of the solution** = a set of precedences between tasks.

If the variables respect the precedences, resource constraints are automatically satisfied.



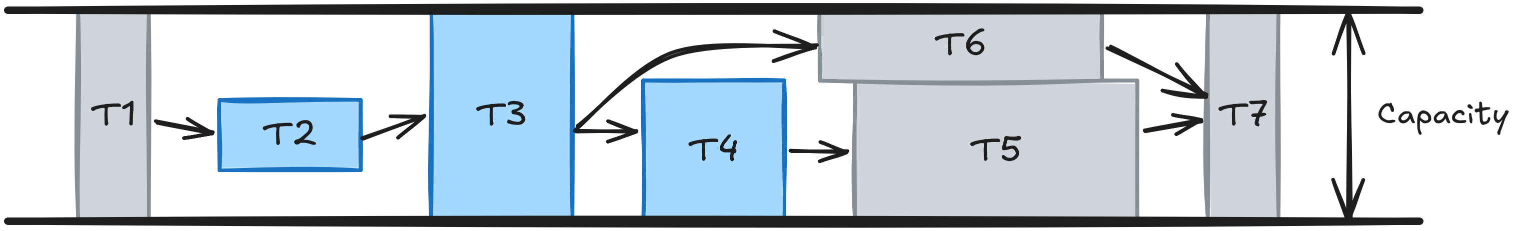
Relaxing with POS



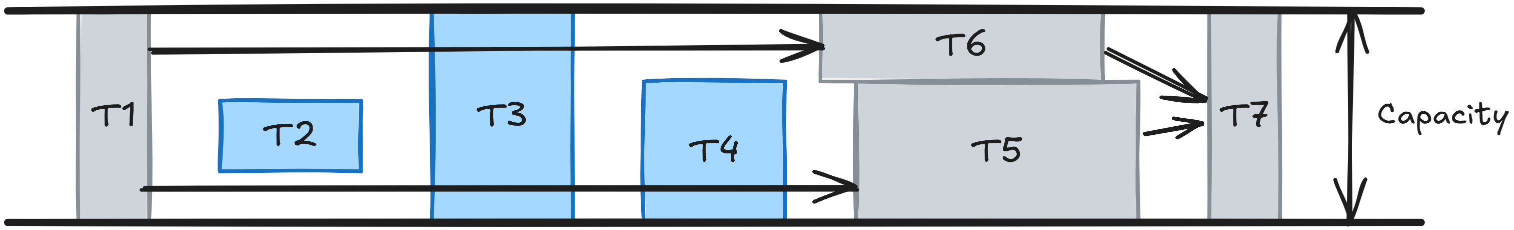
To relax a task, we remove all its precedences.

Transitive precedences between the remaining tasks are added instead.

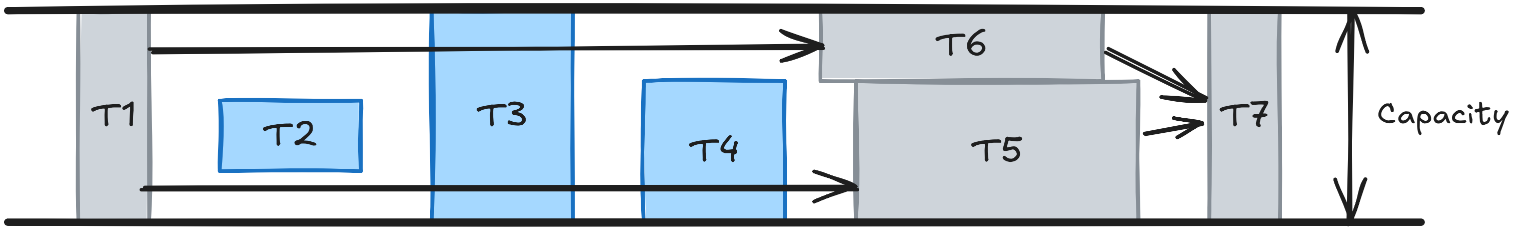
Relaxing with POS



To relax a task, we remove all its precedences.
Transitive precedences between the remaining tasks are added instead.



Solving with POS

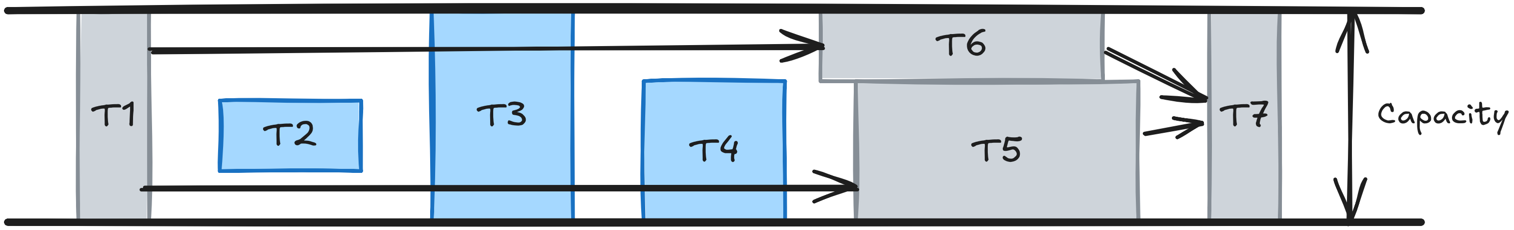


LNS **sub-problem**:

- Has the same variables (and domains).
- But **more constraints** (precedences from relaxed POS).

Sub-problem solution:

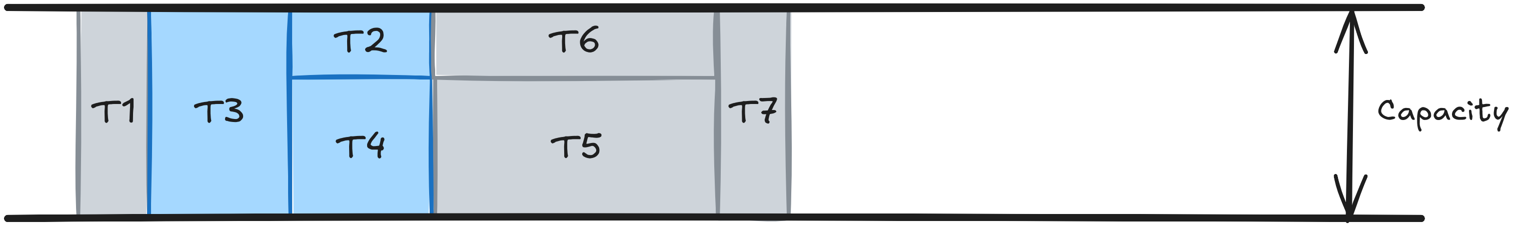
Solving with POS



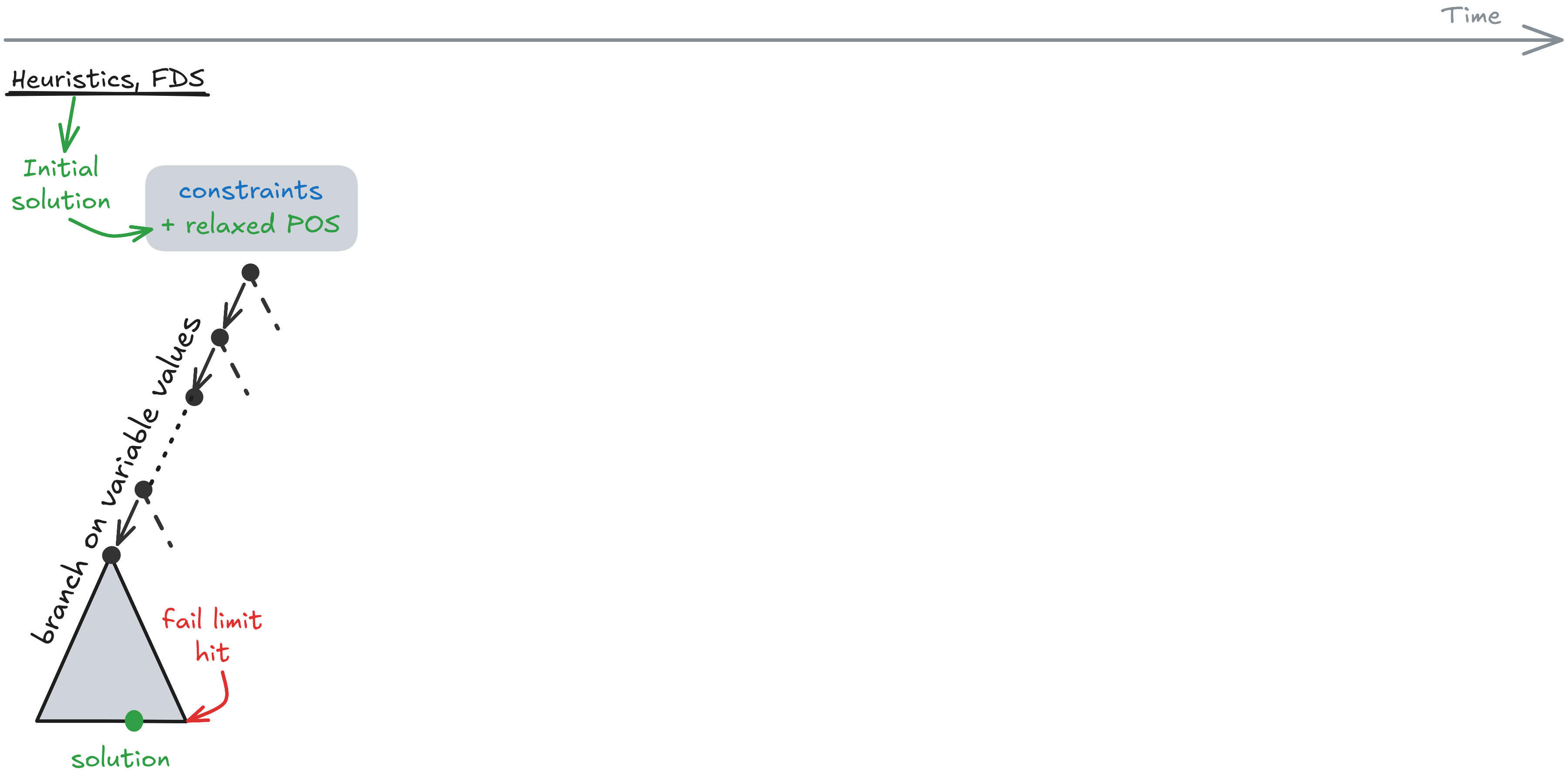
LNS **sub-problem**:

- Has the same variables (and domains).
- But **more constraints** (precedences from relaxed POS).

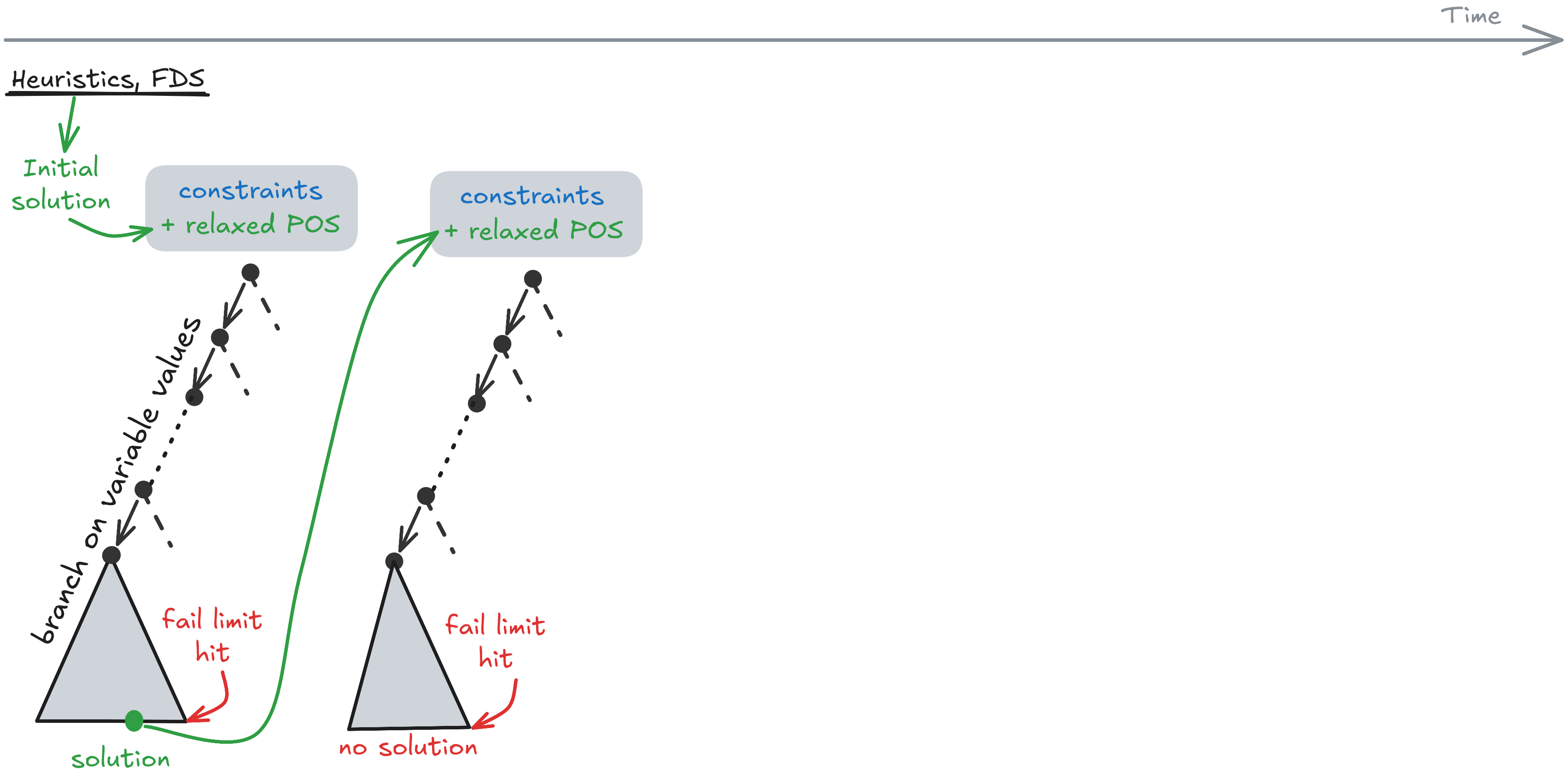
Sub-problem solution:



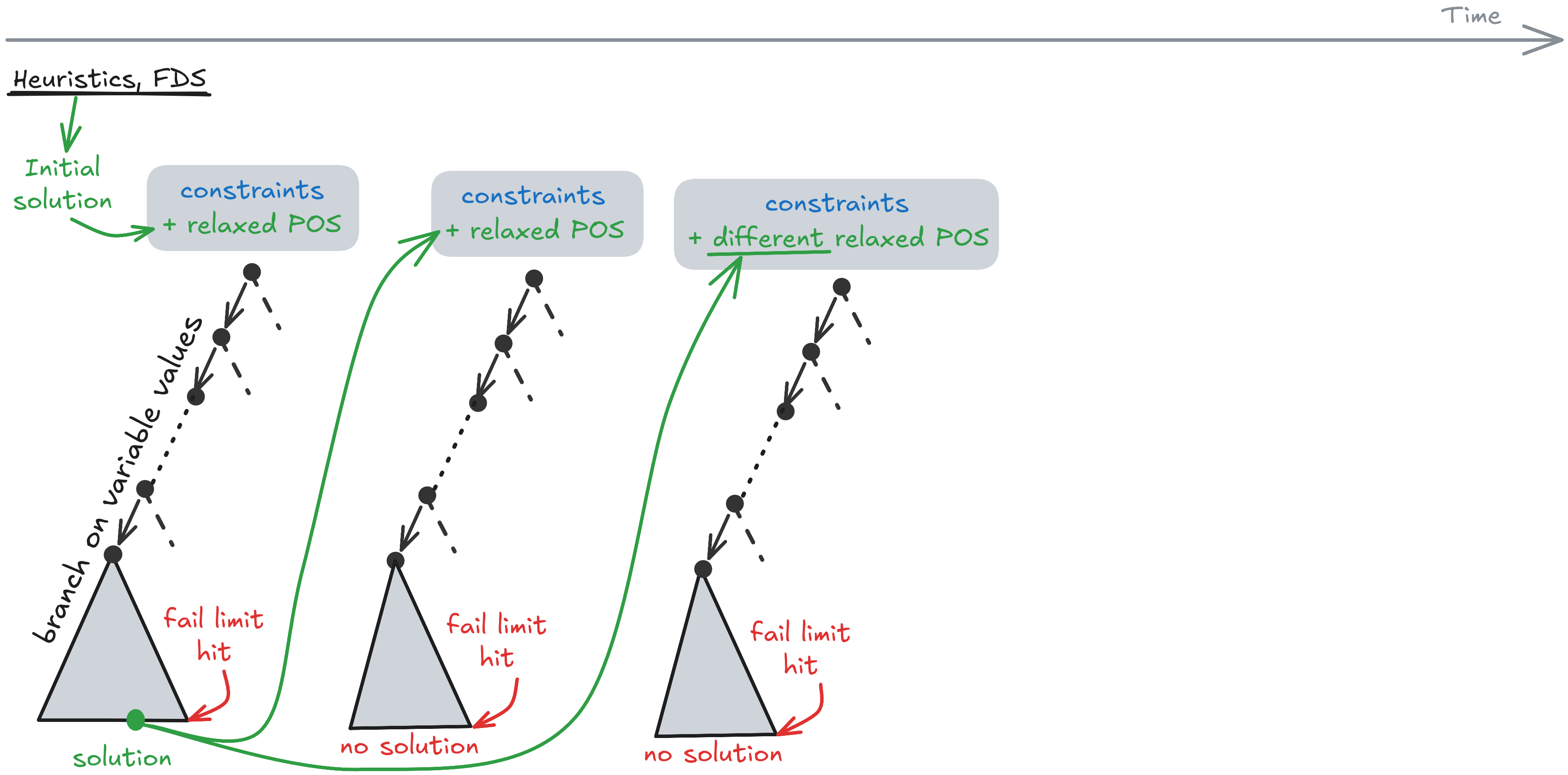
LNS Iterations



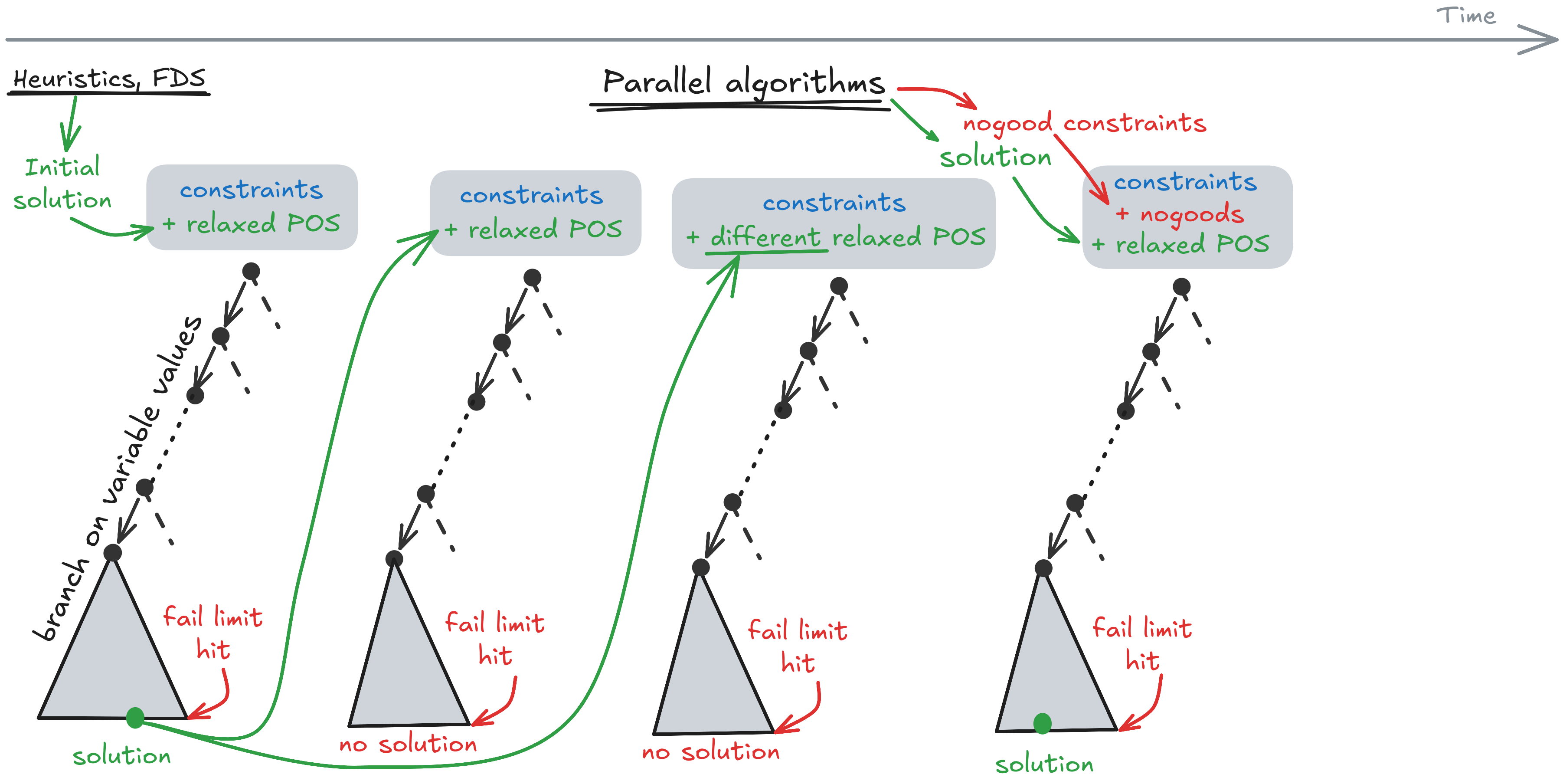
LNS Iterations



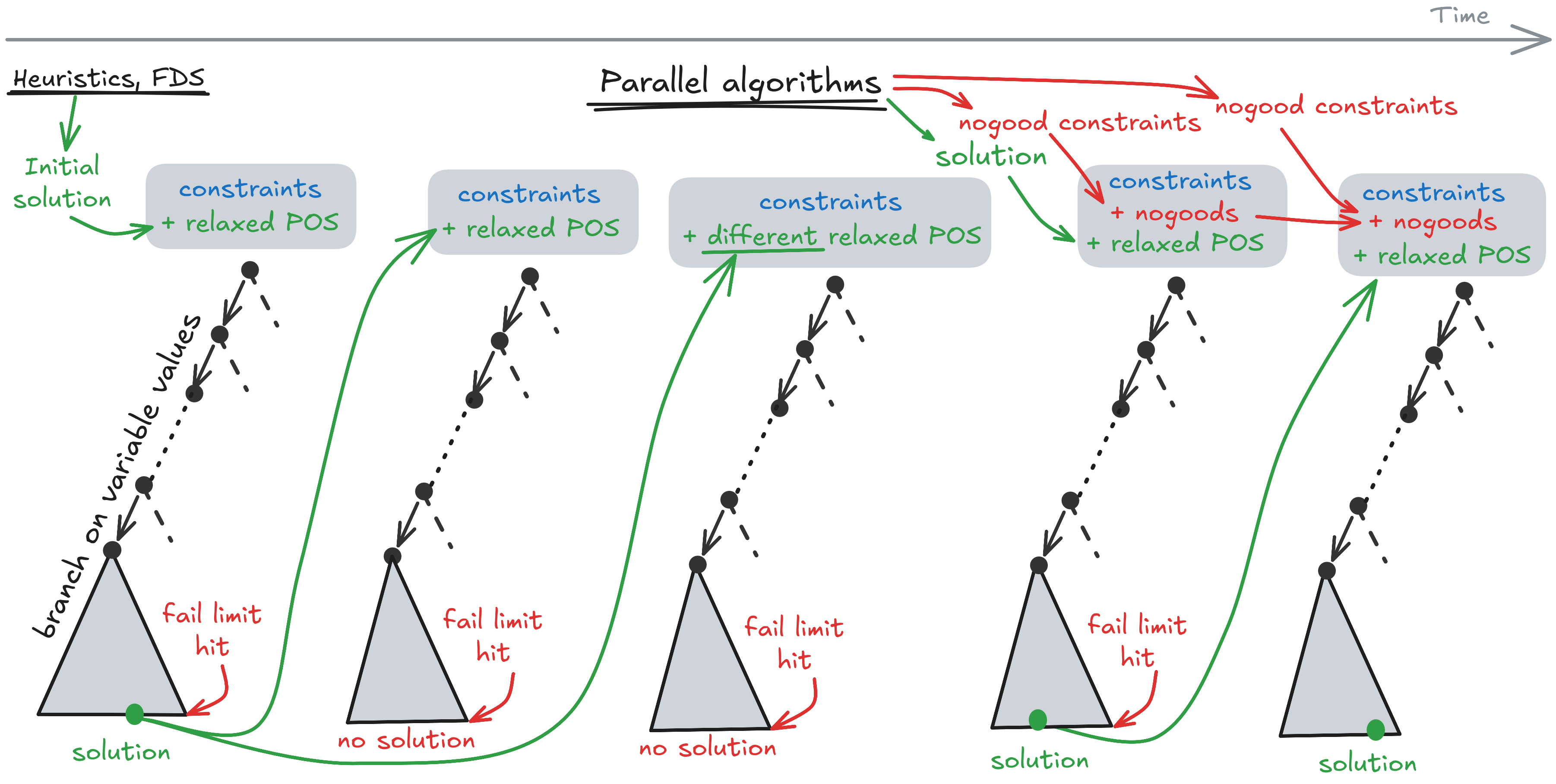
LNS Iterations


Time
Heuristics, FDS
Initial solution
constraints + relaxed POS
constraints + relaxed POS
constraints + different relaxed POS
branch on variable values
fail limit hit
fail limit hit
fail limit hit
solution
no solution
no solution

LNS Iterations



LNS Iterations



Failure-Directed Search (FDS)

FDS

Failure-Directed Search

Type: Systematic tree search

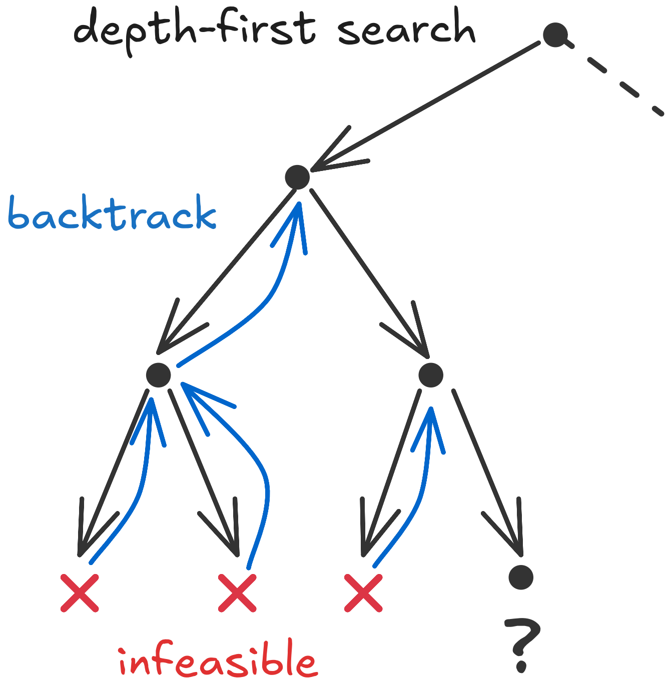
Assumes: Problem is infeasible/hard

Strategy: Learn from failures, restart

Produces: Solutions, proofs, nogoods

✓ Optimality proofs Lower bounds

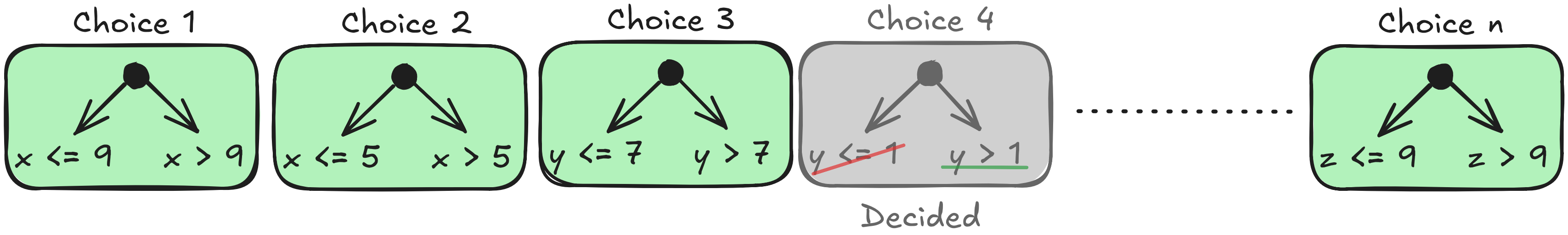
✗ Slow Solutions are a byproduct



Vilém Heinz, Petr Vilím, Zdeněk Hanzálek:
Reinforcement Learning for Search Tree Size Minimization in Constraint Programming: New Results on Scheduling Benchmarks

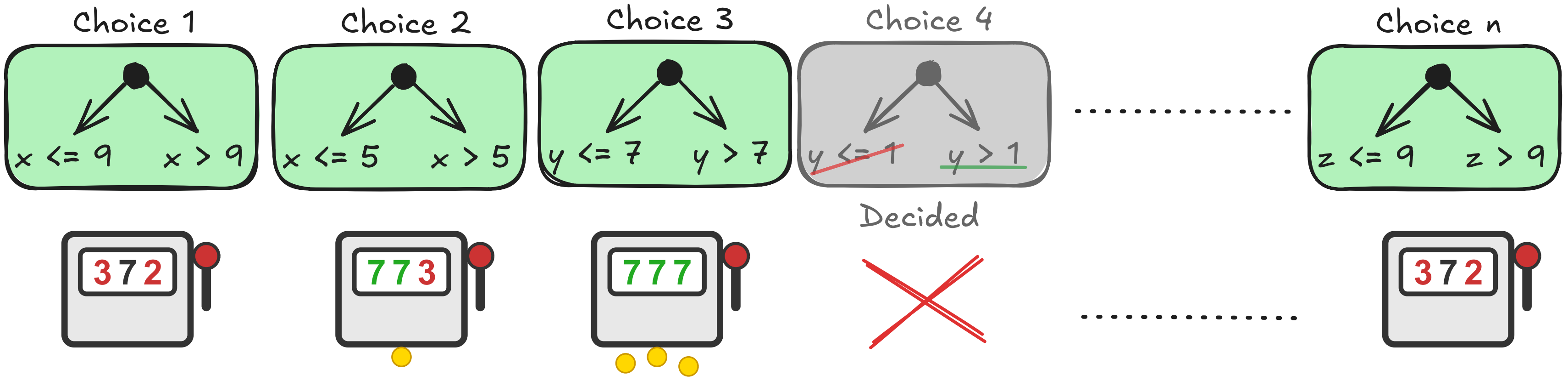
Simplified FDS

FDS maintains **ratings** on *choices*:



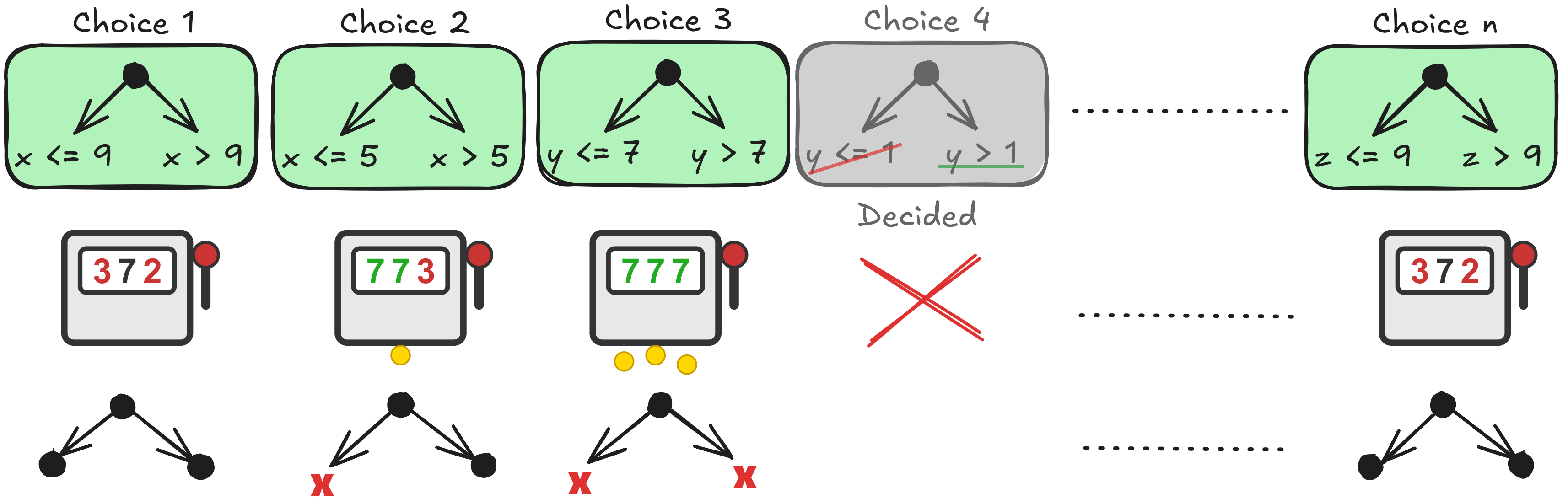
Simplified FDS

FDS maintains **ratings** on *choices*:



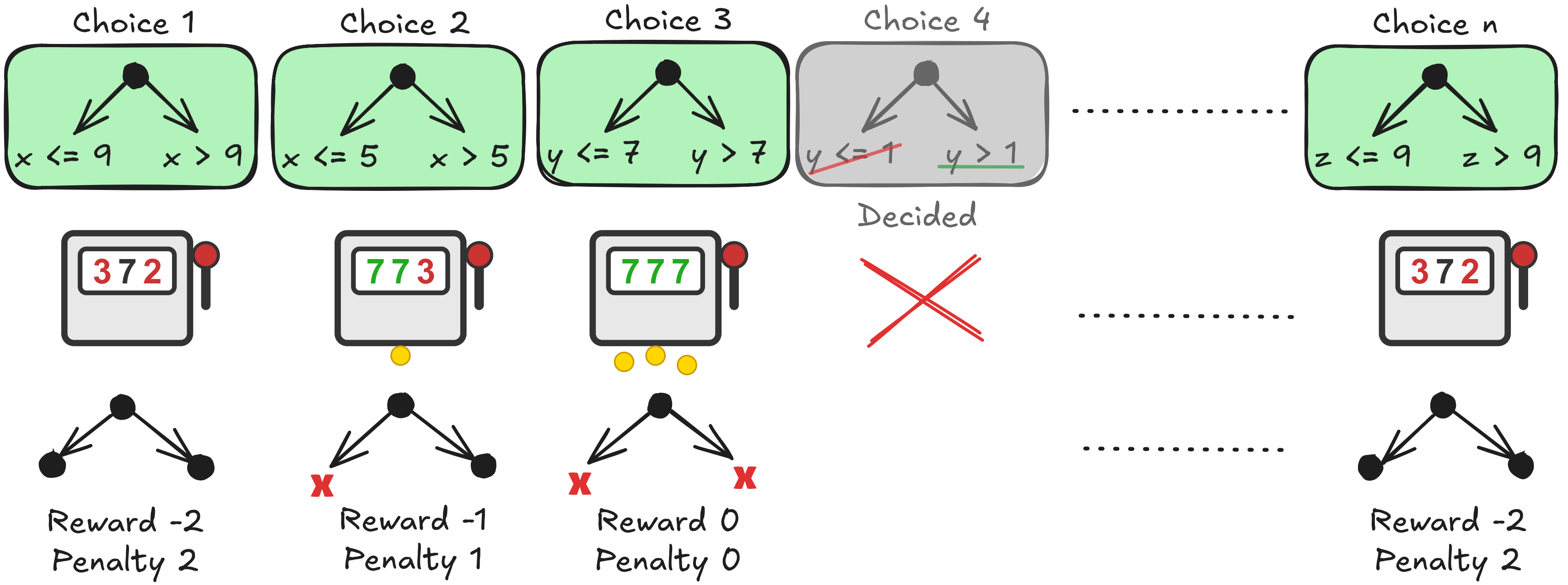
Simplified FDS

FDS maintains **ratings** on *choices*:



Simplified FDS

FDS maintains **ratings** on *choices*:



$$\mathbf{rating}(\text{choice}) \;:=\; \alpha \cdot \mathbf{rating}(\text{choice}) + (1-\alpha)\cdot \text{penalty}$$

The FDS - MAB Connection

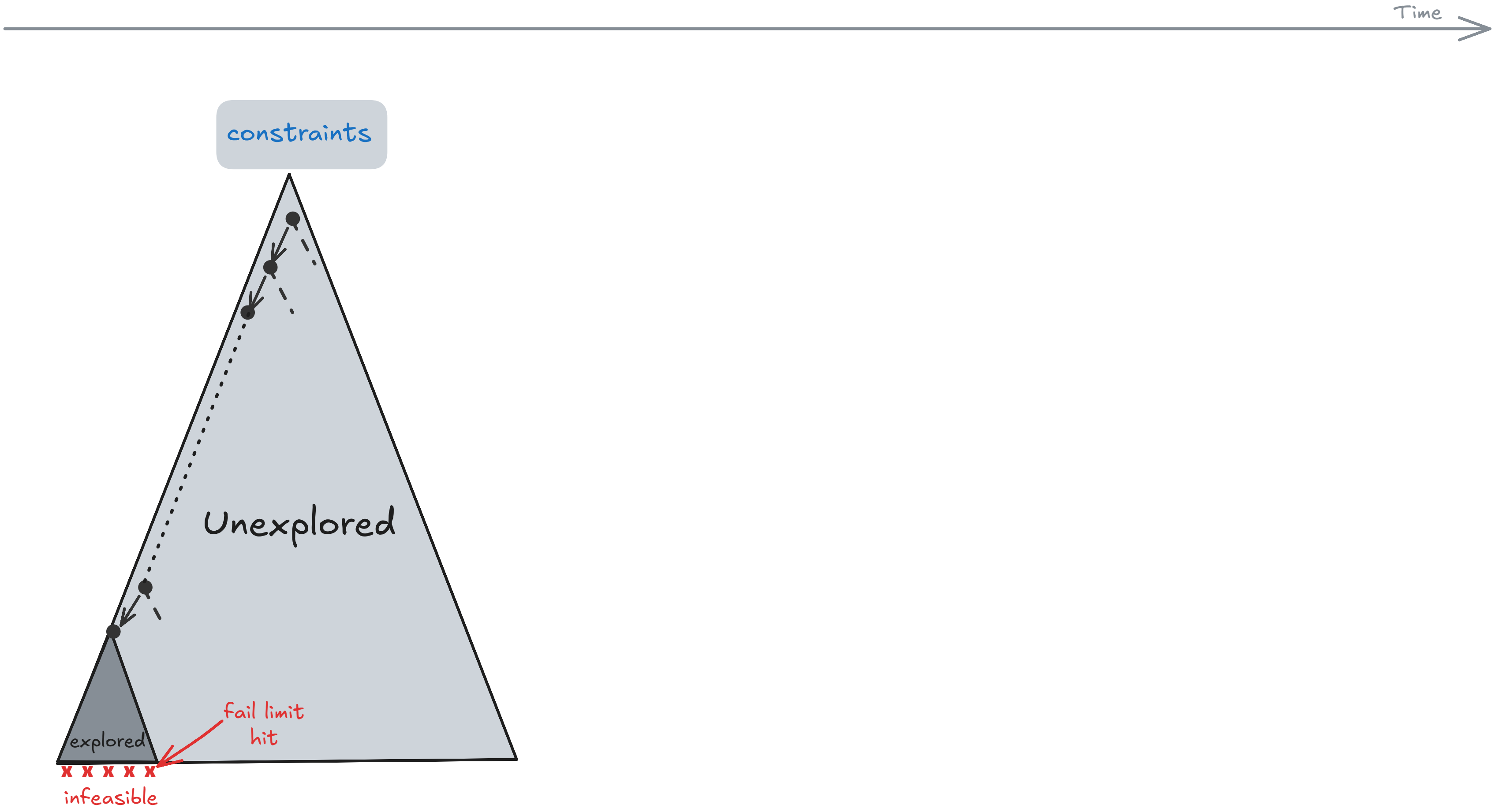
- MAB algorithms minimize sum of penalties.
- In (simplified) FDS, sum of penalties **is the tree size!**

FDS learns to minimize tree size. By design.

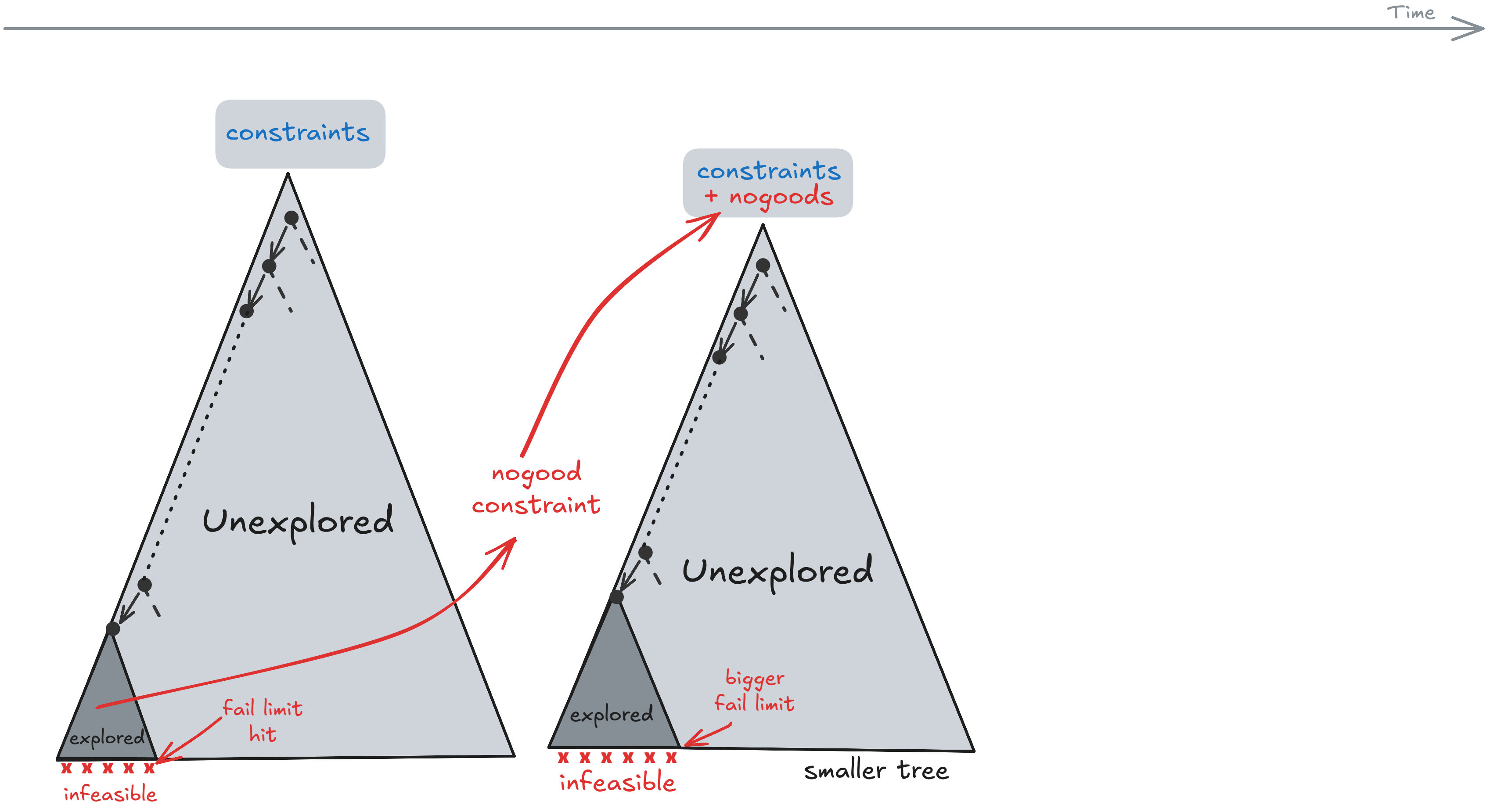
I didn't know this when we designed FDS. But it makes sense now.

Vilém will explore the MAB perspective in more depth.

FDS Restarts

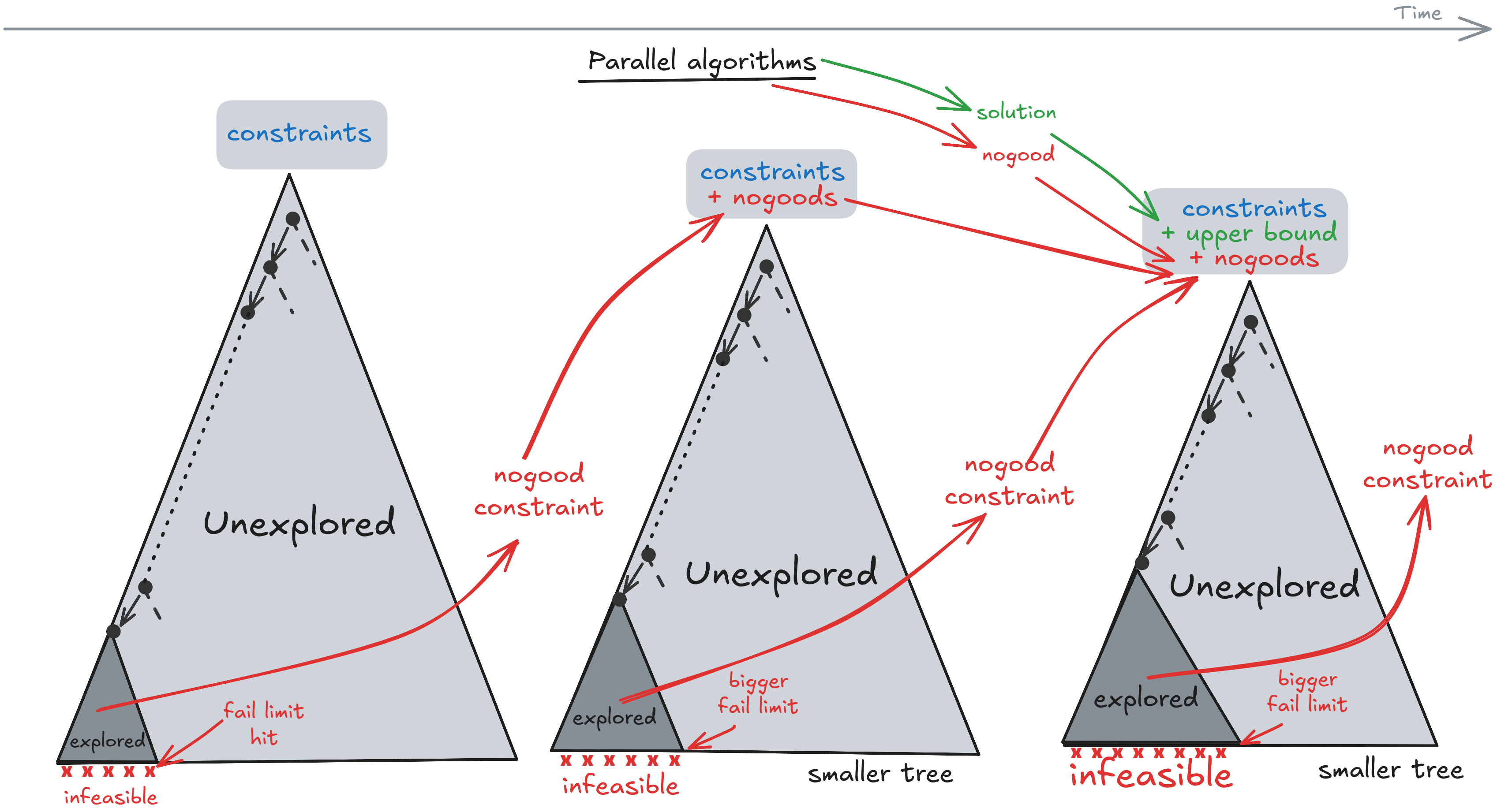


FDS Restarts



Underlying trees get smaller due to better choices and accumulated nogoods.
Explored subtrees get bigger due to increased fail limits.

FDS Restarts



Underlying trees get smaller due to better choices and accumulated nogoods.
Explored subtrees get bigger due to increased fail limits.

FDS Dual

FDS Dual

Lower Bound Prover

Type: Bound-focused search

Assumes: Lower bound can be improved

Strategy: Prove infeasible, increment

Produces: Tighter lower bound, nogoods

✓ Fast bound proofs · Efficient for LB

✗ No solutions · Not good team player

Smarter version of **destructive lower bounds**.

FDS Dual Search

Efficient lower bound proving

Start with a tight bound, prove it infeasible, then relax.

```
bound = current_LB
while solve(objective <= bound) == INFEASIBLE:
  reportLB(bound + 1)              # Assuming integer
  bound += new_bound_to_try(..)   # by parameter FDSDualStrategy
```

FDS Dual workers focus specifically on tightening the lower bound.

The Perfect Combo

LNS
Fast solutions

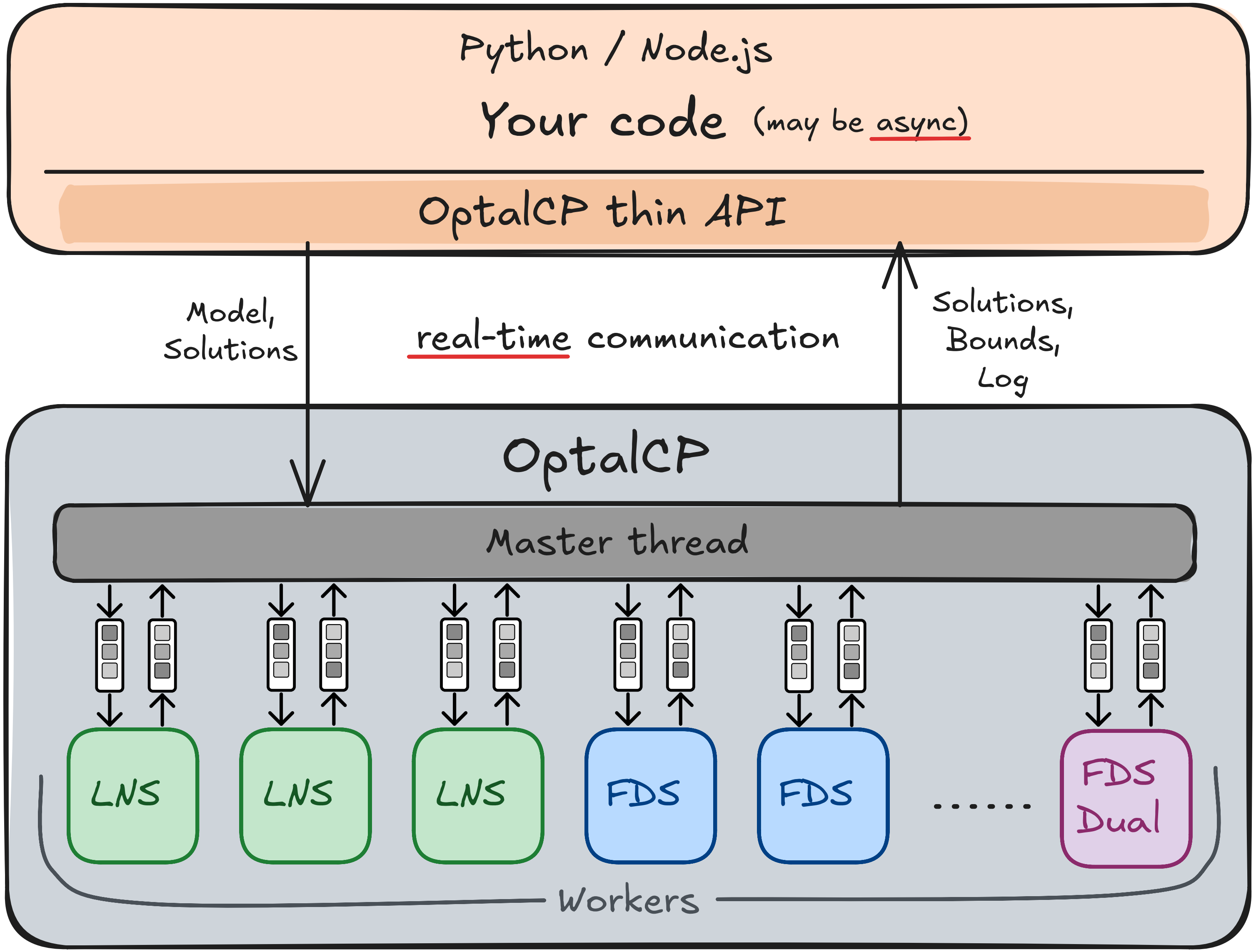
FDS
Proves optimality

FDS Dual
Tightens bounds

They complement each other:

- LNS finds solutions → FDS has a better bound
- FDS finds solutions → LNS escapes local optima
- FDS Dual proves bounds → Gap shrinks from below
- FDS FDS Dual restarts → Generate nogoods for LNS FDS FDS Dual
- LNS FDS FDS Dual prove optimality together → Search ends

Parallel Architecture



Heterogeneous Workers

Each worker can be configured independently:

```
model.solve({
  nbWorkers: 4,
  workers: [
    // Fast exploration:
    { searchType: "LNS", noOverlapPropagationLevel: 2 },
    // Stronger reasoning:
    { searchType: "LNS", noOverlapPropagationLevel: 4 },
    // Optimality focus, escape local optima:
    { searchType: "FDS", noOverlapPropagationLevel: 4 },
    // Prove lower bounds:
    { searchType: "FDSDual", noOverlapPropagationLevel: 4 }
  ]});
```

Or you can just let OptalCP to decide.

Hybrid Solution Using Your Algorithm

Your Algorithm

Your Secret Weapon

Type: Revolutionary

Solutions: Always the best

Speed: Blazing fast

Code: Beautiful and bug-free

✓ Perfect for the problem

✓ Scales effortlessly

✗ Not in OptalCP No lower bounds

Hybrid Solution Using Your Algorithm

Your Algorithm

Your Secret Weapon

Type: Revolutionary

Solutions: Always the best

Speed: Blazing fast

Code: Beautiful and bug-free

✓ Perfect for the problem

✓ Scales effortlessly

✗ Not in OptalCP   No lower bounds

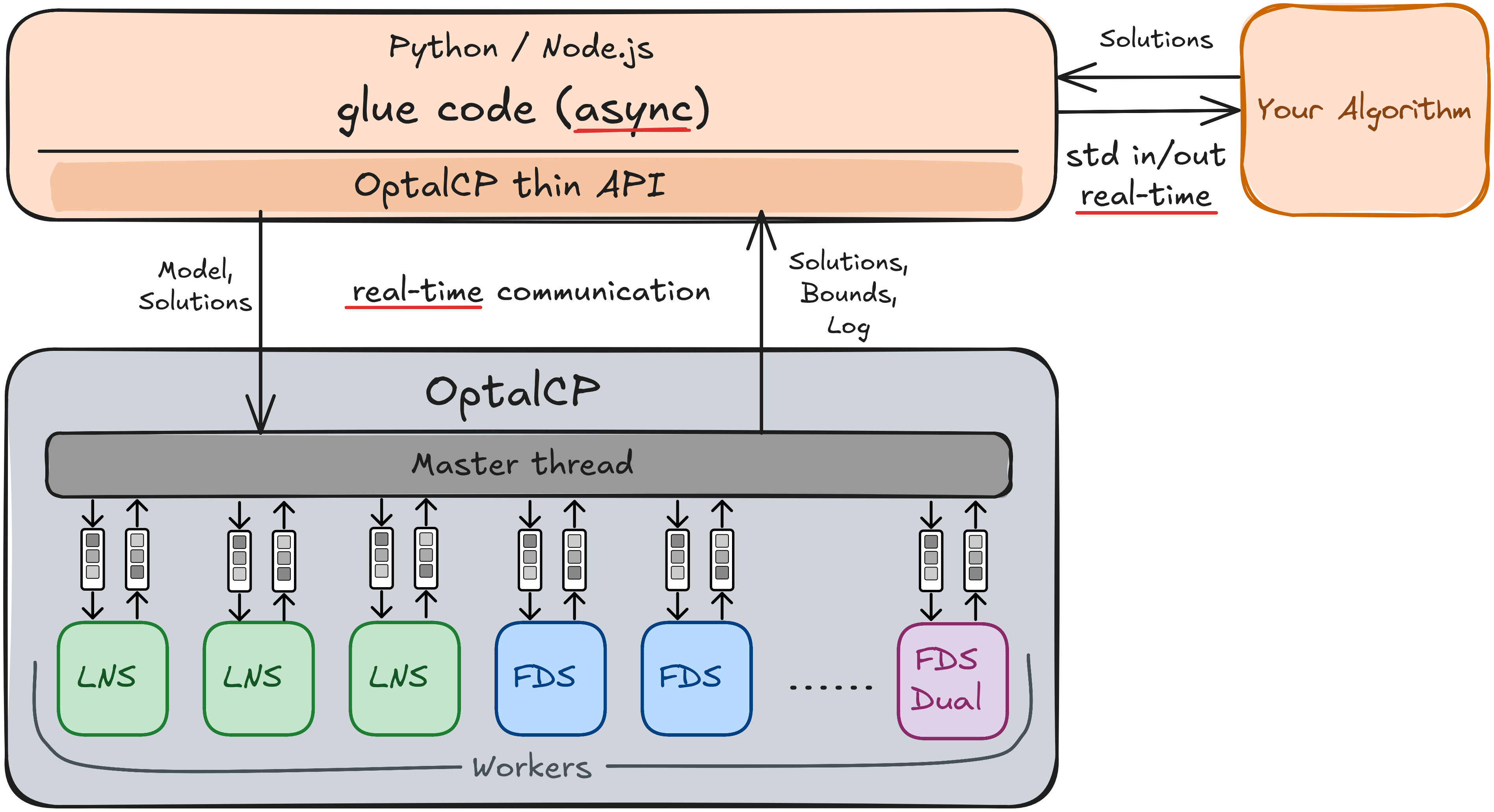
Why hybridize?

- Solution exchange both ways
- Improved robustness
- Escape local optima
- Better than parallel alone
- Adds optimality gap, stops at optimum

How to plug in:

- Your algorithm in any language
- Communicates via stdin/stdout
- Short glue code in Python/*TypeScript*
- Example on GitHub

Architecture Enabling Hybrid Solution



Plugging In Your Algorithm

```python
your_algorithm = await asyncio.create_subprocess_exec(...)
solver = cp.Solver()

def on_optalcp_solution(event: cp.SolutionEvent) -> None:
    serialized = your_solution_format(event.solution)
    your_algorithm.stdin.write(serialized + b'\n')

async def read_your_solutions() -> None:
    while True:
        line = await your_algorithm.stdout.readline()
        solution = to_optalcp_solution(line)
        await solver.send_solution(solution)

solver.on_solution = on_optalcp_solution
asyncio.create_task(read_your_solutions())
await solver.solve(model, parameters)
your_algorithm.kill()
```

Live Demo demo

Research Results

- Hybridization with (Meta)heuristics
- Search Acceleration using Reinforcement Learning

Vilém Heinz

Hybridization with (Meta)heuristics

Experiments on Scheduling and Routing Problems

Research

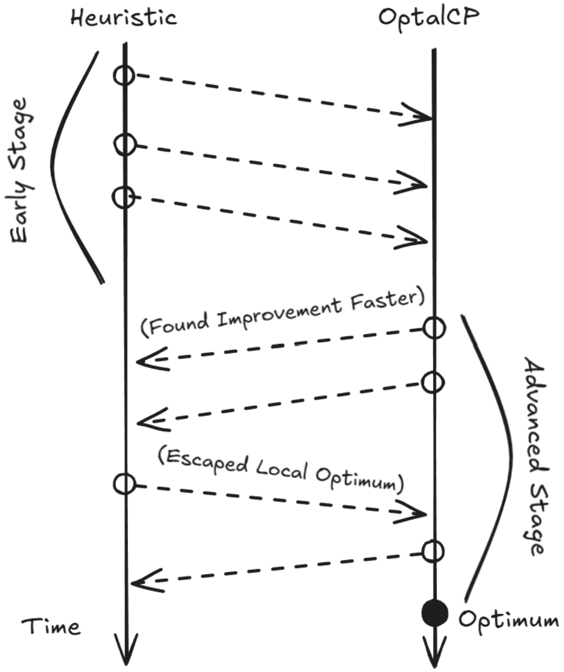
Motivation

Capability	CP Solver	(Meta)heuristics
Bounds and optimality proofs	✓	
Systematic and complete search	✓	
Can prove infeasibility	✓	
Scales to large instances	(✓)	✓
Good anytime behavior		✓
Problem-aware search		(✓)

Question: Can we benefit from their complementary nature?

Goals

- **Early search stage:**
  - Heuristics provide good feasible solutions
  - **Heuristics guide solver's search to promising regions early**
- **Advanced search stage:**
  - Solver incrementally improves and provides bounds
  - **Solver helps heuristics to escape local optima**
- **Overall robustness:**
  - Adversarial instances to one method solved by others

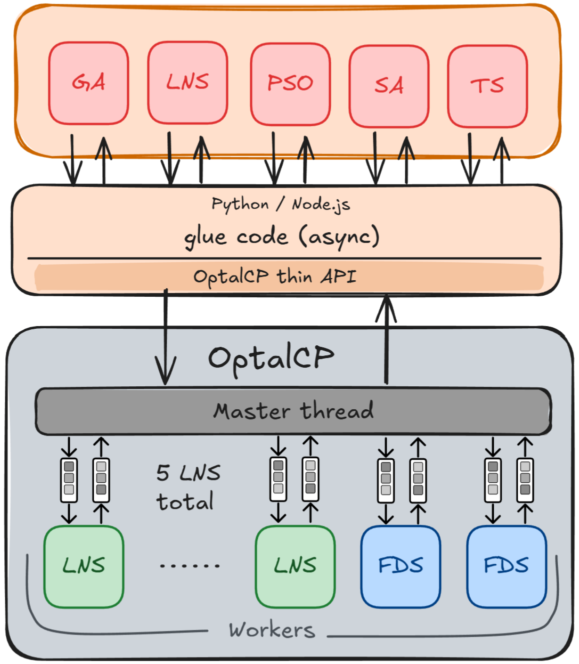


Benchmark Problem Classes

- Two application domains
- Scheduling
  - Flow Shop (FSSP)
  - Job Shop (JSSP)
  - Resource-Constrained Project Scheduling Problem (RCPSP)
- Routing
  - Travelling Salesman Problem (TSP)
  - Capacitated Vehicle Routing Problem (CVRP)
  - Vehicle Routing Problem with Time Windows (VRP-TW)

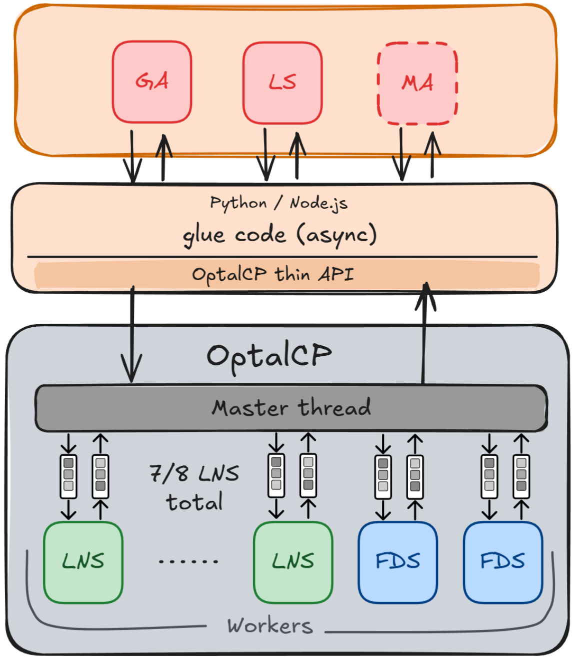
Configuration Scheduling

- 5 different metaheuristics for scheduling problems
  - Genetic Algorithm (GA)
  - Large Neighborhood Search (LNS)
  - Particle Swarm Optimization (PSO)
  - Simulated Annealing (SA)
  - Tabu Search (TS)
- 12 threads
  - one thread for each heuristic (5 threads total)
  - rest for solver (5 LNS workers + 2 FDS workers)



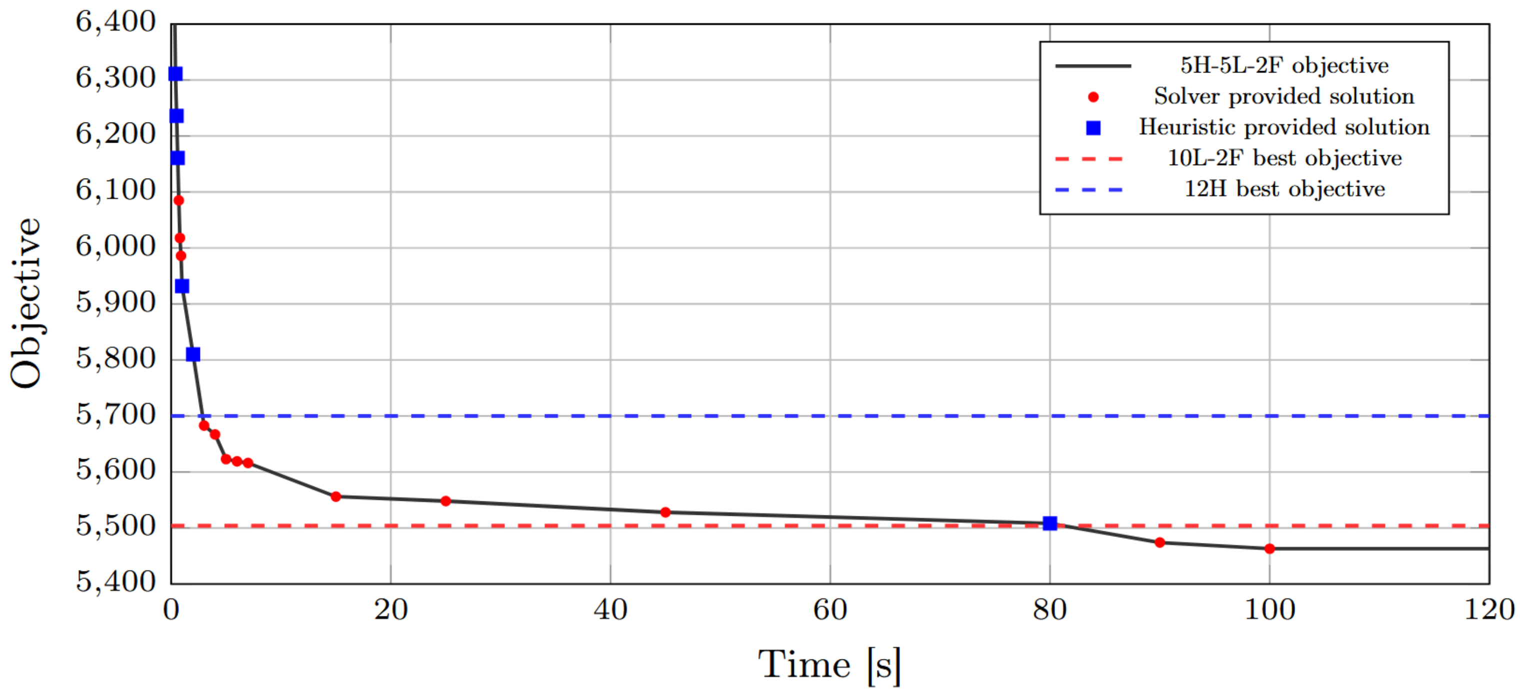
Configuration Routing

- 2/3 different (meta)heuristics for routing problems
  - Genetic Algorithm (GA)
  - Local Search (LS)
  - Memetic Algorithm (MA) - only for VRP-TW
- 12 threads
  - one thread for each heuristic (2/3 threads total)
  - rest for solver (7/8 LNS workers + 2 FDS workers)



Practical Example

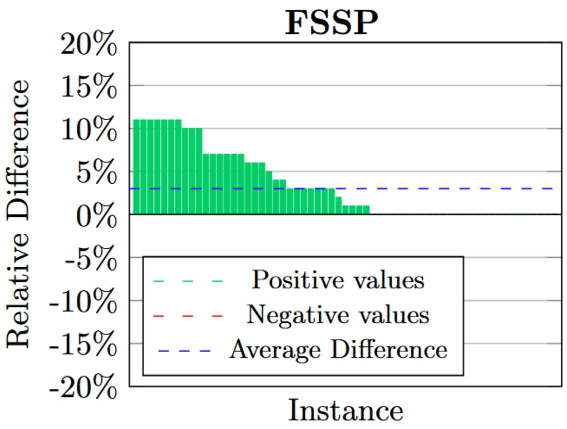
- Job Shop instance cscmax_40_15_7

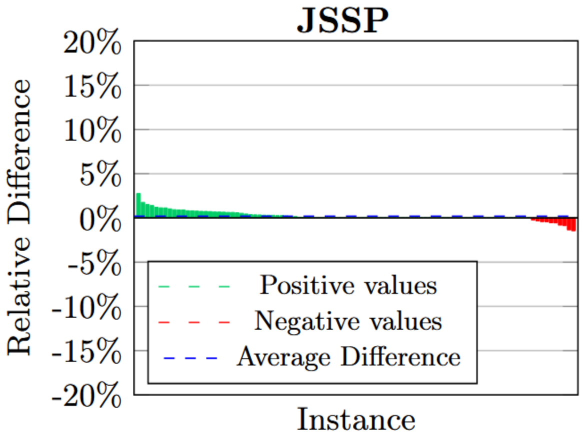


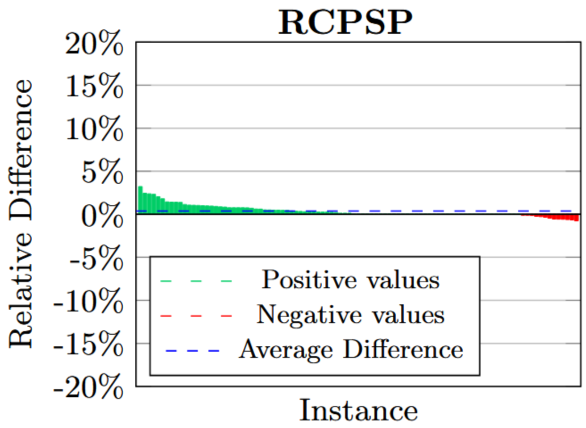
- Produces better solution than solver or heuristic portfolio alone
  - Different methods clearly profit from real-time exchange
  - Heuristics are useful again once local optima is escaped

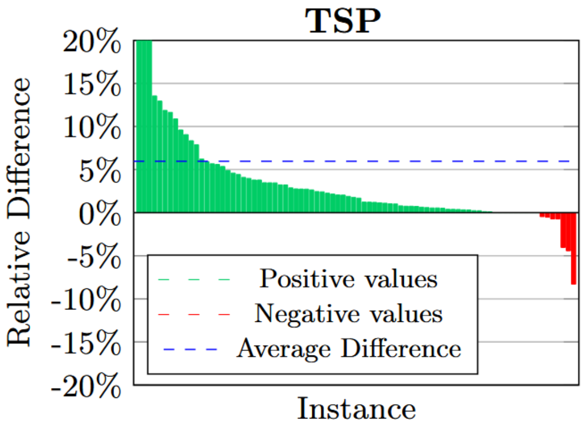
Results: Instance Solutions

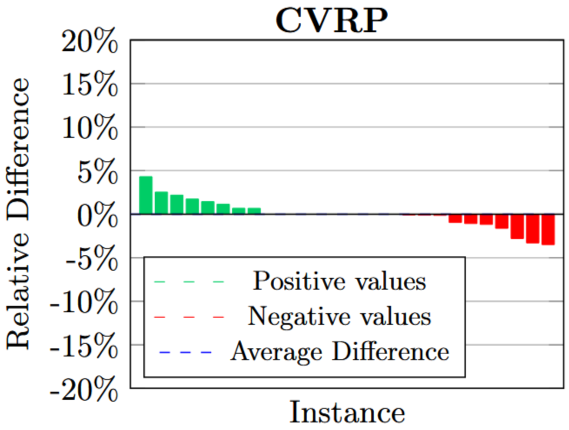
- Green bars denote instances where hybrid outperformed solver, red denote the opposite
- 120s runtime per instance

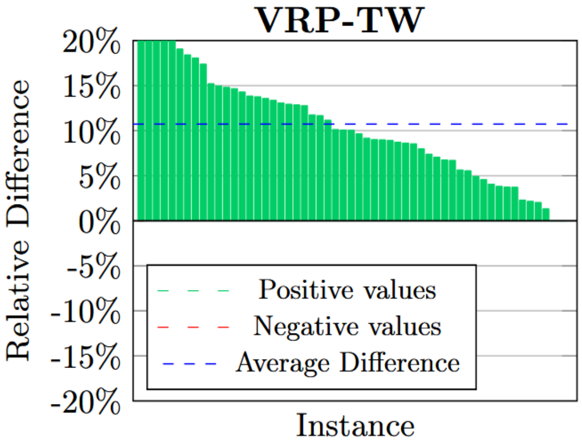












Conclusion

Hybridization improves overall solution quality, anytime behavior and robustness.

- Clear improvements on 3 problem classes
- Overall **60%** of solutions **improved**, only **10% worsened** (mostly slightly) **using same resources**
- Improved overall robustness
  - A few instances unsolved by OptalCP alone were solved by hybrid configuration
- (Meta)heuristics used were not state-of-the-art
  - Still potential for improvements in problems where solver is strong (JSSP, RCPSP)

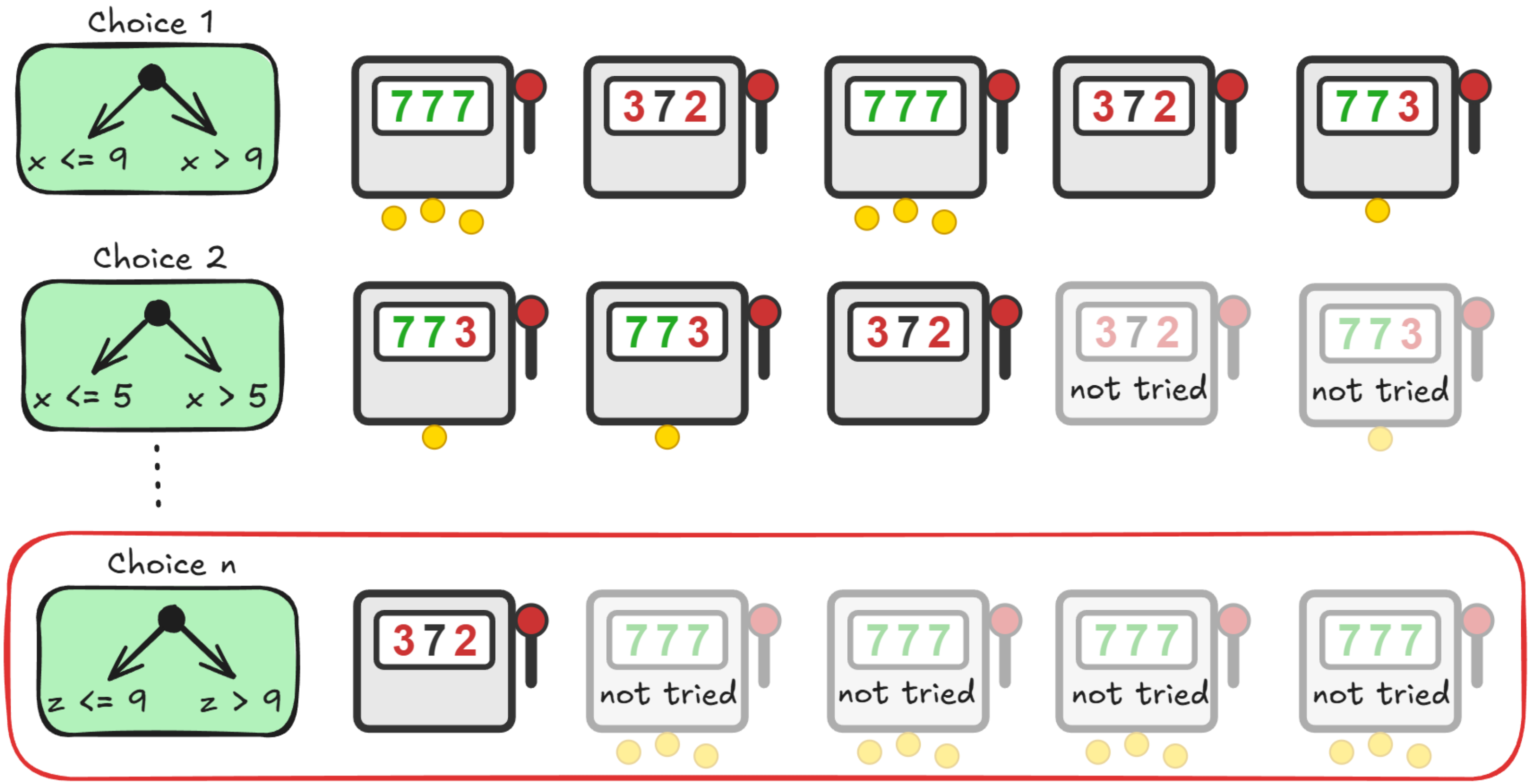
Problem	**Improvement**
FSSP	+3% average
JSSP	marginal
RCPSP	marginal
TSP	+6% average
CVRP	marginal
VRP-TW	+11% average

Accelerating FDS with Reinforcement Learning

Application of Multi-Armed Bandit Algorithms

Research

Motivation



- FDS always picks (undecided) choice with best rating
- Good choices with bad initial/recent performance are ignored
    - Choice success can depend on current search state (previous choices)

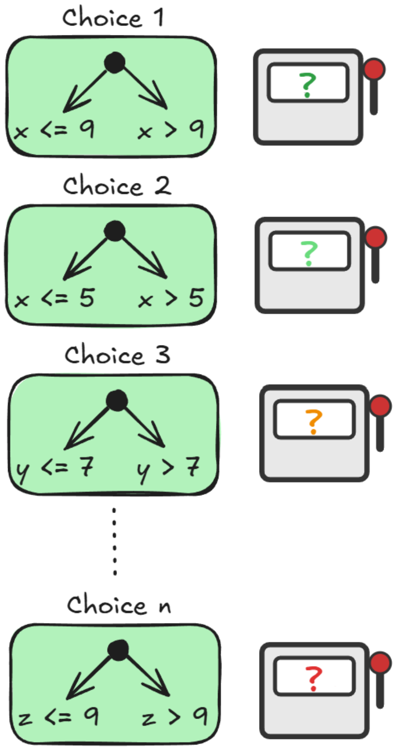
Question: How to prevent missing such good choices?

Goals

- We need to sufficiently test all choices to get an accurate assessment
  - Test all choices initially
  - Revisit bad choices occasionally
- **Enforcing initial choice exploration**
  - Initialize all choices with good rating (optimistic initialization)
- **Reassess choice quality efficiently**
  - FDS choice selection problem ≈ MAB problem

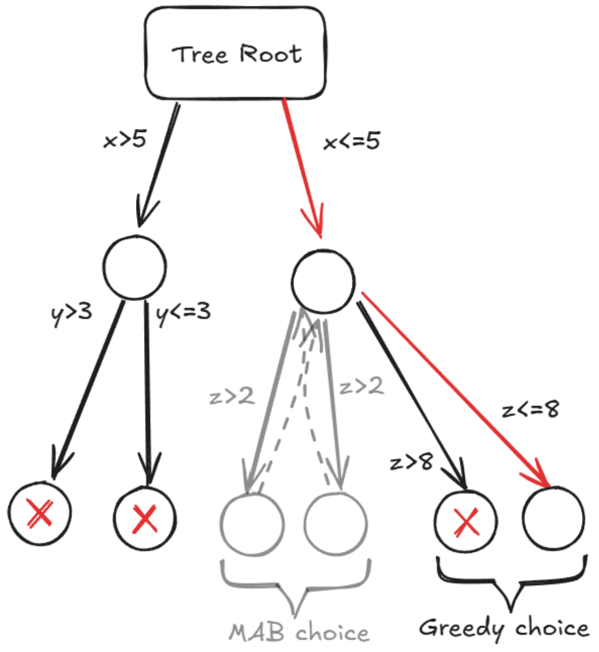
Reinforcement Learning: MAB Problem

- Exploration-Exploitation dilemma (the problem we have)
  - When to pick the best-known action (exploit)
  - When to test new/under-used actions (explore)
- **Multi-Armed Bandit problem**
  - Framework solving exploration-exploitation dilemma
  - Different algorithms/ways to handle exploration
  - Epsilon-greedy (ε), UCB-1 (U), Boltzmann exploration (B), Thompson sampling (T)
  - MAB reward maximization → Search tree size minimization

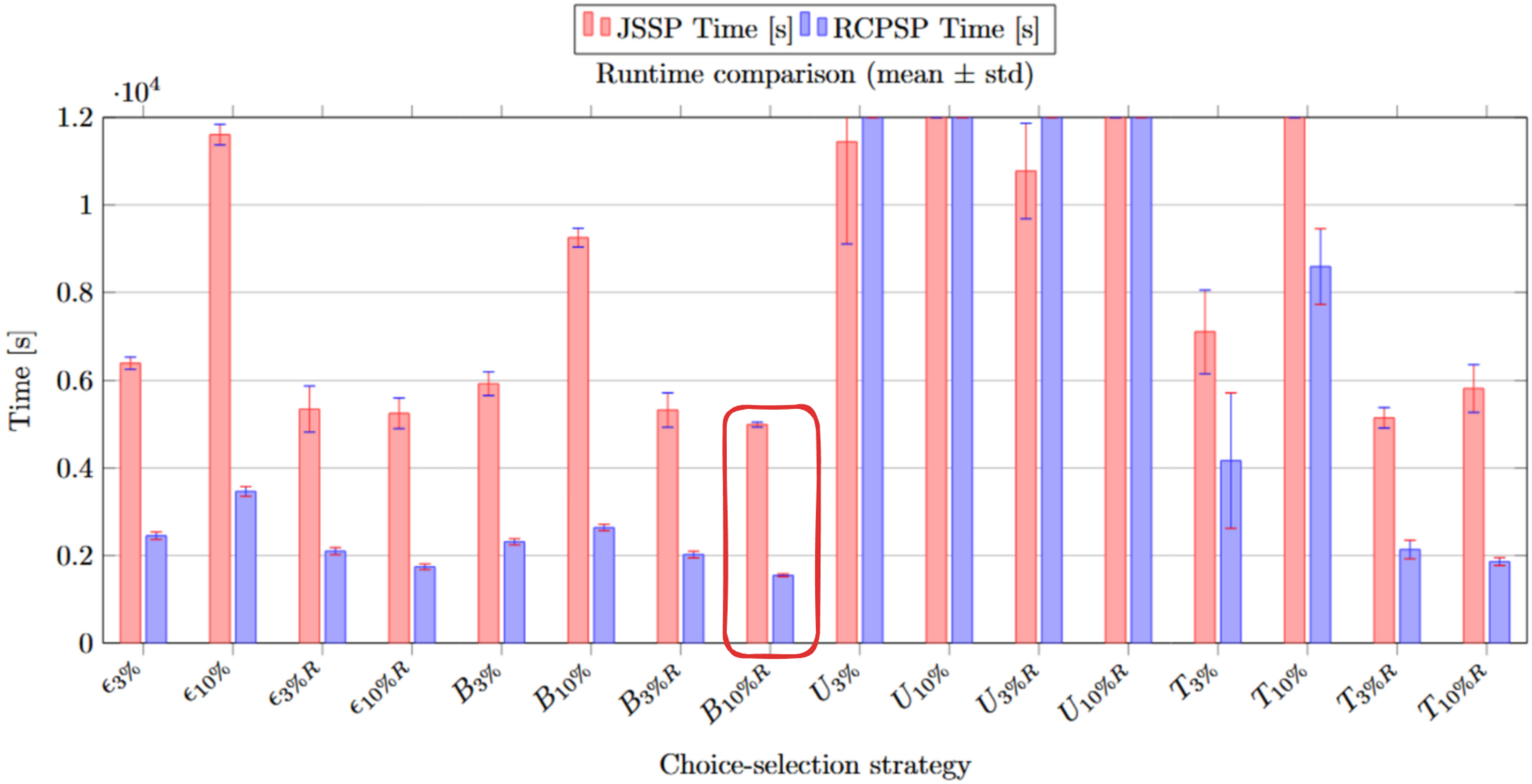


Exploration in FDS Setting

- **MAB-based exploration can be costly in FDS**
  - Bad exploratory choice = doubling the tree size
- **Switch between pure exploitation and MAB strategy**
  - In most cases, we exploit
- **MAB-based choice rollback**
  - “Test run” to evaluate effect and update rating (exploration)
  - Choice is used if it does not increase search tree size, else best-rated choice is used (exploitation)



Results: Selection Strategies

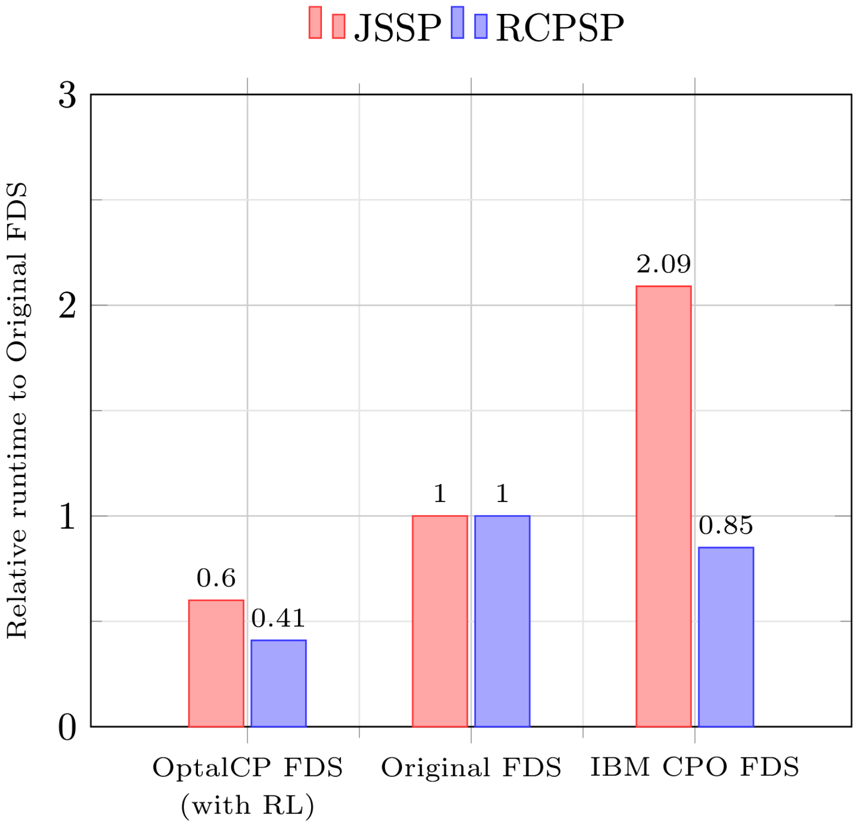


- Measurements on JSSP and RCPSP (percentage denotes MAB choice probability)
- **10% of Boltzmann exploration with Choice rollback performs the best**
- UCB-1 and Thompson embed exploration in action values (actions regain priority)
  - Degrades performance in FDS exploitation-heavy setting

Conclusion

Application of extended MAB algorithm with optimistic initialization roughly halved the computation time required by FDS in JSSP and RCPSP instances.

- Improved a large number of lower bounds for both problems
  - 78/84 of open standard Job Shop (JSSP) instances
  - 226/393 of open standard RCPSP instances
  - A few instances for both problems were closed
  - 900s time limit per instance



Heinz, S., Vilím, P., & Hanzálek, Z. (2025). *Reinforcement learning for search tree size minimization in constraint programming.*

Thank You!

Questions?

Website: https://optalcp.com

Benchmarks GitHub: https://github.com/scheduleopt/optalcp-benchmarks

> **Academic Licenses:** Send me your GitHub username