BOLIDE: BOosted LInear DEcoding

Philippe Laborie
IBM Analytics

April 13, 2016

Overview

- Introduction
- Concepts and notations
- Prototype
- Protocols
- Structures
- Raw performances
- Basic meta-heuristic

Introduction

Objective #1

Search method for computing **reasonable quality** solutions to potentially **very big** scheduling problems expressed with CP Optimizer

- Typical size: 1M interval variables (activities)
- Our customers already have these problems (cpo file available!):
  - Embraer: aircraft assembly scheduling with ∼1M activities
  - Tivoli TWS: RCPSP-like problems with up to ∼300K tasks
  - Voestalpine: production scheduling problem with ∼250K operations
- Current search is not able to handle these problems

Introduction

Objective #1

Search method for computing **reasonable quality** solutions to potentially **very big** scheduling problems expressed with CP Optimizer

- Typical size: 1M interval variables (activities)
- Our customers already have these problems (cpo file available!):
  - Embraer: aircraft assembly scheduling with ∼1M activities
  - Tivoli TWS: RCPSP-like problems with up to ∼300K tasks
  - Voestalpine: production scheduling problem with ∼250K operations
- Current search is not able to handle these problems

Objective #2

Diversify the portfolio of techniques used in CP Optimizer to improve its **robustness** on small/medium size problems

Introduction

Example (a simple but large scheduling model)

```
using CP;
execute {
  cp.param.Workers = 1;
  cp.param.SearchType = 'DepthFirst';
}
int n = 1000000;
dvar interval v[1..n] size 1;
constraints {
  noOverlap(v);
}
```

Introduction

Example (a simple but large scheduling model)

```
using CP;
execute {
  cp.param.Workers = 1;
  cp.param.SearchType = 'DepthFirst';
}
int n = 1000000;
dvar interval v[1..n] size 1;
constraints {
  noOverlap(v);
}
```

- Run time : **6 days** (estimated)
- Memory use : **20TB** (estimated)

What's wrong ?

On this problem at each search node (so n times) CP does the following:

- **Select the next variable** to fix: naive selection would require $O(n)$, for scheduling we are a bit smarter and can do it (usually) in $O(\sqrt{n})$ thanks to scopes
- Fix the selected variable and **propagate on all impacted variables**: this requires at least $O(n)$ operations as there are $O(n)$ variables whose domain changes
- If we are in the search tree, it requires at least $O(n)$ memory in the **trail**

Result: $O(n^2)$ for both time and memory

Introduction

What can we do ?

- **Don't select the next variable** to fix: assume a given static order of decisions is given (e.g. by some meta-heuristics)
- **Don't propagate on all impacted variables**: compute a tight domain only for the (known) variable that is being fixed
- **Don't trail**: use a greedy algorithm for decoding the static order of decisions: no search tree, no backtrack, no reversibility (no bugs ☺)

Result: targeted complexity for decoding a static order of decisions is in $O(n)$ or $O(n \log(n))$ for both time and memory

Introduction

Example (a simple but large scheduling model)

```
using CP;
execute {
  cp.param.Workers = 1;
  cp.param.SearchType = 'DepthFirst';
}
int n = 1000000;
dvar interval v[1..n] size 1;
constraints {
  noOverlap(v);
}
```

- Run time : 6 days $\rightarrow$ <**1s** (ratio >500000)
- Memory use : 20TB $\rightarrow$ <**1GB** (ratio >20000)

Introduction

BOLIDE = BOosted LInear DEcoding

- **BO**osted because it is supposed to be *much faster* than classical CP on large problems
- **LI**near because:
  1. It works on *linear* structures (sequences of decisions) and
  2. Decoding a decision list is supposed to be achievable in *quasi-linear* time and memory
- **DE**coding because it focuses on *decoding* a decision list

Introduction

BOLIDE = BOosted LInear DEcoding

- **BO**osted because it is supposed to be *much faster* than classical CP on large problems
- **LI**near because:
  1. It works on *linear* structures (sequences of decisions) and
  2. Decoding a decision list is supposed to be achievable in *quasi-linear* time and memory
- **DE**coding because it focuses on *decoding* a decision list

The big challenge

Be generic and robust enough !

- We do not want to improve CPO on toy problems only ...
- We want to support all the concepts of CPO

The reality check

All that is presented here has been implemented as a prototype

```
! ------------------------------------------------------------------------
! Number of model variables     : 1000000 ( interval=1000000 )
!                 constraints   : 1
!                 expressions   : 0
! Number of generated decisions : 1000000
! Peak memory usage             : 741.6 MB
! Final memory usage            : 741.6 MB
! Initialization time           : 390.002 ms
! ------------------------------------------------------------------------
! Iteration   Type        Time (s)    Non-fixed
         1   -           0.561604            0
! ------------------------------------------------------------------------
! Number of iterations   : 1
! Peak memory usage      : 762.2 MB
! Total time             : 639.604 ms
! Mean iteration time    : 171.601 ms
! Iteration speed (it/s) : 5.82747
! ------------------------------------------------------------------------
```


Concepts and notations

- Variables
- Constraints
- Structures
- Decisions
- Decision Lists
- Decision DAG
- Decoder
- Re-encoding

Concepts and notations

Example (FJSP: flexible job-shop scheduling problem)

```
dvar interval op[i in ...];
dvar interval alloc[i in ...][k in ...] optional size ...;
minimize max(i in ...) endOf(op[i]);
subject to {
  forall(<i,j> in ...) {
    endBeforeStart(op[i], op[j]);
  }
  forall(i in ...) {
    alternative(op[i], all(k in ...) alloc[i][k]);
  }
  forall(m in ...) {
    noOverlap(all(<i,k> in ...) alloc[i][k]);
  }
}
```

Concepts: Variables

- A **variable** is an object associated with a domain that get fixed during the process to compute a solution. Typical examples include:
  - Integer and interval variables declared in the model
  - Results of numerical expressions (integer, floating point)
- Note that large variable structures like *sequences* or *cumul* or *state functions* are not considered as variables in this context because they usually get fixed only at the very end. They will be called *structures*
- The set of all variables (denoted $\mathcal{V}$) is automatically partitioned into a set of **decision variables** (denoted $\mathcal{V}_\mathcal{D}$) and the **other variables** (denoted $\mathcal{V}_\mathcal{O}$). Typically:
  - Results of numerical expressions are not decision variables
  - Master of alternative or span constraints are not decision variables
- The idea is that it is enough to fix only the decision variables of $\mathcal{V}_\mathcal{D}$

Concepts: Variables

Example

In the FJSP the following variables are created:

- Decision variables:
  - Interval variables `alloc[i][k]`
- Other variables:
  - Interval variables `op[i]`
  - Integer variables `end_i` for expressions `endOf(op[i])`
  - Integer variable for the result of the `max` expression (objective)

Prototype

In the current version of the prototype the following variables are implemented:

- Integer variables
- Interval variables (without intensity function)

Concepts: Constraints

- **Constraints** are objects that express some restriction on the possible value of variables (so this covers the expressions too)
- Notations:
  - The set of all constraints of the problem: $\mathcal{C}$
  - A constraints $c \in \mathcal{C}$ is defined on a set of variables $V \subset \mathcal{V}$
  - $V(c) \subset \mathcal{V}$ denotes the set of variables of a constraint c (its scope)
  - $C(v) \subset \mathcal{C}$ denotes the set of constraints holding on a variable v

Concepts: Constraints

Example

In the FJSP the following constraints are created:

- Precedence constraints:
  - `endBeforeStart(op[i], op[j])`
- Alternative constraints:
  - `alternative(op[i], all(...)  alloc[i][k])`
- No-overlap constraints:
  - `noOverlap(all(...)  alloc[i][k])`
- Interval expression constraints:
  - `eqEndOf(end_i, op[i])`
- Max expression constraint:
  - `eqMax(obj, all(...)  end_i)`

Concepts: Constraints

Prototype

In the current version of the prototype the following constraints are implemented:

- Simple precedence constraints with fixed delay value (0 by default)
- Simple alternative constraint (cardinality=1)
- Simple no-overlap constraint (no transition distance, no transition expressions)
- Expressions:
  - Interval expressions: `startOf`, `endOf`, `lengthOf`, `presenceOf`
  - Arithmetical integer expressions: `max`, `sum`

Concepts: Structures

- Constraints may maintain and share some **Structures**
- A structure is used to store the current state of a constraint
- Structures are usually maintained in an incremental way
- They are used to speed-up the processing of constraints

Example

For instance in the FJSP, the `no-overlap` constraint maintains a gap structure (see later)

Concepts: Decisions

- Decision variables of the problem needs to be fixed during the search
- A **decision** is an action on a given decision variable in order to fix it (or strongly restrict its domain)
- Executing a decision may result in two situations:

  Success The domain restriction implied by the execution of the decision was applied and was not shown to be inconsistent with the current state of the problem

  Failure The domain restriction implied by the execution of the decision was shown to be inconsistent with the current state of the problem

Concepts: Decisions

Prototype

In the current version of the prototype the following decisions are implemented:

- FixStart(v) If interval variable v is not absent: set v to be **present** and fix its **start** value at the **smallest** value in the domain
- FixEnd(v) If interval variable v is not absent: fix the **end** value of v at the **smallest** value in the domain

Decision integer variables are still not handled

Concepts: Decisions

- Notations[1]:
  - $\Delta(v)$ denotes the set of decision on a decision variable $v \in \mathcal{V}_{\mathcal{D}}$
  - Δ denotes the set of all decisions of the problem: $\Delta = \bigcup_{v \in \mathcal{V}_{\mathcal{D}}} \Delta(v)$
  - For a decision $\delta \in \Delta$, we denote $v(\delta)$ its decision variable

Property: Completeness of a set of decisions

A set of decision Δ is said to be **complete** if it ensures that once all the decisions $\delta \in \Delta$ have been executed all the decision variables $\mathcal{V}_{\mathcal{D}}$ of the problem are fixed

[1] As a rule of thumb we denote by Latin letters the elements of the direct model and by Greek letters the elements of the indirect encoding

Concepts: Decision Lists

- A **decision list** is a total order on the set of decisions of the problem
- More formally, we define a decision list π as a permutation of Δ and we denote Π the set of all decision lists
- For $\delta \in \Delta$, we denote $\pi[\delta] \in [1, |\Delta|]$ the position of δ in list π
- The goal of BOLIDE engine is to quickly decode a given decision list π into a feasible solution of the problem

Concepts: Decision DAG

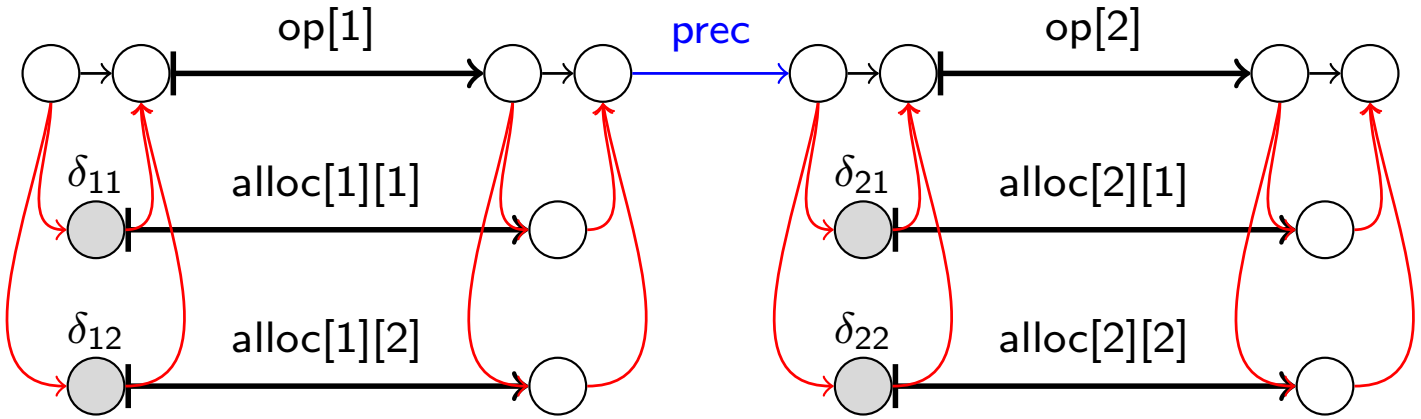
- The BOLIDE approach relies on a strong directionality of the optimization problem being solved
- Typically the case for scheduling problems as the time axis and the temporal relationship between causes and effects is naturally oriented
- In this context, we usually want to enforce that a given decision δ_i cannot be executed before another given decision δ_j has been executed.
- We denote $\Gamma = (\Delta, \prec)$ the resulting **Directed Acyclic Graph of decisions**
- Decision DAG Γ is **automatically** extracted from the problem by analyzing the variables and the constraints
- Additional arcs can be added into the decision DAG as guidelines for the search (search phases, or finer grain: fix y right after x)

Concepts: Decision DAG

Example

In the FJSP, let δ_{ik} denotes decision `FixStart(alloc[i][k])`, the decision DAG $\Gamma(\Delta, \prec)$ will be such that for all precedence constraint `endBeforeStart(op[i], op[j])` and for all k, k': $\delta_{ik} \prec \delta_{jk'}$

The actual DAG for the FJSP contains additional vertices with no decision variables for synchronization purpose. Before simplification, it looks like:



Concepts: Decoder

- The **decoder** is in charge of decoding a decision list π
- The decoding of an instance of decision list by the decoder is called an **iteration**

Concepts: Decoder

- The **decoder** is in charge of decoding a decision list π
- The decoding of an instance of decision list by the decoder is called an **iteration**
- An iteration performs two things:
  1. Reconciliate the decision list π with the decision DAG $\Gamma = (\Delta, \prec)$. This means re-ordering the decisions in π to create a new decision list $\gamma(\pi)$ (or just γ) that respects the partial order implied by Γ. More formally, γ is uniquely defined as follows:

$$\begin{aligned}
&\forall \delta, \delta' \in \Delta, \delta \neq \delta' : \\
&\quad \gamma[\delta] < \gamma[\delta'] \Leftrightarrow \\
&\quad \Big(\delta \prec \delta'\Big) \vee \Big(\neg(\delta' \prec \delta) \wedge \big(\pi[\delta] < \pi[\delta']\big)\Big)
\end{aligned}$$

  2. Sequentially execute the decisions of $\gamma(\pi)$ to (hopefully) produce a solution $s(\gamma(\pi))$

Concepts: Decoder

Property: Soundness of the decoder

The decoder is **sound** if it ensures that when all the decision variables $\mathcal{V}_{\mathcal{D}}$ of the problem have been fixed then all the other variables $\mathcal{V}_{\mathcal{O}}$ are fixed and all the constraints $\mathcal{C}$ of the problem are satisfied

Property: Completeness of the decoder

The decoder is **complete** if it ensures that there exists a feasible optimal solution to the optimization problem s^* and a decision list π^* such that decoding π^* will produce s^* $(s(\gamma(\pi^*)) = s^*)$

Concepts: Decoder

- The decoding of a decision list π may result in two situations:

  Success All decisions could be executed. If the decision set Δ is *complete* and the decoding is *sound*, then all the variables are fixed and all the constraints are satisfied, thus a feasible solution was produced

  Failure A decision failed. This means that the iteration did not allow to produce a feasible solution

Concepts: Re-encoding

- By the DAG reconciliation, different decision lists π may produce the same reconciliated decision list $\gamma(\pi)$
- By decision execution, different decision lists γ may produce the same feasible solution $s(\gamma)$
- Stated otherwise, none of the functions γ and s are injective so there is some amount of redundancy on the space of decision lists Π
- $s \circ \gamma$ partitions the set of decision lists into equivalence classes $\Pi = \Pi_1 \cup ... \cup \Pi_i \cup ... \cup \Pi_n$ such that for all partition element Π_i we have $\forall \pi, \pi' \in \Pi_i, s\big(\gamma(\pi)\big) = s\big(\gamma(\pi')\big)$
- Re-encoding a decision list π consists in selecting a unique representative $\rho(\pi)$ in the equivalence class of π

Prototype

- Currently implemented in a CPO-independent directory that mimics the structure of the CPO one (include/ilcp, src, bench, lib, ...)
- Includes some basic structures defined in CPO (numerical types, arrays, heaps, DAG)
- Classes are all prefixed with `Ilb`
- Protocols are implemented on virtual classes (`IlbVar`, `IlbConstraint`, `IlbDecision`)
- Model is created through a set of `IlbDecoder::createXXX` methods that mimic the layer API so that connection with layer conversion should be straightforward. Example:

```
IlbIntervalVar* createIntervalVar(IlcInt smin, ...);
IlbPrecedence*  createPrecedence(IlcSched::PrecedenceType type,
                                 IlbIntervalVar* source,
                                 IlbIntervalVar* target,
                                 IlcInt delay = 0);
```

Prototype

- The decoder implemented in the prototype is **sound**
- For most *regular* scheduling problems:
  1. The set of decisions `FixStart(v)`, `FixEnd(v)` implemented in the prototype is **complete**
  2. The decoder implemented in the prototype is **complete**
- Example of exceptions:
  1. Cases we want to execute *as few tasks as possible*
  2. Cases when it is *necessary* to delay some tasks only because of the objective function (e.g. earliness costs)

Protocols

- Once the model is created in the engine it performs an **initialization** to initialize the elements that survive from one iteration to the other
- Then, **iterations** are performed to decode decision lists
- Unless stated otherwise when we speak of the *size n* of the problem, we mean:

$$n = |\mathcal{V}| + sum_{c \in \mathcal{C}} |V(c)|$$

Protocols: Initialization

- Each variable v stores an ordered list of its constraints $C(v)$
- Notion of **constraint weight** used for performance reasons: basic idea is that is is better to first consider the constraints that are "light" to process
- Example of weights in the prototype:
  - Interval expressions: $weight = 1$
  - Alternative constraint: $weight = 1$
  - Precedence constraint: $weight = 2$
  - Max/Sum expressions: $weight = 3$
  - No-overlap constraint: $weight = 10$

Protocols: Initialization

- The **initialization** does the following:
  1. Store the constraints in the *constraint list* of each variable by increasing weight
  2. Partition the variables into *decision* and *non-decision* variables by going through the constraints
  3. Perform some very basic *local propagation* on initial domain of variables
  4. Create/initialize the *structures* owned by the constraints
  5. For each decision variable, generates its *decisions* and *decision vertices* in the decision graph
  6. Generate *arcs* between decisions in the decision graph by going through the variables and constraints
  7. *Simplify* the decision graph
  8. Create the *heap nodes* for the binary heap that will be used at each iteration to traverse the decision DAG
- Complexity of initialization step is in $O(n \log(n))$

Protocols: Iteration

- An **iteration** is in charge of decoding a decision list π, it does the following:
  1. *Re-initialize* all the model elements (variables, constraints, structures) to the state they were after the **initialization**. This means in particular that variables have two instances of domain: *original* and *current* domains. At re-initialization the original domain is just copied on the current domain $\rightarrow O(n)$
  2. *Re-initialize* the key of the decision heap nodes by the position of the decision in π: $\forall \delta \in \Delta : key(\delta) = \pi[\delta] \rightarrow O(n)$
  3. Construction of $\gamma(\pi)$: traverse the decision DAG in a topological order inserting into the decision heap the executable decisions (the ones with no predecessor still not executed) and poping the most prioritary decision δ in the binary heap $\rightarrow O(n \log(n))$
  4. Sequentially *execute* decisions in $\gamma(\pi)$ (see later)
  5. Create *re-encoded* list ρ (see later)
  6. Report *failure* or *success*, on *success* a feasible solution is available
- Complexity of an iteration should not be more than $O(n \log(n))$

Protocols: Decision execution

- The **execution** of a decision δ on a decision variable v involves 3 steps:
  1. **Pull** the current domain of v. Pulling a variable means asking the different constraints what is the current domain of v (see later)
  2. Perform the action specified by the decision. This action involves restricting/fixing the domain of v
  3. **Push** decision variable v in the queue of fixed variables in case v was fixed by the decision. Pushing a variable v means notifying the different constraints that the variable was fixed (see later)

Protocols: Push mechanism

- The **push** mechanism is quite similar to constraint propagation limited to `whenValue` events: it pushes fixed variables in a queue
- Two objectives:
  1. Fix the non-decision variables when all the decision variables they depend on are fixed[2]
  2. Update the incremental structures maintained by the constraints so that the **pull** mechanism will be faster

[2] *Provided it is cheap enough* the constraints can also update the domain of other variables but this is optional

Protocols: Push mechanism

Queue propagation

Stamp++ ▷ Used for re-encoding, see later
while *queue* is not empty **do**
    $v \leftarrow queue.extractFirst()$
    $v.stamp \leftarrow Stamp$ ▷ Used for re-encoding, see later
    for all $c \in C(v)$ **do**
        $c.push(v)$ ▷ Can queue some new fixed variables

- Exactly $|\mathcal{V}|$ variables enter the queue during an iteration
- Targeted complexity of $c.push(v)$ is $O(1)$ or $O(\log(n))$

Protocols: Pull mechanism

- The **pull** mechanism is called when a decision on a decision variable v is executed
- Main idea is to call a function $c.pull(v)$ on all constraints $c \in C(v)$ to restrict the current domain of v until a fix point is reached for the domain

Pull algorithm

$v.changed \leftarrow true$
while $v.changed$ **do**
    $v.changed \leftarrow false$
    for all $c \in C(v)$ **do**
        $c.pull(v)$ ▷ May change the domain of v and mark it changed

Protocols: Pull mechanism

Actual **pull algorithm** is a bit more complex:

- Several optimizations are possible
  - Some constraints may be non-reentrant meaning that for them $c.pull(v)$ can be called only once
  - Constraints $c \in C(v)$ are visited by increasing weight so that "light" constraints are treated first
  - If v was not changed since last call to $c.pull(v)$ fo a given c one can exit the loop
- It may involve pulling (a few) other variables than just v (for instance: alternative masters)
  - Currently this is just implemented as recursive calls to $v.pull()$
  - Protection against cycles: pulling a variable that is already being pulled has no effect

Protocols

In the end, the virtual class of constraint looks like this:

```
class IlbObject {
public:
  virtual IlcBool init() =0; // Initialization, returns IlcFalse if inconsistent
  virtual void  reinit() =0; // Re-initialization
  virtual void display(ILOSTD(ostream)& str) {}
};

class IlbConstraint :public IlbObject {
public:
  virtual void generateArcs(){}          // Decision DAG arcs at initialization
  virtual IlcBool push(IlbVar*) =0;      // Returns IlcFalse if inconsistent
  virtual IlcBool pull(IlbVar*) =0;      // Returns IlcFalse if inconsistent
  virtual IlcInt  getWeight() const =0; // Constraint weight
  virtual IlcBool check()     const =0; // For solution checker [DEBUG]
};
```

Protocols: Re-encoding

- **Re-encoding** uses the notion of *stamp* that reflects how the fixation of the different decision variables was sequenced and synchronized during the iteration

Protocols: Re-encoding

- Each decision δ is associated the following information:

  stamp The stamp of variable $v(\delta)$. Decisions with identical **stamp** were resolved in the same propagation loop, they define a **decision block**. Each block was initiated by the execution of one decision of the block: the **initiator**

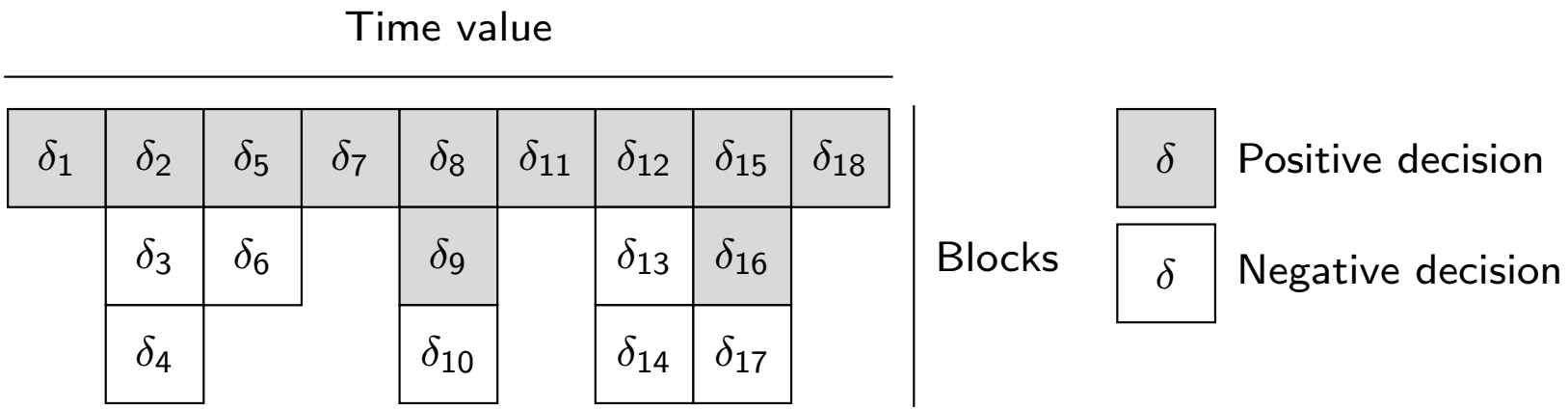
  positive A boolean stating whether or not the variable was fixed according to the preference specified in the decision

  time In a scheduling context, this is the *date* at which the endpoint of the interval variable related with the **initiator** decision was fixed

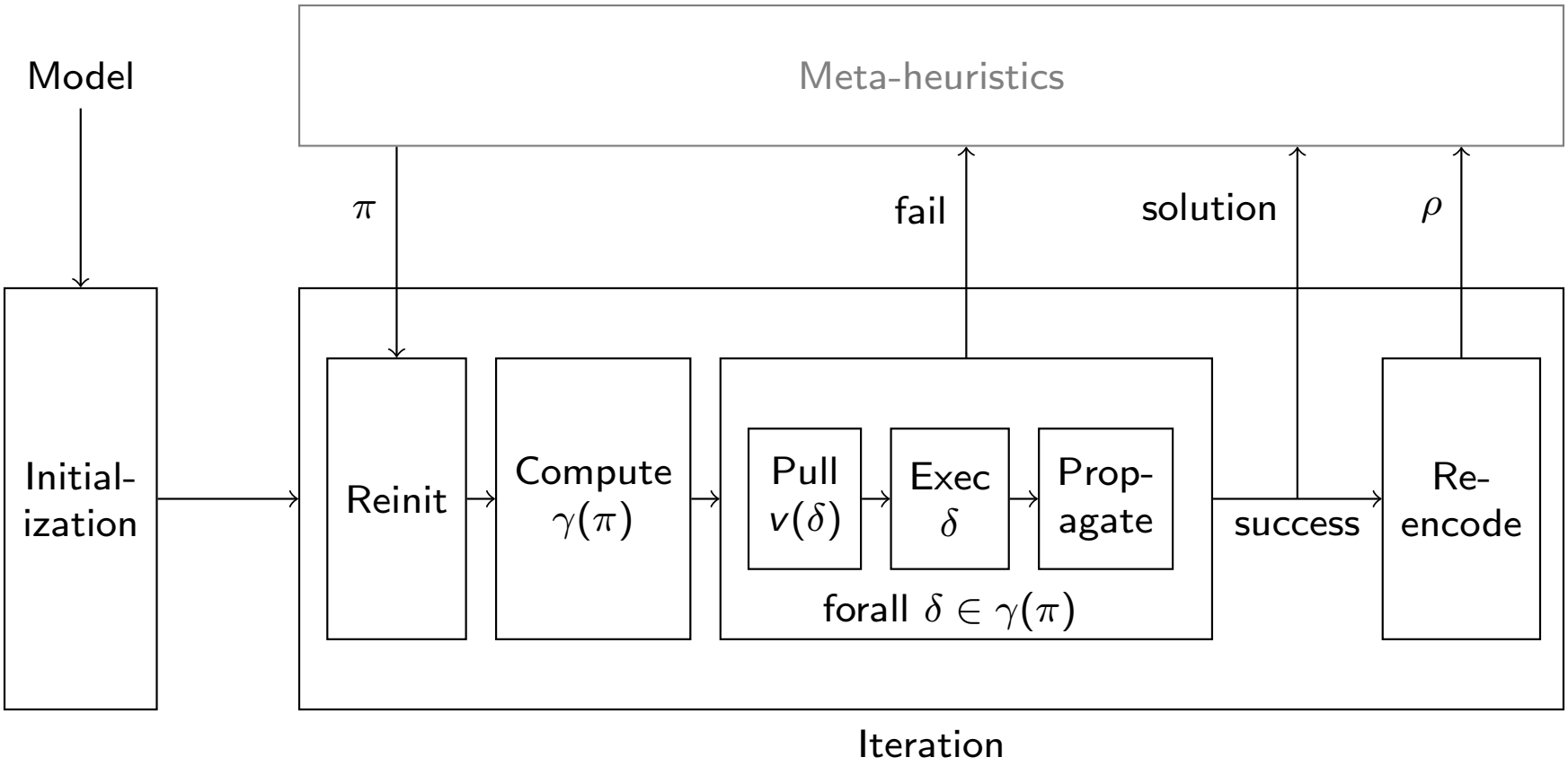
- Note that decoding is not strictly chronological: a decision decoded later (greater **stamp**) may have smaller **time** value (because of holes)

Protocols: Re-encoding

- **Re-encoding** stores the **decisions blocks** by increasing **time** value
- The structure of re-encoded decision lists can be exploited by the meta-heuristics



Protocols


Model
Meta-heuristics
π
fail
solution
ρ
Initial-
ization
Reinit
Compute
$\gamma(\pi)$
Pull
$v(\delta)$
Exec
δ
Prop-
agate
forall $\delta \in \gamma(\pi)$
success
Re-
encode
Iteration

Structures

- So far, two structures are implemented:
  1. Gap Binary Tree structure for the simple noOverlap constraint
  2. Cumul Step Function for the simple monotonous cumul functions

Structures: Gap Binary Tree for noOverlap

- The GBT maintains a balanced binary tree whose nodes are the gaps where non-fixed interval variables can fit in (Gap Binary Tree or GBT)
- A node n consists of 3 numbers:

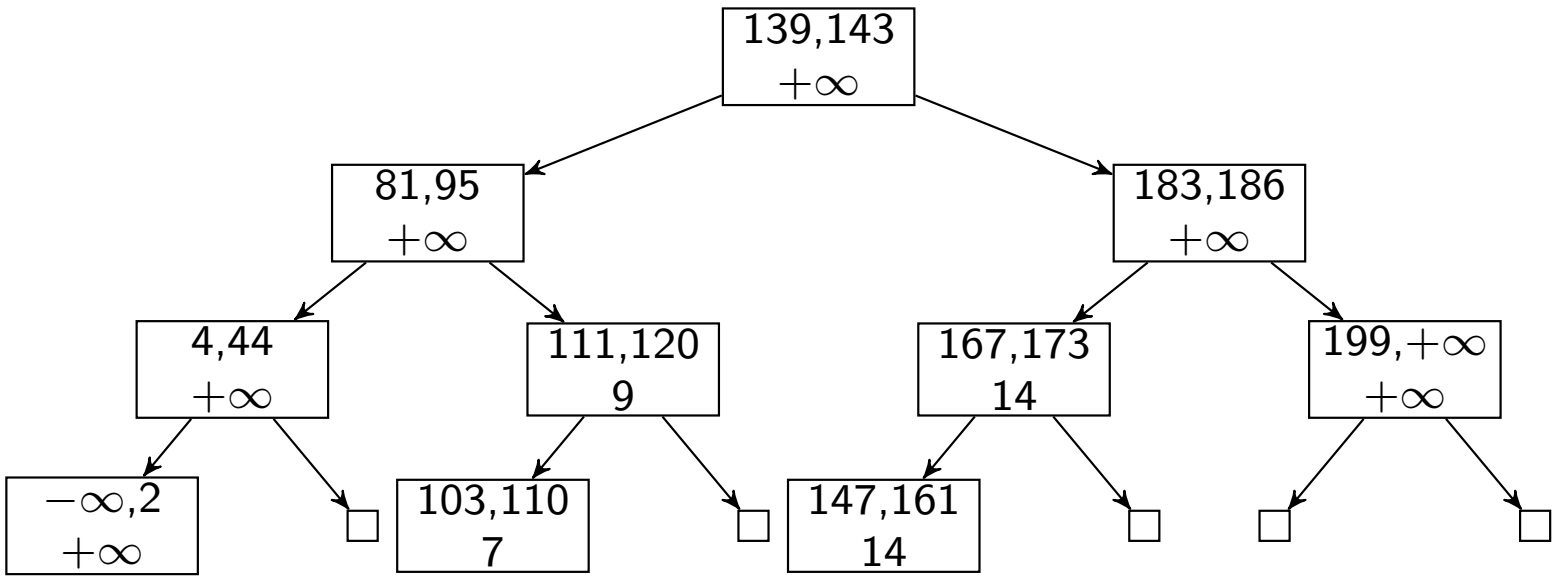
start	The start value of the gap
end	The end value of the gap
lmax	The maximal length ($m.end - m.start$) of all the gaps m in the subtree of n (including n itself)

- The interest of this structure is that it is $O(n)$ in memory and both the $pull(v)$ and the $push(v)$ functions can be performed in $O(\log n)$ in the **worst case**

Structures: Gap Binary Tree for noOverlap

Example

Suppose a `noOverlap` constraint on which a number of interval variables have already been fixed: $[2, 4)$, $[44, 81)$, $[95, 103)$, $[110, 111)$, $[120, 139)$, $[143, 147)$, $[161, 167)$, $[173, 183)$, $[186, 199)$



Structures: Gap Binary Tree for noOverlap

Pull algorithm is given as input an interval variable v with a given minimal start/end/length value ($smin$,$emin$,$lmin$) and is in charge of finding the earliest gap where such an interval v can fit and thus, a possible update of the minimal start time $smin$. It works in 2 steps:

1. Walk down the tree to find the latest gap n that is before $smin$. Interval v will have to fit into a gap that is after n. Because the tree is balanced, this step is in the worst case in $O(\log n)$
2. Starting from node n find the earliest gap after n with a length larger than $lmin$. In this second step, we do not care any more of the minimal start of v but only consider the maximal gap lengths $lmax$ stored on the nodes. In the worst case it requires moving up to the root node and moving down the right sub-tree of the root, so again $O(\log n)$

Structures: Gap Binary Tree for noOverlap

Push algorithm is given as input a freshly fixed interval variable $v = [s, e)$. It updates the GBT in 2 steps:

1. Walk down the tree to find the gap n that contains v. If none is found, report inconsistency. This steps is performed in the worst case in $O(\log n)$
2. In general the gap n will have to be split in two: the gap $[n.start, s)$ and the gap $[e, n.end)$. Some of these two gaps may be removed if their length is strictly smaller than $minl$, so we end up with 3 possibilities:
   - either completely **delete** node n, or
   - update its start or end time, or
   - update its start or end time and **insert** an additional node

   The only costly operations are node deletion and insertion which may require re-balancing the binary tree. The tree is implemented as an AVL tree, both operations can be done in $O(\log n)$ in worst case

Structures: Cumul Step Function

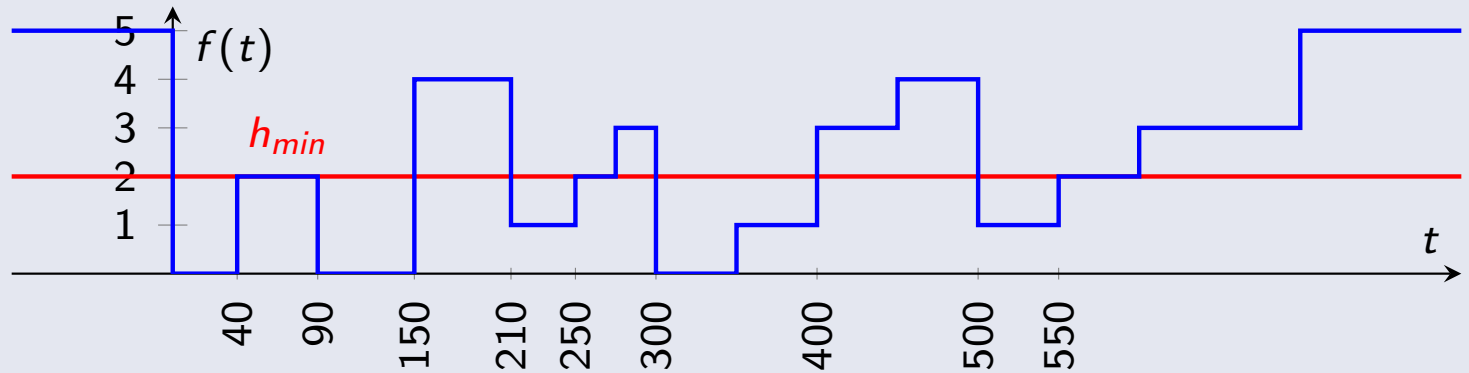
- This structure is used for cumul functions with the following characteristics:
  - Sum of **pulses** of **fixed height** only
  - Constraints alwaysIn on **fixed time-windows** with maximal value only (**no minimal value**)
- In spite of these restrictions, this is a very common case (cumulative resource with maximal availability profile)

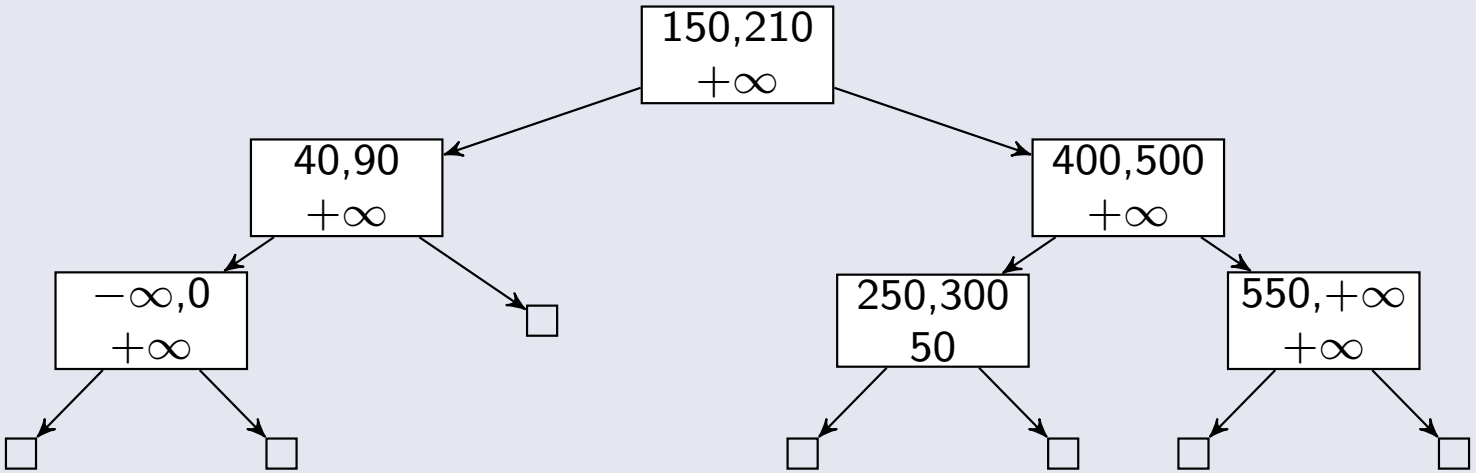
Structures: Cumul Step Function

- The structure is an association of a step function and a GBT structure
  - A step function $f(t)$ represents the remaining space of the cumul function. Steps are stored in a skip-list (average random access in $O(\log n)$)
  - If h_{min} denotes the smallest height of the pulses of the cumul function, a GBT structure maintains the segments of the step function of value $f(t) \geq h_{min}$

Structures: Cumul Step Function

Example





Structures: Cumul Step Function

- **Pull algorithm** is given as input an interval variable v with a given minimal start/length value ($smin$,$lmin$) and its height h. It is in charge of finding the earliest start $newsmin$ for v such that:
  - $smin \leq newsmin$ and
  - $\forall t \in [newsmin, newsmin + lmin), h \leq f(t)$
- If $h = h_{min}$, the GBT exactly captures all the gaps where v can fit in: same algorithm as for GBT on noOverlap
  - Worst case complexity in $O(\log n)$
- If $h > h_{min}$, v will have to fit into a gap of the GBT but the algorithm has to iterate over the steps of the function in each gap of length greater than $lmin$. Gaps are used as accelerators. Worst case complexity in $O(n)$ but much faster in practice
- Several GBTs could be maintained for different values of h. Experiments shown that it does not really pay off in practice and maintaining only the GBT for h_{min} is enough

Structures: Cumul Step Function

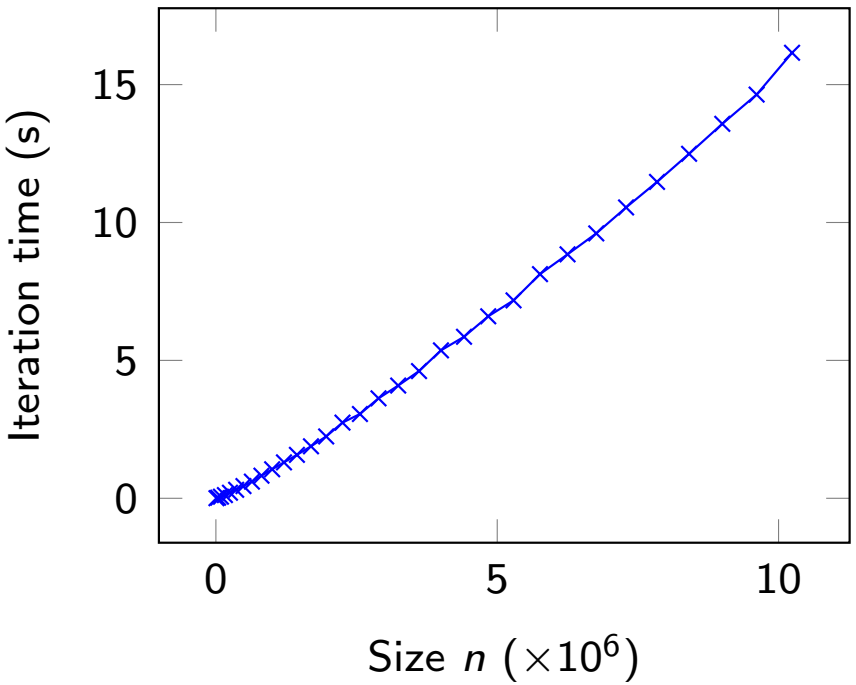
- **Push algorithm** is given as input a freshly fixed interval variable $v = [s, e)$ that contributes with a height h to the cumul function and iis in charge of updating the step function $f(t)$ and the GBT structure
- It first finds the step of s in $O(\log n)$ in average (skip-list)
- It then iterates over all the steps between s and e (we can assume this number of steps small in practice) and for each step, decreases the step value v by h. If $v - h < h_{min} \leq v$, the step is pushed in the GBT structure in $O(\log n)$ as described earlier

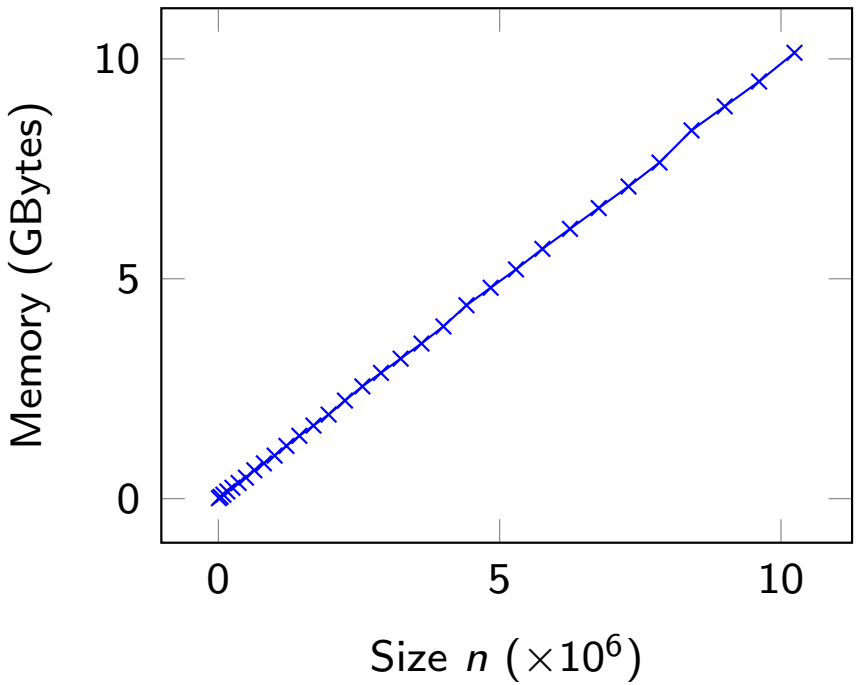
Raw performances

- Objective is to study the memory usage and average iteration time on some large scheduling problems
- Tested on large *job-shop*, *flexible job-shop* and *RCPSP* instances with up to 10M variables
- Tested on decoding randomly shuffled decision lists
- So far, no evaluation of *solution quality* (more on this later)

Raw performances: job-shop

- Average iteration time and peak memory usage for decoding a randomly generated square **job-shop** scheduling problem of size n (thus $n^{1/2}$ jobs $\times$ $n^{1/2}$ operations)





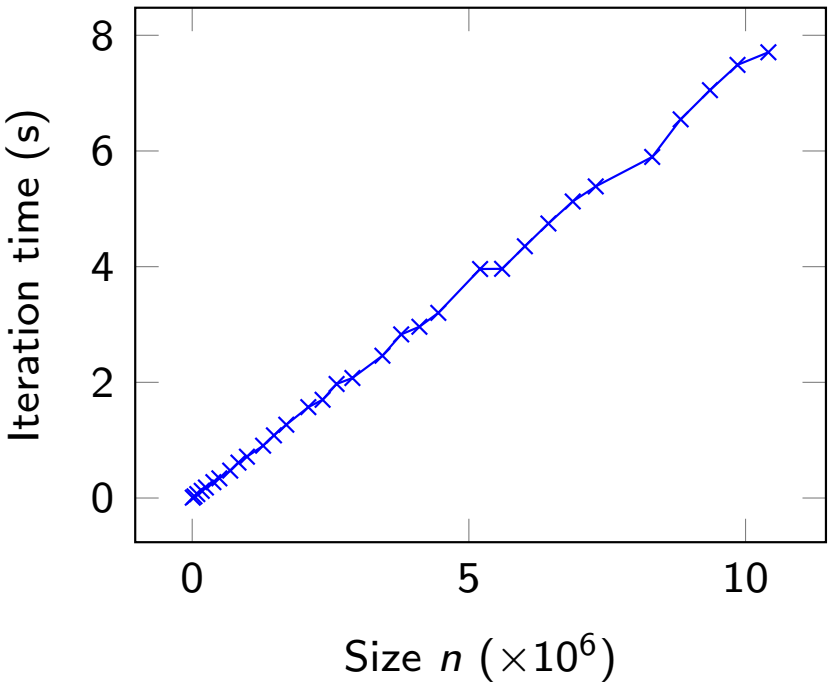
Raw performances: job-shop

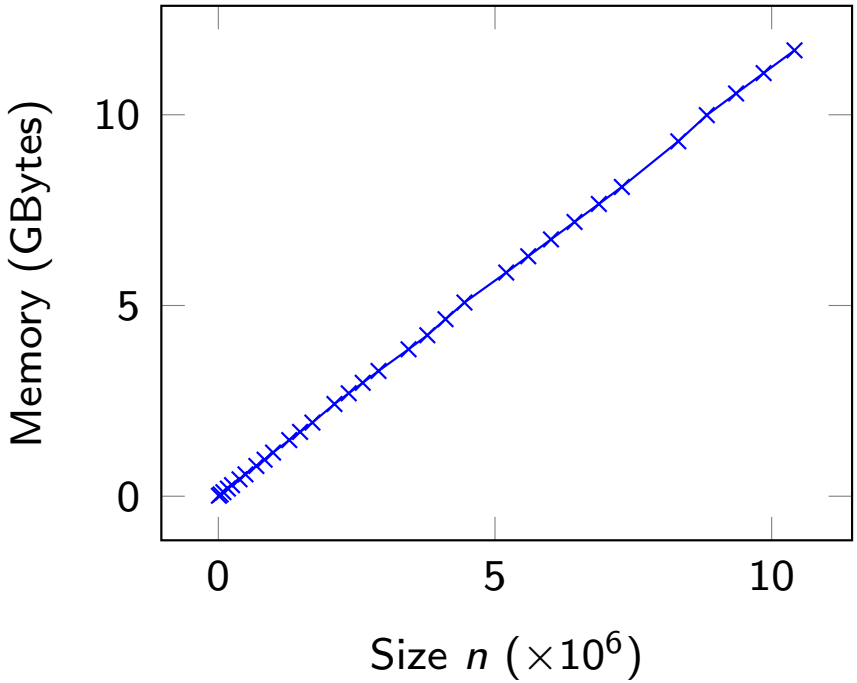
- Comparison with CPO on small instances (*"small"* means *"small for BOLIDE"* ☺)
  - BOLIDE: decode a random decision list to produce a solution
  - CPO: find a feasible solution with Depth-First and 1 worker

Size	BOLIDE time (s)	CPO time (s)	Time ratio	BOLIDE mem. (MB)	CPO mem. (MB)	Memory ratio
10000	0.005	1.45	320	10.8	60.1	5.5
40000	0.028	31.65	1130	39.8	305.7	7.7
90000	0,071	256.09	3606	86.5	832.5	9.6

Raw performances: flexible job-shop

- Average iteration time and peak memory usage for decoding a randomly generated **flexible job-shop** scheduling problem of size n ($n^{1/2}$ jobs $\times$ $n^{1/3}$ operations $\times$ $n^{1/6}$ alternative machines)





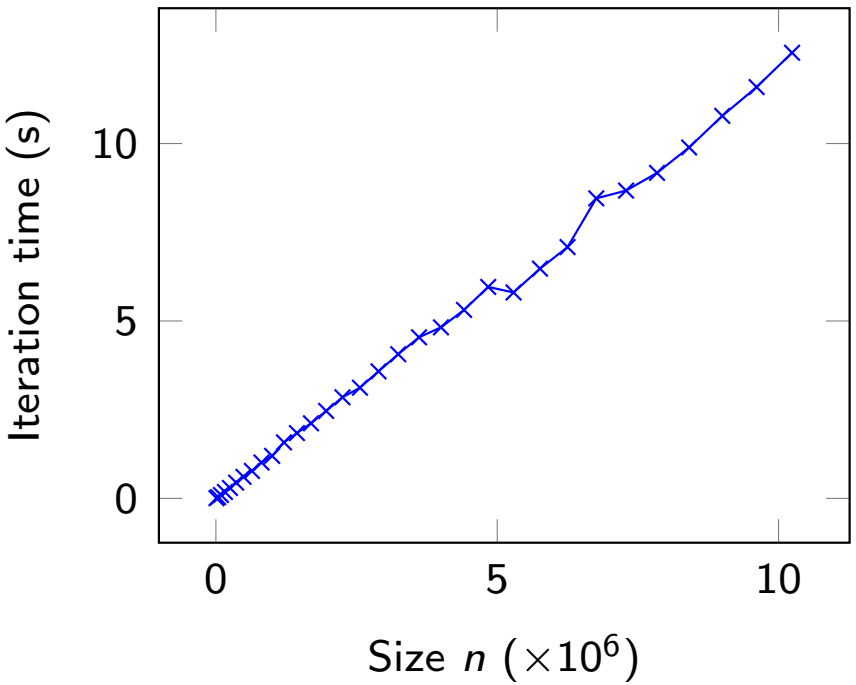
Raw performances: flexible job-shop

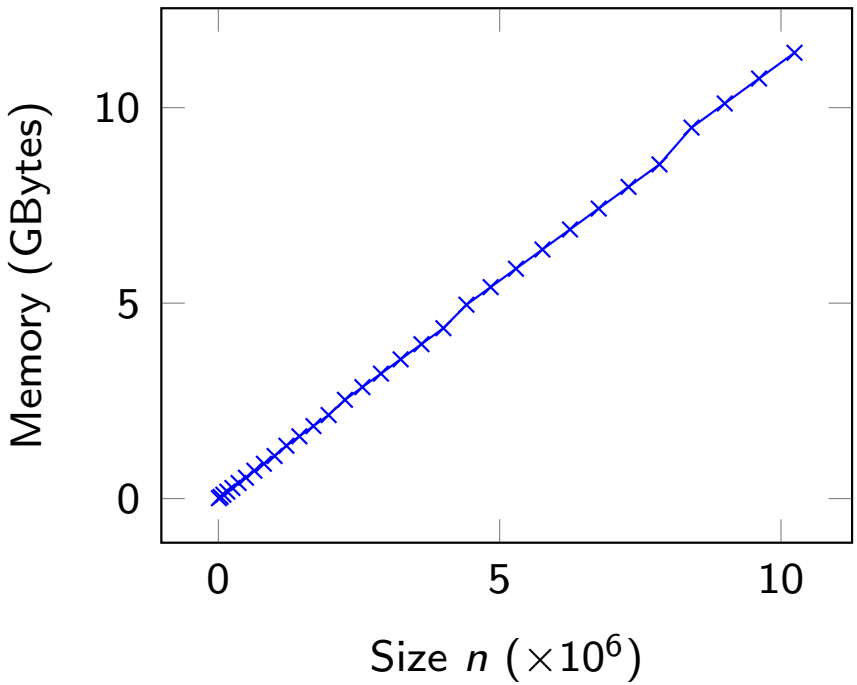
- Comparison with CPO on small instances (*"small"* means *"small for BOLIDE"* ☺)
  - BOLIDE: decode a random decision list to produce a solution
  - CPO: find a feasible solution with Depth-First and 1 worker

Size	BOLIDE time (s)	CPO time (s)	Time ratio	BOLIDE mem. (MB)	CPO mem. (MB)	Memory ratio
10395	0.006	1.62	270	12.2	89.6	7.3
40596	0.029	29.94	1032	46.9	564.4	12.0
92092	0.068	149.65	2200	105.1	1900	18.1

Raw performances: RCPSP

- Average iteration time and peak memory usage for decoding a randomly generated **RCPSP** of size n with a unique cumulative resource used by each of the n tasks





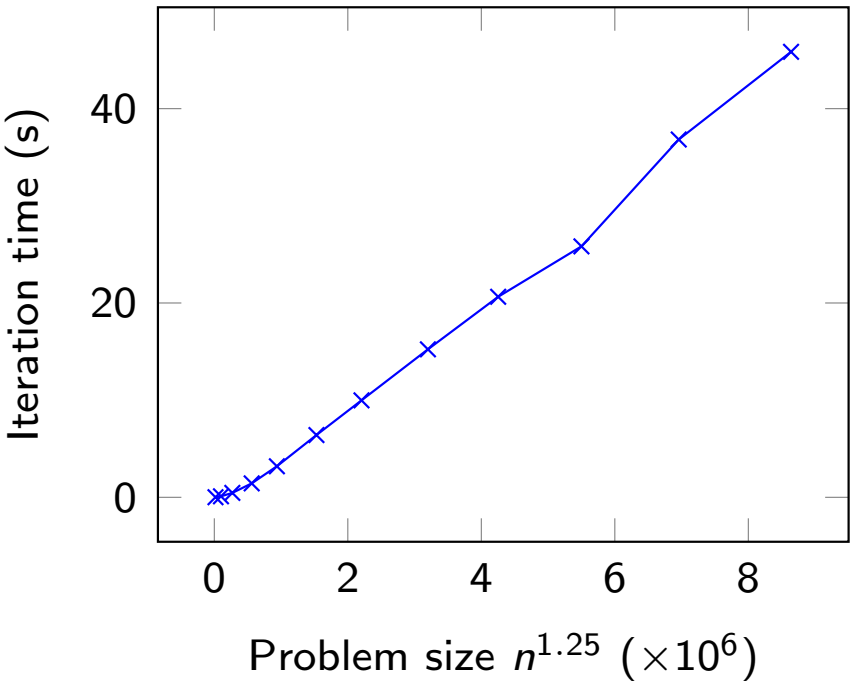
Raw performances: RCPSP

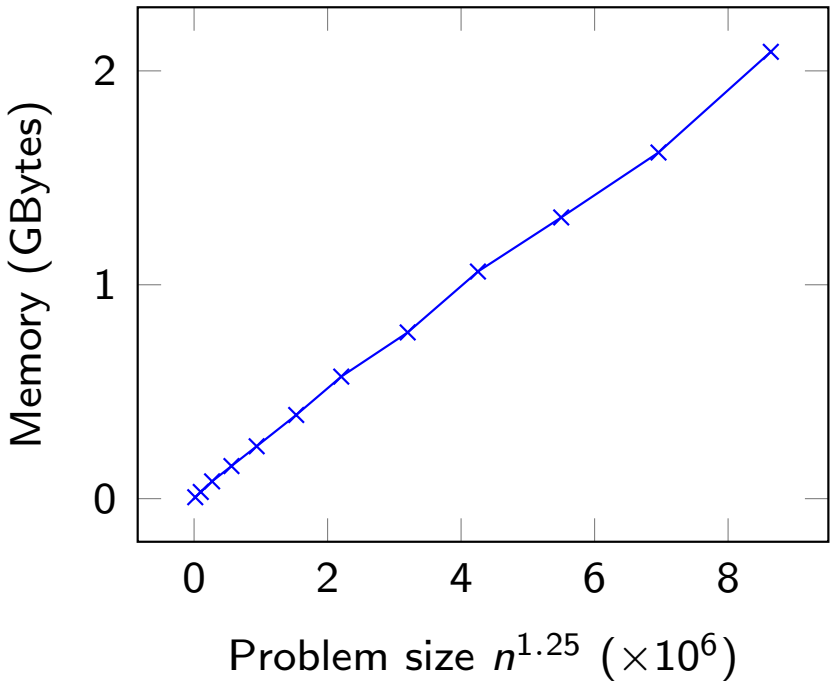
- Comparison with CPO on small instances (*"small"* means *"small for BOLIDE"* ☺)
  - BOLIDE: decode a random decision list to produce a solution
  - CPO: find a feasible solution with Depth-First and 1 worker

Size	BOLIDE time (s)	CPO time (s)	Time ratio	BOLIDE mem. (MB)	CPO mem. (MB)	Memory ratio
2500	0.001	0.74	678	2.9	73.3	25.7
3600	0.002	1.44	923	4.0	147.4	36.8
4900	0.002	2.89	1235	5.5	275.3	49.8
6400	0.003	5.59	1629	7.1	465.6	65.3
8100	0.005	10.99	2348	9.0	750.3	83.6
10000	0.006	18.78	3087	11.2	1200	107.0
12100	0.008	30.94	3967	13.4	1700	126.5
14400	0.010	49.76	5145	15.9	2400	151.2
16900	0.012	70.46	6104	19.2	3400	177.4
19600	0.014	97.58	7028	22.0	4500	204.8
22500	0.016	133.75	8244	25.0	6000	240.0
25600	0.021	185.33	8932	28.3	7800	276.0
28900	0.022	237.47	11031	31.8	9900	311.4
32400	0.025	301.47	11856	35.5	12500	352.6

Raw performances: RCPSP

- Average iteration time and peak memory usage for decoding a randomly generated **RCPSP** with n tasks and $n^{1/4}$ cumulative resources, each resource is used by each of the n tasks
- For the largest problem this is 360000 tasks, each of them requiring a conjunction of 25 resources





Basic meta-heuristic

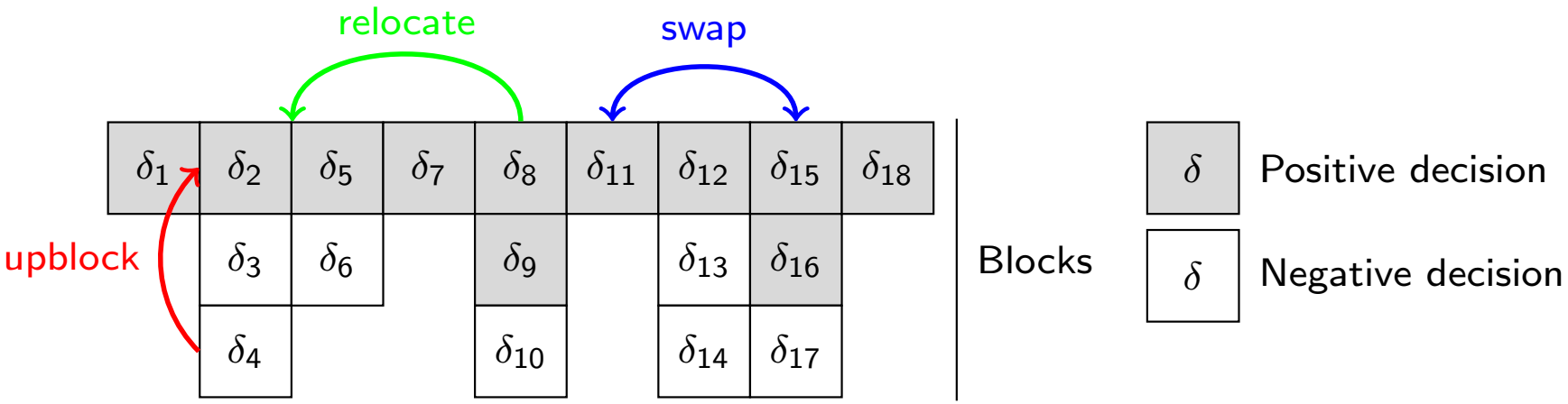
- So far we focused our efforts on providing a generic decoding scheme
- Still, we implemented a **basic** meta-heuristic ("meta-heuristic 0.0"):
  1. For testing purpose
  2. To get a very initial idea of the potential of the approach
- It works in two steps:
  1. Run a predefined set of heuristically defined decision lists
  2. Perform a local search on the best solution found in first step using very simple random moves

Basic meta-heuristic

- Initial heuristics:
  1. Randomly shuffled decision list
  2. Initial order (depends on the order of creation of variables/constraints)
  3. Reverse initial order
  4. A DFS traversal of decision DAG
  5. A BFS traversal of decision DAG
  6. Increasing length of intervals $\rightarrow$ DFS traversal of decision DAG
  7. DFS traversal of decision DAG $\rightarrow$ Increasing length of intervals
  8. Increasing length of intervals $\rightarrow$ BFS traversal of decision DAG
  9. BFS traversal of decision DAG $\rightarrow$ Increasing length of intervals
  10. Increasing length of intervals $\rightarrow$ Random tie-break
  11. Decreasing length of intervals $\rightarrow$ DFS traversal of decision DAG
  12. DFS traversal of decision DAG $\rightarrow$ Decreasing length of intervals
  13. Decreasing length of intervals $\rightarrow$ BFS traversal of decision DAG
  14. BFS traversal of decision DAG $\rightarrow$ Decreasing length of intervals
  15. Decreasing length of intervals $\rightarrow$ Random tie-break

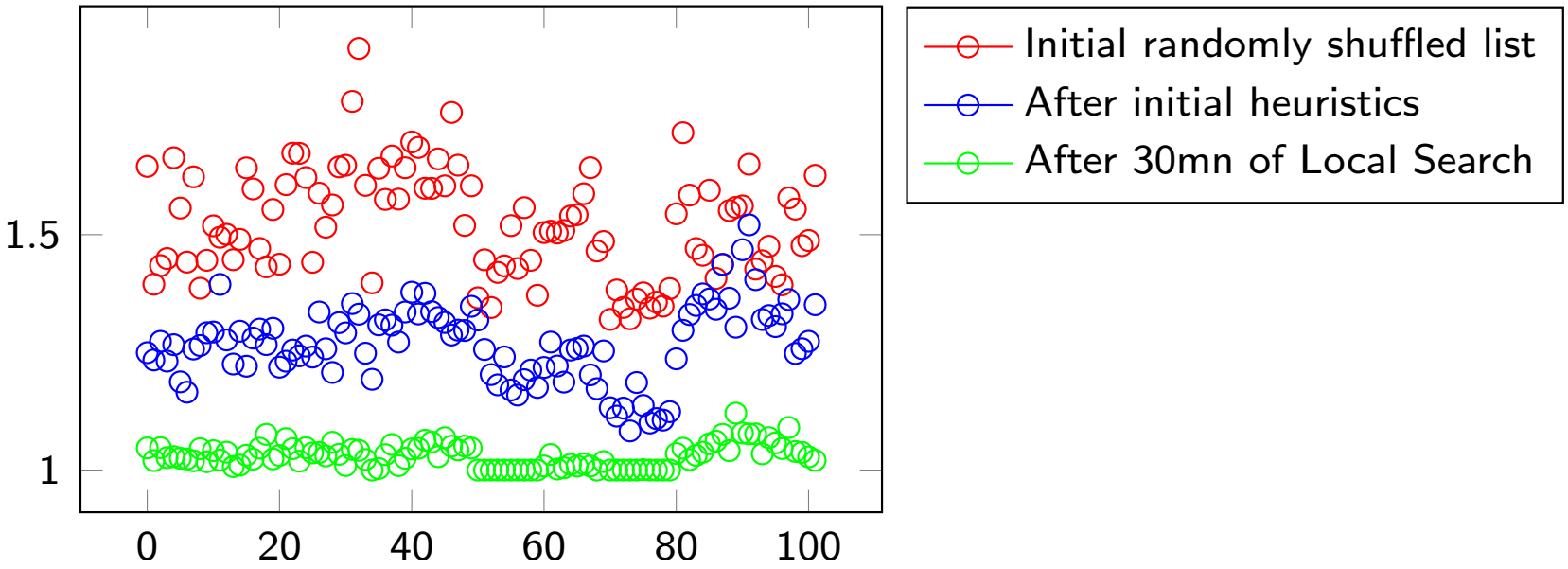
Basic meta-heuristic

- Local Search with random moves using 3 types of moves: relocate, swap, upblock
- At each iteration, k randomly selected moves are applied together on the re-encoded incumbent decision list
- k is small: $k = i$ with probability $1/2^i$
- Random walk on plateaus is allowed



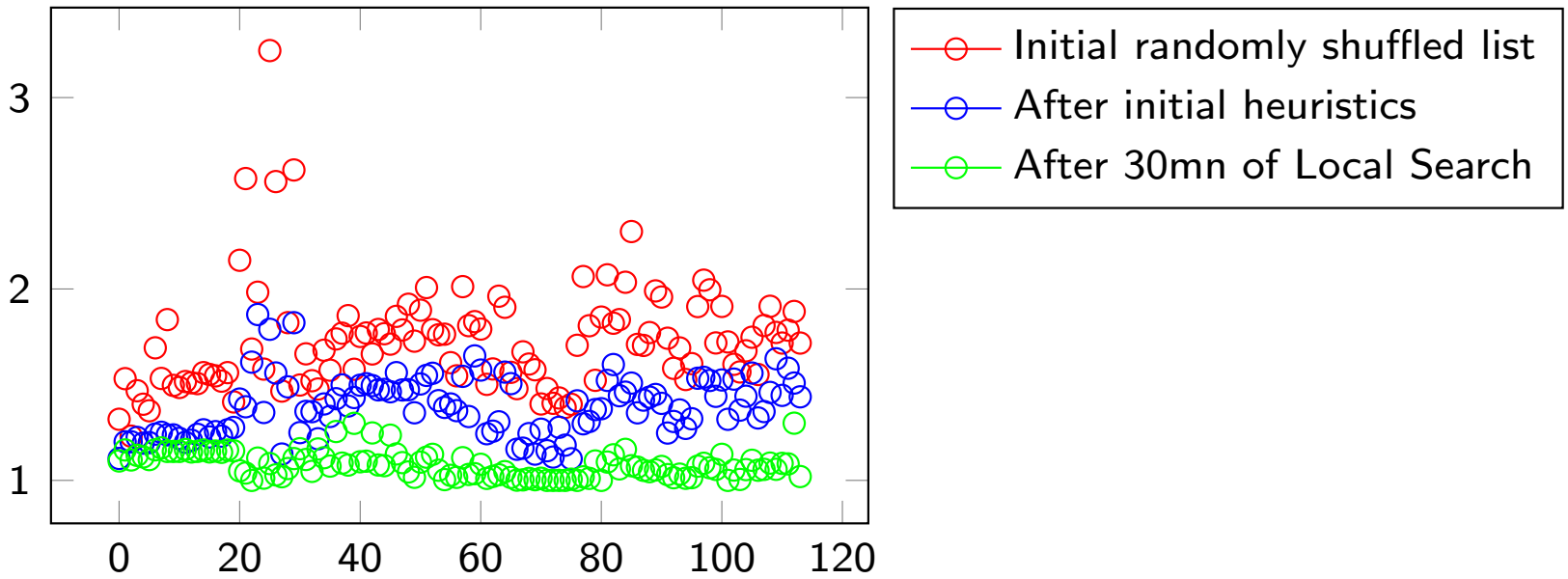
Basic meta-heuristic

- Ratio to optimal or best known solution for 103 classical *job-shop* scheduling problems (ft10, tail1-80, abz7-9, swv1-15, yam1-4)
- Size in range [100,2000] variables
- After 30mn ratio is in the range [1,1.12], average gap is around 3%



Basic meta-heuristic

- Ratio to optimal or best known solution for 115 classical *flexible job-shop* problems (Barnes, Brandimarte, Dauzere, Hurink *R*)
- After 30mn ratio is in the range [1,1.3], average gap is around 8%

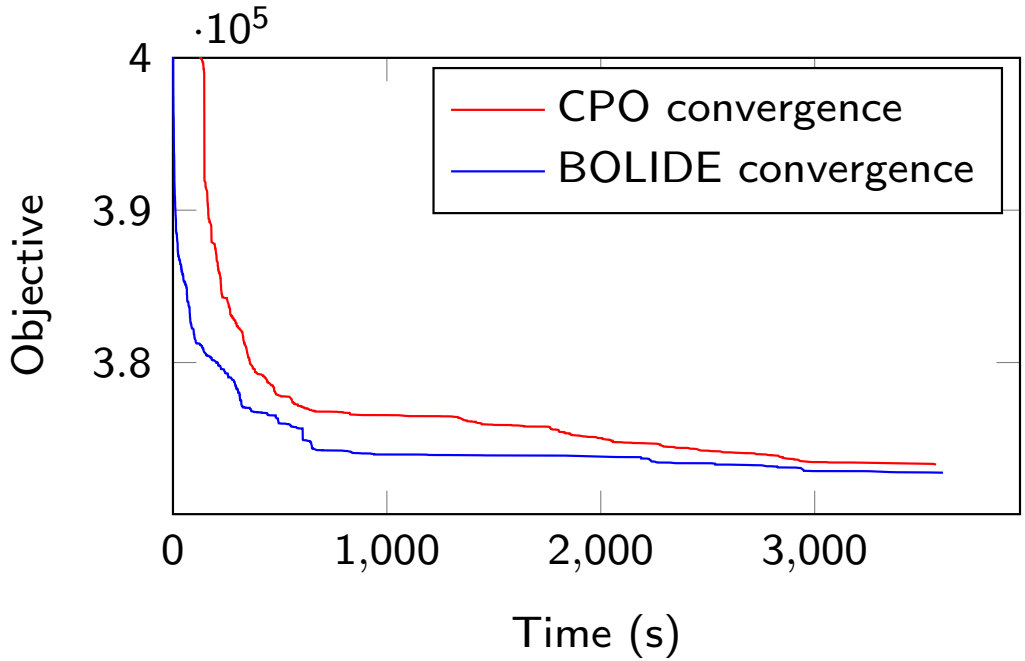


Basic meta-heuristic

- Test on *job-shop* scheduling problems with *minimization of total flow time* (sum) instead of makespan (max)

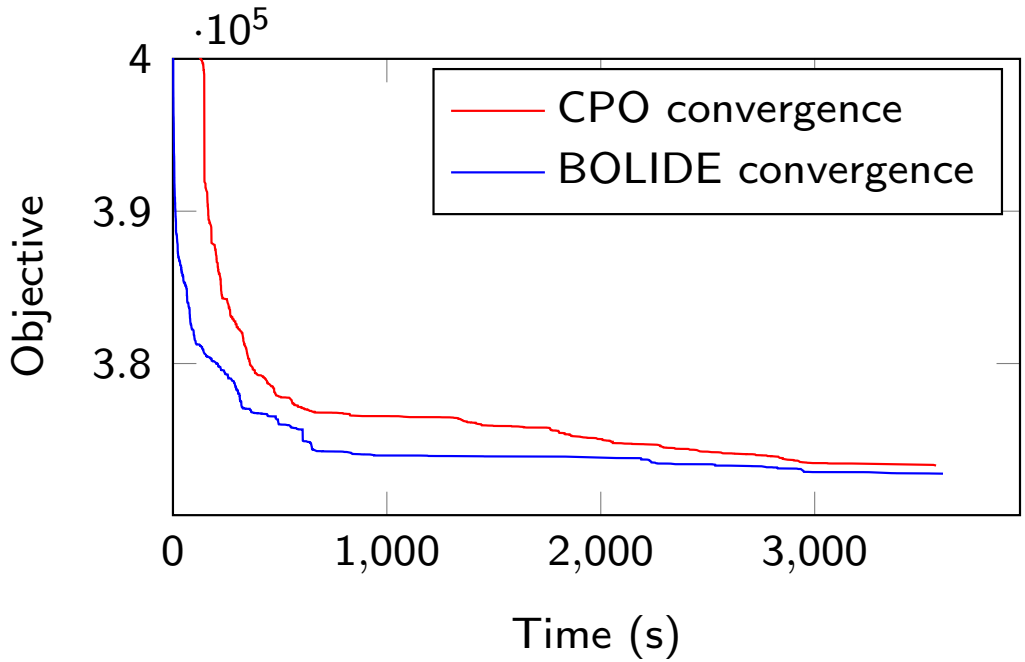
Basic meta-heuristic

- Test on *job-shop* scheduling problems with *minimization of total flow time* (sum) instead of makespan (max)
- Convergence comparison CPO[automatic search, workers=1] v.s. BOLIDE on the largest Taillard instance (tail80: 100×20)



Basic meta-heuristic

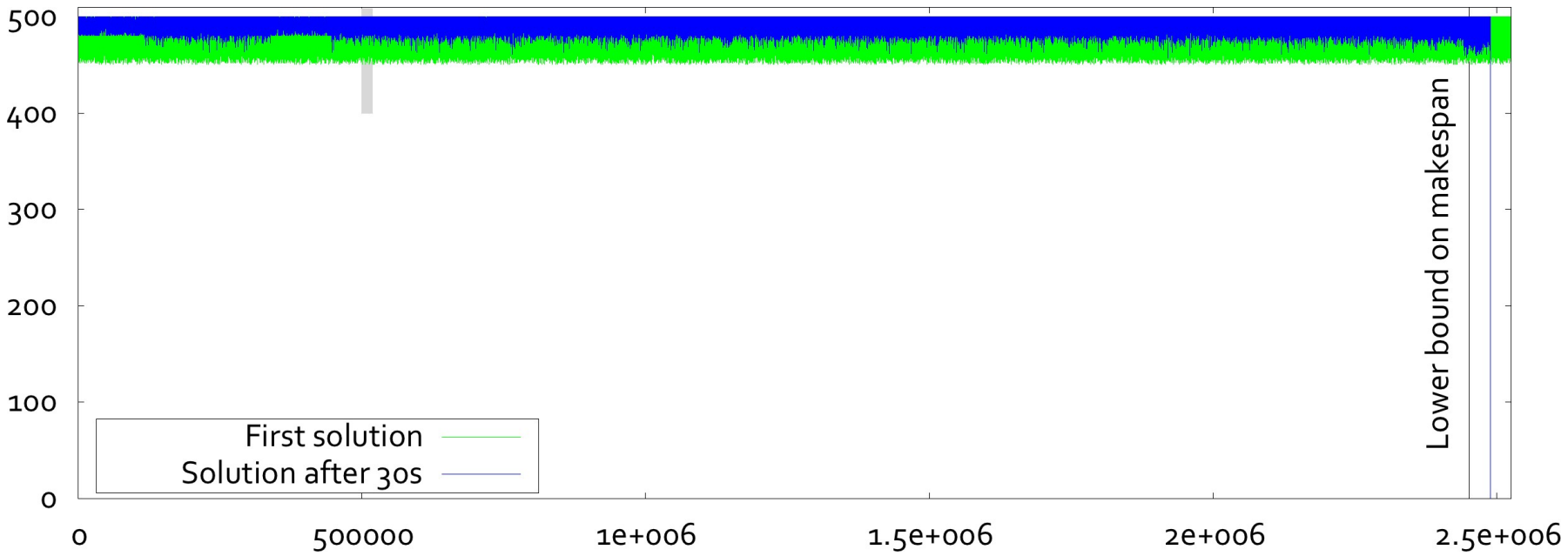
- Test on *job-shop* scheduling problems with *minimization of total flow time* (sum) instead of makespan (max)
- Convergence comparison CPO[automatic search, workers=1] v.s. BOLIDE on the largest Taillard instance (tail80: 100×20)



- To be honest there are also instances where the basic meta-heuristic is stuck at larger objective value than CPO

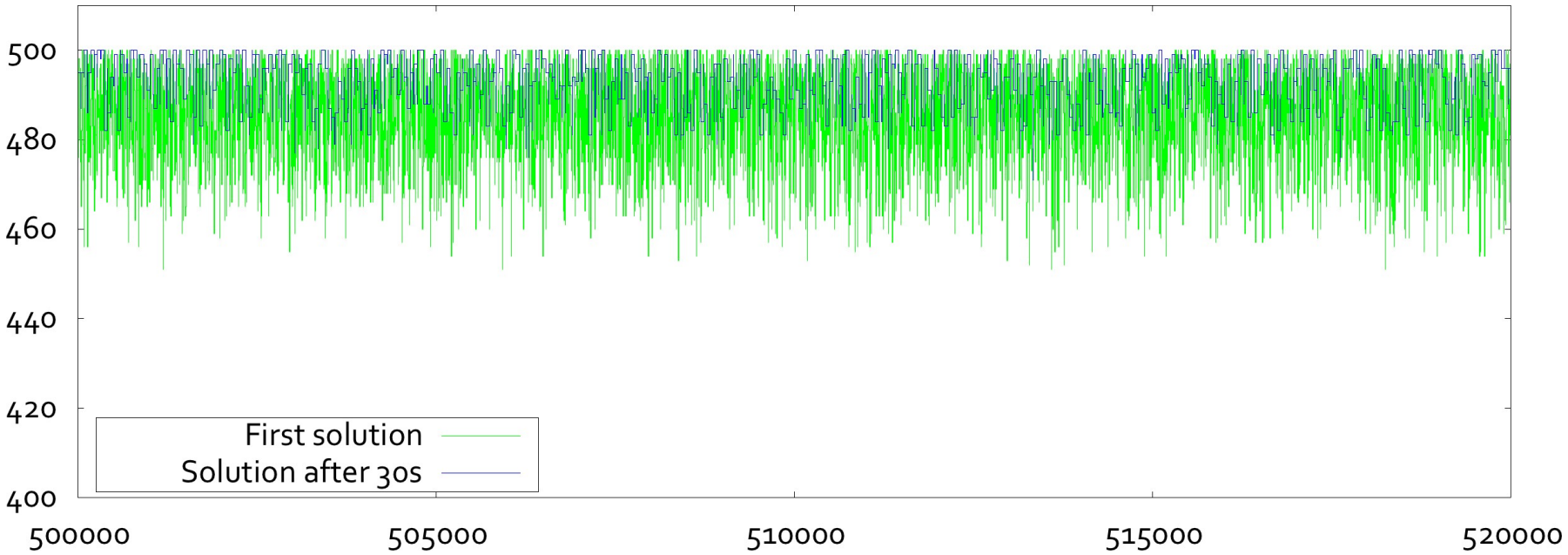
Basic meta-heuristic

- Test on a *single cumulative resource* scheduling problem with *minimization of makespan* with 1M tasks
- Task duration in [20,50], demand in [20,50], capacity is 500
- Iteration time is less than 2s



Basic meta-heuristic

- Test on a *single cumulative resource* scheduling problem with *minimization of makespan* with 1M tasks
- Task duration in [20,50], demand in [20,50], capacity is 500
- Iteration time is less than 2s



Where do we go from here?

- Scope extension
  - Work on maximizing solution completion for hard feasibility problems
    $\rightarrow$ Test on RCPSP with max delays
  - Sequence variables and transition distances/expressions
    $\rightarrow$ Test on TSP / VRP
  - Intensity functions and forbidXXX (calendars) ...
- Integration with CP Optimizer
  - Layer conversion into BOLIDE
  - Cooperation with current CPO search
- Meta/Hyper-heuristic
  - More informed initial heuristics (essential for huge problems)
  - More serious Local Search (exploit decision DAG, time net, resources)
  - Multi-Point search (GA, path relinking, ...)


Thank you
Questions?