Operations Research Perspectives 9 (2022) 100249

Contents lists available at ScienceDirect

Operations Research Perspectives

journal homepage: www.elsevier.com/locate/orp

Industrial-size job shop scheduling with constraint programming

Giacomo Da Col [a,*], Erich C. Teppan [a,b]

[a] Fraunhofer Austria, Innovationszentrum für Digitalisierung und Künstliche Intelligenz (KI4LIFE), Lakeside B13a, 9020 Klagenfurt, Austria
[b] Universität Klagenfurt, Universitätsstr. 65-67, 9020 Klagenfurt, Austria

ARTICLE INFO

Keywords:
Large scale optimization
Job shop scheduling problem
Constraint programming
CP Optimizer
OR-Tools

ABSTRACT

The job shop scheduling problem is one of the most studied optimization problems to this day and it becomes more and more important in the light of the fourth industrial revolution (Industry 4.0) that aims at fully automated production processes. For a long time exact methods like constraint programming had problems to solve real large-scale problem instances and methods of choice were to be found in the area of (meta-) heuristics. However, developments during the last decade improved the performance of state-of-the-art constraint solvers dramatically, to the point that they can be applied also on large-scale instances. The presented work's main target is to elaborate the performance of state-of-the-art constraint solvers with respect to industrial-size job shop scheduling problem instances. To this end, we analyze and compare the performance of two cutting-edge constraint solvers: OR-Tools, an open-source solver developed by Google and recurrent winner of the MiniZinc Challenge, and CP Optimizer, a proprietary constraint solver from IBM targeted at industrial optimization problems. In order to reflect real-world industrial scenarios with heavy workloads like found in the semi-conductor domain, we use novel benchmarks that comprise up to one million operations to be scheduled on up to one thousand machines. The comparison is based on the best makespan (i.e. completion time) achieved and the time required to solve the problem instances. We test the solvers on single-core and quad-core configurations.

1. Introduction

A schedule is a plan to organize activities within a time frame. In the context of every day life, typical examples of schedules include the timetable of a bus or a lecture plan for a semester. In a more technical context, schedules can define the ordering of processes on a computer, or the weekly production plan of a car factory. It can be assumed that scheduling played a major role in the realization process of any work requiring the contribution and coordination of groups of workmen to be accomplished. Large ceremonial buildings, military campaigns and complex infrastructures like canals and streets were planned and organized for thousands of years. In some cases, like the Art of War treatise by Sun Tzu [1], we have an account of these organization processes. In others, like the pyramids, no written record indicating the schedule planing has been found, although it is unlikely that these works could be completed without rigorous planning of time and resources [2].

Despite scheduling problems interested mankind since the dawn of history, it was not until the twentieth century that the formalization and categorization of these problems started: One of the first models for the schedules' visualization and analysis is the Gantt chart, invented by the American mechanical engineer Henry Gantt (1861–1919) [3]. This bar chart offers a two-dimensional representation by listing the tasks to be accomplished on the vertical axis and the time on the horizontal axis. It is used to illustrate a schedule, which is a solution to a scheduling problem. Widely used by the US Army during the Great War, it was later adopted all over the world. Even the first 5-year-plan of the Soviet government was drafted with the help of Gantt charts. Thanks to its flexibility in representing resources and tasks, the Gantt chart has been used in all sorts of scheduling problems (not just project schedules) and it is still in use today as a practical graphical representation of schedules [4].

One of the domains that had a big impact on the evolution of scheduling theory is industry [5–7]. This is becoming even more important as an application field of scheduling theory nowadays, thanks to the advent of "internet of things" applications that open many opportunities of improvement for existing systems, to the point that the current period is commonly referred to as the beginning of the *fourth industrial revolution* [8]. The increasing automation of facilities is leading more and more towards a scenario where machines are able to interact with each other in the most efficient way possible, "free" from the error-prone intervention of human personnel. This will be the case in transportation, where it is estimated that roads will be far safer when

* Corresponding author.
E-mail addresses: giacomo.dacol@fraunhofer.at (G. Da Col), erich.teppan@aau.at (E.C. Teppan).

https://doi.org/10.1016/j.orp.2022.100249
Received 29 March 2022; Received in revised form 6 August 2022; Accepted 7 August 2022
Available online 12 August 2022

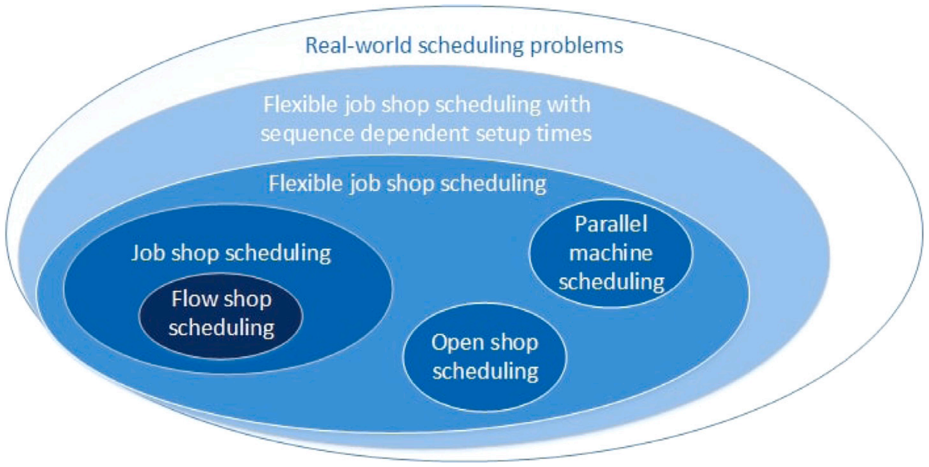
Real-world scheduling problems
Flexible job shop scheduling with sequence dependent setup times
Flexible job shop scheduling
Job shop scheduling
Flow shop scheduling
Parallel machine scheduling
Open shop scheduling

Fig. 1. Diagram of the relations between different types of scheduling problems (inclusion means to be a special case).

free from human drivers [9], and it is already the case in logistics, with Amazon's autonomous shelf-robots in their warehouses and automated packaging machines that are five times faster than humans [10,11]. In this context, advancements in scheduling theory can have stronger impacts on real-world industry than ever before.

The organization of the production is one of the most important activities of an industrial company [12], in order to remain competitive in the market: An effective management of production schedules results in reduction of production costs, efficient use of resources and increased quality of service (*e.g.* respect of product due-dates).

The need of organization of time and resources ignited the flame of scheduling optimization studies in the last century. Research works like Johnson's on the flow shop problem (1953) and Muth and Thompson's on the job shop (1963) are considered pioneering examples of research in industrial scheduling [5,13]. In the attempt to formalize and study the many variations of scheduling issues in different industrial contexts, a lot of mathematical formulations of scheduling problems have been defined over the years [14–16]. To identify the various entities of an industrial scheduling problem, scholars typically refer to *machines* to indicate resources and *jobs* to indicate tasks. Following this terminology, the factory layout (*i.e.* the number of machines and their functionality) is called a *shop*, and when a job is composed by sub-tasks ordered in a specific sequence, they are called *operations*. Operations are linked to machines by the concept of *operation type*. Every operation has a type and each machine can process operations of certain types, but not others. In general, shops represent factories with different machines needed for the creation of products (jobs) that afford multiple production steps (operations). In general, products cannot be on two machines at the same time, and machines process one operation per time, therefore, operations assigned to the same machine must not overlap in a schedule (*i.e.* a solution).

Fig. 1 depicts the relation of some of the most well-known production scheduling problems. When a problem is included in another one, it means it is a special case.

Considering multi-machine scheduling, the most simplistic problem is the *parallel machine scheduling problem* (PMSP) [17,18]. In a PMSP, jobs are atomic and are not further divided into operations, or alternatively, jobs can be seen as single-stage and consist of only a single operation. Each job/operation has the same type and each of the perfectly equal machines is able to perform each of the jobs. Hence, the problem consists in assigning all jobs to one of the machines (called routing) and order them (or give them a priority, called sequencing) in order to optimize a certain optimization criterion. Already this simplistic problem is NP-hard if there are at least three machines [7]. An important application field for this problem class is the scheduling of processes (*i.e.* the jobs) in multi-processor (or multi-core, *i.e.* the machines) computer systems.

In the job shop scheduling problem (JSSP), for every job there is a (typically different) predefined order of operation types/machines as part of the input [14,19,20]. Real world examples are complex factories that produce diverse types of products each of which need a different workflow to be created. For example in the semiconductor domain, the production of chips for different purposes is always depending on the same machines, however, depending on the realized integrated circuits, workflows of production can be totally different.

The *flow shop scheduling problems* (FSSP) is a special case of JSSP, where the order of operations for a job is predetermined [21,22]. In contrast to JSSP, the order of operation types/machines is the same for every job. Although operation lengths may vary also between operations of the same type (i.e. the same machine) the order of operation types (i.e. order of machines) is the same for all jobs. An example for this is a bakery that produces breads. Although recipes can be different for different breads (e.g. because of size or flour type) the overall structure of bread production and consequently the order of machines is the same for every product.

A similar problem to JSSP is the *open shop scheduling problem* (OSSP). In fact, an OSSP can be seen as a JSSP where the ordering of the operations of a job is arbitrary [23]. Thus, there is no predefined order of operation types/machines as part of the input. Contrary to FSSP, however, OSSP is not to be considered a special case of JSSP, because it is not possible to express OSSP as a JSSP problem, due to the constraint of having a predefined operation type orders in JSSP. Real world examples for the OSSP are factory lines that produce rather simple product parts of which the production only relies on independent assembly steps.

In the most simplistic versions of OSSP, FSSP and JSSP there exists only a single machine for each operation type and as a consequence operation to machine assignments are predetermined (hence there is no routing problem). Clearly, this can be generalized in that there could be more machines for an operation type which leads to the *flexible* JSSP (FJSSP) [16,24]. If furthermore there are machines that support multiple operation types and switching from one operation mode to another affords setup time, this leads to the FJSSP with sequence dependent setup times [13,25].

All these formulations of increasing generality aim at capturing the core issues from the complexity of the real world that are most relevant. At the same time, the abstract problem formulations prune real world factors that are of negligible importance for scheduling itself. Even considering such abstracted problems, most of them are already NP-hard optimization problems [21], meaning that there is no efficient algorithm to solve them.[1]

[1] Unless P = NP.

The intrinsic challenge presented by these problems allowed research based on these abstract versions to gain momentum on its own. On the one hand, this led to increased awareness of this problem in the scientific community, which is positive. On the other hand, though, scholars focused more and more on finding optimal solutions for synthetic benchmark instances (often with marginal improvements of solution quality) and lost contact with industrial realities. This becomes especially obvious when we compare the size of those instances in literature with the actual workloads of nowadays industry.

The discrepancy between theory and practice is even exacerbated by the often conflicting objectives of industry and academy. On the one hand, scholars find value in theoretical proofs or formal explanations of why an approach works better than another. The abstraction of problems helps to concentrate on the mathematical aspects and disregard the others (*e.g.* reducing scheduling to a graph theory problem). On the other hand, people from industry are interested in an effective solution limited to the problem they are currently facing, valuing also the costs of integrating the new solution on the old system, training personnel on the new approach, etc. As a consequence, despite the vast corpus of publications on production scheduling, few of the advancements discovered by research are used in real industrial scenarios [12,26].

Already in the Sixties, it was clear that scheduling was considered a hard combinatorial problem by scholars, but managers of production factories did not even recognize scheduling as a problem [5]. Instead, the production time was seen as a constant, and when the volume of jobs increased, other factors were adapted to prevent the actual scheduling problem to arise (*e.g.* buying more machines or hiring more personnel). In the Nineties, a survey by Maccarthy and Liu [26] showed that the situation did not improve, with many factory managers delegating scheduling to shift-leaders or foremen, and little to no value was given by companies to theoretical results that could improve productivity. However, a more recent survey (2017) by Fuchigami and Rangel [12] identifies 46 publications targeted at real production scheduling problems between 1992 and 2016, which is a sign that the gap between theory and practice is slowly beginning to shrink.

The aim of our work is to continue to shrink this gap by showing how state-of-the-art methods work in practice. In particular, we want to assess the capabilities of state-of-the-art tools to scheduling problems that are representative for the current industrial landscape. Following the examples of other scholars, we select the Job Shop Scheduling Problem (JSSP) as a representative for an industrial problem with practical relevance. However, we map it to current industrial scenarios by creating a number of new benchmark instances that better reflect nowadays large-scale production problems. In fact, the typical size of academic benchmarks ranges between 100 and 2000 operations to be scheduled [19,27,28], while in real scenarios it is not uncommon to reach more than 100 000 operations within a certain planning horizon (*e.g.* a week).

As state-of-the-art method, we use Constraint Programming (CP), for two main reasons:

1. CP is a declarative method, meaning that the encoding of the problem is done in terms of what is to be solved, instead of how to solve it (as typical for procedural languages like C). In particular, CP relies on logical rules and constraints to model the problem. This results in various advantages, like the compactness of the code that can be cheaply maintained in the long run[2] and quickly adapted to new needs. Moreover, it is easier to prove correctness of models based on logic rules, compared to the slew of if-statements and nested loops typical of the procedural paradigm. Declarative approaches have been applied successfully in various real scenarios, from industry [30,31] to biology and medicine [32,33].
2. CP approaches of the early days were promising, but severely limited by memory. However, recent advancement in methods to explore the search space (*e.g.* Large Neighborhood Search (LNS) and relaxation techniques [34,35]) combined with the fact that memory is becoming cheaper and cheaper as a resource, led to a revival of constraint approaches. Nevertheless, recent works about scheduling in real scenarios are still concentrating on MILP as a tool of choice [12], even if CP seems better suited for scheduling problems, especially for large-scale instances [36].

The experiments reported herein constitute a (significant) extension of the experiments reported in [37,38]. In particular, we repeated the experiments with updated versions of the solvers, and extended the experimental setup to cover configurations that were not treated in [37,38] by also testing quad-core configurations for all solvers and for all the benchmark instances. By this, we are able to paint a complete and homogeneous picture. Furthermore, we provide an in-depth analysis of the results and the solvers' behaviors.

The remainder of this work is organized as follows: Section 2 contains an overview on the JSSP, including the definition of the problem, the main approaches proposed by scholars over the years, and the most well-known benchmark suites from literature. Section 3 introduces constraint programming (CP) by giving a general description of the approach, and showing how it can be applied to the JSSP. This includes a discussion of high-level variable and constraint types needed for solving large-scale JSSP instances. Section 4 gives all details about the conducted experiment. This includes a description of the tested state-of-the-art CP solvers, the general experimental setup and an in-depth description of our novel (publicly available) benchmarks. Section 5 contains the results and detailed discussion with respect to all three used benchmark suites, completed with a comparison of the state of the art results. As well we provide some technical insights on the reasons behind the solvers' behaviors. Section 6 concludes the paper.

2. Job shop scheduling

The Job Shop Scheduling Problem (JSSP) [6,7] is among the most studied combinatorial problems. For example, with respect to other famous scheduling problems, JSSP has a count on Google Scholar of about 63 000 references, compared to 12 000 for parallel machine problems and 17 000 for flow shop problems (at the time of release of this work). Several factors contributed to its fame. One of them is definitely the work by Fisher and Thompson from 1963, where they describe two problem instances that, albeit their small size, were particularly hard to solve [5]. The challenge lured many researchers into finding the best methods to reach the optimal solutions. The easier of the two, comprising 20 jobs to be scheduled on 5 machines, was solved 10 years later, while the instance of 10 jobs on 10 machines took more than 25 years to be solved optimally.

From the practical point of view, the JSSP captures many issues of real scheduling problems: jobs with sequences of operations of different types to be scheduled on a limited set of machines. The JSSP constitutes a very hard problem also from the theoretical point of view. Indeed, the JSSP is a strongly NP-hard optimization problem [21], meaning that a polynomial-time algorithm is not possible.[3] Even if some aspects of real world environments are not covered (*e.g.* setup times, parallel machines, etc.), an effective model for JSSP could also be useful to solve problems with more complex requirements. In particular, JSSP could be used as a relaxed version of a more complex problem at hand, providing the means to both build an effective heuristic, and to learn the bounds of a solution for the non-relaxed problem. Moreover, an effective method to solve the JSSP could be used as a subroutine for solving more complex problems, *e.g.* a flexible JSSP can be solved by assigning operations to machines first, and then proceed to solve the resulting JSSP [39,40].

[2] At Siemens, it was reported a reduction of maintainable cost by 80% thanks to declarative approaches [29].

[3] Unless P = NP.

Operation-Machine Assignment

	Op 1	Op 2	Op 3
Job 1	M1	M2	-
Job 2	M2	M3	M1
Job 3	M2	M3	M1

Operations Lengths

	Op 1	Op 2	Op 3
Job 1	4	2	-
Job 2	1	2	3
Job 3	4	2	1

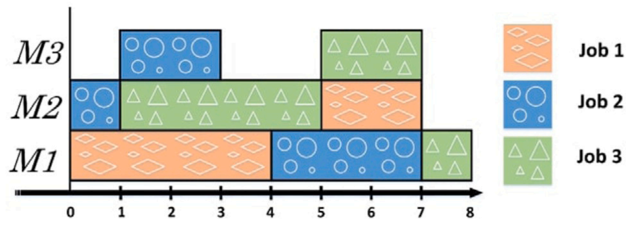

Fig. 2. Tables define an instance of a job shop scheduling problem, *i.e.* the operation-machine assignment (left) and the length of the processing times (right). The chart is a Gantt representation of a possible solution (schedule) to the given problem.

2.1. Problem definition

The JSSP can be defined as follows:

- Given is a set of machines M and a set of jobs J.
- For each job $j \in J$ we have a sequence of operations $\langle j_1, \dots, j_{last_j} \rangle$ such that, for each j_i, j_{i-1} is called predecessor and j_{i+1} is called successor.
- As a trivial exception, the first operations of each job do not have a predecessor and the last operations do not have a successor.
- Each operation x is processed by a $machine_x \in M$.
- Each operation x has associated a $length_x \in \mathbb{N}$, which represents the processing time of the operation on the assigned $machine_x$.
- There is no preemption. An operation cannot be interrupted.

A solution is an assignment of starting times $start_x$ for all operations x such that the following constraints are respected:

- Precedence constraint: For each job j

$$start_{j_{i+1}} \geq start_{j_i} + length_{j_i}, i \in \{1, \dots, last_j - 1\} \tag{1}$$

- NoOverlap constraint: For each operation couple $x \neq y$ with $machine_x = machine_y$

$$start_x \neq start_y \tag{2}$$

$$start_x < start_y \Rightarrow start_y \geq start_x + length_x \tag{3}$$

We also call a solution for a scheduling problem a *schedule*. An optimal solution is a schedule such that a certain optimization criterion is minimized(maximized).

One of the most used optimization criteria is the makespan minimization (also called completion time, that is the total time needed for accomplishing all operations). In this case, we define an optimal solution to be a schedule such that:

$$max\{start_{j_{last_j}} + length_{j_{last_j}} \mid j \in J\} \rightarrow min \tag{4}$$

2.2. Example

As an example, let us consider the packing section of a beer brewery. In this scenario, presented in Fig. 2, the three products are beer bottles to be handled in three production stages: labeling, filling and capping. In this case, the resources are industrial machineries that are designed to cap the bottles (M1), label them (M2) and fill them with the right type of beer (M3). Tables at the top define the scheduling problem: The table on the right shows which machines handle the different production stages of the products. Product 2 and 3 have the same production flow of labeling, filling and capping, while Product 1 is already filled and only needs to be capped and labeled. The table on the left illustrates how long each production stage takes, for each product. Since each product has its own beer type, label and cap, the duration of production stages may differ. The Gantt chart in Fig. 2 shows an optimal schedule for the described problem, listing the machines on the vertical axis, and when the production stages are handled on the horizontal axis. In this case, the achieved (optimal) makespan/completion time is 8.

2.3. Solving approaches

Fig. 3 offers a rough classification of methods and techniques that were applied to JSSP over the years. The subdivision of the methods are not intended to be a strict partitioning. In fact, a single approach can span over more categories, especially considering that several new approaches emerge from the combination of existing ones (*e.g.* [41–43]).

The first approaches to JSSP were exact optimization methods aimed at finding the optimal solution. The first models for the problem came out at the end of the fifties, by means of mixed integer linear programming (MILP). The simplex algorithm to solve linear programs was formulated 10 years earlier [44], but it was not until 1958 that Ralph Gomory invented an algorithm that used the simplex algorithm to deal with integer variables [45]. Integer variables can be used in the context of scheduling to model the decision of ordering the operations in the machines. The first MILP encoding for JSSP was formulated by Edward Wagner in 1959 [46], closely followed by Bowman (1959) [15] and Manne (1960) [14]. Despite their age, these encodings were recently tested against a modern encoding using state-of-the-art solvers, and proved to be still effective. In particular, the best overall performer on the JSSP is still the version by Manne [36].

Another approach widely used was the branch and bound technique [47–49]. It consists in an enumeration of all the possible solutions organized in a tree structure, to define subsets of solutions. Each branch indicates a choice for a certain task to be scheduled. Every subset is evaluated by calculating a lower bound of the solution based on the choices of a particular branch. The lower bound is an optimistic estimate of what kind of solutions a branch can lead to. Therefore, considering a minimization problem, if the lower bound for a branch is larger than the overall current best solution, the whole branch can be pruned from the search tree. Consequently, research on this method has been focusing on finding strong lower bounds to shrink the search space.

The lower bounds are typically calculated using a relaxed version of a problem. For JSSP, this can be done using efficient algorithms on subproblems, like in [49]. In particular, this can involve the resolution of m one-machine problems (where m is the number of machines) to optimality. Although one-machine scheduling problems are also NP-hard there are methods to efficiently deal with them in the average case, like the method proposed by Carlier in 1982 [47]. In 1989, he used such a one-machine algorithm to find an optimal solution of the famous 10 × 10 instance by Fisher and Thompson [48]. Lower bounds are also useful to prove the optimality of a solution. The lower bound indicates that the optimal solution cannot be better than the lower bound value, while the upper bound indicates the best solution found yet. When the two bounds coincide, it is proved that the solution is optimal.

Considering exact methods, one famous example is constraint programming (CP) [50,51]. First applications of constraint programming to JSSP reach back to the eighties. In 1982, Mark Fox presented a constraint directed reasoning system targeted at scheduling problems [52]. At the end of the decade, Keng and Yun applied two domain independent policies (most-constrained and least-impact) that help to reduce the number of backtracks during search [53]. In the same year,

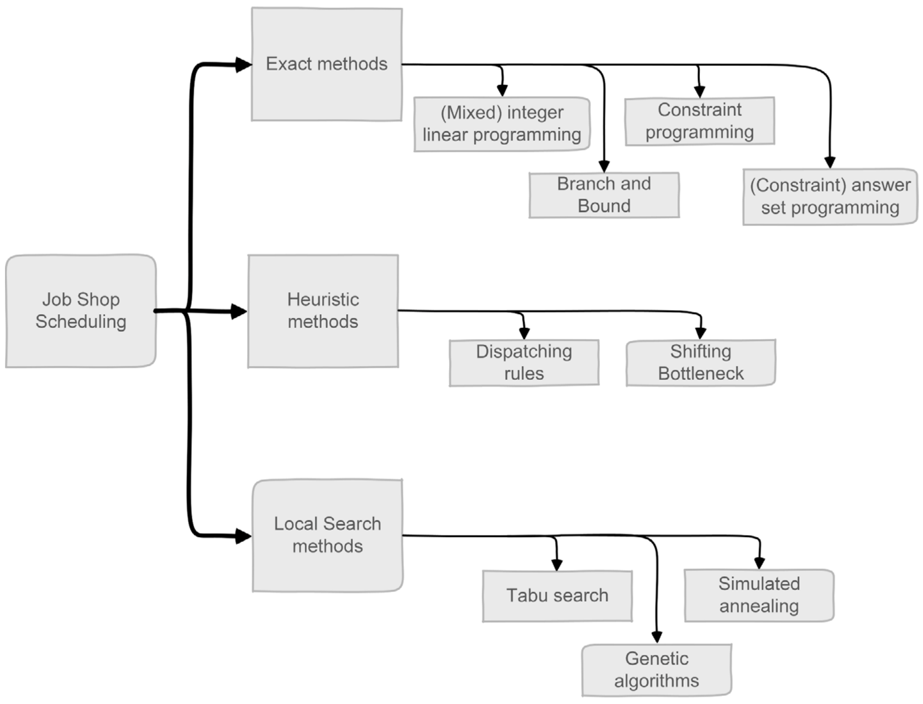

Fig. 3. Approaches for solving the job shop scheduling problem.

a collaboration between Fox and Norman Sadeh paved the way to a series of publications focused on heuristic search on CP applied to JSSP [54–56].

Answer set programming (ASP) is another exact method used in the context of combinatorial problems [57–60]. Based on the stable model semantics, ASP offers high level problem representation capabilities, thanks to predicate logic constructs and aggregate functions. This approach has been applied successfully even in real world scenarios, from industry [30,31] to biology and medicine [32,33]. However, the high level representation comes at the cost of the translation of the encoded problem in propositional logic (*i.e.* grounding). This step is particularly costly for problems like scheduling, since every time-point needs to be represented explicitly [61].

A common problem of these exact methods was/is that they perform well on instances of limited size, but tend to not scale well with larger instances. To deal with large-scale problems, other approaches aim to find reasonably good solutions in reasonable time, even at the cost of missing the optimal solution. This can be done using heuristics, general rules applied to explore the search tree in an organized manner. Thus, with increasing problem sizes it becomes crucial to concentrate the search on the most promising branches.

Priority rules (also called dispatching rules) are one of the first heuristic methods that have been applied to scheduling problems [62, 63]. They define an ordering on the operations to be scheduled, using attributes that are typically easy to calculate, like the processing time of the operation or the number of remaining operations in a job. On the one hand, they do not excel like exact methods on problems where the quality of the solution is highly regarded. On the other hand, their inexpensive computation makes them well suited for large problem instances, where the size of the search space does not allow an extensive search.

Due to their low computational complexity, heuristic approaches like priority rules have been the only way to deal with moderate size schedules (*i.e.* hundreds of operations) for a long time, and are still considered a state-of-the-art technique when dealing with large-scale problems [64]. These simple rules can also be combined to form more powerful heuristics [65,66]. This approach, called hyper-heuristic, typically uses simple heuristics as building blocks, but can be abstracted even further, *e.g.* using a method that selects hyper-heuristics, effectively creating a hyper-hyper heuristic [67].

One turning point was the shift from constructive to iterative approaches. Instead of creating the schedule from scratch, the idea of iterative methods is to start from a solution and improve it step-wise. Local search methods like genetic algorithms, simulated annealing and, in particular, tabu search, evolved in this context. All these approaches revolve around the concept of the *neighborhood* of a solution s, *i.e.* the set of solutions that can be generated from s by applying some changes (in JSSP, the swap of two operations).

A famous iterative heuristic algorithm for scheduling is the shifting bottleneck procedure. The idea is to start from an initial schedule and identify the machine that creates a bottleneck in the schedule (*e.g.* the busiest machine) and use the one-machine schedule method by Carlier [47] to solve the scheduling problem on that machine *optimally*. Then, the schedule is adapted on all the other machines, and the algorithm carries over to find the new bottleneck. After the first formulation by Adams et al. [19], the heuristic was later improved by Balas and Vazacopoulos, using a more effective algorithm to solve the one-machine schedule [68].

One of the local search methods that resulted more effective on scheduling problems is Tabu search (TS) [20,69–71]. This technique improves a given solution by exploring the neighborhood and trying to find an alternative solution better than the previous one. While searching, a tabu list is kept in order to mark the list of actions that have been done in order to not undo a change made in the near past. The first application of a TS algorithm in the context of job shop scheduling was formulated by Taillard [69], in 1992. This approach takes advantage from the notion of critical path to reduce the neighborhood of solutions to the most promising subset.

About a year later, 1993, Dell'Amico and Trubian proposed an improved version of the TS approach [70]. Albeit performing better, the refinement regarded mostly the heuristic used to create the initial solution, rather than the tabu search phase itself.

Smutnicki and Nowicki [20] proposed in 1996 a refinement on the neighborhood, dividing the critical path into blocks of consecutive operations on the same machine. The critical path is the sequence of operations on machines that is most responsible for the current makespan.

Table 1
Most used JSSP benchmark instances, divided by benchmark size (values represent the number of instances).

Benchmark	6 × 6	10 × 5	10 × 10	15 × 5	15 × 10	15 × 15	20 × 5	20 × 10	20 × 15	20 × 20	30 × 10	30 × 15	30 × 20	40 × 10	40 × 15	40 × 20	50 × 10	50 × 15	50 × 20	100 × 20
ft [5]	1		1				1													
la [72]		5	5	5	5	5	5	5			5									
abz [19]			2						3					2			2			
orb [28]			10																	
swv [73]								5	5								10			
ta [27]						10			10	10		10	10					10	10	10
yn [74]										4										
dm [75]									5	5		5	5		5	5		5	5	

When a schedule is represented as a disjunctive graph, the critical path is identified as the longest path from the start of the schedule to the end. Their tabu search algorithm defines the neighborhood of a solution as the *interchanges near the borderline of blocks of a single critical path*. This increases the likelihood that a change will affect the makespan. In 2005 they created an enhanced version of their algorithm [71]. These successful applications started a trend of TS based ideas proposed over the following years, combining TS with other local search methods such as simulated annealing [43] or evolutionary algorithms [41] that proved to be successful, at least on the classic benchmarks of the literature.

All cited publications use benchmarks problem instances which can be seen only as small sized problem instances when compared to nowadays industrial realities. Although it is clear that size is not the only source of practical hardness for solution calculation, it is still an important one. Even if you had a theoretically efficient algorithm, complexity could be too high for practical purposes.

For example, having an algorithm that runs in quadratic time in the number of operations and needs a millisecond to schedule a single operation on average, the scheduling of 1000 operations seems to be manageable in acceptable time ($(10^3)^2 = 10^6$ ms = 1000 s). However, having 10^5 operations to schedule (which is quite common in certain industries nowadays for weekly workloads) would result in 10^7 s (nearly 4 months) of calculation time. The next section offers an overview of the (problem) sizes of the common used benchmarks.

2.4. Benchmarks from the literature

Benchmarks offer an effective method to compare different approaches on a particular problem. Moreover, in the case of job shop scheduling, the benchmark of Fisher and Thompson [5] functioned as a challenge, increasing the popularity of the problem. After them, many other researchers followed their example and created a vast variety of scheduling instances, with different sizes and difficulties.

Even if it is not a strict requirement, it is convention in the scheduling community to create rectangular instances of jobs and machines. This means that each job has exactly one operation for each machine and each machine has exactly one operation for each job assigned to it. Therefore, each machine has exactly n operations assigned and each job is composed by exactly m operations, where n is the number of jobs and m the number of machines. This way, it is possible to express a JSSP instance with two matrices of $n \times m$. One matrix is used for the associations of operation to machine, and the other for the operation lengths. The number of operations is quickly calculated as $n \cdot m$. This convention does not undermine the hardness of the instance and simplifies its representation, therefore, it is widely used in literature. Table 1 resumes the most commonly used benchmarks, derived from the following works:

- **Lawrence (la)**: A set of 40 instances for JSSP [72];
- **Adams/Balas/Zawack (abz)**: 9 instances spanning from 100 to 500 operations to be scheduled [19];
- **Applegate and Cook (orb)**: 5 instances (orb1–orb5) were generated in a challenge in Bonn. Each instance has a duplicate with the same operation sequence but different processing times, for a total of 10 instances [28];
- **Storer et al. (swv)**: 20 instances spanning from 200 to 500 operations to be scheduled [73];
- **Yamada/Nakano (yn)**: 4 instances of size 20 × 20 with processing times drawn from a uniform distribution in the interval [10,50] [74];
- **Taillard (ta)**: Taillard created a benchmark of 260 instances for various scheduling problems, emulating the size of real industrial problems. 80 of those instances are JSSPs [27];
- **Demirkol et al. (dm)**: A collection of 600 scheduling instances (40 JSSPs) with both completion time and lateness optimization criteria [75].

Of the benchmarks listed in Table 1, the largest instances are in **ta**, created by Taillard in 1992. He published a benchmark simulating the size of real industrial data, with the largest JSSP instances comprising 100 jobs to be scheduled on 20 machines, for a total of 2000 operations. However, problems encountered in nowadays real-world industrial domain are far larger.

2.4.1. On large-scale instances

Few works in literature try to address the problem of large-scale scheduling. Clearly, size is not the only differentiating factor between academic benchmarks and real-world scheduling scenarios. However, while factors like flexibility, setup times etc, are addressed by the literature [24,25,76], we find that the size factor is generally much less regarded.

Some recent studies, like Zhai et al. in 2014 [77], address the problem, but are still reluctant to test instances beyond 100 jobs on 50 machines (5000 operations). Others, like one proposed by IBM, mention tests done on large-scale scheduling instances that go beyond 1000 jobs, but said benchmark is not of public domain [78]. When looking at benchmarks with due date optimization criteria, it is possible to find instances up to 20 000 operations [79], but also in this case, the benchmark is not public. Yet, in domains like semi-conductor manufacturing, the weekly workload reaches up to more than 10 000 operations to be scheduled on more than 100 machines only in the shipping section, while in the production section about more than 100 000 are scheduled every week on more than 1000 machines [65,80].

3. Constraint programming

Constraint programming (CP) [51] is a declarative approach that builds on solving constraint satisfaction problems (CSPs). Hence, a problem at hand must be modeled as a CSP, but after that, the whole solving process is handled by a general CP solver. This results in various advantages, like the compactness of the code that can be cheaply maintained in the long run and quickly adapted to new needs [31]. Also, this typically gives the opportunity of an easy inclusion of knowledge by domain experts, in order to create domain specific heuristics. Moreover, it is easier to prove correctness of models based on logic rules, compared to the slew of large and complex testing methods typical the procedural paradigm.

A CSP is defined as a triple $\langle V, D, C\rangle$, whereby V is a set of variables, D is a set of value domains so that for each $v \in V$ there is exactly one $d \in D$, and C is a set of constraints imposed on the variables. A solution s for a CSP is a complete assignment of domain values to variables so that no constraint is violated. Since there can be more solutions for a CSP, let S be the set of all the solutions for a certain CSP. A *constraint optimization problem* COP is a CSP with an objective function $f : S \Rightarrow \mathbb{R}$ that associates each solution $s \in S$ with a value. An optimal solution is a solution for a s that minimizes(maximizes) $f(s)$.

One of the most classic variants of CSPs uses variables with finite and discrete domains. This means that the domain sizes are limited in form of lower and upper bounds and the domain values are discrete, normally integers, in which case variables are called *integer variables*. Depending on the solver that is used, different constraints can be employed to restrict value combinations for variables that are incorporated in the constraint. For example, *primitive constraints* express that some variable's value must be equal/unequal or smaller/greater than some other variable's value or constant value (e.g. $v1 \neq v2$, $v1 > 6$). *Arithmetic constraints* express arithmetic operations like $v1 + v2 = v3$. *Global constraints* restrict value combinations for a set (unrestricted in size) of variables and typically implement algorithms for solving special sub-problems which, without such a global constraint, have to be represented with primitive and arithmetic constraints. Global constraints significantly increase the expressive power in the way that for each global constraint, the set of primitive/arithmetic constraints used instead can be very large.

Take as an example the *alldifferent* global constraint, assuring that, out of some predefined set of variables, there are no two variables with the same value. For instance, having three variables $v1$, $v2$, and $v3$ with equal domains ranging from 1 to 3, $alldifferent(\{v1, v2, v3\})$ states that exactly one variable takes the value 1, 2 and 3, respectively. With primitive constraints, three constraints would be needed for expressing the same, i.e. $v1 \neq v2, v2 \neq v3, v1 \neq v3$. When there are 1000 variables that should get different values, still a single *alldifferent* constraint suffices whereas, without this global constraint, roughly half a million primitive constraints are needed.

Similarly to global constraints that lift expressive power with respect to certain sub-problems, higher order types of variables, e.g. *set variables*, have been introduced. Whereas a finite-domain integer variable takes values from a finite set of integers, a finite-domain set variable takes values from the power set of a finite set of integers. Consequently, the expressive power of set variables is exponentially higher, since for a domain with n values the corresponding power set contains 2^n values.

Another type of higher order variable are *interval variables*. Interval variables are designed to meet the special requirements of scheduling problems. Interval variables are in particular well-suited for representing time intervals, since they consist of a start time, a length and an end time, all from integer domains. An interval variable automatically enforces an arithmetic constraint, i.e. $start + length = end$. Employing interval variables, a job operation can be represented by a single (interval) variable.

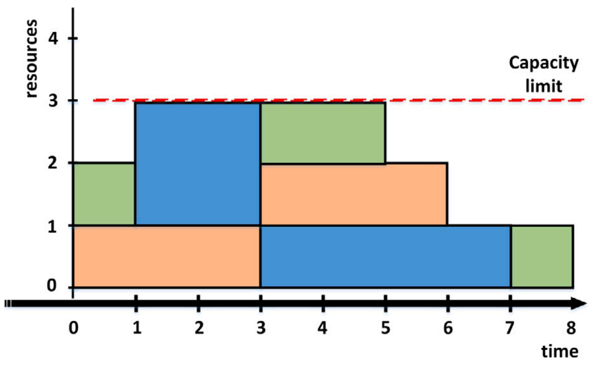

Fig. 4. An example of a cumulative schedule. The rectangles represent tasks, with production time indicated by the width and resource consumption indicated by the height. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

3.1. A constraint model for JSSP

The JSSP can be easily represented as a discrete finite COP. In fact, many solvers implement problem-dependent variable types and global constraints to deal with scheduling problems. For example, the *NoOverlap* constraint defined in Section 2.1 is provided out-of-the-box by both solvers we take into consideration, OR-Tools and CP Optimizer.

A more general formulation of the overlap constraint is the *cumulative* constraint [81]. The cumulative constraint ensures that at each time point, operations in the scope of the same constraint do not exceed a certain resource limit (i.e. capacity) in total. Fig. 4 shows an example of a schedule regulated by a cumulative constraint. Each task has a processing time (width) and a certain resource consumption (height). This comes handy in contexts where the resources are consumable entities like electricity or water (*e.g.* in electrical plants), and it is typical for different tasks to have different resource requirements. In JSSP, however, resources (machines) are reusable, because they can be used by one process at a time and are not depleted by the use. Thus, the resource consumption of an operation is always 1, because an operation cannot be processed by more than one machine at the same time. Similarly, also the capacity of the machines is 1, because an operation can only be processed by one machine. Summing up, for JSSP, the NoOverlap constraint is equivalent to a cumulative constraint with the capacity of 1 for all machines and the resource consumption of 1 for all operations.

Another structure that is targeted towards scheduling problems is the *interval variable*, which both CP Optimizer and OR-Tools support natively. The formulation of the model is shown in Fig. 5.

Two integer matrices are expected as input: for each operation, *opLength* contains the operation length and *opSuccessor* contains the machine index of its successor. The input data is used to represent each operation as an interval variable of given length. Two sets of constraints over the interval variables are needed: First, precedence constraints are represented as primitive constraints on the start and end times. Second, a NoOverlap constraint is instantiated for each machine. Finally, the objective function defines the makespan/completion time to be optimized, *i.e.* it minimizes the end time of the last operation in a schedule.

4. Evaluation

The goal of this study is to assess the capabilities of state-of-the-art CP solvers on JSSP instances that are representative for current industrial scheduling problems. For simulating different application scenarios we use benchmark instances that range from small to real large-scale.

```
INPUT
opLength: IntegerArray[1..numJobs][1..numMachines]
opSuccessor: IntegerArray[1..numJobs][1..numMachines]

VARIABLES
ops: IntervalVariableArray[1..numJobs][1..numMachines]

$\forall j \in \{1,\dots,numJobs\}, \forall m \in \{1,\dots,numMachines\}$:
ops[j][m].length = opLength[j][m]

CONSTRAINTS
$\forall j \in \{1,\dots,numJobs\}, \forall m \in \{1,\dots,numMachines\}$
$with\ opSuccessor[j][m] \neq NULL$:
ops[j][m].end $\leq$ ops[j][opSuccessor[j][m]].start

$\forall m \in \{1,\dots,numMachines\}$:
noOverlap($\{op | op \in ops[1..numJobs][m]\}$)

OBJECTIVE
minimize max($\{op.end | op \in ops\}$)
```

Fig. 5. Constraint model for JSSP.

4.1. Cutting-edge CP solvers

OR-Tools (ORT) is an open-source constraint solver created by Google and originally developed to deal with their complex routing problems. In the last decade, OR-Tools took part in the yearly held *MiniZinc Challenge*, a global competition to test the capabilities of CP solvers that support the MiniZinc modeling-language [82]. Since 2018, OR-Tools took the lead on almost every category, placing himself among the best CP solvers currently available.

CP Optimizer (CPO) is a constraint solver tool part of the IBM ILOG CPLEX Optimization Studio. ILOG was a renowned software company focused on optimization and supply-chain problems. Owner of the famous CPLEX mathematical programming software, ILOG was absorbed by IBM in 2008 and since then, a lot of focus was put in the development of a general optimization suite, with CPLEX to deal with MILP problems and CP Optimizer for CP problems. CP Optimizer uses its own modeling language, OPL, and does not support MiniZinc. For this reason, CPO and ORT were never compared directly. Since 2020, the IBM ILOG CPLEX Optimization Studio has been entering the MiniZinc Challenge. However, just the CPLEX library is used to solve the MILP-translated problems of the challenge, while the constraint solving library is not used.[4]

4.2. Benchmark instances

Problem instances are divided in three benchmarks:

- Classic benchmark: A selection of 74 of the most used benchmark instances in literature (Section 2.4).
- Large-TA benchmark: Novel benchmark inspired by the work by Taillard [27], with 90 instances up to 1 million operations.
- Known-optima benchmark: 24 novel instances with known optimal makespan.

Fig. 6 shows the distribution of the sizes of the benchmark instances, with respect to number of machines and operations. The size of the

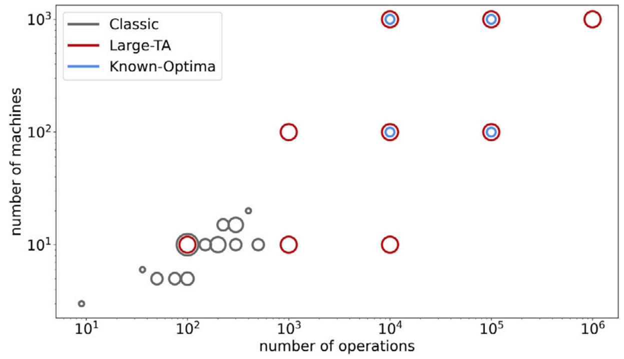

Fig. 6. Distribution of the size of problem instances used in our experiment. The size of the circles reflects the number of instances with that number of operations/machines in the corresponding benchmark. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 2
List of the instance classes and number of instances in the Large-TA benchmark.

Machs	Jobs	Operations	Num instances
10	10	100	10
100	10	1 000	10
1000	10	10 000	10
10	100	1 000	10
100	100	10 000	10
1000	100	100 000	10
10	1000	10 000	10
100	1000	100 000	10
1000	1000	1 000 000	10

indicator gives an idea of the number of instances corresponding to that number of operations and machines. We can see that the green circles, representing the classic benchmark, are concentrated at the lower end of both machines and operations. The large-TA benchmark (orange circles) covers a good amount of the area, from the lower end, to the extremely high end with a grand total of 1 million operations to be scheduled on 1000 machines. The known-optima benchmark covers a subset of the sizes of the large-TA benchmark, but offers instances with known optimal makespan.

The classic benchmark is composed by 74 instances among the most used in literature. The selection followed the criteria of the MiniZinc challenge, using directly the problem instances available for JSSP in their repository on github.[5] It is a subset of the instances described in Section 2.4. In particular, it comprises:

- 3 instances from *ft*;
- 40 instances from *la*;
- 5 instances from *abz*;
- 10 instances from *orb*;
- 14 instances from *swv*;
- 1 instance from *yn*;
- In addition, a single instance 3×3 from [83]

The Large-TA benchmark is composed of 90 rectangular JSSP instances. This means that once fixed the number of jobs n and machines m, every job has to go through every machine exactly once, therefore the number of operations per job is always the same and coincides with m. Table 2 shows the instances grouped by size. The instances with 100 000 operations on 1000 machines represent the order of magnitude of real weekly workloads encountered in the semi-conductor industry.[6]

[4] For details https://www.minizinc.org/challenge2020/results2020.html.

[5] https://github.com/MiniZinc/minizinc-benchmarks/tree/master/jobshop.

[6] E.g. our project partner, Infineon Technologies Austria AG.

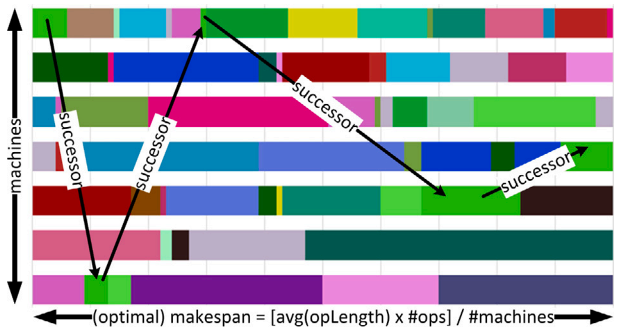

Fig. 7. Principle of instance generation of the instances of the known-optima benchmark. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

The instances with 1 million operations, while not having a correspondence to real life cases, are useful to test the limits of the solvers. The operation lengths are generated randomly using the *utils* library of Java 8 and are in the range of [0 … 100] for smaller instances and [0 … 1000] for larger instances.

The known-optima benchmark, contrary to the large-TA, does not comprehend rectangular instances, *i.e.* a job can have less operations than the number of machines, and it can happen that two operations of the same job are processed by the same machine (like in reality).

A problem instance of the known-optima benchmark is created as follows: First an optimal solution is produced without (idle) holes for (1) a given number of job operations to be scheduled, (2) a given number of machines and (3) the desired optimal makespan. This is done by randomly partitioning the machines' time continuum into the predefined number of partitions. Each partition corresponds to the processing period of one operation. Consequently, optimal makespan, average operation length (avg(length)), number of machines ($\#machines$) and number of operations ($\#ops$) all relate conforming to $makespan = \frac{\#ops \times avg(opLength)}{\#machines}$. Based on such a partitioning, successor relations are randomly generated and this way form the jobs. Each partition can have maximally one successor (and also maximally one predecessor). Thus, for a partition a successor is a partition with a starting time greater than its finishing time. Fig. 7 shows the principle. This creates scheduling problem instances where the best makespan is known before-hand, giving the opportunity to immediately check the quality of a given solution.

We applied two different procedures for generating random successor relations based on a pre-calculated solution:

1. **Short-jobs instances:** For each operation op (in random order) define as successor suc a random operation such that
   - suc is not on the same machine as op.
   - suc starts later than op ends.
   - suc is not yet a successor of another operation.
   - If no such suc exists op has no successor.
2. **Long-jobs instances:** For each operation op (in random order) define as successor suc an operation such that
   - suc is not on the same machine as op.
   - suc starts later than op ends.
   - suc is not yet a successor of another operation and
   - the time between op ends and suc starts is minimal.
   - In case that there are multiple possible successors, a random one is chosen.
   - If no such suc exists op has no successor.

The two different generating approaches result in benchmark instances that are different in nature: (1) produces many jobs consisting of a small number of operations. On the contrary, (2) produces fewer jobs but with a larger number of operations per job. This way, we can simulate complex products that need a lot of steps to be completed, as well as cases when there is a vast variety of simple jobs. The total amount of generated instances is 24 (12 short-jobs, 12 long-jobs). The total number of operations goes up to 100 thousand operations to be scheduled on up to one thousand machines. All instances have a minimal makespan of 600 000, which is roughly a week in seconds, i.e. a very common planning horizon, at least in the semi-conductor domain.

Tables 3 and 5 show the number of jobs and the min, max and average number of operations per job, for the long and short-jobs instances, respectively. The first group of three instances in Table 3 shows that in the long-jobs, the max number of operations exceeds the number of machines. This means that there are jobs that are processed twice or more by the same machine. Similarly, there are jobs that go through just a subset of the machines, since the min #ops value is smaller than the number of machines. The second group has 1000 machines, and presents much shorter jobs on average. If we compare it with the same group of the short-job instances in Table 5, the long-jobs ones have three times the operations on average (3.5 vs. 10), compared to the 20 times multiplication factor of the first group (4.6 vs. 97.1). The min #ops of instance 1000-10000-1 reveal that there are also jobs composed by a single operation even in the long-job instances. Table 5 shows that in the short-jobs instances there are no cases where a job is processed by all machines, in fact, the max number of operations per job is 19. The low average number of operations results in a high number of jobs (up to 20 thousand) with 3 to 5 operations on average. Tables 4 and 6 show the distribution of the operation lengths of the operations. They vary a lot in both short-jobs instances (max gap between 1 and 600 000) and long-jobs instances (max gap between 6 and 600 000). Given the optimal makespan at 600 000, it means that in instances 1000-10000-1 long-jobs and 1000-10000-3 short-jobs, there is at least one operation which lasts as long as the whole optimal makespan.

Table 3
Long-jobs benchmark: operations and job counts.

#ma-#ops-file	#jobs	min #ops	max #ops	avg #ops
100-10000-1	103	37	134	97.1
100-10000-2	103	54	125	97.1
100-10000-3	103	28	128	97.1
1000-10000-1	1002	1	22	10.0
1000-10000-2	1002	2	22	10.0
1000-10000-3	1002	2	24	10.0
100-100000-1	109	1	1021	917.4
100-100000-2	114	1	1019	877.2
100-100000-3	109	185	1011	917.4
1000-100000-1	1002	74	133	99.8
1000-100000-2	1002	69	131	99.8
1000-100000-3	1003	68	129	99.7

Table 4
Long-jobs benchmark: operation lengths.

#ma-#ops-file	min opLen	max opLen	avg opLen
100-10000-1	2	72 196	6000
100-10000-2	1	49 264	6000
100-10000-3	1	53 090	6000
1000-10000-1	6	600 000	60 000
1000-10000-2	2	547 505	60 000
1000-10000-3	1	567 640	60 000
100-100000-1	1	6549	600
100-100000-2	1	7777	600
100-100000-3	1	6982	600
1000-100000-1	1	70 829	6000
1000-100000-2	1	63 685	6000
1000-100000-3	1	70 263	6000

Table 5
Short-jobs benchmark: operations and job counts.

#ma-#ops-file	#jobs	min #ops	max #ops	avg #ops
100-10000-1	2162	2	12	4.6
100-10000-2	2192	1	13	4.6
100-10000-3	2169	1	13	4.6
1000-10000-1	2882	1	9	3.5
1000-10000-2	2863	1	10	3.5
1000-10000-3	2897	1	9	3.5
100-100000-1	20 685	2	16	4.8
100-100000-2	20 870	2	19	4.8
100-100000-3	20 767	2	16	4.8
1000-100000-1	21 280	2	16	4.7
1000-100000-2	21 349	2	15	4.7
1000-100000-3	21 338	2	16	4.7

Table 6
Short-jobs benchmark: operation lengths.

#ma-#ops-file	min opLen	max opLen	avg opLen
100-10000-1	1	69 723	6000
100-10000-2	1	53 463	6000
100-10000-3	1	68 936	6000
1000-10000-1	2	503 525	60 000
1000-10000-2	3	471 671	60 000
1000-10000-3	2	600 000	60 000
100-100000-1	1	9158	600
100-100000-2	1	8373	600
100-100000-3	1	6505	600
1000-100000-1	1	86 811	6000
1000-100000-2	1	68 044	6000
1000-100000-3	1	65 721	6000

The large-TA benchmark and the known-optima benchmark are both available for download at [84], together with the files of the CP encodings and a detailed technical report. The instances for the classic benchmark are collected in the git repository of MiniZinc (link provided in Section 4.2).

4.3. Experimental setup

The experiment is organized in three tests, one for each benchmark described in Section 4.2. The main research question that we aim to answer is whether it is possible to tackle real industrial scheduling problems with the tools provided by the state-of-the-art scheduling research, in particular regarding constraint programming solvers. With this aim in mind, we designed the experiment with further research questions specific for each test. Solvers are tested with two configurations, single-core and quad-core, depending on how many worker threads are allowed to run (one or four respectively). The optimization criterion for all the instances of the various benchmarks is the makespan, also referred to as completion time, which indicates the length of the time interval between the start of the first operation in a schedule and the completion of the last. Another common optimization criterion in scheduling problems is the tardiness, *i.e.* the minimization of due date violation of each job. We decided to use the makespan because it is the most used criterion in literature (as highlighted in Section 4.2) and, more importantly, it makes the problem harder. In fact, in tardiness minimization problems, the information of the jobs' due date can be used to infer an ordering of the jobs to rapidly stir the search towards a (near) optimal solution. This effect would be particularly exacerbated in the case of the known-optima benchmark. In makespan minimization, it is effectively like all jobs had the same due date, hence, all job orderings have potentially the same probability to lead to an optimal solution (excluding factors like the number of operations per job and their length, which can also be derived in tardiness minimization problems).

The **classic benchmark test** serves the purpose of showing that cutting-edge CP solvers have no problems to solve problem instances comprised in the benchmarks that are typically used in literature also in a very short time. Inspired by the MiniZinc challenge, we allow a timeout of 20 min for each instance. We want to replicate the conditions of the challenge as well as possible, also in the calculation of the score to determine the winner. We will use the *complete scoring procedure*, based on a Borda count voting [85] that takes into account both the quality of the solution and the time needed to find it.[7] This test is designed around the following research question:

- **RQ1.** How would CPO perform in a MiniZinc challenge scenario on JSSP instances, in comparison with the current winner (ORT)?

After establishing this baseline, we target the core issue with the **Large-TA benchmark test**. This benchmark is close to the type of instances found in literature, but projected in a scale that is typical of current industrial domains (up to 1 million operations). In this case, we set the timeout for each instance to six hours, in order to simulate a scenario where the calculation of the next factory schedule is done overnight. The research question to be answered is presented as follows:

- **RQ2.** Is it possible to solve industrial-size JSSPs within six hours (overnight scenario) using state-of-the-art CP solvers?

The instances of the large-TA benchmark cover a large span of sizes, as shown in Fig. 6, but since the optimal solution is unknown, it is not always easy to determine the effectiveness of the approaches.

The **known-optima benchmark test** has the advantage that it is always possible to evaluate a solution against the known optimal value. The value of the optimal makespan is not given as input to the solvers, therefore, they still have to search for the solution and prove the optimality with the lower bound comparison. Also in this case, the timeout for each instance is set to six hours. With this test, we aim to answer the following research questions:

- **RQ3.** How close to an optimum are the solutions found by the solvers on industrial-size JSSPs?

Furthermore, we explore further research questions that span across the tests:

- **RQ4.** How much do the solvers benefit from additional CPU cores in the search of solutions?
- **RQ5.** Which solver is the best performer overall?
- **RQ6.** How do the considered solvers compare with the state of the art?

Concerning the solvers, we use version 12.10.0 for CP Optimizer and version 7.7.7810 for OR-Tools. The two solvers use different modeling languages, but they are both implemented in C++ and offer a Java wrapping to directly interface with the solver. To make the comparison as fair as possible, we implemented the CP encoding using the Java interfaces. In CPO we selected the default search parameters, which turned out to be the most effective ones after a preliminary test. In ORT we decided to use the CP-SAT solver, because the old CP solver is not updated any more by the Google researchers, and because CP-SAT proved to be better on average after a pre-test. The experiment is conducted on a system equipped with a 2 GHz AMD EPYC 7551P 32 Cores CPU and 256 GB of RAM.

[7] Complete description of the scoring procedure can be found at https://www.minizinc.org/challenge2020/rules2020.html.

5. Results

In the experiment, we refer to each couple of solver/core configurations as a *system*. We will test the capabilities of four systems: CPO/single-core, CPO/quad-core, ORT/single-core, ORT/quad-core. In particular, when talking about CPO systems, we refer to CPO/single-core and CPO/quad-core, when talking about single-core systems, we refer to CPO/single-core and ORT/single-core, etc. Each system will continue to improve the solution of an instance until the optimal makespan is found, or the timeout established for the test occurs. At the timeout, the system outputs the solution with the best makespan found until that point.[8]

5.1. The classic benchmark test

To evaluate the performance of the systems, the results were analyzed conforming to the *complete score procedure* of the MiniZinc Challenge. The scores were calculated considering all four systems together (as opposed to pairwise confrontation). The systems have 1200 s per instance to find a solution. With this scoring procedure, each instance acts like a "judge" that assigns points to the systems. For each instance, a system scores 1 point for each other system that generated a worse solution than itself (in this case, a larger makespan). If two solutions have the same makespan, then the score is calculated using the solving time: given two systems a and b, let t_a and t_b be the solving times of the systems for one instance. The score of a is $\frac{t_b}{t_a+t_b}$, while the score of b is $\frac{t_a}{t_a+t_b}$. Thus, a system receives a score between 0 and 1, proportional to time needed by a concurrent system.

With respect to **RQ1** *(relative performance of CPO and ORT on MiniZinc challenge scenario)* we summarize the outcome of the four systems as follows:

1. CPO/single-core: 124.1
2. CPO/quad-core : 114.9
3. ORT/quad-core : 114.0
4. ORT/single-core: 94.9

To dive deeper into the results obtained by the systems, we can look at the number of instances solved optimally within the imposed timeout of 20 min. When talking about optimal solutions, it is intended a solution that is *proved* to be optimal, *i.e.* there is no gap between the lower bound calculated by the solver and the solution found. Considering the total of 74 instances, ORT/single-core solved optimally 56 instances and ORT/quad-core 55, while CPO solved optimally 59 instances in single-core and 60 in quad-core configuration. In particular, CPO/single-core was the only system capable of finding an optimal solution for *abz7*, while CPO/quad-core was the only one to solve optimally *swv3* and *swv14*. In conclusion, CPO systems performed slightly better, but in general, all systems solved optimally about 75% of the instances. Another interesting case is instance *la29*, where both ORT/single-core and CPO/quad-core found a solution with a makespan of 1152, but were not able to prove its optimality (this makespan was found to be optimal in [86]).

In the context of this test, We denote an instance for which at least one system was not able to reach the optimal makespan as a *non-optimal instance*. The optimal makespan was achieved by *all* systems in 55 out of 74 instances, therefore, the remaining 19 constitutes non-optimal instances. The performance of the systems are pretty close in general: CPO/quad-core finds 9 best solutions (*i.e.* solutions strictly better than the ones found by the other solvers), followed by ORT/single-core with 7, CPO/single-core with 6 and ORT/quad-core with 5 (Note that it is possible for more than one system to find the same best solution). Even considering the instances where the difference between the best and worst system is at its highest (*swv11* and *swv12*), this difference still remains below 10% of the total makespan.

[8] The interested reviewer can find additional details on the results of each test in the appendix (supplementary material).

Table 7
Average makespans for the Large-TA benchmark, comparing OR-Tools and CP Optimizer with different core configurations. Every row shows the average value calculated on a group of 10 instances with the size specified by the columns "Machs" and "Jobs".

Group	Machs	Jobs	ORT		CPO	
ID			Single-core	Quad-core	Single-core	Quad-core
1	10	10	8169.9	8169.9	8169.9	8169.9
2	100	10	55 224.5	55 224.5	55 224.5	55 224.5
3	1000	10	514 393.3	514 393.3	514 393.3	514 393.3
4	10	100	54 876.9	54 863.6	54 858.6	54 858.6
5	100	100	114 476.2	113 692	80 565.5	80 976.9
6	1000	100	600 376.1	597 618.2	543 981.7	544 216.0
7	10	1000	550 235.0	549 996.4	515 429.7	515 429.7
8	100	1000	686 409.4	686 513.3	536 414.7	536 325.0
9	1000	1000	No solution	No solution	1 018 093[a]	958 664.2

[a]Average computed on only 6 out of 10 instances (no solution was found in the other 4).

The results of the classic benchmark showed that both solvers are capable of dealing with small-size instances within a short time horizon. The worst solution registered was about 8% off the best solution found. Therefore, the difference between solutions is quite limited, regardless of the solver or the number of cores used. Concluding, to answer **RQ4** *(on the benefit of additional cores)* in the context of the classic benchmark, every system was able to find the optimal makespan in about 75% of instances, regardless of the number of cores used. In the non-optimal instances, the limited margin between solutions show that there is not a significant benefit in using additional cores, in both solvers. Considering the MiniZinc Challenge scores, ORT seems to benefit slightly from additional cores, while CPO is better off with the single-core configuration.

5.2. The large-TA benchmark test

This test aims to assess the capabilities of the systems on large-scale JSSP instances. The benchmark is composed of 90 instances, divided by size in 9 groups of 10 instances each. The timeout for the solving time of each instance is set to six hours.

The results are presented in Table 7, where the average makespan for each instance group is shown. The evolution of the search is depicted with a plot for each instance group (Fig. 8). Each line on the plot represents the average makespan of the 10 instances of the corresponding group and system, calculated at each time-point from 0 to the timeout of 21 600 s (6 h). Red lines identify the CPO systems and blue lines identify the ORT systems (dotted lines correspond to the quad-core systems).

Considering the instances with 10 jobs (groups 1, 2, 3), all systems converge to optimal solutions. Note that, even if the benchmark was not constructed to have a known optimum, the solvers are able to prove the optimality of their solution by converging to the calculated lower bound. The first row of plots in Fig. 8 shows the evolution of the search in the instances with 10 jobs. In group 1, all systems converge to the optimal solutions in 2 s or less. In the other two cases, CPO systems are always able to find the optima in less than 2 s, while ORT/single-core needs 15 and 120 s for group 2 and 3 respectively.

Considering the instances with 100 jobs (groups 4, 5, 6), a difference between the two solvers starts to be noticeable, in favor of CPO. In group 4, the gap between the systems is small, about 1%. Fig. 8 shows that, also in this case, both CPO systems converge to the (proven) optimal makespans almost immediately, while ORT systems need about 3 h to reach near optimal solutions, slowly improving them until the timeout occurs. In group 5, the gap rises to 42%, but the shape of the

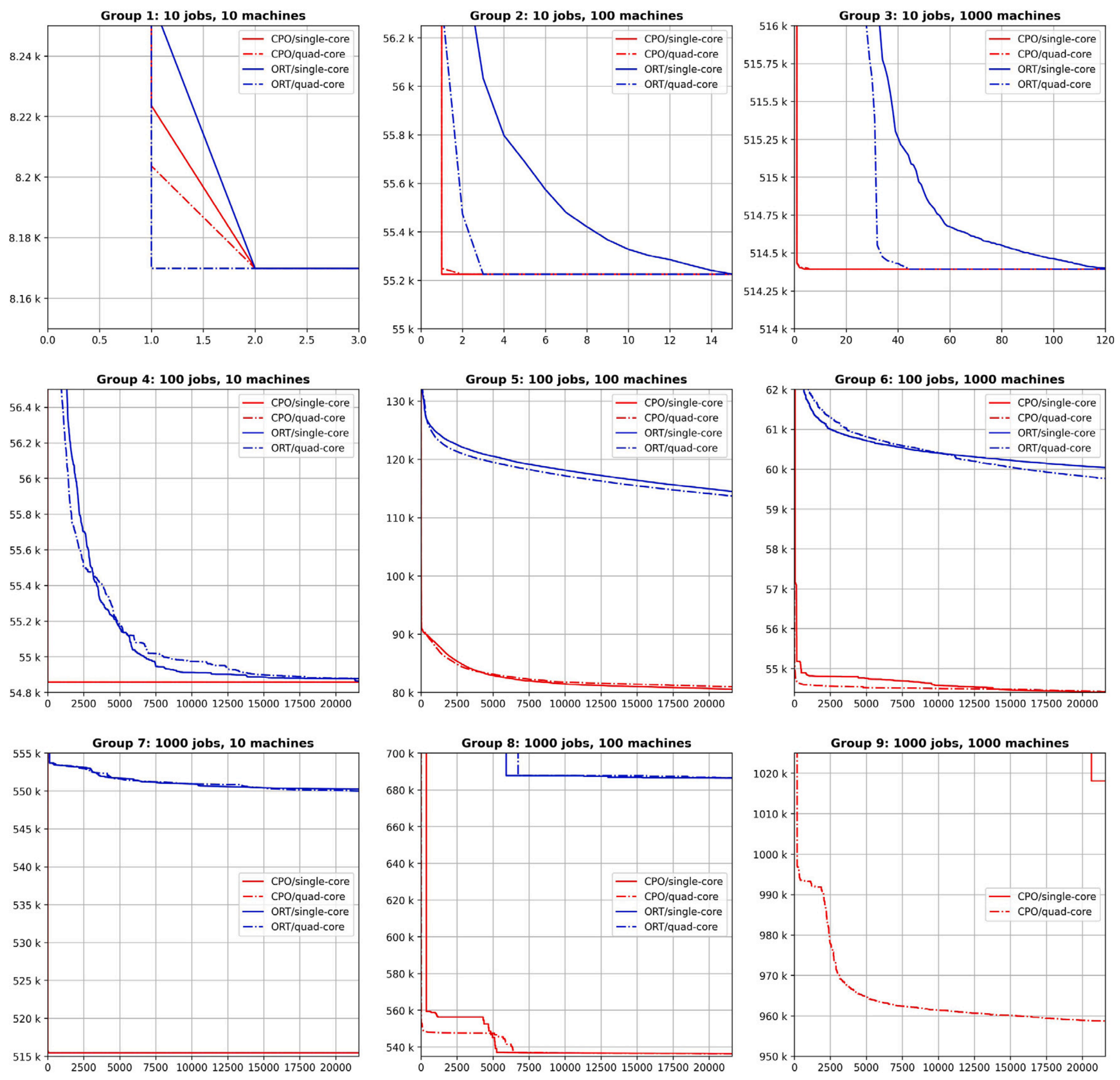

Fig. 8. Results of the Large-TA experiment. The 9 plots illustrate how the evolution of the solving process over time for the four systems. The y axis shows the average makespan of the instances of the correspondent group, while the x axis is the temporal line from zero until the timeout of six hours (21 600 s). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

curves is similar, with CPO systems also converging slowly to the final solutions and improving them until the end (the final solutions were not proven to be optimal). In group 6 the gap between CPO and ORT systems is 10%, always in favor of the former, and also the convergence is shown to be much faster.

Considering the instances with 1000 jobs (groups 7, 8, 9), the gap between the two single/core systems is around 6% in group 7, 28% in group 8 and "infinite" in group 9. In fact, the ORT systems were not able to solve any of the ten instances within the timeout, while CPO/single-core was able to find solutions for 6 out of 10 instances and CPO/quad-core solved all of them.

RQ4. *(on the benefit of additional cores)* Concerning CPO, the performance of the quad-core systems is mostly in line with the findings of the classic benchmark test. Looking at Fig. 8, we can see that the improvement is limited and often negligible up to a certain instance size. In group 6, however, the additional cores help to find the solutions 4 h faster, on average (even if the end solutions are the same). The most remarkable improvement is in the group of the largest instances (1000 × 1000), where CPO/quad-core was not only capable of finding solutions for all instances in the group, but it was capable to do it within 10 min (In comparison, CPO/single-core took almost 6 h to find solutions for just six of the instances). This is due to a search strategy called *Iterative Diving* (ID),introduced in CPO 12.9, which is tried when 4 or more cores are used in search. It focuses on quickly produce feasible solutions, and it is particularly beneficial on large problems, at the expense of a slower convergence on smaller instances.

We conducted an additional test, activating ID on a single core on one large instance (1000 × 1000) and one smaller instance (1000 × 10). In the large instance, the single-core with diving showed the same capabilities of the quad-core, implying that it is indeed the search strategy that allows CPO to promptly find solutions for the large instances. In the small instance, ID on single core found the optimal solution faster than single core with the default strategy (1 s vs. 46 s). The default strategy, however, was able to find the best lower bound and prove the optimality of the solution immediately, while ID in single core was not able to do it in 6 h. The 4 cores system, using both strategies in parallel, is able to find the best solution in one second *and* prove its optimality immediately.

Concerning ORT, there is a significant improvement in solving time in the 10 jobs instances (about 50% faster) with the additional cores. In the larger instances, however, the improvement is not so relevant, averaging around 1%. Similar to CPO, also ORT applies alternative search strategies when multiple cores are available. In particular, one strategy that is used is to set the value of the current lower bound as an hard constraint for the optimization criterion. This is a valuable strategy when the solver is able to find a good lower bound, but can lead to unsatisfiability is the lower bound is smaller than the optimal solution.

Summing up, to answer **RQ2** *(on the applicability of CP solvers to large JSSPs)* the two solvers perform equally well in the smaller instances of the benchmark, but the supremacy of CPO becomes increasingly evident the larger the instances are. CPO is able to deal with instances up to the extreme size of 1 million operations (group 9) for the most part. ORT, on the other hand, while not being able to match the results of CPO, it is still able to cope with large-scale problems up to 100 000 operations, maintaining a somehow respectable gap with its opponent (between 6% and 42% off CPO's makespans). The quad-core systems provide alternative search strategies that work in parallel with the default and showed to be effective on the smaller instances on ORT and on the largest instances on CPO.

5.3. The known-optima benchmark test

The asymptotic behavior shown in Fig. 8 for the large-TA test may suggest that the final solutions found are close to the actual optimal solutions. However, there is no way to be certain until the optimum is actually found. The known-optima test aims at solving this problem, evaluating the systems on a benchmark suite where the optimal solutions are known by design. Each instance is designed to have the optimal makespan of 600 000 s. Note that there can be more than one solution with the optimal makespan. As such, when talking about the optimal solution of an instance, we refer to any solution with a makespan of 600 000. Also in this case, the timeout for the solving time of each instance is set to six hours.

Results are presented in Table 8, where values in the column clarify if the optimal makespan was reached, or specify the gap (in percentage) between the solution found and the optimal one. Instances are divided in groups of 3 conforming to their size (number of machines/operations) and type (long/short jobs). Fig. 9 offers a graphical representation of the evolution of solutions over time, from 0 to the timeout of six hours (21 600 s). There is one plot for each group of instances. In each plot, a line represents the average makespan of the three instances of the corresponding group, at each time-point. This gives an idea of how fast the systems are converging to the final solution. Also in this case, red lines correspond to the CPO systems and blue to ORT systems, with dotted lines identifying the quad-core systems.

Concerning **RQ3** *(on the gap between achieved and optimal makespan in large-scale instances)*, CPO/single-core is able to solve optimally 16 out of 24 instances (66.6%). In general, it was able to solve almost all the short-jobs instances to optimality (beside 2 instances in group 7 and 8, which were still very close to the optimal makespan). Concerning the long-jobs instances, CPO/single-core found the optimal makespan in all instances with 10 000 operations (groups 1 and 3), while it was not possible to hit the optimum in the cases with 100 000 operations (groups 2 and 4). The hardest instances to solve were the ones in group 2, the three long-jobs instances with 100 machines and 100 000 operations: the worst result was about 80% off the optimum. This poor performance can be attributed to the behavior of the *NoOverlap* constraint. The reason is that the model instantiates a *NoOverlap* constraint for each machine, which operates on the operations on that machine. Since the instances are generated following a uniform distribution, we can assume the number of operations per machine to be roughly $Ops/Machs$ on average, *e.g.* group 1 has 100 machines and 10 000 operations, with an average of 100 operations per machine. The larger this number, the harder the work for each single *NoOverlap* constraint. Conforming to this assertion, instances in group 3 should be the easiest to solve, with 10 ops per machine, and instances in group 2 should be the hardest, with 1000 operations per machine. We can see that this is the case for the long-jobs instances with CPO systems, which are able to solve all

Table 8
Gap between the achieved makespan and the optimum (600 000), comparing OR-Tools and CP Optimizer with different core configurations.

Group	Type	Machs	Ops	ORT		CPO	
ID				Single-core	Quad-core	Single-core	Quad-core
1	Long	100	10 000	+136%	Optimum	Optimum	Optimum
	Long	100	10 000	+139.1%	Optimum	Optimum	Optimum
	Long	100	10 000	+134.6%	Optimum	Optimum	Optimum
2	Long	100	100 000	+174.5%	+174.5%	+80%	+80%
	Long	100	100 000	+167.3%	+167.3%	+78.1%	+78%
	Long	100	100 000	+172.8%	+172.8%	+78.3%	+78.4%
3	Long	1000	10 000	+45.2%	Optimum	Optimum	Optimum
	Long	1000	10 000	+36.2%	Optimum	Optimum	Optimum
	Long	1000	10 000	+48.9%	Optimum	Optimum	Optimum
4	Long	1000	100 000	+189.2%	Optimum	+34.3%	+33.5%
	Long	1000	100 000	+189.6%	Optimum	+36.2%	+33.7%
	Long	1000	100 000	+185.1%	Optimum	+39.9%	+39.1%
5	Short	100	10 000	+31.4%	+31.4%	Optimum	Optimum
	Short	100	10 000	+23.4%	+22.5%	Optimum	Optimum
	Short	100	10 000	+23.4%	+22.9%	Optimum	Optimum
6	Short	100	100 000	+8.7%	+8.7%	Optimum	Optimum
	Short	100	100 000	+8.3%	+8.3%	Optimum	Optimum
	Short	100	100 000	+10.2%	+10.2%	Optimum	Optimum
7	Short	1000	10 000	+134.3%	+134.3%	Optimum	Optimum
	Short	1000	10 000	+107.5%	+195.4%	Optimum	Optimum
	Short	1000	10 000	+125%	+85.4%	<+1%	<+1%
8	Short	1000	100 000	+37.1%	+37.1%	Optimum	<+1%
	Short	1000	100 000	+32.5%	+32.5%	Optimum	<+1%
	Short	1000	100 000	+34.8%	+34.8%	<+1%	<+1%

the instances in group 1 and 3 to optimality, but struggles slightly with instances in group 3 (100 ops per machine) and even more with group 2 (1000 ops per machine).

ORT/single-core is not able to solve optimally any of the instances. The best results are achieved in group 6, containing the smallest instances of the short-jobs type, while the worst results are collected on the long-jobs instances of group 2 (174% off the optimum) and group 4 (189% off the optimum). Like CPO, the long-jobs instances are typically harder to solve compared to the short-jobs ones, with one exception: The instances in group 7 should be to easiest instances of the short-jobs, considering their low number of ops per machine. However, they seem particularly problematic for ORT systems, which are not able to reach results below 85%. To explain this behavior, it is necessary to look back to Section 4.2, where the statistics of the known-optima benchmark is depicted: Looking at Table 6, we can see that instances in group 7 (1000–10000 instances) have an average operation length of 60 000 (about 10% of the length of the optimal makespan), with some operations reaching 600 000, which *is* the length of the whole optimal makespan. This means that every mistake on the operations' placement is very costly, because even the misplacement of a single operation can result in a significant delay in the schedule completion.

As an example, think of a scenario where the optimal schedule should take 5 days, but an operation which needs 2 days to be processed is scheduled on the machine only in the last day: the completion of the whole schedule will be delayed of 2 days, even if all the other operations are finished.

An analog situation is found in the instances of group 3, however, the long-jobs instances have longer chains of operations for each job, meaning that there are more constraints. On the one hand, more constraints mean that it is harder to find a solution (that respect all the constraints), *i.e.* the solution space is smaller. On the other hand, the constraint propagation eliminates a lot of schedules that would lead to poorly optimized results.

Summing up the results of the single-core systems, CPO/single-core is capable of finding a (near) optimal solution on every instance group of the short-jobs instances and on half of the long-jobs instances.

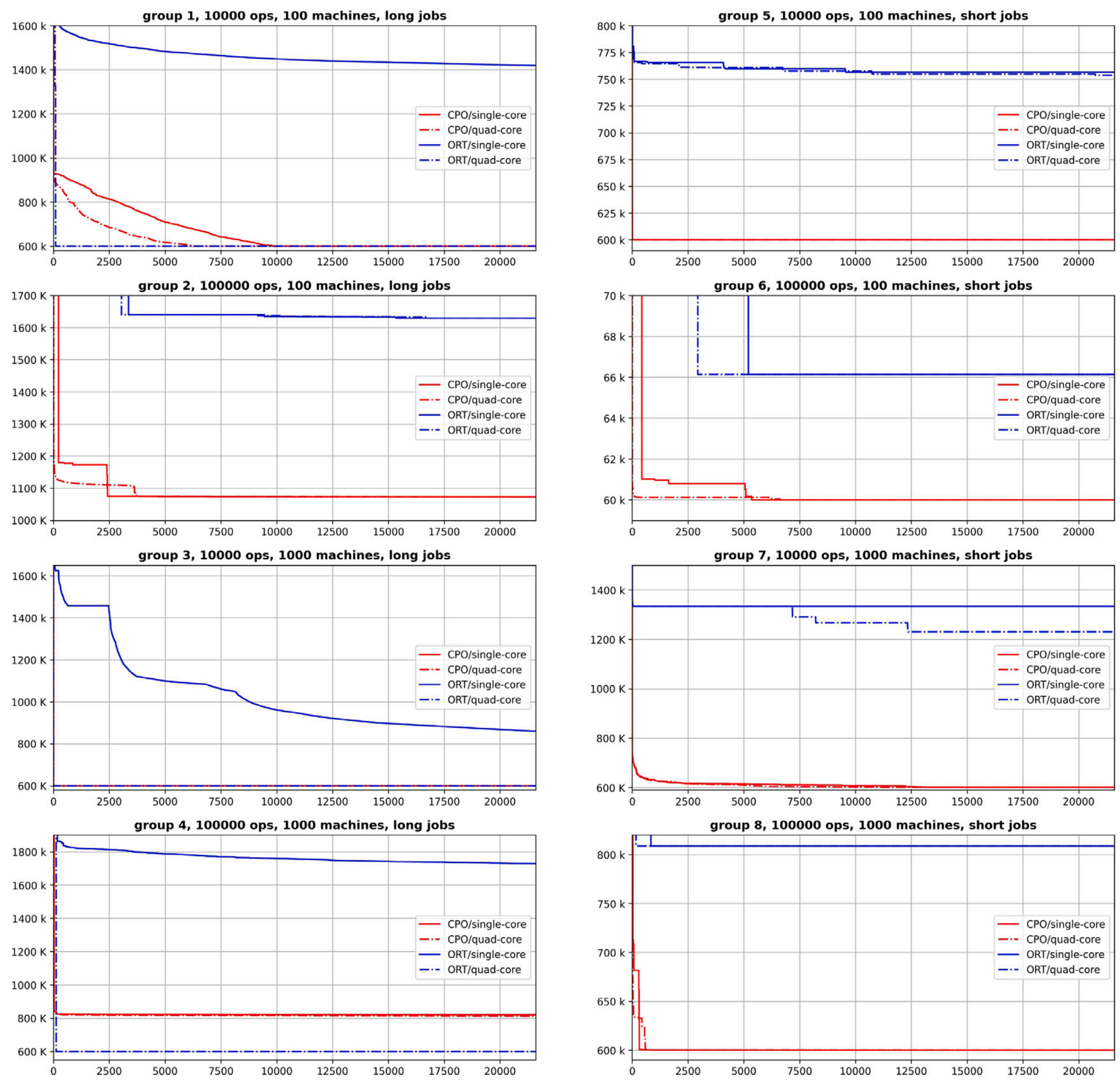

Fig. 9. Evolution of solutions over time in the known-optima benchmark. The y axis shows the average makespan of the instances of the correspondent group, while the x axis is the temporal line from zero until the timeout of six hours (21 600 s). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

ORT/single-core is not able to reach the performance of CPO/single-core in any of the groups, and is not able to find good solutions for about half the instances, where the achieved makespan was more than double the optimum.

RQ4. Concerning the quad-core systems, there is a drastic improvement for ORT. In fact, ORT/quad-core is able to find the optimum in 9 of the long-jobs instances (group 1, 3 and 4). The results on group 4 are particularly significant, in the sense that CPO systems were not able to solve any of the instances of this group to optimality. This peak in performance is due to the alternative strategy used by ORT in multi-core, which consists in using the current lower bound as a hard constraint for the makespan. In the known-optima benchmark, this strategy is particularly effective, because in this benchmark, the value of the optimal makespan is relatively easy to guess. In fact, being no gaps between operations in the optimal solution by design, the value of the optimal makespan is simply the sum of the lengths of the operations assigned to each machine. This technique worked best on the long-job instances, because the value of the optimal makespan combined with long chains of operations constrained the problem to the point that finding the optimal schedule was trivial. The propagation was not so effective in group 2, due to the high number of operations per machine, and in the short job instances, due to the short jobs being less constrictive than the long ones. CPO, apparently, does not benefit at all from additional cores: the solutions found by CPO/quad-core were just marginally better and sometimes worst than the CPO/single-core ones.

In Fig. 9 it is possible to see the difference between the four systems as the search progresses. In all the groups, for CPO, the difference between 1 and 4 cores is mostly insignificant in term of the end result, but there is an improvement on some instance groups concerning the solving time of the instances: the best improvement is found in group 6 where CPO is able to converge almost immediately to near optimal solutions with 4 cores, but needs about 90 min (5000 s) with a single core to reach similar results. Similar situation in group 1, where CPO/quad-core reaches optimality in less than two hours, while CPO/single-core needs about three to achieve the same results.

Concerning ORT, there is no large difference between single and quad-core systems in groups 5 and 2. In group 6, the ORT/quad-core is able to converge to the same solution of ORT/single-core in almost half of the time. In group 7, the additional cores improve the results of the solver by an average of 15% of the makespan. In group 1, the difference is remarkable, with ORT/quad-core being able to converge to optimality almost immediately, beating the ORT/single-core and also the CPO systems by more than an hour. Similar case in group 3, with ORT/quad-core matching the performance of CPO systems and

Table 9
Comparison of the state of art best solution against our best performer on the classic benchmark. The best lower bounds (LB) and upper bounds (UP) are highlighted in bold.

Instances	State of the art		CPO single-core		Instances	State of the art		CPO/single-core	
	LB	UB	LB	UB		LB	UB	LB	UB
abz5	**1234** [28]	**1234** [28]	**1234**	**1234**	*la30*	**1355** [19]	**1355** [19]	**1355**	**1355**
abz6	**943** [28]	**943** [19]	**943**	**943**	*la31*	**1784** [19]	**1784** [19]	**1784**	**1784**
abz7	**656** [87]	**656** [87]	**656**	**656**	*la32*	**1850** [19]	**1850** [19]	**1850**	**1850**
abz8	**648** [87]	**667** [88]	608	682	*la33*	**1719** [19]	**1719** [19]	**1719**	**1719**
abz9	**678** [87]	**678** [88]	630	686	*la34*	**1721** [19]	**1721** [19]	**1721**	**1721**
ft06	**55** [89]	**55** [89]	**55**	**55**	*la35*	**1888** [19]	**1888** [19]	**1888**	**1888**
ft10	**930** [48]	**930** [90]	**930**	**930**	*la36*	**1268** [91]	**1268** [91]	**1268**	**1268**
ft20	**1165** [92]	**1165** [92]	**1165**	**1165**	*la37*	**1397** [28]	**1397** [28]	**1397**	**1397**
la01	**666** [19]	**666** [19]	**666**	**666**	*la38*	**1196** [93]	**1196** [94]	**1196**	**1196**
la02	**655** [19]	**655** [19]	**655**	**655**	*la39*	**1233** [28]	**1233** [28]	**1233**	**1233**
la03	**597** [28]	**597** [95]	**597**	**597**	*la40*	**1222** [28]	**1222** [28]	**1222**	**1222**
la04	**590** [28]	**590** [95]	**590**	**590**	*orb01*	**1059** [28]	**1059** [28]	**1059**	**1059**
la05	**593** [19]	**593** [19]	**593**	**593**	*orb02*	**888** [28]	**888** [28]	**888**	**888**
la06	**926** [19]	**926** [19]	**926**	**926**	*orb03*	**1005** [28]	**1005** [28]	**1005**	**1005**
la07	**890** [19]	**890** [19]	**890**	**890**	*orb04*	**1005** [28]	**1005** [28]	**1005**	**1005**
la08	**863** [19]	**863** [19]	**863**	**863**	*orb05*	**887** [28]	**887** [28]	**887**	**887**
la09	**951** [19]	**951** [19]	**951**	**951**	*orb06*	**1010** [93]	**1010** [93]	**1010**	**1010**
la10	**958** [19]	**958** [95]	**958**	**958**	*orb07*	**397** [93]	**397** [93]	**397**	**397**
la11	**1222** [19]	**1222** [19]	**1222**	**1222**	*orb08*	**899** [93]	**899** [93]	**899**	**899**
la12	**1039** [19]	**1039** [19]	**1039**	**1039**	*orb09*	**934** [93]	**934** [93]	**934**	**934**
la13	**1150** [19]	**1150** [19]	**1150**	**1150**	*orb10*	**944** [93]	**944** [93]	**944**	**944**
la14	**1292** [19]	**1292** [19]	**1292**	**1292**	*swv01*	**1407** [42]	**1407** [42]	**1407**	**1407**
la15	**1207** [19]	**1207** [19]	**1207**	**1207**	*swv02*	**1475** [88]	**1475** [88]	**1475**	**1475**
la16	**945** [91]	**945** [91]	**945**	**945**	*swv03*	**1398** [87]	**1398** [88]	1346	1417
la17	**784** [91]	**784** [95]	**784**	**784**	*swv04*	**1464** [87]	**1464** [87]	1405	1520
la18	**848** [28]	**848** [95]	**848**	**848**	*swv05*	**1424** [87]	**1424** [87]	1414	1464
la19	**842** [28]	**842** [95]	**842**	**842**	*swv06*	**1630** [87]	**1667** [96]	1572	1711
la20	**902** [28]	**902** [28]	**902**	**902**	*swv07*	**1513** [87]	**1595** [88]	1406	1690
la21	**1046** [93]	**1046** [93]	**1046**	**1046**	*swv08*	**1671** [87]	**1751** [41]	1614	1841
la22	**927** [28]	**927** [95]	**927**	**927**	*swv09*	**1633** [87]	**1655** [88]	1594	1698
la23	**1032** [19]	**1032** [19]	**1032**	**1032**	*swv10*	**1663** [87]	**1743** [88]	1588	1819
la24	**935** [28]	**935** [28]	**935**	**935**	*swv11*	**2983** [88]	**2983** [88]	2983	3033
la25	**977** [28]	**977** [28]	**977**	**977**	*swv12*	**2972** [96]	**2972** [96]	2972	3141
la26	**1218** [19]	**1218** [95]	**1218**	**1218**	*swv13*	**3104** [88]	**3104** [88]	3104	3191
la27	**1235** [19]	**1235** [97]	**1235**	**1235**	*swv14*	**2968** [88]	**2968** [88]	2968	3048
la28	**1216** [19]	**1216** [28]	**1216**	**1216**	*vw3* × *3*	**256** [83]	**256** [83]	**256**	**256**
la29	**1152** [86]	**1152** [86]	1119	1153	*yn04*	**929** [87]	**968** [88]	885	988

immediately converging to optimum, and in group 4, where ORT 4 cores is the only system able of finding optimal solutions, in just a few minutes.

Summing up, CPO is a very capable system, especially considering the results reached with just a single core. However, the additional computing power and search strategies brought to the table by the additional cores, did not improve the performance of the solver like in the Large-TA benchmark. ORT, on the contrary, suffers a lot in the single-core configuration, presenting results that could be considered poorly optimized even in a real scenario, where near-optimality is sufficient. However, ORT seems to benefit from the increased number of cores in this benchmark, being able to beat CPO in the instances of group 4, by a non-negligible margin (about 33% better than CPO). This group is of particular interest because it is the closest, in terms of size and job type, to the real semi-conductor scenario that inspired the creation of our benchmarks. However, given that the remarkable results of ORT are due to the easy calculation of the best lower bound in this particular benchmark, replicating the same performance might be harder on a real scenario, where the best makespan is not usually given.

Finally, to answer **RQ5** *(on the best overall CP solver)*, the best overall performer across the three different tests was CPO. The closest gap between the two solvers was in the classic benchmark test, where both solvers solved about 75% of the instances to optimality. The largest gap was on the large-TA benchmark test, where CPO was able to deal with instances up to 1 million operation, and ORT could not keep up. The known-optima benchmark results are still in favor of CPO, but ORT is able to keep up, and sometimes overtake, the CPO performance, at least when looking at the results of the quad-core systems.

5.4. Comparison with the state of the art

In Section 5.1 we conducted our experiments on a collection of benchmarks among the most studied in the literature. This allow us to easily compare the achieved results with the state of the art, in response to **RQ6** *(on the comparison with state of the art approaches)*. Table 9 collects all the currently discovered upper bounds (UB) and lower bounds (LB) of the makespan of the instances. When LB and UB coincide, it means that the optimal solution was found, and the instance is called *closed*. Conversely, instances where the bounds do not coincide are called *open*. The lower bound indicates that the optimal makespan cannot be lower than the bound value, and it is typically calculated by solving a simplified version of the problem at hand. The upper bound is calculated by finding a solution to the problem, proving that a solution with that makespan actually exists.

For example, consider instance *swv07*, with UB of 1595 [88]. It is possible that the solution with makespan 1595 is the optimal solution, but until the lower bound does not reach the same value, it is not possible to prove that the upper bound is not further improvable. Note that while with the upper bound "lower is better", with the lower bound is the opposite. In instance *abz8*, for example, the LB found by our system is 608, however the best LB has been found by [87] at 648.

Table 9 is organized in two columns, each showing LBs and UBs: The **State of the Art** column shows the best achieved bounds from works in the literature, with the corresponding reference to the paper where the bound was found. In the column representing our results, we show the bounds achieved by CPO/single-core, which proved to be the best performer in our classic benchmark test.

While the comparison it is not completely fair, considering that the computing resources and the solving time allowed vary across

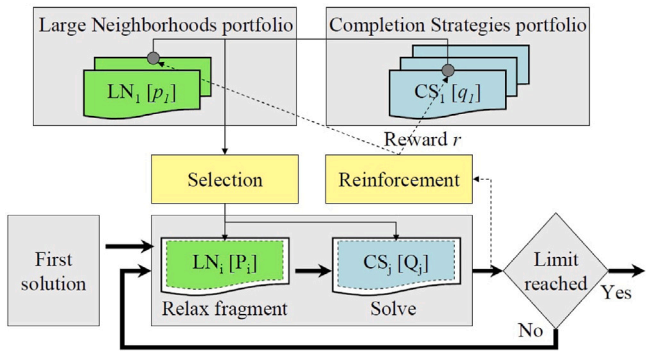

Fig. 10. Schema of underlying procedure of CP solvers (ORT and CPO) showing the iterative method based on large neighborhood search.
Source: Figure taken from [34].

the studies, it offers a good general indication of the value of the considered systems. CPO/single-core is able to achieve state of the art results in 58 out of 74 instances. Of the remaining 15 instances, 7 are still open (abz8, swv6, swv7, swv8, swv9, swv10, yn4). In these 15 instances, interestingly, the vast majority of lower bounds (and some upper bounds) where found by Vilim, Laborie and Shaw [87], the main developers of CP Optimizer at IBM. To achieve these results, they fine-tuned the solver and used a search procedure that is not available in the CPO public API (at least not in the version they used). This shows that CPO is indeed at pair with the state of the art in the most studied benchmarks. Considering the limited difference in performance among the tested systems on the classic benchmark (Section 5.1), we find that the conclusions one can draw from the comparison presented in Table 9 can be extended to the other systems as well.[9]

At the time of writing, there are no works by other authors in the literature that target the other two benchmark treated in this work, the known-optima benchmark and the large-TA. To allow an easy access for confrontation, we published the benchmarks and made them freely downloadable at [84].

5.5. Solvers' behavior

To explain the difference in performance, we analyzed the technical characteristics of the two solvers. Both approaches are relying on a similar search strategy based on large neighborhood search (LNS), which consists in an iterative relaxation and re-optimization of the scheduling problem, as depicted in Fig. 10 taken from [34]. The relaxation of a problem is done by resolving an easier version of the problem at end, and using the solution of the relaxed problem to improve the real solution. In scheduling, this is typically done by relieving a constraint (*e.g.* removing the precedence relation of the operations) or by considering just a portion of the instance, *e.g.* by solving the scheduling of just one machine at a time.

While CPO uses portfolio strategies in combination with machine learning to converge to the best neighborhoods and to decide which completion strategy to use, ORT uses a more simplistic approach based on random variables and constraint selection. Both use interval variables to express the job operations, but in CPO the interval variables are represented compactly using primitive types to express the bounds ($start_{min}$, $start_{max}$, end_{min}, end_{max}) and few other parameters to deal with optional intervals and no-fixed lengths (which are not used in our JSSP scenario). On the other hand, ORT instantiates two integer variables for start and end bounds in addition to the interval variable (for each operation). Therefore, ORT has effectively three times the number of variables of CPO, allegedly slowing down the propagation of constraints.

Concerning the multi-core performance of the two solvers, it is interesting to see that, in both solvers, the additional cores are not only used to spread the calculations, but also as a mean to try different search strategies on the problem at hand. In CPO, for example, the iterative diving technique[10] was used to quickly find a solution for the 1000 × 1000 instances, overcoming the solutions of the other systems by a wide margin. In ORT, the quad-core system was able to quickly find the optimal solution in the long job instances of the Known-Optima benchmark, using a clever alternative strategy that enforces the current lower bound as an hard constraint on the solution. In general, it seems like ORT benefits more from additional cores, compared to CPO. This can be attributed to the fact that CPO enforces determinism in the solution search process. This means that every time the solver is launched on a problem, the solutions are found always in the same order. This property, while beneficial in terms of debugging and reproducibility, becomes a problem when more than one core is involved in the search, since the parallelization is "disturbed" by the need to synchronize the search between the cores. CPO partly offsets this with exceptional performance in single core. In fact, while both solvers use different search strategies when more cores are used, CPO is also able to switch search strategy when running on a single core. An example of these "plan B" strategies is called failure directed search (FDS), which is triggered when the LNS is not able to improve the current solution [87].

6. Discussion

Like every research work, also our work possesses limitations, as well as topics that were not treated because regarded as out of scope.

First of all, for this research work we decided to concentrate solely on the JSSP. Albeit being a hard scheduling problem, there are several aspects of some industrial problems that are not covered by plain JSSP. In fact, real problems often incorporate complicated requirements (*e.g.* energy-related issues, variable-state machines, setup times, flexibility in the machines' selection, connected shift planning aspects, etc.) that can impact the ability of the solver to find solutions effectively, especially if such requirements cannot be represented by global constraints provided by the solver and they have to be encoded via primitive constraints.

Second, we performed our tests on a wide variety of instances, which allowed us to draw a good picture of the capabilities of the tested solvers. However, the instances used in the benchmarks were mostly generated using random procedures, while real problem instances would present an underlying structure emerging from the sequence of the production steps of actual products (*e.g.* following the line of work in [98]) and the given factory layout (*e.g.* [99]). It is possible that such instances would be actually easier to solve, thanks to more predictable production sequences that can be exploited during search.

Third, we did not investigate probabilistic and random influences, *e.g.* deviation of the processing times, randomness due to the human factors or unpredictable events like machine breakdowns. While relevant in practice, we felt that these aspects were out of the scope of this paper.

Another limitation concerns parameter tuning. In fact, we decided to stick with the default configurations for both solvers. We imagined a scenario where the practitioner could use the solver directly "out of the box", without the need of tuning the parameters for search. While it is almost certain that time invested in fine tuning the search parameters would lead to better results overall for the solvers, for the JSSP instances used in this work we performed a quick evaluation on a number of configurations and did not find significant improvements.

[9] The interested reader can find the complete results of all the other systems in the Appendix.

[10] See Section 5.2 for details.

However, for other scheduling problem types parameter tuning could be more impacting.

Finally, we report that the contribution of this work is an empirical evaluation of state-of-the-art CP solvers on industrial-size JSSPs, with no theoretical implications. Nevertheless, we think that this work provides very good insights that can be useful to researchers as well as to practitioners, even as a gentle introduction to constraint programming in the scheduling context.

7. Conclusion

In this work we present an evaluation of state-of-the-art constraint programming approaches for JSSP on various problem instances with a size that is representative of scenarios of nowadays industrial realities. In particular, we test the capabilities of two of the best constraint solvers available: Google's OR-Tools (ORT) and IBM's CP Optimizer (CPO). The benchmarks used for the evaluation include:

- a selection of classic problem instances from the scheduling literature;
- a benchmark based on Taillard's template, but with a vast range of instances from 100 to 1 million operations;
- a benchmark with large-scale instances with known optimal solutions inspired by the semi-conductor industry.

The novel large-scale instances that we provide[11] were generated using random procedures to create jobs and operations and to associate them to machines. In our tests we concentrate on the job shop scheduling problem, which is among the most studied scheduling problems in the field, and it is a hard problem in terms of complexity. To better simulate a real scheduling environment, we created new benchmark instances with sizes comparable with problem sizes found in nowadays industrial application fields like the semiconductor domain. In the extreme case this is up to a million operations to be scheduled on a thousand machines.

We test the solvers using the default configurations, evaluating both single and multi-core performance (quad-core). On the classic benchmark (comprising smaller well-known instances), both solvers perform well, finding optimal solutions on about three quarter of the instances in less than 20 min (often in a few seconds), matching the results of the best algorithms in literature. CPO is the best performer, but the performance gap is small across the systems.

The difference in performance becomes obvious when dealing with larger instances. In particular, CPO shows its capabilities on both the known-optima and the large-TA benchmarks: It is able to deal with all the instances of the large-TA, including the largest 1 million instances (at least with quad-core configuration), often finding (near-) optimal solutions. Furthermore, it finds optimal solutions for more than half of the instances of the known-optima benchmark. ORT, albeit not fully reaching the performance of CPO, benefits more from the multi-core configuration, allowing it to find solutions that are at least comparable with the ones of CPO (*e.g.* about 30% worst than the CPO ones on the large-TA benchmark in instances with 100 000 operations to be scheduled). Thus, while CPO is the first choice, also ORT shows extraordinary performance in the context of large-scale JSSP.

In conclusion, we show the capabilities of state-of-the-art approaches in general, and constraint programming approaches in particular, on realistic-size JSSP instances. The fact that constraint programming approaches are so effective is not only relevant for the dedicated literature, but for industry as well. In fact, the need for easy adaptability and cheap maintenance in the long run (typical requirements in industrial realities) are easily met by such approaches, thanks to compact problem formulations and logic-based encodings. Thus, we expect a rise in the quantity of applications of constraint programming in industrial scenarios in the upcoming future.

[11] Benchmarks and encodings available for download at [84].

CRediT authorship contribution statement

Giacomo Da Col: Methodology, Software, Validation, Investigation, Data curation, Writing – original draft, Visualization. **Erich C. Teppan:** Conceptualization, Software, Resources, Data curation, Writing – review & editing, Supervision, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.orp.2022.100249.

References

[1] Griffith SB. Sun Tzu: The art of war, Vol. 39. London: Oxford University Press; 1963.
[2] Hyatt C, Weaver P. A brief history of scheduling. Melbourne, Australia: Mosaic Project Services Pty Ltd; 2006.
[3] Clark W. The Gantt chart: A working tool of management. Ronald Press Company; 1922.
[4] Petersen PB. The evolution of the Gantt chart and its relevance today. J Manag Issues 1991;131–55.
[5] Muth J, Thompson G. Industrial scheduling. International series in management, Prentice-Hall; 1963.
[6] Blazewicz J, Ecker K, Pesch E, Schmidt G, Weglarz J. Handbook on scheduling: Models and methods for advanced planning (International handbooks on information systems). Secaucus, NJ, USA: Springer-Verlag New York, Inc.; 2007.
[7] Baker KR, Trietsch D. Principles of sequencing and scheduling. John Wiley & Sons; 2013.
[8] Lasi H, Fettke P, Kemper H-G, Feld T, Hoffmann M. Industry 4.0. Bus Inf Syst Eng 2014;6(4):239–42.
[9] Sparrow R, Howard M. When human beings are like drunk robots: Driverless vehicles, ethics, and the future of transport. Transp Res C 2017;80:206–15.
[10] Delfanti A, Frey B. Humanly extended automation or the future of work seen through Amazon patents. Sci Technol Human Values 2020;0162243920943665.
[11] Anonymous. Amazon to introduce more automated packaging machines. CBC 2019. URL https://www.cbc.ca/news/technology/amazon-machines-pack-orders-1.5134201.
[12] Fuchigami HY, Rangel S. A survey of case studies in production scheduling: Analysis and perspectives. J Comput Sci 2018;25:425–36.
[13] Johnson S. Optimal two- and three-stage production schedules with setup times included. Rand Corporation; 1953.
[14] Manne AS. On the job-shop scheduling problem. Oper Res 1960;8(2):219–23.
[15] Bowman EH. The schedule-sequencing problem. Oper Res 1959;7(5):621–4.
[16] Brucker P, Schlie R. Job-shop scheduling with multi-purpose machines. Computing 1990;45(4):369–75.
[17] Cheng T, Sin C. A state-of-the-art review of parallel-machine scheduling research. European J Oper Res 1990;47(3):271–92.
[18] Mokotoff E. Parallel machine scheduling problems: A survey. Asia-Pac J Oper Res 2001;18(2):193.
[19] Adams J, Balas E, Zawack D. The shifting bottleneck procedure for job shop scheduling. Manage Sci 1988;34(3):391–401.
[20] Nowicki E, Smutnicki C. A fast taboo search algorithm for the job shop problem. Manage Sci 1996;42(6):797–813.
[21] Garey MR, Johnson DS, Sethi R. The complexity of flowshop and jobshop scheduling. Math Oper Res 1976;1(2):117–29.
[22] Zheng D-Z, Wang L. An effective hybrid heuristic for flow shop scheduling. Int J Adv Manuf Technol 2003;21(1):38–44.
[23] Gonzalez T, Sahni S. Open shop scheduling to minimize finish time. J ACM 1976;23(4):665–79.
[24] Müller D, Müller MG, Kress D, Pesch E. An algorithm selection approach for the flexible job shop scheduling problem: Choosing constraint programming solvers through machine learning. European J Oper Res 2022. http://dx.doi.org/10.1016/j.ejor.2022.01.034.
[25] Kim S, Bobrowski P. Impact of sequence-dependent setup time on job shop scheduling performance. Int J Prod Res 1994;32(7):1503–20.
[26] Maccarthy BL, Liu J. Addressing the gap in scheduling research: a review of optimization and heuristic methods in production scheduling. Int J Prod Res 1993;31(1):59–79.

[27] Taillard E. Benchmarks for basic scheduling problems. European J Oper Res 1993;64(2):278–85. http://dx.doi.org/10.1016/0377-2217(93)90182-M.
[28] Applegate D, Cook W. A computational study of the job-shop scheduling problem. ORSA J Comput 1991;3(2):149–56. http://dx.doi.org/10.1287/ijoc.3.2.149.
[29] Falkner A, Friedrich G, Haselböck A, Schenner G, Schreiner H. Twenty-five years of successful application of constraint technologies at Siemens. AI Mag 2016;37(4):67–80.
[30] Balduccini M. Industrial-size scheduling with ASP+ CP. In: International conference on logic programming and nonmonotonic reasoning. Springer; 2011, p. 284–96.
[31] Falkner A, Friedrich G, Schekotihin K, Taupe R, Teppan EC. Industrial applications of answer set programming. KI-Künstl Intell 2018;32(2–3):165–76.
[32] Dal Palù A, Dovier A, Formisano A, Pontelli E. Exploring life: answer set programming in bioinformatics. In: Declarative Logic Programming: Theory, Systems, and Applications. Association for Computing Machinery and Morgan; Claypool; 2018, p. 359–412, https://doi.org/10.1145/3191315.3191323.
[33] Dal Palù A, Dovier A, Formisano A, Policriti A, Pontelli E. Logic programming applied to genome evolution in cancer. In: CILC. 2016, p. 148–57.
[34] Laborie P, Godard D. Self-adapting large neighborhood search: Application to single-mode scheduling problems. In: Proceedings MISTA-07, Paris, Vol. 8. 2007.
[35] Laborie P, Rogerie J. Temporal linear relaxation in IBM ILOG CP optimizer. J Sched 2016;19(4):391–400.
[36] Ku W-Y, Beck JC. Mixed integer programming models for job shop scheduling: A computational analysis. Comput Oper Res 2016;73:165–73.
[37] Da Col G, Teppan EC. Industrial size job shop scheduling tackled by present day cp solvers. In: International conference on principles and practice of constraint programming. Springer; 2019, p. 144–60.
[38] Da Col G, Teppan E. Google vs IBM: A constraint solving challenge on the job-shop scheduling problem. 2019, arXiv preprint arXiv:1909.08247.
[39] Xia W, Wu Z. An effective hybrid optimization approach for multi-objective flexible job-shop scheduling problems. Comput Ind Eng 2005;48(2):409–25.
[40] Fattahi P, Mehrabad MS, Jolai F. Mathematical modeling and heuristic approaches to flexible job shop scheduling problems. J Intell Manuf 2007;18(3):331–42.
[41] Cheng T, Peng B, Lü Z. A hybrid evolutionary algorithm to solve the job shop scheduling problem. Ann Oper Res 2013;1–15.
[42] Peng B, Lü Z, Cheng T. A tabu search/path relinking algorithm to solve the job shop scheduling problem. Comput Oper Res 2015;53:154–64.
[43] Zhang CY, Li P, Rao Y, Guan Z. A very fast TS/SA algorithm for the job shop scheduling problem. Comput Oper Res 2008;35(1):282–94.
[44] Danzig G. The Dantzig simplex method for linear programming. 1947.
[45] Gomory RE. Outline of an algorithm for integer solutions to linear programs. Bull Amer Math Soc 1958;64(5):275–8.
[46] Wagner HM. An integer linear-programming model for machine scheduling. Nav Res Logist Q 1959;6(2):131–40.
[47] Carlier J. The one-machine sequencing problem. European J Oper Res 1982;11(1):42–7.
[48] Carlier J, Pinson E. An algorithm for solving the job-shop problem. Manage Sci 1989;35(2):164–76.
[49] Brucker P, Jurisch B. A new lower bound for the job-shop scheduling problem. European J Oper Res 1993;64(2):156–67.
[50] Jaffar J, Lassez J-L. Constraint logic programming. In: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on principles of programming languages. 1987, p. 111–9.
[51] Rossi F, Van Beek P, Walsh T. Handbook of constraint programming. Elsevier; 2006.
[52] Fox MS, Allen BP, Strohm G. Job-shop scheduling: An investigation in constraint-directed reasoning. In: AAAI. 1982, p. 155–8.
[53] Keng N, Yun DY. A planning/scheduling methodology for the constrained resource problem. In: IJCAI. 1989, p. 998–1003.
[54] Fox MS, Sadeh N, Baykan C. Constrained heuristic search. In: Proceedings of the eleventh international joint conference on artificial intelligence. 1989, p. 309–15.
[55] Sycara K, Roth SF, Sadeh N, Fox MS. Distributed constrained heuristic search. IEEE Trans Syst Man Cybern 1991;21(6):1446–61.
[56] Sadeh NM, Fox MS. Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. Artificial Intelligence 1996;86:1–41.
[57] Gelfond M, Lifschitz V. The stable model semantics for logic programming. In: ICLP/SLP, Vol. 88. 1988, p. 1070–80.
[58] Lifschitz V. Answer set programming. Springer; 2019.
[59] Dovier A, Formisano A, Pontelli E. A comparison of CLP (FD) and ASP solutions to NP-complete problems. In: International conference on logic programming. Springer; 2005, p. 67–82.
[60] Brewka G, Eiter T, Truszczyński M. Answer set programming at a glance. Commun ACM 2011;54(12):92–103.
[61] Da Col G, Teppan EC. Declarative decomposition and dispatching for large-scale job-shop scheduling. In: Joint German/Austrian conference on artificial intelligence (Künstliche intelligenz). Springer; 2016, p. 134–40.
[62] Panwalkar SS, Iskander W. A survey of scheduling rules. Oper Res 1977;25(1):45–61.
[63] Haupt R. A survey of priority rule-based scheduling. Oper-Res-Spektrum 1989;11(1):3–16.
[64] Jun S, Lee S, Chun H. Learning dispatching rules using random forest in flexible job shop scheduling problems. Int J Prod Res 2019;57(10):3290–310.
[65] Teppan EC. Dispatching rules revisited-a large scale job shop scheduling experiment. In: 2018 IEEE symposium series on computational intelligence (SSCI). IEEE; 2018, p. 561–8.
[66] Teppan E, Da Col G. Automatic generation of dispatching rules for large job shops by means of genetic algorithms. In: CIMA@ ICTAI. 2018, p. 43–57.
[67] Vela A, Cruz-Duarte JM, Ortiz-Bayliss JC, Amaya I. Beyond hyper-heuristics: A squared hyper-heuristic model for solving job shop scheduling problems. IEEE Access 2022;10:43981–4007.
[68] Balas E, Lenstra JK, Vazacopoulos A. The one-machine problem with delayed precedence constraints and its use in job shop scheduling. Manage Sci 1995;41(1):94–109.
[69] Taillard ED. Parallel taboo search techniques for the job shop scheduling problem. ORSA J Comput 1994;6(2):108–17. http://dx.doi.org/10.1287/ijoc.6.2.108.
[70] Dell'Amico M, Trubian M. Applying tabu search to the job-shop scheduling problem. Ann Oper Res 1993;41(3):231–52.
[71] Nowicki E, Smutnicki C. An advanced tabu search algorithm for the job shop problem. J Sched 2005;8(2):145–59.
[72] Lawrence S. Resource constrained project scheduling: An experimental investigation of heuristic scheduling techniques (supplement). Graduate School of Industrial Administration, Carnegie-Mellon University; 1984.
[73] Storer RH, Wu SD, Vaccari R. New search spaces for sequencing problems with application to job shop scheduling. Manage Sci 1992;38(10):1495–509.
[74] Yamada T, Nakano R. A genetic algorithm applicable to large-scale job-shop problems. In: PPSN. 1992, p. 283–92.
[75] Demirkol E, Mehta S, Uzsoy R. Benchmarks for shop scheduling problems. European J Oper Res 1998;109(1):137–41. http://dx.doi.org/10.1016/S0377-2217(97)00019-2.
[76] Lackner M-L, Mrkvicka C, Musliu N, Walkiewicz D, Winter F. Minimizing cumulative batch processing time for an industrial oven scheduling problem. In: 27th international conference on principles and practice of constraint programming (CP 2021). Schloss Dagstuhl-Leibniz-Zentrum für Informatik; 2021.
[77] Zhai Y, Liu C, Chu W, Guo R, Liu C. A decomposition heuristics based on multi-bottleneck machines for large-scale job shop scheduling problems. J Ind Eng Manage (JIEM) 2014;7(5):1397–414.
[78] Laborie P, Rogerie J, Shaw P, Vilím P. IBM ILOG CP optimizer for scheduling. Constraints 2018;23(2):210–50.
[79] Zhang R, Wu C. A hybrid approach to large-scale job shop scheduling. Appl Intell 2010;32(1):47–59.
[80] Friedrich G, Frühstück M, Mersheeva V, Ryabokon A, Sander M, Starzacher A, Teppan E. Representing production scheduling with constraint answer set programming. In: Operations research proceedings 2014. Springer; 2016, p. 159–65.
[81] Schutt A, Feydy T, Stuckey PJ, Wallace MG. Explaining the cumulative propagator. Constraints 2011;16(3):250–82.
[82] Nethercote N, Stuckey PJ, Becket R, Brand S, Duck GJ, Tack G. MiniZinc: Towards a standard CP modelling language. In: International conference on principles and practice of constraint programming. Springer; 2007, p. 529–43.
[83] Vazquez M, Whitley LD. A comparison of genetic algorithms for the dynamic job shop scheduling problem. In: 2nd annual conference on genetic and evolutionary computation. Morgan Kaufmann Publishers Inc.; 2000, p. 1011–8.
[84] Da Col G, Teppan E. Large-scale benchmarks for the job shop scheduling problem. 2021, arXiv preprint arXiv:2102.08778.
[85] Emerson P. The original Borda count and partial voting. Soc Choice Welf 2013;40(2):353–8.
[86] Martin PD. A time-oriented approach to computing optimal schedules for the job-shop scheduling problem. Cornell University; 1996.
[87] Vilím P, Laborie P, Shaw P. Failure-directed search for constraint-based scheduling. 2015, http://dx.doi.org/10.1007/978-3-319-18008-3_30.
[88] Shylo O, Shams H. Boosting binary optimization via binary classification: A case study of job shop scheduling. 2018.
[89] Florian M, Trepant P, McMahon G. An implicit enumeration algorithm for the machine sequencing problem. Manage Sci 1971;17(12):B–782.
[90] Van Laarhoven PJ, Aarts EH, Lenstra JK. Job shop scheduling by simulated annealing. Oper Res 1992;40(1):113–25.
[91] Carlier J, Pinson E. A practical use of Jackson's preemptive schedule for solving the job shop problem. Ann Oper Res 1990;26(1):269–87.
[92] McMahon G, Florian M. On scheduling with ready times and due dates to minimize maximum lateness. Oper Res 1975;23(3):475–82.
[93] Vaessens RJM, Aarts EH, Lenstra JK. Job shop scheduling by local search. Informs J Comput 1996;8(3):302–17.
[94] Nowicki E, Smutnicki C. A fast tabu search algorithm for the permutation flow-shop problem. European J Oper Res 1996;91(1):160–75.
[95] Matsuo H. A controlled search simulated annealing method for the general jobshop scheduling problem. Technical Report Working Paper 03-04-88, U. Texas at Austin; 1988.

[96] Constantino O, Segura C. A parallel memetic algorithm with explicit management of diversity for the job shop scheduling problem. Appl Intell 2022;141–53. http://dx.doi.org/10.1007/s10489-021-02406-2.

[97] Carlier J, Pinson E. Adjustment of heads and tails for the job-shop problem. European J Oper Res 1994;78(2):146–61.

[98] Vela A, Cruz-Duarte JM, carlos Ortiz-Bayliss J, Amaya I. Tailoring job shop scheduling problem instances through unified particle swarm optimization. IEEE Access 2021;9:66891–914.

[99] Teppan EC. Types of flexible job shop scheduling: A constraint programming experiment. In: 14th int. conf. on agents and artificial intelligence (ICAART 2022), Vol. 3. 2022, p. 516–23.