Industrial Size Job Shop Scheduling Tackled by Present Day CP Solvers

Giacomo Da Col and Erich C. Teppan(✉)

Alpen-Adria Universität Klagenfurt, 9020 Klagenfurt, Austria
{giacomo.da,erich.teppan}@aau.at

Abstract. The job shop scheduling problem (JSSP) is an abstraction of industrial scheduling and has been studied since the dawn of the computer era. Its combinatorial nature makes it easily expressible as a constraint satisfaction problem. Nevertheless, in the last decade, there has been a hiatus in the research on this topic from the constraint community; even when this problem is addressed, the target instances are from benchmarks that are more than 20 years old. And yet, constraint solvers have continued to evolve and the standards of today's industry have drastically changed. Our aim is to close this research gap by testing the capabilities of the best available CP solvers on the JSSP. We target not only the classic benchmarks from the literature but also a new benchmark of large-scale instances reflecting nowadays industrial scenarios. Furthermore, we analyze different encodings of the JSSP to measure the impact of high-level structures (such as interval variables and no-overlap constraints) on the problem solution. The solvers considered are OR-Tools, Google's open-source solver and winner of the last MiniZinc Challenge, and IBM's CP Optimizer, a proprietary solver targeted towards industrial scheduling problems.

Keywords: Constraint programming · Job shop scheduling · JSSP · OR-Tools · CP Optimizer · Large-scale benchmark

1 Introduction

The job shop scheduling problem (JSSP) is among the first combinatorial problems ever studied [11]. Due to its relevant application to, both, computer and manufacturing systems, it is one of the most studied and analyzed.

The problem is presented as a set of jobs that must be processed by a set of machines. In the classical formulation, every job has to go through each machine exactly once. The processing of a job by one machine is called operation and the processing time is called duration. Every job has a specific ordering of operations that must be respected. An admissible solution for an instance of this problem is a sequence of operations on every machine where there is no time overlap between two operations in the same machine and the ordering of the operations is respected. The most typical optimization criterium is the minimization of the makespan, *i.e.* the time interval between the start of the first operation and the end of the last.

The structure of the problem makes it easily representable as a constraint satisfaction problem. In fact, there have been many successful applications of constraint-based approaches to this problem over the past years, *e.g.* [2,5,14].

Among the various CP techniques, a particularly successful one is Large Neighborhood Search (LNS) [7]. This method consists of an iterative process of relaxation and re-optimization of the problem, progressively selecting the most promising partial schedules to improve the final solution. This idea was also applied to MIP approaches (in the form of Relaxation Induced Neighborhood Search [4]). Hybrid CP-MIP methods have been considered as well, in order to take the best of the two worlds [13], and in some cases it has been shown that hybrid approaches perform better than MIP alone [6].

Despite these advancements in constraint solving, the last decade has experienced a decrease of research interest of CP applied to the classic JSSP. The major contributor in this period was IBM, which stole the scene with their proprietary CP Solver, CP Optimizer. This solver has been capable of finding better solutions for many JSSP instances from the classic benchmarks [19]. They also use a hybrid CP-MIP approach in case of non-regular objective functions, as in scheduling problems with earliness costs [8].

This evolution of solving capabilities, however, does not correspond to an evolution of benchmarks instances, which are almost the same as twenty years ago. In the meantime, industrial standards have changed to the point that scheduling problems from modern manufacturing systems can easily require up to 2000 jobs to be scheduled on 100 machines [3,17]. In comparison, the biggest instance of the Taillard benchmark [16], which reflected real dimensions of industrial problems in 1993 and it is still among the largest available, has 50 jobs on 20 machines. To close this gap, we created a new benchmark of JSSP instances, based on Taillard's specification, but in line with today's industrial scenarios.

The aim of this paper is twofold. First, we want to close the gap on the JSSP research, testing the capabilities of the best available CP solvers on, both, classic benchmarks and our large-scale one. Second, we want to investigate how different design choices affect the search process. In fact, while CP allows a compact representation, it still offers multiple ways to encode a problem. In particular, we focus on the application of high-level structures like interval variables and no-overlap global constraints.

As anticipated, one of the most successful CP solvers on scheduling problems is CP Optimizer (abbreviated CPO). To find a worthy opponent, we took the winner of the last years MiniZinc challenge[1]. The MiniZinc challenge is a recurring competition where all the constraint solvers that support the MiniZinc modeling language [12] compete on various combinatorial problems, including scheduling. OR-Tools[2] (ORT), an open-source solver developed by Google, won the gold medal in all categories in 2018. While preliminary studies done on

[1] https://www.minizinc.org/challenge2018/challenge.html.
[2] https://developers.google.com/optimization/.

global constraints suggest the advantage of CP Optimizer over OR-Tools [9], a direct comparison of these two solvers on the JSSP has never been conducted, especially on industrial-size instances.

2 Experimental Setup

We implement three encodings for the JSSP to measure the effectiveness of high-level constraint structures:

- A Naive encoding, which uses primitive CP constraints and integer variables;
- A SemiNaive encoding, which takes advantage of the interval variables, particularly well suited to represent operations in JSSP;
- An Advanced encoding, which combines the interval variables with global constraints designed for scheduling problems, such as the no-overlap constraint.

The comparison is carried out measuring the quality of the final solution (makespan) and the time needed to reach that solution. Each solver is tested with all three encodings on all benchmark instances. The search procedure runs on a single core.

On the classic instances, the time allowed for each instance is 20 min, instantiation time included; this time limit complies with the MiniZinc challenge rules.

On the large-scale instances, the time allowed is 6 h, instantiation time included. The time limit is extended because the size of large-scale instances demands more time to have a thorough exploration of the search space. Nevertheless, given that these instances were generated with industrial scenarios in mind, we selected a time limit which would allow the calculation to be completed overnight, to minimize gaps in the production flow. Concerning the solvers' version, we use version 12.8.0 for CP Optimizer and version 6.10.6025 for OR-Tools. Since CPO does not support MiniZinc as modeling language, but both solvers offer Java APIs, we decided to use Java to interface with the solvers, to avoid the bias that different modeling languages might introduce. We conducted a short preliminary experiment to test the various solver configurations. Concerning CPO, we found the default configuration to be the most performant. In ORT, we tested both the classic CP Solver and the new CP-SAT Solver, finding the latter to be better than the former in most cases.

The experiment is conducted on a system equipped with a 2 GHz AMD EPYC 7551P 32 Cores CPU and 128 GB of RAM. Each run was performed on a single core with a maximum cap of 10 GB of RAM.

2.1 Problem Instances

Our tests on the various models are conducted on the classic benchmark and the large-scale benchmark. All the instances of both benchmarks are rectangular JSSP instances. This means that every job has to go through all the machines, therefore every job will have a number of operations equal to the total number of machines and every machine will have assigned a number of operations equal to the total number of jobs. The classic benchmark consists of a selection of problem instances and comprises the most used JSSP benchmarks in the literature. In particular, we used the same problem instances selected for the MiniZinc benchmark[3]:

- **FT:** This is one of the oldest benchmarks for JSSP [11]. It includes 3 problem instances of sizes 6×6, 10×10 and 20×5. The square instance 10×10 is famous for remaining unsolved for more than 20 years.
- **LA:** This benchmark contains 40 problem instances from 10×5 to 30×10 [10].
- **ABZ:** 5 problem instances from the work about shifting bottleneck by [1].
- **ORB:** 10 problem instances proposed by [2].
- **YN:** 1 randomly generated problem instance of size 20×20 [20].
- **SWV:** A set of 14 problem instances from [15].
- **VW:** 1 instance from [18].

The large-scale benchmark consists of 90 instances that we generated following the Taillard benchmark specification [16]. We generated our own benchmark instead of using Taillard's because our aim is to verify the performance of the two solvers on problem instances with a size comparable with modern industrial problems[4]. The benchmark is structured with instances of increasing sizes, as follows:

- 10 instances 10×10 (*i.e.* 10 jobs to be scheduled on 10 machines, for a total of 100 operations)
- 10 instances 10×100 (*i.e.* 10 jobs to be scheduled on 100 machines, for a total of 1000 operations)
- 10 instances 10×1000 (10000 operations)
- 10 instances 100×10 (1000 operations)
- 10 instances 100×100 (10000 operations)
- 10 instances 100×1000 (100000 operations)
- 10 instances 1000×10 (10000 operations)
- 10 instances 1000×100 (100000 operations)
- 10 instances 1000×1000 (1000000 operations)

2.2 Encodings

One of the main advantages of adopting a constraint programming approach is that it allows a compact and formal definition of the problem, which is easily maintainable and adaptable to sudden changes in the problem configuration. Nevertheless, various decisions can be made during the modeling process to affect the search phase. In particular, we are interested in the impact of high-level structures and constraints on the solving capabilities of solvers, *i.e.* :

[3] https://github.com/MiniZinc/minizinc-benchmarks/tree/master/jobshop.
[4] complete encodings and benchmarks are available at https://goo.gl/qarP3m.

- Interval variable: a special type of variable well suited to represent job operations in scheduling. This variable incorporates a start time, an end time and a duration, and automatically enforces a duration constraint, such that $start + duration = end$;
- No overlap constraint: given a sequence of interval variables $v_1 \dots v_n$, if v_i starts before v_j, then v_j cannot start before the end of v_i (for every variable index $i, j \in \{1 \dots n\} | \; i \neq j$). This constraint is designed to work with interval variables, and cannot be used without them.

```
INPUT
opDurations : IntegerArray[1..numJobs][1..numMachines]
opSuccessors : IntegerArray[1..numJobs][1..numMachines]

VARIABLES
opStarts : IntegerVariableArray[1..numJobs][1..numMachines]
opEnds  : IntegerVariableArray[1..numJobs][1..numMachines]

CONSTRAINTS
opStarts[j][m] + opDurations[j][m] = opEnds[j][m],
$\forall j \in \{1, \dots, numJobs\}, \forall m \in \{1, \dots, numMachines\}$

opEnds[j][m] $\leq$ opStarts[j][opSuccessors[j][m]],
$\forall j \in \{1, \dots, numJobs\}, \forall m \in \{1, \dots, numMachines\}$
with $opSuccessors[j][m] \neq NULL$

opEnds[j][m] $\leq$ opStarts[k][m] $\vee$ opEnds[k][m] $\leq$ opStarts[j][m],
$\forall j \in \{1, \dots, numJobs\}, \forall k \in \{1, \dots, numJobs\}, \forall m \in \{1, \dots, numMachines\}$
with $j \neq k$

OBJECTIVE
minimize max($\{end | end \in opEnds\}$)
```

Encoding 1. Naive encoding for the job shop scheduling problem

Based on these constructs, we created three encodings for the JSSP:

The **Naive** encoding does not take advantage of any of the specialized structures, and relies on integer variables and primitive constraints to model the JSSP. Encoding 1 shows the model of the naive encoding. Keeping in mind that all the adopted instances are rectangular, we use a matrix structure with a number of rows equal to the number of jobs and a number of columns equal to the number of machines of an instance. In the input data there are two such matrices, one to store the durations and one for the succession of the operations of each job on the machines. In the model, *opStarts* and *opEnds* are matrices of integer variables that store respectively the start and end variables of each operation. The first constraint imposes that for each operation of each job, $start + duration = end$. The second constraint enforces the precedence relation between operations within a job (as long as a certain operation has a successor). The third constraint ensures that on each machine, for every couple of operations o_j and o_k (with j different from k), either o_j comes before than o_k or viceversa, without any overlap. The objective function aims to minimize the largest of the *opEnds* (*i.e.* the makespan).

```
INPUT
opDurations : IntegerArray[1..numJobs][1..numMachines]
opSuccessors : IntegerArray[1..numJobs][1..numMachines]

VARIABLES
ops : IntervalVariableArray[1..numJobs][1..numMachines]
with $ops[j][m].duration = opDurations[j][m]$
$\forall j \in \{1, \ldots, numJobs\}, \forall m \in \{1, \ldots, numMachines\}$

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

ops[j][m].end $\leq$ ops[k][m].start $\vee$ ops[k][m].end $\leq$ ops[j][m].start,
$\forall j \in \{1, \ldots, numJobs\}, \forall k \in \{1, \ldots, numJobs\}, \forall m \in \{1, \ldots, numMachines\}$
with $j \neq k$

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

Encoding 2. SemiNaive encoding for the job shop scheduling problem

The **SemiNaive** encoding makes use of interval variables to encode job operations. The model described in Encoding 2 is more compact compared to the Naive encoding. In fact, the interval variables already contain the information about start and end of each operation, as well as the duration constraints. Therefore, the only constraints that have to be explicitly expressed are the precedence constraints and the no-overlap constraints, done by manually instantiating a disjunctive constraint for every couple of operations in the machine, as in the Naive encoding.

The **Advanced** encoding (Encoding 3) also exploits interval variables, like the SemiNaive encoding, but instead of a quadratic number of primitive constraints per machine, one no overlap global constraint is used. Given that this procedure condenses two iterations over the number of jobs, a wise implementation of such procedure can lead to a great speed-up of both the instantiation and the solving process.

```
INPUT
opDurations : IntegerArray[1..numJobs][1..numMachines]
opSuccessors : IntegerArray[1..numJobs][1..numMachines]

VARIABLES
ops : IntervalVariableArray[1..numJobs][1..numMachines]
with $ops[j][m].duration = opDurations[j][m]$
$\forall j \in \{1,\ldots,numJobs\}, \forall m \in \{1,\ldots,numMachines\}$

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

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

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

Encoding 3. Advanced encoding for the job shop scheduling problem

3 Results

This section illustrates the results of the experiment carried out on the classic benchmark and on the large-scale benchmark. The tests are conducted on pairs of solver-encoding configurations, *e.g.* OR-Tools solver using Naive encoding, CP Optimizer solver using Advanced encoding, and so on. For the sake of synthesis, from now on we will refer to any solver-encoding configuration simply as a system.

3.1 Results of Classic Benchmark

The results of the classic benchmark are summarized in Table 1. In the makespan columns, the listed values correspond to the best solution achieved after 20 min of computation (or earlier, if the optimal solution is detected). The best solutions found are highlighted in bold. The last column indicates whether the best solution found is optimal or not. In the solving time columns, the time is measured in seconds. The best values are highlighted in bold.

Table 1. Results of the experiment on the classic benchmarks.

Problem	**Makespan**						**Solving Time**						
Instances	**CP Optimizer**			**OR-Tools**			**CP Optimizer**			**OR-Tools**			
	naive	seminaive	advanced	naive	seminaive	advanced	naive	seminaive	advanced	naive	seminaive	advanced	optimal
abz5	1238	**1234**	**1234**	**1234**	**1234**	**1234**	1200	6.32	2.19	2.03	2.01	**1.75**	Yes
abz6	**943**	**943**	**943**	**943**	**943**	**943**	1200	1.25	0.68	**0.43**	1.19	0.7	Yes
abz7	685	700	**656**	689	671	660	1200	1200	**1170.24**	1200	1200	1200	Yes
abz8	700	705	682	706	685	**679**	1200	1200	1200	1200	1200	1200	No
abz9	728	739	**685**	703	687	695	1200	1200	1200	1200	1200	1200	No
ft06	**55**	**55**	**55**	**55**	**55**	**55**	0.04	0.04	**0**	0.03	0.02	0.01	Yes
ft10	**930**	**930**	**930**	**930**	**930**	**930**	1200	22.81	**3.76**	5.42	4.4	4.85	Yes
ft20	1196	1197	**1165**	1206	1174	**1165**	1200	1200	**1.36**	1200	1200	4.88	Yes
la01	**666**	**666**	**666**	**666**	**666**	**666**	1200	0.66	**0**	0.25	0.25	0.08	Yes
la02	**655**	**655**	**655**	**655**	**655**	**655**	1200	2.25	0.3	0.51	0.46	**0.04**	Yes
la03	**597**	**597**	**597**	**597**	**597**	**597**	816.38	2.14	0.07	0.53	0.52	**0.06**	Yes
la04	**590**	**590**	**590**	**590**	**590**	**590**	1200	1.27	0.34	0.19	0.35	**0.18**	Yes
la05	**593**	**593**	**593**	**593**	**593**	**593**	1200	1.5	**0**	0.76	0.64	0.02	Yes
la06	**926**	**926**	**926**	**926**	**926**	**926**	1200	1200	**0**	1200	1200	1.04	Yes
la07	**890**	**890**	**890**	**890**	**890**	**890**	1200	1200	**0.02**	1200	1200	0.11	Yes
la08	**863**	**863**	**863**	**863**	**863**	**863**	1200	1200	**0.02**	1200	1200	0.26	Yes
la09	**951**	**951**	**951**	**951**	**951**	**951**	1200	1200	**0**	1200	1200	0.48	Yes
la10	**958**	**958**	**958**	**958**	**958**	**958**	1200	1200	**0**	1200	1200	0.87	Yes
la11	**1222**	**1222**	**1222**	**1222**	**1222**	**1222**	1200	1200	**0.01**	1200	1200	0.7	Yes
la12	**1039**	**1039**	**1039**	**1039**	**1039**	**1039**	1200	1200	**0.14**	1200	1200	0.64	Yes
la13	**1150**	**1150**	**1150**	**1150**	**1150**	**1150**	1200	1200	**0.02**	1200	1200	3.02	Yes
la14	**1292**	**1292**	**1292**	**1292**	**1292**	**1292**	1200	1200	**0.01**	1200	1200	1.92	Yes
la15	**1207**	**1207**	**1207**	**1207**	**1207**	**1207**	1200	1200	**0.14**	1200	1200	5.61	Yes
la16	**945**	**945**	**945**	**945**	**945**	**945**	1200	2.38	1.45	0.73	0.95	**0.6**	Yes
la17	**784**	**784**	**784**	**784**	**784**	**784**	1200	1.71	1.12	0.54	1.7	**0.32**	Yes
la18	**848**	**848**	**848**	**848**	**848**	**848**	93.2	1.67	0.9	0.55	1.85	**0.88**	Yes
la19	**842**	**842**	**842**	**842**	**842**	**842**	1200	3.83	2.93	0.93	1.44	**1.65**	Yes
la20	**902**	**902**	**902**	**902**	**902**	**902**	1200	2.03	1.6	0.55	2.49	**0.7**	Yes
la21	1059	1051	**1046**	1048	**1046**	**1046**	1200	1200	**22.56**	1200	534.25	83.03	Yes
la22	932	927	**927**	**927**	**927**	**927**	1200	173.47	**5.28**	103.93	80.29	6.62	Yes
la23	**1032**	**1032**	1032	**1032**	**1032**	**1032**	1200	303.5	**0.12**	118.82	257.44	2.8	Yes
la24	969	**935**	**935**	**935**	**935**	**935**	1200	314.42	**15.42**	113.99	47.07	24.5	Yes
la25	982	**977**	**977**	**977**	**977**	**977**	1200	218.82	**14.63**	60.28	47.45	18.82	Yes
la26	1218	1246	**1218**	1237	**1218**	**1218**	1200	1200	**7.35**	1200	1200	78.47	Yes
la27	1293	1275	**1235**	1289	1266	**1235**	1200	1200	**129.38**	1200	1200	509.21	Yes
la28	1297	1248	**1216**	1250	1224	**1216**	1200	1200	17.53	1200	1200	**13.97**	Yes
la29	1198	1226	**1152**	1236	1191	1153	1200	1200	1200	1200	1200	1200	No
la30	1426	**1355**	**1355**	**1355**	**1355**	**1355**	1200	1200	**0.28**	1200	1200	20.79	Yes
la31	1791	**1784**	**1784**	1841	**1784**	**1784**	1200	1200	**0.44**	1200	1200	23.74	Yes
la32	1853	**1850**	**1850**	1882	**1850**	**1850**	1200	1200	**0.04**	1200	1200	29.09	Yes
la33	1747	**1719**	**1719**	1746	**1719**	**1719**	1200	1200	**0.25**	1200	1200	14.16	Yes
la34	1793	**1721**	**1721**	1781	1746	**1721**	1200	1200	**1.57**	1200	1200	69.39	Yes
la35	**1888**	1898	**1888**	1922	**1888**	**1888**	1200	1200	**0.24**	1200	1200	25.14	Yes
la36	1281	**1268**	**1268**	**1268**	**1268**	**1268**	1200	108.6	**10.37**	32.34	40.16	10.79	Yes
la37	1399	**1397**	**1397**	**1397**	**1397**	**1397**	1200	330.43	**4.03**	179.52	158.46	8.5	Yes
la38	1202	**1196**	**1196**	**1196**	**1196**	**1196**	1200	856.21	**84**	462.06	135.24	260.72	Yes
la39	1248	**1233**	**1233**	**1233**	**1233**	**1233**	1200	110	**5.93**	51.89	40.1	13.81	Yes
la40	1240	**1222**	**1222**	**1222**	**1222**	**1222**	1200	707.27	**9.87**	396.79	187.83	52.46	Yes
orb01	1079	**1059**	**1059**	**1059**	**1059**	**1059**	1200	167.49	**7.09**	110.33	38.87	22.69	Yes
orb02	**888**	**888**	**888**	**888**	**888**	**888**	1200	3.6	2.23	**0.92**	1.56	1.9	Yes
orb03	**1005**	**1005**	**1005**	**1005**	**1005**	**1005**	1200	48.16	**6.54**	56.56	23.45	20.6	Yes
orb04	1011	**1005**	**1005**	**1005**	**1005**	**1005**	1200	5.03	2.73	2.99	**1.96**	2.86	Yes
orb05	**887**	**887**	**887**	**887**	**887**	**887**	1200	7.98	3.62	**1.81**	2.71	2.4	Yes
orb06	1023	**1010**	**1010**	**1010**	**1010**	**1010**	1200	41.35	**4.64**	15.61	10.04	8.59	Yes
orb07	**397**	**397**	**397**	**397**	**397**	**397**	1200	4.77	1.57	1.53	**1.25**	1.33	Yes
orb08	**899**	**899**	**899**	**899**	**899**	**899**	1200	9.78	**1.09**	2.45	2.29	1.36	Yes
orb09	**934**	**934**	**934**	**934**	**934**	**934**	1200	8.48	**1.19**	2.36	3.92	1.38	Yes
orb10	**944**	**944**	**944**	**944**	**944**	**944**	1200	4.68	**0.72**	1.57	2.6	1.8	Yes
swv01	1517	1544	1445	1541	1483	**1412**	1200	1200	1200	1200	1200	1200	No
swv02	1620	1590	1491	1571	1502	**1475**	1200	1200	1200	1200	1200	**901.91**	Yes
swv03	1491	1582	1420	1547	1493	**1410**	1200	1200	1200	1200	1200	1200	No
swv04	1568	1667	1520	1597	1553	**1482**	1200	1200	1200	1200	1200	1200	No
swv05	1537	1618	**1424**	1559	1518	1436	1200	1200	**1134.75**	1200	1200	1200	Yes
swv06	1947	1886	**1728**	1841	1784	1746	1200	1200	1200	1200	1200	1200	No
swv07	1699	1816	**1672**	1759	1711	1677	1200	1200	1200	1200	1200	1200	No
swv08	2072	2063	**1785**	1920	1881	1855	1200	1200	1200	1200	1200	1200	No
swv09	1942	1841	**1713**	1825	1779	1715	1200	1200	1200	1200	1200	1200	No
swv10	1953	1974	1823	1920	1872	**1807**	1200	1200	1200	1200	1200	1200	No
swv11	3472	3468	**3041**	3815	3696	3317	1200	1200	1200	1200	1200	1200	No
swv12	3447	3282	**3114**	3753	3794	3358	1200	1200	1200	1200	1200	1200	No
swv13	3616	3477	**3205**	3934	3938	3421	1200	1200	1200	1200	1200	1200	No
swv14	3474	3293	**3032**	3832	3807	3162	1200	1200	1200	1200	1200	1200	No
vw3x3	**256**	**256**	**256**	**256**	**256**	**256**	**0**	**0**	**0**	**0**	**0**	**0**	Yes
yn4	1048	1054	**980**	1030	1005	994	1200	1200	1200	1200	1200	1200	No

While perfect for a complete view on the classic benchmark experiment, Table 1 falls short when it comes to direct comparisons of systems. Figure 1 offers a more eye-friendly summary of the experiment. Every cell of the map represents the *difference* between number of instances where the row system beats the column system *and* number of instances where the row system is beaten by the column one. In this case, **to beat** means to achieve a makespan **strictly** lower or, in case of a draw, to achieve the same makespan faster. in case of a draw in both makespan and solving time (*e.g.* instance $\mathbf{vw3} \times \mathbf{3}$), the corresponding instance is not counted. The scale goes from −74 (the corresponding row system is beaten in all the instances by the column one) to 74 (the system beats the corresponding column one in every instance). Hence, the map has to be read row-wise, and the "greener" the row, the better the corresponding system.

It is noticeable that the greenest of the rows corresponds to CPO Advanced, which is better than the other CPO systems in almost all the instances, and scores a positive 35 against ORT Advanced, winning on 53 instances and losing on 18. ORT Advanced also achieves good performance against the Naive and SemiNaive encodings, going "in the red" only against CPO Advanced. The worst performer is unarguably CPO Naive, which is defeat on almost all the instances by the Advanced encodings and on more than 50% of the instances by the other systems. In general, CPO performs better than ORT with the Advanced encoding, but poorly with the other encodings. If we consider the percentage of instances solved optimally, the difference between the two solvers is even narrower on the Advanced encodings, with CPO solving optimally 77.03% of the instances and ORT following at 75.68%. In both solvers, there is an improvement from Naive to SemiNaive encoding (better results on about 60% of the instances), but the improvement is even larger from SemiNaive to Advanced (better on 89% of instances for ORT and 99% for CPO).

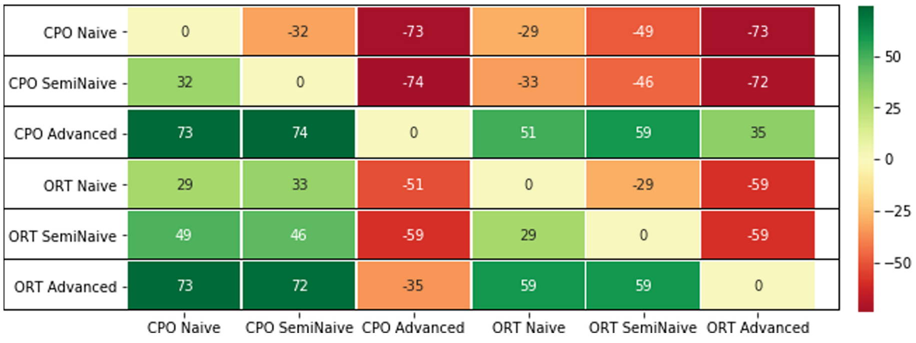

Fig. 1. Map on the pairwise confrontation between systems on the classic benchmarks. To be read row-wise, the greener the better. (Color figure online)

3.2 Results of the Large-Scale Benchmark

The results of the large-scale benchmark are summarized in Table 2. The results are grouped by instance size. Given that there are 10 instances per group, every value represents the mean of the final makespans of the corresponding group. In the first two groups, every system is able to solve all the instances optimally. From the **10 × 1000** group on, we do not refer anymore to optimal solutions, because it is not possible for any of the systems to solve all the instances optimally. On **10 × 1000**, all systems converge to the same solution besides ORT SemiNaive, which reaches almost double the makespan compared to the others.

OR-Tools provides worst solutions on the **100 × 10** and **100 × 100** instances; in fact, in CP Optimizer the Advanced and the SemiNaive encodings achieve the best results among the other systems, with even the Naive encoding achieving better results than the OR-Tools ones. In the remaining instances, the role of the global constraint becomes crucial. In fact, it is possible to solve those instances only with the Advanced encodings, while with Naive and SemiNaive encodings it is not manageable to even instantiate the problem. The high number of operations, and in particular the number of jobs, makes the number of disjunctive constraints explode. Concerning the biggest instances **1000 × 1000**, only CP Optimizer is able to find a solution in each of the 10 instances. OR-Tools is able to instantiate the problem instances but does not manage to find any solutions within the timeout of 6 h.

Table 2. Average makespan over the 9 instance groups of the large-scale benchmark.

Problem instances	CP Optimizer			OR-Tools		
	Naive	SemiNaive	Advanced	Naive	SemiNaive	Advanced
10 × 10	8169.9	8169.9	8169.9	8169.9	8169.9	8169.9
10 × 100	55224.5	55224.5	55224.5	55224.5	55224.5	55224.5
10 × 1000	514393.3	514393.3	514393.3	514393.3	1139284.3	514393.3
100 × 10	55084.9	54858.6	54858.6	59623.6	82619.6	55493.5
100 × 100	93827.1	89311.4	80570.5	141524.5	3327955.9	117332.8
100 × 1000	Timeout	Timeout	545687.7	Timeout	Timeout	604119.2
1000 × 10	Timeout	Timeout	515429.7	Timeout	Timeout	550534.4
1000 × 100	Timeout	Timeout	536403.6	Timeout	Timeout	686811.1
1000 × 1000	Timeout	Timeout	1017974.1	Timeout	Timeout	Timeout

We analyzed also the evolution of intermediate solutions during the search, in order to have a better understanding of the solving process. Figures 2, 3 and 4 offer a detailed view of the search process: every plot shows the results of one instance class, showing time (in seconds) on the x-axis and makespan on the y-axis. In each plot, every line is linked with a system and, conforming to Table 2, represents the mean makespan of the intermediate solutions at each time x. Basically, Table 2 shows the situation when $x = timeout$, while the plots show the evolution of the search as x varies from 0 to $timeout$.

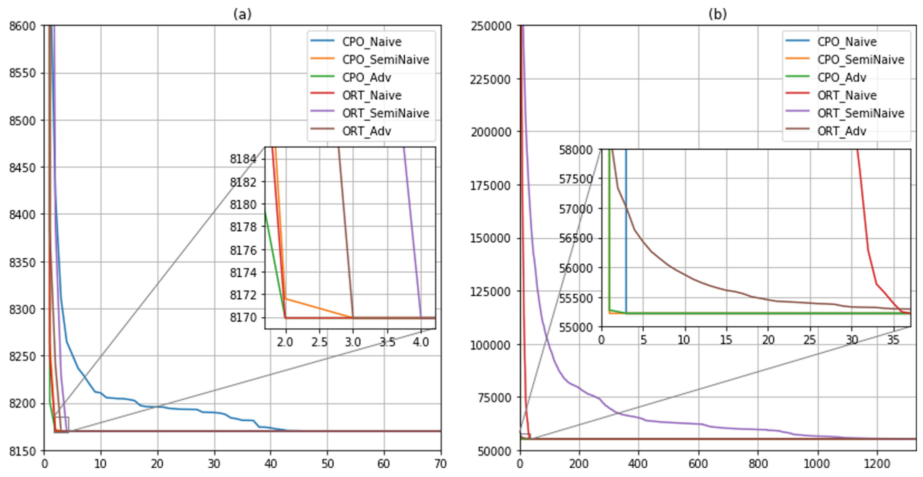

Fig. 2. (a) Plot of instance group $\mathbf{10 \times 10}$ (b) Plot of instance group $\mathbf{10 \times 100}$.

Figure 2 shows the results regarding $\mathbf{10 \times 10}$ (a) and $\mathbf{10 \times 100}$ (b), the only instance groups where it was possible to achieve optimal makespan in all the instances. In (a) the time interval is limited between 0 and 70 s, since all the systems converge hastily to the optimal solution in all cases. The slowest is CPO Naive, achieving the optimal makespan on all instances after 70 s. Concerning the other systems, we can see that both CPO Advanced and ORT Naive converge to the optimum after 2 s, ORT Advanced after 3 and ORT SemiNaive after 4. In (b) the slowest system is ORT SemiNaive, reaching optimal results after 1330 s. The other ORT encodings follow at 77 s for the Advanced and 37 for the Naive. All CPO encodings reached the optimum within 3 s.

The fact that in OR-Tools the Naive encoding is faster than both SemiNaive and Advanced is peculiar. Concerning the Advanced encoding, this is due to one particular instance ($\mathbf{10 \times 100\_4}$) which needs 77 s to be solved by ORT Advanced and 32 by ORT Naive. However, the average solving time of the $\mathbf{10 \times 100}$ group (without instance **4**) is 15 s for ORT Advanced and 25 for ORT Naive. Thus, we can see this case as an outlier. Concerning ORT SemiNaive, we see a general performance decline in all the instances of this benchmark. The reason for this behavior is more complex and will be treated in detail in Sect. 4.

Figure 3 illustrates the last instance group with 10 jobs ($\mathbf{10 \times 1000}$) and all groups with 100 jobs. In Fig. 3(a) we omitted the legend for space reasons, but we maintained the line colors consistent across plots, therefore the legend from any other plot can be used as a reference. The worst performer is ORT SemiNaive, which is visibly far from all other systems. In fact, the improvement of the solutions is very slow compared to the others and, albeit continuing until the timeout

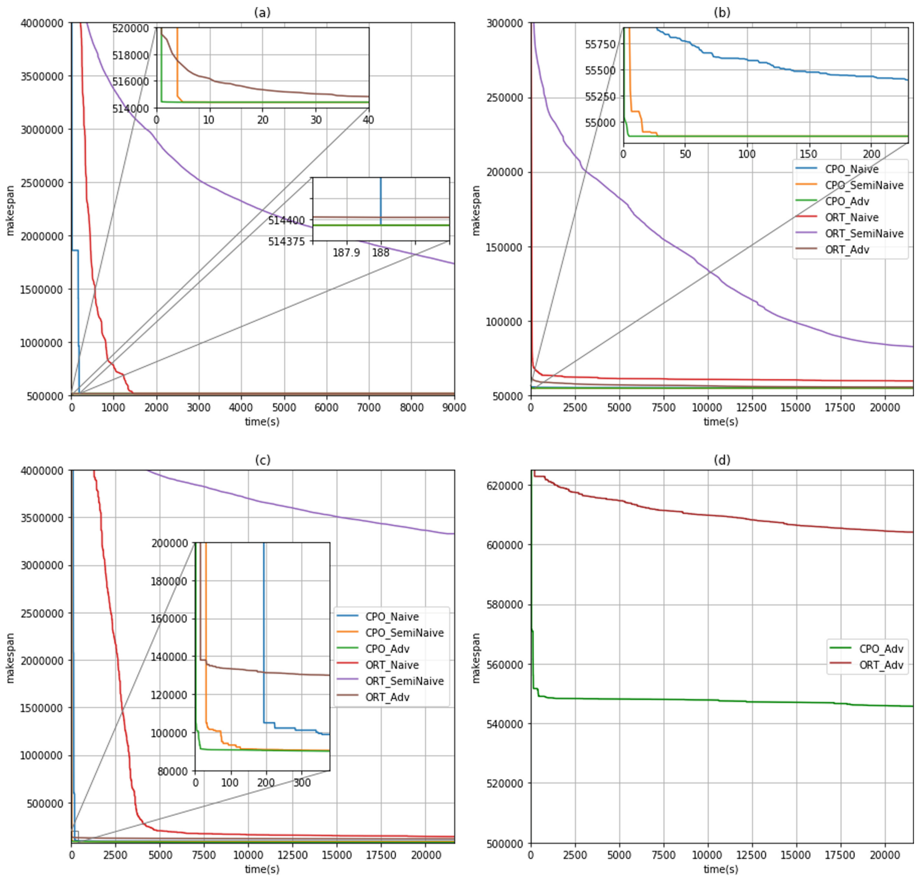

Fig. 3. (a) Plot of instance group **10** $\times$ **1000** (b) Plot of instance group **100** $\times$ **10** (c) Plot of instance group **100** $\times$ **100** (d) Plot of instance group **100** $\times$ **1000**. (Color figure online)

occurs, it stops being effective after around 8000 s (about 2 h and 20 min). The other systems are much faster and they all converge to the same result: ORT Naive in 1484 s, CPO Naive in 188 s, ORT Advanced in 202 (although already at 40 s is just 0.08% off CPO's result, and at 188 is 0.002% off) and finally CPO SemiNaive in 10 s and CPO Advanced in 4.

In Fig. 3(b) the trend continues, with ORT SemiNaive being the worst performer, followed by ORT Naive, which comes as close as 8% off CPO Advanced after 195 s, and then does not improve this result. ORT Advanced arrives at 10% off CPO Advanced already after 100 s, and then continues to slowly improve the solutions until 1.16% off. CPO Naive behaves in a similar way, being able to find solutions just 1% off in the first 30 s, improving them until 0.41% off when

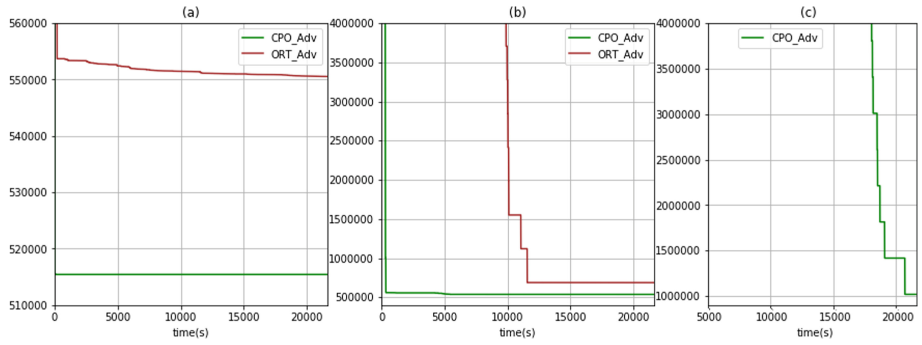

Fig. 4. (a) Plot of instance group **1000** × **10** (b) Plot of instance group **1000** × **100** (c) Plot of instance group **1000** × **1000**.

the timeout occurs. Both CPO SemiNaive and Advanced converge to the same result after 27 and 4 s respectively.

Figure 3(c) does not show a significant difference from (b) in terms of systems' behavior. In (d) however, we see the first instance with 100000 operations. In this case, only the Advanced encodings are able to solve the instances, while the others cannot even instantiate the problem. CPO finds good solutions already after 160 s, with just a marginal improvement in the following 6 h. ORT shows a more regular improvement, on average reaching final solutions which is 10% worse than the ones from CPO.

Figure 4 shows the results of the three instance groups with 1000 jobs. As anticipated in Table 2, only the Advanced encodings are able to instantiate these instances. In (a) we see a rapid convergence to final solutions in CPO after 70 s, while ORT manages to achieve a result that is about 7% worse than the CP Optimizer one. In (b), CP Optimizer achieves the average makespan of 536454.1 in 11 s, with an improvement in the following 6 h of just 400.5 makespan on average (about 0.07% improvement). OR-Tools, after the 3-hour mark, manages to greatly improve its solutions, achieving a result 28% above CPO's.

Finally, (c) shows the test on the largest instances of the benchmark. As we can see, CP Optimizer finds first solutions after 5 h of computation, and then gradually improves its results until it reaches an average makespan of 1017974.1. Although it is hard to judge these results without any comparison from other solvers, it is still remarkable that CP Optimizer tackles all 10 instances with 1 million operations within 6 h. Moreover, based on the plots of all the other instance groups, the fast convergence to a value often corresponds to a (near) optimal solution. This belief is strengthened by a further test that we launched on the largest instance group, doubling the solving time. In 7 out of 10 instances, there was no improvement in the solutions. In the remaining 3, the improvement was less than 0.01% of the makespan.

In the large-scale benchmark, CP Optimizer dominates the scene, being both faster and better than the OR-Tools counterparts, and managing to solve instances up to one million operations. OR Tools, albeit with higher makespans, is able to solve all the instances of the large benchmark up to 100 thousand operations, and is even able to instantiate (but not solve) the biggest instances $\mathbf{1000} \times \mathbf{1000}$. In general, we can confirm the conclusions on the classic benchmark on the benefits of interval variables and no-overlap constraint[5]. Moreover, we can add that the global constraints are not only helpful during the solving process, but also crucial to instantiate the larger instances.

4 Discussion

In our experiment we thoroughly tested two state of the art CP solvers with different encodings on a vast selection of instances. The best overall combination of solver-encoding is, without a doubt, CP Optimizer with the Advanced encoding. OR-Tools with Advanced encoding performed not much worse than CPO on the classic benchmark but was not able to keep the same pace on the large-scale instances. To understand why, we looked at the different implementation choices of the two solvers.

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, slowing down the propagation of constraints.

The two solvers use a similar search strategy based on large neighborhood search (LNS), which iteratively relax and re-optimize the problem instance. To decide what to fix and what to vary on the partial schedules, CPO uses portfolio strategies in combination with machine learning to converge to the best neighborhoods [8]; ORT uses a less sophisticated method with random variables, random constraints and local neighborhood in the var-constraint graph. Furthermore, CPO uses a "plan B" strategy called failure directed search (FDS), which is triggered when LNS is not able to improve the current solution [19].

While negligible on the small instances, these differences appear to be particularly impactful on the large problem instances, as it appears from our experiment. In fact, considering the Naive and SemiNaive encoding, OR-Tools performs even better than CPO on the classic benchmarks. Even looking at the $\mathbf{10} \times \mathbf{10}$ instances of the large-scale benchmark (which are the most similar in size to the classic benchmarks), we see that ORT Naive is at pair with CPO Advanced (Fig. 2(a)) and much better than CPO Naive to converge to the optimum. However, CPO takes the lead already with the $\mathbf{10} \times \mathbf{100}$ instances, which, albeit being among the smallest of our benchmark, are still bigger than any of the instances of the classic benchmark.

A particularly unexpected performance is registered by the SemiNaive encoding on OR-Tools, which is among the best performer on the classic benchmark,

[5] With the exception of ORT SemiNaive.

but is by far the worst in the large-scale one, even compared with the Naive encoding on ORT. This unforeseen behavior is explained by the fact that in the SemiNaive encoding the interval variables ensure that start and end values always respect the duration, without an explicit constraint. However, in ORT, the relaxation process will pick up the equality constraint on the Naive encoding, but not the interval equivalent. Thus, even if the Naive encoding has to instantiate more constraints, it can take full advantage of the relaxation process which, as the results show, becomes more impactful the bigger the instances. To be useful in OR-Tools, the interval variables have to be coupled with the no overlap constraint, which is also picked up in the relaxation process.

In CP Optimizer, on the other hand, there is no such issue because the interval variables are also treated during the relaxation process. In particular, each interval is treated as four numerical variables, while the duration of the interval is imposed with a "precedence" constraint with a delay between the start and the end of the interval. As a result, the use of interval variables is always beneficial in CP Optimizer, both in the classic and the large-scale benchmark.

5 Conclusions

In this paper, we tested two of the best current CP Solvers, OR-Tools and CP Optimizer. The experiment was conducted on, both, a set of famous benchmark instances from literature as well as a large-scale benchmark generated by us, testing three encodings of increasing sophistication for each solver. Based on the results, we can draw the following conclusions:

- CPO with the Advanced encoding was the best overall solver-encoding system, followed by OR-Tools Advanced.
- The difference between ORT and CPO is marginal in small instances (*e.g.* classic benchmark) but becomes more significant the larger the problem instances.
- The use of interval variables in the model brings a tangible benefit compared to a naive approach, both in terms of solving time and quality of solution, but does not help during the instantiation of the problem.
- The use of global constraints further improves solving time and solution quality, and makes possible to instantiate large problem instances.

A more efficient implementation of the interval variables, an additional strategy to avoid the local optima and more a sophisticated algorithm to select the best partial schedules played a major role in the supremacy of CP Optimizer, with was able to solve instances up to 1 million operations. However, one has to consider that CP Optimizer is a proprietary solution mainly targeted at solving scheduling problems. Despite being an open-source solver, OR-Tools performance was able to solve instances up to 100000 operations, which means that it could be applied even for real-world industrial problems.

References

1. Adams, J., Balas, E., Zawack, D.: The shifting bottleneck procedure for job shop scheduling. Manage. Sci. **34**(3), 391–401 (1988). http://www.jstor.org/stable/2632051
2. Applegate, D., Cook, W.: A computational study of the job-shop scheduling problem. ORSA J. Comput. **3**(2), 149–156 (1991). https://doi.org/10.1287/ijoc.3.2.149
3. Da Col, G., Teppan, E.C.: Declarative decomposition and dispatching for large-scale job-shop scheduling. In: Friedrich, G., Helmert, M., Wotawa, F. (eds.) KI 2016. LNCS (LNAI), vol. 9904, pp. 134–140. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46073-4_11
4. Danna, E., Rothberg, E., Le Pape, C.: Integrating mixed integer programming and local search: a case study on job-shop scheduling problems. In: Fifth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'2003), pp. 65–79 (2003)
5. Fox, M.S., Allen, B.P., Strohm, G.: Job-shop scheduling: an investigation in constraint-directed reasoning. In: AAAI, pp. 155–158 (1982)
6. Ku, W.Y., Beck, J.C.: Mixed integer programming models for job shop scheduling: a computational analysis. Comput. Oper. Res. **73**, 165–173 (2016)
7. Laborie, P., Godard, D.: Self-adapting large neighborhood search: application to single-mode scheduling problems. In: Proceedings MISTA-07, Paris, vol. 8 (2007)
8. Laborie, P., Rogerie, J.: Temporal linear relaxation in IBM ILOG CP optimizer. J. Sched. **19**(4), 391–400 (2016)
9. Laborie, P., Rogerie, J., Shaw, P., Vilím, P.: IBM ILOG CP optimizer for scheduling. Constraints **23**(2), 210–250 (2018)
10. Lawrence, S.: Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques (Supplement). Carnegie-Mellon University, Graduate School of Industrial Administration (1984)
11. Muth, J., Thompson, G.: Industrial Scheduling. International Series in Management. Prentice-Hall, New Jersey (1963)
12. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74970-7_38
13. Refalo, P.: Linear formulation of constraint programming models and hybrid solvers. In: Dechter, R. (ed.) CP 2000. LNCS, vol. 1894, pp. 369–383. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45349-0_27
14. Sadeh, N.M., Fox, M.S.: Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. Artif. Intell. **86**, 1–41 (1996)
15. Storer, R.H., Wu, S.D., Vaccari, R.: New search spaces for sequencing problems with application to job shop scheduling. Manage. Sci. **38**(10), 1495–1509 (1992)
16. Taillard, E.: Benchmarks for basic scheduling problems. Eur. J. Oper. Res. **64**(2), 278–285 (1993)
17. Teppan, E.C., Da Col, G.: Automatic generation of dispatching rules for large job shops by means of genetic algorithms. In: 8th International Workshop on Combinations of Intelligent Methods and Applications (CIMA 2018), pp. 43–57 (2018)
18. Vazquez, M., Whitley, L.D.: A comparison of genetic algorithms for the dynamic job shop scheduling problem. In: 2nd Annual Conference on Genetic and Evolutionary Computation, pp. 1011–1018. Morgan Kaufmann Publishers Inc. (2000)

19. Vilím, P., Laborie, P., Shaw, P.: Failure-directed search for constraint-based scheduling. In: Michel, L. (ed.) CPAIOR 2015. LNCS, vol. 9075, pp. 437–453. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18008-3_30
20. Yamada, T., Nakano, R.: A genetic algorithm applicable to large-scale job-shop problems. In: PPSN, pp. 283–292 (1992)