Types of Flexible Job Shop Scheduling: A Constraint Programming Experiment

Erich C. Teppan[1,2][a]
[1] *Universität Klagenfurt, Austria*
[2] *Fraunhofer Austria - KI4Life, Austria*

Keywords: Flexible Job Shop Scheduling, Constraint Programming, Empirical Evaluation.

Abstract: The scheduling of jobs is a crucial task in every production company and becomes more and more important in the light of the fourth industrial revolution (Industry 4.0) that aims at fully automated processes. One such problem formulation with big practical relevance is the flexible job shop scheduling problem (FJSSP). Since the classic problem formulation is more general than what can be found in most of nowadays industrial environments, this paper introduces different types of flexibility and investigates how the flexibility type, the amount of allowed flexibility and the presence of machine dependent processing times influence the solution quality that can be achieved by a state-of-the-art constraint solver within limited time. Results show that certain forms of flexibility, higher flexibility factors and the absence of machine dependent processing times can ease the problem.

1 INTRODUCTION

Scheduling and particularly the scheduling of jobs (Blazewicz et al., 2014) has been an active field of research at least since the mid of the last century (Muth and Thompson, 1963; Johnson, 1953; Manne, 1960; Bowman, 1959). In the industrial context scheduling has become more important than ever in the light of Industry 4.0 (also sometimes called the fourth industrial revolution) which aims at full automation and digitization of processes, in particular production processes (Lasi et al., 2014). Among the various definitions of different scheduling problems the flow shop scheduling problem (FSSP) (Johnson, 1953) and the job shop scheduling problem (JSSP) (Brucker and Schlie, 1990; Brucker and Jurisch, 1993; Applegate and Cook, 1991) play an especially important role for production plants. Both, FSSP and JSSP, are known to be NP-hard problems (Garey et al., 1976).

Informally spoken, the problems can be described as follows: In FSSP and JSSP a set of jobs is to be accomplished by a set of machines (or resources in general). A job can be thought of as the production process carried out to create a certain product. This process, i.e. the job, is sub-divided into job operations each of which has to be performed by a dedicated machine and possesses a predetermined processing time on the assigned machine (preemption is not allowed). Hence, the machine and the corresponding processing time for each job is part of the input. The problem consists in ordering the job operations on each machine such that some optimization criterion is optimized, e.g. minimization of the total completion time. What makes the FSSP different from the JSSP is that the order in which jobs go through the different machines is the same for all jobs (though the processing times can vary). Take as an explanatory example for a FSSP a bakery. In principle, every bread has to go through the same production steps in the same order although recipes and related processing times differ for different products. In the JSSP, the order in which jobs go through the machines differ between jobs. Examples for that can be found in factories where different types of products are produced (e.g. for some products the cutting operation might be before the screwing operation and for other products it might be the other way round). Thus, the FSSP can be seen as a special of the JSSP. For the classic JSSP (and as well for the FSSP) for each operation there exists only one possible machine which can/must execute it. Thus, there is no guessing about operation-to-machine assignments. However, for many real scenarios these assumptions are to restrictive.

The flexible job shop scheduling problem (FJSSP) (Chaudhry and Khan, 2015) is a direct generalization

[a] https://orcid.org/0000-0001-8397-9303

Teppan, E.
Types of Flexible Job Shop Scheduling: A Constraint Programming Experiment.
DOI: 10.5220/0010849900003116
In *Proceedings of the 14th International Conference on Agents and Artificial Intelligence (ICAART 2022) - Volume 3*, pages 516-523
ISBN: 978-989-758-547-0; ISSN: 2184-433X

of the JSSP allowing more than one machine for a job operation, also with different processing times for the same operation. This introduces the routing problem, hence, the assignment of job operations to machines is not predetermined and is part of the solution and not part of the input as for the JSSP. Although, the JSSP as well as the FJSSP are NP-hard and their decision problem versions[1] are NP-complete, the FJSSP is widely assumed to be generally harder than the JSSP because of the additional routing sub-problem. This view is supported by the fact that by increasing the flexibility (i.e. more than one machine could perform a job operation) also the number of possible (though not forcibly optimal) schedule solutions is increased at the same time. Consequently, the search space for an optimal schedule is bigger.

However, from a practical point of view, some aspects could ease the problem significantly:

1. First of all, for many application areas there are not (significantly) different processing times for the same operation on different machines. Take as a simple example the bakery described above. How long an operation takes is dependent on the type of bread but not on the type of the oven.
2. A second important point where the FJSSP allows more generality than needed for many application areas is that in the FJSSP there are no restrictions on which machines can perform the same operations and as a matter of fact typical benchmark problems use totally random machine sets. In real application scenarios, though, the reason why a certain amount of flexibility for machine assignment is possible is that machines are similar or equal. As a consequence, they can execute the same set of operations and, thus, implement parallelism.
3. A third issue that is not taken into account when stating that the FJSSP is harder than the JSSP is the question of optimality. It is rather intuitive that in the worst case to find an optimal solution for a FJSSP is harder than for a JSSP due to the bigger search space. However, most often in real application environments absolute optimality is not needed and near-optimal solutions are enough. The bigger search space should also increase the number of good enough solutions so that it is not clear whether higher flexibility factors really increase hardness when searching only a near-optimal solution.

[1]Deciding whether there exists a solution with an objective value equal or better than some given constant (e.g. completion time $\leq \kappa$).

In this paper we try to identify easier classes of FJSSP from a practical constraint solving point of view. To this end, we introduce different types of flexibility and report on an experiment based on problem instances derived from a large scale JSSP benchmark suite (Da Col and Teppan, 2021) where we systematically change flexibility factors, types of flexibility and whether processing times are machine dependent or not. For solving the problem instances we use constraint programming (CP) out of three reasons: First of all, CP has a long and successful history in solving job scheduling problems (Fox et al., 1982; Fox et al., 1989; Sycara et al., 1991; Sadeh and Fox, 1996; Keng and Yun, 1989; Rodler et al., 2021). Second, CP solvers are currently the strongest for solving scheduling problems, with IBM's CP Optimizer and Google's OR Tools leading the way (Laborie et al., 2018; Da Col and Teppan, 2019a; Da Col and Teppan, 2019b). The third important reason is that in CP it is possible to give a worst case estimation on how far some current best solution is off the optimum since lower and upper bounds are inherently calculated by constraint propagation techniques. This is crucial for many real life problems for which optimal solutions are generally out of reach.

The further reading is as follows: In Section 2 we formally define the FJSSP and some special flexibility types that are sufficient for many real life application scenarios. In Section 3 we discuss a state-of-the-art constraint model to be used with IBM's CP Optimizer. In Section 4 we present an empirical evaluation that analyzes the impact different types of flexibility, the amount of allowed flexibility and machine dependent processing times have on the proven quality of (near-) optimal solutions found within limited calculation time.

2 THE FLEXIBLE JOB SHOP SCHEDULING PROBLEM

The flexible job shop scheduling problem (FJSSP) can be defined as follows:

Definition 1. *Given a set of machines $M = \{m_1, \dots, m_v\}$ and a set of jobs $J = \{j_1, \dots, j_w\}$:*

(1) Each job $j \in J$ consists of a sequence of operations $O_j = \{j_1, \dots, j_{l_j}\}$ whereby j_{l_j} is the last operation of job j.

- *For a job j and its operation j_i, the operation j_{i+1} is called successor, and the operation j_{i-1} is called predecessor.*
- *We refer to the set of all operations as $O = \bigcup_{j \in J} O_j$*

(2) *For each operation o, there is a non-empty set of machines $M_o \subseteq M$ representing the machines that are able (or allowed) to process operation o.*
   - *We refer to the maximum size of the operations' machine sets $max\{|M_o| : o \in O\}$ as the problem's flexibility factor $flex$.*

(3) *For each operation $o \in O$ and machine $m \in M_o$, there is a predefined processing time $time_{o,m} \in \mathbb{N}$.*

(4) *A (consistent and complete) schedule consists of an assigned machine $m_o \in M_o$ and a start time $start_o \in \mathbb{N}$ for each operation $o \in O$ such that:*
   - *An operation's successor starts only after the operation is finished:* $start_{j_{i+1}} \geq start_{j_i} + time_{j_i,m_{j_i}}$
   - *Operations on the same machine do not overlap:* $start_{o1} \leq start_{o2} \longrightarrow start_{o1} + time_{o1,m_{o1}} \leq start_{o2}$

(5) *An objective function is optimized.*
   - *One of the most classic optimization criteria in this context is the completion time C_{max}, that is the time span needed for processing all operations:* $minimize\ \max\{start_o + time_{o,m_o} : o \in O\}$

The FJSSP is strongly NP-hard. Already the job shop scheduling problem (JSSP), being a special case, is strongly NP-hard (Garey et al., 1976). As in the JSSP for each operation there exists only one machine that can process it, the JSSP can be seen as a FJSSP with flexibility factor $flex = 1$.

Observation 1. *Definition 1 (2) does not specify how machine sets $M_o \subseteq M$ are composed and why. In real production environments operations and machines are tied together via the concept of operation types. Each operation o possesses a type $t_o \in T$ and a machine m supports one or more operation types $T_m \subseteq T$. A machine m can process an operation o if (and only if) $t_o \in T_m$.*

Depending on the actual production line layout and machine capabilities, the concept of operation types offers the possibility of identifying different types of flexibility that can be present and we expect this to have a significant impact on the hardness of a FJSSP.

Observation 2. *In many cases, the processing times for a certain operation is the same for every machine that can process it (or differences are small and can be neglected). Consequently, there would be only one machine independent processing time ($time_o$) for each operation o instead of machine dependent processing times ($time_{o,m}$, compare Definition 1 (3)).*

With respect to Observation 1 we introduce three special types of flexibility in job shops, which can be found in nowadays industrial production environments (see Figure 1).

Imagine a classic job shop, i.e. a production environment with exactly one machine for each operation type (Figure 1-a). Now it is possible to flexibilize this job shop by adding new machines such that multiple machines operate in parallel for the same operation type (Figure 1-b). In this case, although the number of possible machines for an operation type increases, the number of operation types a machine supports does not increase.

Another common shop configuration that can be found are closed machine sets (Figure 1-c). In this case, machines support multiple operation types and can be sub divided into closed subsets containing machines with the same capabilities.

Another shop configuration is given when multi purpose machines have asymmetric capabilities, thus do not form closed machine sets, yet are linked by their capabilities (Figure 1-d). In the most extreme case this forms a chain where there is a machine that supports type A and B, the second machine supports type B and C, the third type C and D and so on.

All of those different FJSSPs remain NP-hard as they all contain the classic JSSP as a special case that is strongly NP-hard itself. The basic question treated in this paper is how increased flexibility factors, the flexibility type (see Figure 1) and machine dependent processing times (see Observation 2) influence the hardness of the FJSSP (in particular for CP solvers as they are currently the means of choice for job shop scheduling problems).

3 CP MODEL

A state-of-the-art CP model for the FJSSP expressed in IBM's modelling language OPL[2] is depicted in Listing 1.

```
using CP;

tuple paramsT {
  int nbJobs; int nbMchs;
};

tuple Operation {
  int opId; int jobId; int pos;
};

tuple Mode {
  int opId; int mch; int pt;
```

[2] https://www.ibm.com/docs/en/icos/20.1.0?topic=opl-optimization-programming-language

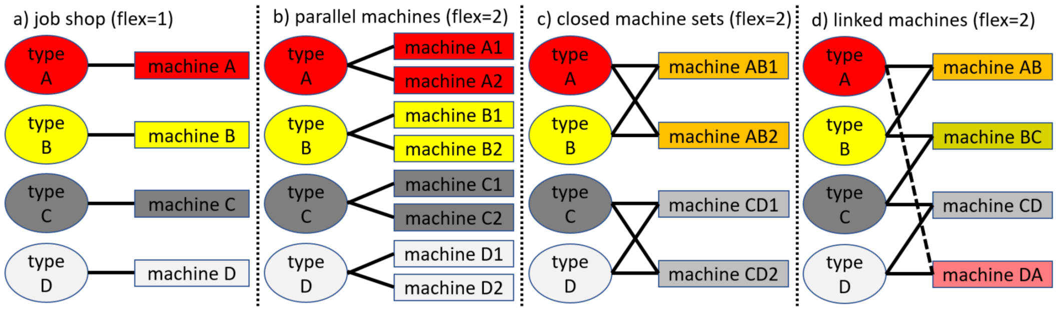

Figure 1: Different types of flexibility in job shops with flexibility factor $flex = 2$.

```
13 };
14
15 paramsT   Params    = ...;
16 {Operation} Ops      = ...;
17 {Mode}     Modes     = ...;
18
19 range Jobs = 1..Params.nbJobs;
20 range Mchs = 1..Params.nbMchs;
21
22 int jlast[j in Jobs] = max(o in Ops:
       o.jobId==j) o.pos;
23
24 dvar interval ops  [Ops];
25 dvar interval modes[md in Modes]
       optional size md.pt;
26 dvar sequence mchs[m in Mchs] in
       all(md in Modes: md.mch==m)
       modes[md];
27
28 minimize max(j in Jobs, o in Ops:
       o.pos==jlast[j]) endOf(ops[o]);
29
30 subject to {
31   forall (o in Ops)
       alternative(ops[o], all(md in
       Modes: md.opId==o.opId) modes[md]);
33   forall (m in Mchs)
34     noOverlap(mchs[m]);
35   forall (j in Jobs, o1 in Ops, o2 in
       Ops: o1.jobId==j && o2.jobId==j
36     && o2.pos==1+o1.pos)
       endBeforeStart(ops[o1],ops[o2]);
37 }
```

Listing 1: OPL encoding of the flexible job-shop scheduling problem (FJSSP).

Line 1 states that the CP library is used[3]. Lines 3-18 define three tuple data types[4] (*paramsT*, *Operation*, *Mode*) to be filled with the *Params* tuple, an *Ops* tuple or a *Modes* tuple from the input file. Lines 20-22 do the actual read-in of the input data. In particular, the *Params* tuple from the input is read into the variable of type *paramsT*, the *Ops* tuples are read into a variable that takes a set of *Operation* tuples and analog for the *Modes* tuples. Modes actually represent the set of possible machines for an operation.

In Lines 24 and 25 two integer ranges are defined for reuse with array data structures and loops. In Line 27 an integer array is defined and initialized that saves the last job positions for all jobs, corresponding to a pointer to the last operation in a job. Up to here, the encoding is concerned with the read-in of the input and some preparations for convenience reasons.

The rest of the encoding, Lines 29-41, contains the actual declarative problem representation. In Line 29, an array of interval variables is defined so that for each *Operation* in *Ops* there is an interval variable. Note that the domains are not set explicitly such that the lower bound is 0 and the upper bound is set automatically by the solver.

In Line 30, an array of interval variables for each mode is defined. Those variables also take the processing times as input (i.e., *size md.pt*). Hence, there is an interval variable for each possible operation-to-machine combination. Recall that it is not clear beforehand to which machine a particular operation will be assigned to. For this reason, the variables are set as *optional*, which means that they can be active or not, depending on whether an operation is assigned to a specific machine or not.

The interval variables for operations (Line 29) and the (optional) interval variables for the modes are linked together in Lines 36-37. The *alternative* statements express that among all possible alternatives for an operation variable, i.e., all corresponding mode variables, exactly a single one can be active. The operation variable's start, end and processing time values are set equal to its active alternative. The question about which alternative to set active is part of the problem and to be determined by the solver.

In Line 31, an array of helping *sequence* variables is defined, each of which stores a list of modes (i.e. operation assignments) for a machine. Those variables are used in Lines 38-39 for establishing *noOverlap* constraints, assuring that a machine can process only one operation at a time. Finally, Lines 40-41

[3] OPL also supports other types of programs.

[4] This is similar to structs in the C language.

define constraints that guarantee that each successor operation can start only after its preceding operation has finished.

4 EVALUATION

We conducted an experiment in order to investigate the following research questions:

- How does the type of flexibility in job shops (see Section 2) impacts on FJSSP hardness?
- Does FJSSP becomes harder or easier with increased flexibility factors?
- Is there a difference between having one processing time per operation or having one processing time per operation-machine combination (i.e. machine dependent vs. machine independent processing times)?

As IBM's CP Optimizer is currently (at least one of) the most powerful solver(s) for scheduling problems, in particular job shop scheduling problems (Laborie and Rogerie, 2016; Laborie et al., 2018; Laborie and Godard, 2007; Da Col and Teppan, 2019b), we implemented the model in Section 2 to be used with CP Optimizer. Since it is also possible to interface via Java (and C++) with CP Optimizer, we implemented the model in Java, instead of directly using the OPL program. The intuition behind was to use a single language for all experimental components, i.e. the model, the benchmarking routines and the systematic creation of new benchmark problem instances.

Dataset. In order to give first answers to our research questions and systematically investigate the effects of the actual type of flexibility (Observation 1, Figure 1), different flexibility factors, or whether processing times for operations vary depending on the machine that execute them or not (Observation 2), novel benchmark problems were used.

As a starting point we used the 100x100 JSSP instances from (Da Col and Teppan, 2021) as they are on the one hand conceptually similar to Taillard's famous benchmark (Taillard, 1993) but bigger in size (100 jobs on 100 machines = 10000 operations), and thus better reflect nowadays industrial realities. For these instances we built different variants, that differ in the used type of flexibility (*random*, *parallel*, *closed*, *linked*), the flexibility factor ($flex = 1$, $flex = 2$, $flex = 4$), and whether processing times of operations are machine dependent or not.

In order to produce *random* FJSSP instances, for each operation o, its machine set M_o was composed of the machine to which o was assigned to in the original JSSP instance and the remaining machines (e.g. with flex=4 there are three remaining machines) were chosen randomly, avoiding doublets. In order to produce *parallel* FJSSP instances (compare Figure 1), for each machine m, $flex - 1$ additional machines were defined. For each operation o, its machine set M_o was composed of m and the $flex - 1$ additional machines. *Closed* machine sets were produced by partitioning the set of machines in the original JSSP into sets of size $flex$, i.e. $\{\{m_1, ..., m_{flex}\}, \{m_{flex+1}, ..., m_{2*flex}\}, ...\}$. For each operation o, its machine set M_o was composed of the closed machine set containing the machine it was assigned to in the original JSSP instance. Similarly, for creating *linked* FJSSP instances, for each operation o assigned to m_i in the original JSSP instance, the machine set M_o was composed as $\{m_i, m_{i+1}, ..., m_{i+flex-1}\}$. For the problem instances without machine dependent processing times, the original processing times were used for all machines in an operation's machine set. For the problem instances using machine dependent processing times, random offsets between 0% and 100% were added to the original processing times.

Experimental Setting. The experiment was conducted on a system equipped with a 2 GHz AMD EPYC 7551P 32 Cores CPU and 256 GB of RAM[5]. As a CP solver CP Optimizer 12.10 was employed and configured to use a single worker thread. We defined a 1000 seconds timeout and measured the (relative) objective gap in the end. The objective gap, which can be directly retrieved from CP Optimizer, represents a worst case estimation of how much a current solution is above the optimum ($\frac{objectiveValue - lowerBound}{objectiveValue}$). An optimal solution has a gap of zero.

Results. Table 1 depicts all measured results. Figure 2 visualizes the average gaps for the different classes of FJSSP instances. Having a flexibility factor of $flex = 1$, the type of flexibility (*random*, *closed*, *linked*, *parallel*) does not make any difference as it always boils down to the same non flexible JSSP (gaps around 30% off the optimum). However, higher flexibility have different impacts on the measured objective gaps depending on the type of flexibility and whether processing times are machine dependent ($*DiffTimes$) or not:

In all settings the objective gaps shrinked with increased flexibility factors except for closed ma-

[5] Memory usage for one experimental run was always below 16GByte.

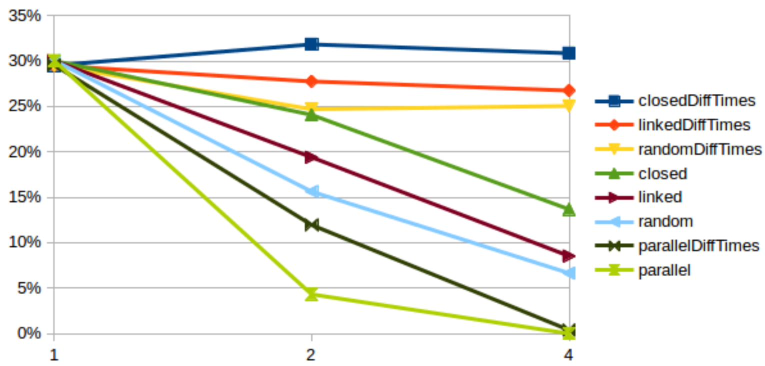

Figure 2: Average objective gaps with respect to different flexibility factors 1, 2 and 4.

chine sets with machine dependent processing times ($closedDiffTimes$). Here, the objective gaps even increased slightly (though the trend turns down again with $flex = 4$). The linked machines ($linked*$) and closed machine sets ($closed*$) settings were even harder than cases with random operation to machine set assignments. Parallel machine settings ($parallel*$) are the by far easiest cases and it can be stated that introducing new parallel machines makes it easier to find/prove (near-) optimal solutions. For all different flexibility types the use of machine dependent processing times ($*DiffTimes$) makes the problem significantly harder for the CP Optimizer solver to find better solutions (or prove their optimality).

5 CONCLUSION

Referring to the three aspects identified in the introduction of this paper that might have the potential to ease the FJSSP from a practical point of view, the results of the experimental evaluation supports the assumptions that:

1. Machine independent processing times, i.e. operations possess the same processing time for each machine on which they could be processed, eases the challenge of finding/proving (near-) optimal solutions.
2. The architecture of a (flexible job) shop, i.e. which type of flexibility is allowed/possible in a shop, highly impacts on problem solvability. In particular, the case where flexibility is based on parallel machines each of which only supporting the same single operation type, though still NP-hard from a theoretical point of view, was identified to be a quite easy type of FJSSP.
3. Increased flexibility factors do not decrease the (proven) solution quality. In contrary, in our evaluations the objective gap, i.e. the proven worst-case difference between the found solution and an optimal solution, decreased by increasing the allowed flexibility.

From a practical point of view the results of this paper has several implications:

- The parallel machine FJSSP variant, that was identified to be the easiest one, is often found also in nowadays modern production environments. For such application areas exact methods, in particular constraint programming (CP), should at least be tried also in the light of large instances, as it could be a strong alternative to heuristic methods.
- If operation processing times are machine dependent but do not differ largely, it could be a good idea to make these processing times machine independent (e.g. by using the average or the maximum) and correct the processing times in the near-optimal solution afterwards, since machine independent processing times lead to an easier problem variant from a practical CP point of view.
- The findings of this paper can be taken into account in order to or produce benchmark problem instances of differing hardness.
- The often heard/read assessment that the FJSSP is harder than the 'normal' JSSP is not generally true, especially when targeting at near-optimal solutions.

Future work might concentrate on the following issues:

- The results account for constraint programming (CP). It can be assumed that the findings also apply for other constraint based methods like (mixed) integer programming or SAT-solving. However, it is not clear whether heuristic methods (e.g. evolutionary and genetic algorithms (Teppan and Col, 2020; Teppan, 2018b; Teppan E.C.,

Table 1: Measured objective gaps.

	#	flex=1	flex=2	flex=4		#	flex=1	flex=2	flex=4
randomDiffTimes	1	0.29	0.24	0.21	random	1	0.30	0.14	0.04
randomDiffTimes	2	0.29	0.27	0.26	random	2	0.29	0.19	0.10
randomDiffTimes	3	0.29	0.26	0.26	random	3	0.31	0.16	0.07
randomDiffTimes	4	0.29	0.22	0.26	random	4	0.29	0.13	0.05
randomDiffTimes	5	0.32	0.25	0.26	random	5	0.31	0.17	0.08
randomDiffTimes	6	0.32	0.25	0.26	random	6	0.34	0.17	0.07
randomDiffTimes	7	0.27	0.26	0.26	random	7	0.29	0.17	0.08
randomDiffTimes	8	0.29	0.24	0.23	random	8	0.30	0.14	0.06
randomDiffTimes	9	0.32	0.24	0.25	random	9	0.32	0.15	0.06
randomDiffTimes	10	0.28	0.23	0.26	random	10	0.28	0.15	0.06
	avg	0.30	0.25	0.25		avg	0.30	0.16	0.07
linkedDiffTimes	1	0.29	0.28	0.27	linked	1	0.30	0.18	0.09
linkedDiffTimes	2	0.29	0.28	0.28	linked	2	0.30	0.22	0.11
linkedDiffTimes	3	0.29	0.28	0.27	linked	3	0.31	0.20	0.08
linkedDiffTimes	4	0.29	0.27	0.27	linked	4	0.29	0.17	0.07
linkedDiffTimes	5	0.31	0.30	0.28	linked	5	0.30	0.21	0.10
linkedDiffTimes	6	0.32	0.31	0.26	linked	6	0.34	0.21	0.09
linkedDiffTimes	7	0.27	0.27	0.26	linked	7	0.29	0.20	0.10
linkedDiffTimes	8	0.30	0.26	0.25	linked	8	0.30	0.17	0.07
linkedDiffTimes	9	0.32	0.26	0.27	linked	9	0.31	0.19	0.07
linkedDiffTimes	10	0.28	0.27	0.26	linked	10	0.28	0.19	0.08
	avg	0.30	0.28	0.27		avg	0.30	0.19	0.09
parallelDiffTimes	1	0.29	0.13	0.00	parallel	1	0.30	0.03	0.00
parallelDiffTimes	2	0.29	0.11	0.00	parallel	2	0.29	0.06	0.00
parallelDiffTimes	3	0.29	0.12	0.04	parallel	3	0.31	0.04	0.00
parallelDiffTimes	4	0.29	0.10	0.00	parallel	4	0.29	0.02	0.00
parallelDiffTimes	5	0.32	0.14	0.00	parallel	5	0.30	0.06	0.00
parallelDiffTimes	6	0.32	0.16	0.00	parallel	6	0.34	0.06	0.00
parallelDiffTimes	7	0.27	0.11	0.00	parallel	7	0.29	0.06	0.00
parallelDiffTimes	8	0.30	0.10	0.00	parallel	8	0.30	0.03	0.00
parallelDiffTimes	9	0.31	0.10	0.00	parallel	9	0.32	0.04	0.00
parallelDiffTimes	10	0.28	0.13	0.00	parallel	10	0.28	0.03	0.00
	avg	0.30	0.12	0.00		avg	0.30	0.04	0.00
closedDiffTimes	1	0.29	0.33	0.30	closed	1	0.30	0.23	0.13
closedDiffTimes	2	0.29	0.32	0.33	closed	2	0.29	0.27	0.17
closedDiffTimes	3	0.29	0.32	0.31	closed	3	0.31	0.24	0.14
closedDiffTimes	4	0.29	0.31	0.30	closed	4	0.29	0.22	0.11
closedDiffTimes	5	0.32	0.34	0.32	closed	5	0.30	0.25	0.16
closedDiffTimes	6	0.31	0.35	0.30	closed	6	0.34	0.25	0.14
closedDiffTimes	7	0.27	0.30	0.31	closed	7	0.29	0.25	0.14
closedDiffTimes	8	0.29	0.31	0.30	closed	8	0.30	0.23	0.12
closedDiffTimes	9	0.32	0.30	0.32	closed	9	0.32	0.23	0.13
closedDiffTimes	10	0.28	0.31	0.30	closed	10	0.28	0.24	0.13
	avg	0.30	0.32	0.31		avg	0.30	0.24	0.14

2018), dispatching rules(Teppan, 2018a)) could benefit form this paper's findings.

- More data sets including even larger problem instances could bring further insights. In particular, the question about whether there can be found a phase transition, e.g. e.g. with respect to flexibility factors, is imposed.
- Analysis of the makespan values should reveal whether the decreased objective gaps measured for settings with machine independent processing times and with larger flexibility factors are mainly due to smaller makespans or better proven lower bounds.

REFERENCES

Applegate, D. and Cook, W. (1991). A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3(2):149–156.

Blazewicz, J., Ecker, K. H., Pesch, E., Schmidt, G., and Weglarz, J. (2014). *Handbook on Scheduling: From Theory to Applications*. Springer Publishing Company, Incorporated.

Bowman, E. H. (1959). The schedule-sequencing problem. *Operations Research*, 7(5):621–624.

Brucker, P. and Jurisch, B. (1993). A new lower bound for the job-shop scheduling problem. *European Journal of Operational Research*, 64(2):156–167.

Brucker, P. and Schlie, R. (1990). Job-shop scheduling with multi-purpose machines. *Computing*, 45(4):369–375.

Chaudhry, I. A. and Khan, A. (2015). A research survey: Review of flexible job shop scheduling techniques. *International Transactions in Operational Research*.

Da Col, G. and Teppan, E. (2019a). Google vs IBM: A constraint solving challenge on the job-shop scheduling problem. In Bogaerts, B., Erdem, E., Fodor, P., Formisano, A., Ianni, G., Inclezan, D., Vidal, G., Villanueva, A., Vos, M. D., and Yang, F., editors, *Proc. 35th Int. Conf. on Logic Programming (ICLP'19), Technical Communications*, volume 306 of *EPTCS*, pages 259–265.

Da Col, G. and Teppan, E. (2019b). Industrial size job shop scheduling tackled by present day cp solvers. In Schiex, T. and de Givry, S., editors, *Principles and Practice of Constraint Programming*, pages 144–160, Cham. Springer International Publishing.

Da Col, G. and Teppan, E. (2021). Large-scale benchmarks for the job shop scheduling problem.

Fox, M. S., Allen, B. P., and Strohm, G. (1982). Job-shop scheduling: An investigation in constraint-directed reasoning. In *Proc. of the AAAI*, pages 155–158.

Fox, M. S., Sadeh, N., and Baykan, C. (1989). Constrained heuristic search. In *Proc. of the Eleventh International Joint Conference on Artificial Intelligence*, pages 309–315.

Garey, M. R., Johnson, D. S., and Sethi, R. (1976). The complexity of flowshop and jobshop scheduling. *Mathematics of operations research*, 1(2):117–129.

Johnson, S. (1953). *Optimal Two- and Three-stage Production Schedules with Setup Times Included*. Rand Corporation.

Keng, N. and Yun, D. Y. (1989). A planning/scheduling methodology for the constrained resource problem. In *IJCAI*, pages 998–1003.

Laborie, P. and Godard, D. (2007). Self-adapting large neighborhood search: Application to single-mode scheduling problems. In *Proc. of MISTA-07*, pages 276–284.

Laborie, P. and Rogerie, J. (2016). Temporal linear relaxation in ibm ilog cp optimizer. *Journal of Scheduling*, 19(4):391–400.

Laborie, P., Rogerie, J., Shaw, P., and Vilím, P. (2018). Ibm ilog cp optimizer for scheduling. *Constraints*, 23(2):210–250.

Lasi, H., Fettke, P., Kemper, H.-G., Feld, T., and Hoffmann, M. (2014). Industry 4.0. *Business & information systems engineering*, 6(4):239–242.

Manne, A. S. (1960). On the job-shop scheduling problem. *Operations Research*, 8(2):219–223.

Muth, J. and Thompson, G. (1963). *Industrial Scheduling*. International series in management. Prentice-Hall.

Rodler, P., Teppan, E., and Jannach, D. (2021). Randomized problem-relaxation solving for over-constrained schedules. In Bienvenu, M., Lakemeyer, G., and Erdem, E., editors, *Proc. of the 18th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'21)*, pages 696–701.

Sadeh, N. M. and Fox, M. S. (1996). Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence*, 86:1–41.

Sycara, K., Roth, S. F., Sadeh, N., and Fox, M. S. (1991). Distributed constrained heuristic search. *IEEE Transactions on systems, man, and cybernetics*, 21(6):1446–1461.

Taillard, E. (1993). Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285.

Teppan, E. and Col, G. (2020). *Genetic Algorithms for Creating Large Job Shop Dispatching Rules*, pages 121–140.

Teppan, E. C. (2018a). Dispatching rules revisited-a large scale job shop scheduling experiment. In *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 561–568.

Teppan, E. C. (2018b). Light weight generation of dispatching rules for large-scale job shop scheduling. In *Proc. of the 21st Int'l Conf on Artificial Intelligence (ICAI'19)*, pages 330–333.

Teppan E.C., F. G. (2018). Heuristic constraint answer set programming for manufacturing problems. In Hatzilygeroudis, I.and Palade, V., editor, *Advances in Hybridization of Intelligent Methods. Smart Innovation, Systems and Technologies*, volume 85.