Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning
Applications and Systems Track

Randomized Problem-Relaxation Solving for Over-Constrained Schedules

Patrick Rodler , Erich Teppan , Dietmar Jannach
Alpen-Adria Universität Klagenfurt
patrick.rodler@aau.at, erich.teppan@aau.at, dietmar.jannach@aau.at

Abstract

Optimal production planning in the form of job shop scheduling problems (JSSP) is a vital problem in many industries. In practice, however, it can happen that the volume of jobs (orders) exceeds the production capacity for a given planning horizon. A reasonable aim in such situations is the completion of as many jobs as possible in time (while postponing the rest). We call this the Job Set Optimization Problem (JOP). Technically, when constraint programming is used for solving JSSPs, the formulated objective in the constraint model can be adapted so that the constraint solver addresses JOP, i.e., searches for schedules that maximize the number of timely finished jobs. However, also highly specialized solvers which proved very powerful for JSSPs may struggle with the increased complexity of the reformulated problem and may fail to generate a JOP solution given practical computation timeouts. As a remedy, we suggest a framework for solving multiple randomly modified instances of a relaxation of the JOP, which allows to gradually approach a JOP solution. The main idea is to have one module compute subset-minimal job sets to be postponed, and another one effectuating that random job sets are found. Different algorithms from literature can be used to realize these modules. Using IBM's cutting-edge CP Optimizer suite, experiments on well-known JSSP benchmark problems show that using the proposed framework consistently leads to more scheduled jobs for various computation timeouts than a standalone constraint solver approach.

1 Introduction

The scheduling of jobs (Blazewicz et al. 2007) is a crucial task for production industries and several formal problem variants were defined that capture properties of different manufacturing environments. Many of those problems, like the open shop-, the flow shop- or the *Job Shop Scheduling Problem (JSSP)*, are NP-hard in the general case. Here, we focus on the JSSP due to its relevance to a wide range of industrial production fields (Drótos, Erdős, and Kis 2009; Sun et al. 2021). Different methods have been applied to approach JSSPs. Among those, constraint programming (CP) has a long and successful history and present-day CP solving systems are able to handle large-scale problem instances. However, today's often highly dynamic production regimes, supporting, e.g., make-to-order or lean production, lead to optimization problems on top of the underlying JSSPs which may significantly increase computation times.

One typical problem of that type arises when the set of orders (jobs) exceeds the current production capacities with respect to a given planning horizon (e.g., a week). Reasons for that to happen can be found in seasonal order fluctuations, unforeseen machine breakdowns, incoming high-priority orders of key accounts, and many more. In such a situation the producer is confronted with an over-constrained JSSP, i.e., it is not possible to work off all the orders in time and it must be decided which of the jobs to postpone. We call the task of finding a job set of maximal utility (e.g., revenue) that can be finished within a given planning horizon the *Job Set Optimization Problem (JOP)*, which is NP-hard (Baptiste 2003).

When using constraint programming, a straightforward approach to solving JOPs is to adapt the CP encoding of the over-constrained JSSP by adding an optimization statement, which effectuates that the utility of jobs scheduled and finished prior to a given deadline is maximized. Since however the JOP represents a hard problem on top of the JSSP, also the most powerful state-of-the-art CP solvers may struggle with the increased problem complexity.

We therefore propose a framework to tackle JOPs based on the observation that the problem of computing a subset-minimal job set to be postponed *(i)* is a relaxation of the JOP, *(ii)* can be solved using a linear number of CP solver calls, each for one JSSP decision problem, and *(iii)* is not directly supported in current CP solvers. The idea is to compute multiple such subset-minimal job sets in a random way, thus intuitively taking a random sample in a solution space that covers all JOP solutions. By always storing the best found solution, the framework allows to successively approach a JOP solution. Each module in the framework tackles a well-understood and well-investigated subproblem of JOP and is viewed as a black-box which can be implemented by different specialized state-of-the-art algorithms.

In evaluations on Taillard's well-known JSSP benchmark suite (Taillard 1993), we show that, given different practical computation timeouts, the use of the proposed framework consistently leads to solutions involving more in-time finished jobs than a standalone CP solver approach.

2 Problem Definition and Approach

The *Job Shop Scheduling Problem (JSSP)* (Blazewicz et al. 2007) is defined as follows:

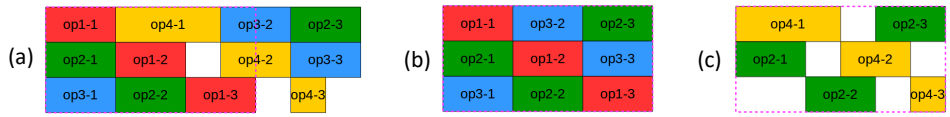

Figure 1: In the plots, the k-th row refers to machine k, the horizontal axis is the time line starting from zero and "op$j$-i" refers to the i-th operation of job j. (a) JSSP solution for all jobs; (b) JOP/JMP solution ($\Delta = \{\text{job 4}\}$); (c) JMP solution ($\Delta = \{\text{job 1, job 3}\}$).

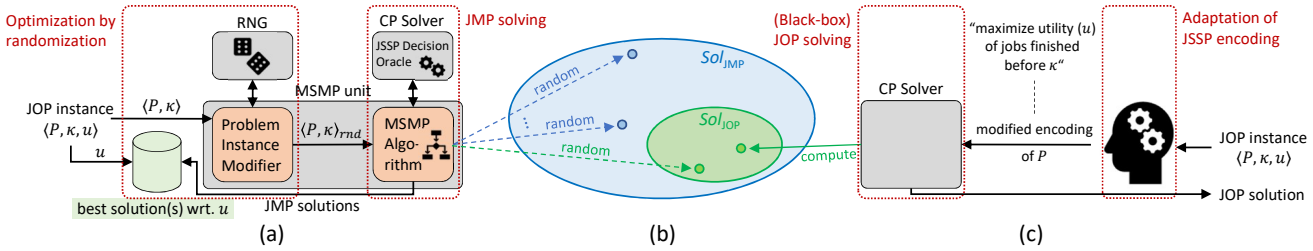

Figure 2: (a) Proposed framework for JOP based on decomposing JOP into subproblems (i) JMP solving and (ii) optimization via randomization. (b) Illustration of solution spaces for JOP and JMP. (c) A direct CP approach to JOP. Remarks: Both approaches require a CP encoding of the JSSP P. Gray rectangles denote black-boxes and can be realized by different suitable algorithms. "RNG" = random number generator.

Definition 1 (JSSP). ***Given:*** *A set of machines M and a set of jobs J where every job $j \in J$ consists of an ordered set of operations $Ops_j = \{op_1, \ldots, op_{k_j}\}$ and each $op \in Ops_j$ has a length $l_{op} \in \mathbb{N}$ and has to be executed on a particular machine $m_{op} \in M$,* ***Find:*** *A schedule σ which maps every operation op in $AllOps := \bigcup_{j \in J} Ops_j$ to a start time $\sigma(op) \in \mathbb{N}_0$ on its respective machine m_{op} such that*

1. *$\sigma(op_{x+1}) \geq \sigma(op_x) + l_{op_x}$ for each pair of successive operations op_x, op_{x+1} in Ops_j for all jobs $j \in J$,*
2. *on each machine in M, a next operation may only start after the current operation has been finished, i.e., for every pair of operations $op, op' \in AllOps$ with $m_{op} = m_{op'}$, either $\sigma(op) \geq \sigma(op') + l_{op'}$ or $\sigma(op') \geq \sigma(op) + l_{op}$ and*
3. *completion time is minimized, i.e., among all schedules, σ has minimal $time(\sigma) := \max_{op \in AllOps}(\sigma(op) + l_{op})$.*

The decision version of a JSSP involves a deadline $\kappa \in \mathbb{N}$ as an additional input, and asks whether there is a schedule σ satisfying criteria 1 and 2 such that $time(\sigma) \leq \kappa$. This decision version is NP-complete (for $|M| \geq 2$) (Garey, Johnson, and Sethi 1976) which is why JSSP is NP-hard in general.

Example: Assume a set of four jobs, each consisting of three operations, that should be scheduled on three machines. Let the lengths of all operations of the first three jobs be 2, while the three operations of the fourth job have lengths (3,2,1). Finally, let the machine numbers where the operations have to be processed be (1,2,3) for both jobs 1 and 4, (2,3,1) for job 2, and (3,1,2) for job 3. A solution for this JSSP instance is shown in Fig. 1(a) and has a completion time of 9. □

If not all jobs can be scheduled until a given deadline, a reasonable remedy is to postpone or reject a set of jobs that implies the least negative effect on a given utility function (e.g., company revenue), i.e., the utility of the scheduled jobs should be maximal. We refer to this problem as the *Job Set Optimization Problem (JOP)* and define it as follows:

Definition 2 (JOP). ***Given:*** *A deadline $\kappa \in \mathbb{N}$, a JSSP instance P with job set J, and a utility function u that assigns a utility $u_j \in \mathbb{N}$ to each job $j \in J$.* ***Find:*** *Some $\Delta \subseteq J$ such that* (i) *P with the reduced job set $J \setminus \Delta$ has a solution schedule σ with $time(\sigma) \leq \kappa$, and* (ii) *there is no other such $\Delta' \subseteq J$ that satisfies $\sum_{j \in J \setminus \Delta'} u_j > \sum_{j \in J \setminus \Delta} u_j$.*

A straightforward approach to solving JOPs is to rely on highly-optimized CP solvers. Given a JOP instance $\langle P, \kappa, u \rangle$, first, the CP encoding of the original JSSP P is adapted by adding an optimization criterion which achieves that the utility sum (as per u) of the jobs finished before κ is maximized. Second, this adapted encoding is fed into the CP solver to generate a JOP solution. However, as we found, even state-of-the-art CP solvers may fail to compute a JOP solution within practical timeouts of one or two hours. Therefore, we devised an alternative approach to solving JOPs, which is based on the following observations:

(Obs1) Since the job utilities u_j are positive numbers, each JOP solution Δ must be a *subset-minimal* job set that satisfies JOP criterion (i). Thus, by modifying JOP criterion (ii) accordingly, we obtain a relaxed problem where the goal is to schedule a set of jobs that is maximal (w.r.t. $\subseteq$). We call this the *Job Set Maximization Problem (JMP)*:

Definition 3 (JMP). ***Given:*** *A deadline $\kappa \in \mathbb{N}$ and a JSSP instance P with job set J.* ***Find:*** *Some $\Delta \subseteq J$ such that* (i) *P with the reduced job set $J \setminus \Delta$ has a solution schedule σ with $time(\sigma) \leq \kappa$, and* (ii) *there is no other such $\Delta' \subseteq J$ that satisfies $\Delta' \subset \Delta$.*

Example (cont'd): Reconsider our JSSP instance and assume that the deadline $\kappa = 6$ (dashed line in Fig. 1) is given. Further, let all jobs have equal utility. The (only) JOP solution for this instance is shown in Fig. 1(b). Another JMP (but not JOP) solution is presented in Fig. 1(c). □

(Obs2) Every JOP solution is also a JMP solution. Fig. 2(b) schematically illustrates the solution spaces Sol_X of both problems $X \in \{\text{JMP}, \text{JOP}\}$. That is, among k randomly selected JMP solutions, at least one is a JOP solution with a probability of $1 - q^k$ where $q = 1 - |Sol_{\text{JOP}}|/|Sol_{\text{JMP}}|$. E.g., if every tenth JMP solution is a JOP solution and $k = 20$, the chance of finding a JOP solution is already 88 %.

(Obs3) JMP is a manifestation of the MSMP (*M*inimal *S*et w.r.t. a *M*onotone *P*redicate) Problem (Marques-Silva, Janota, and Belov 2013; Marques-Silva, Janota, and Mencia 2017). To see this, let the predicate $p(\Delta)$ be defined over a JMP instance $\langle P, \kappa \rangle$ as *"For job set $J \setminus \Delta$, there exists a schedule σ with $time(\sigma) \leq \kappa$"*. This predicate is monotone, i.e., $p(\Delta)$ implies $p(\Delta')$ whenever $\Delta \subseteq \Delta'$. In other words, if there is a schedule such that all jobs $J \setminus \Delta$ are finished in time, then there must also be a schedule which allows to finish as many or fewer jobs $J \setminus \Delta'$. By Def. 3, JMP exactly calls for a minimal set Δ subject to p. Hence, any *sufficiently general* (Rodler 2020) MSMP algorithm can be used to solve JMP. Examples of such algorithms are QuickXplain (Junker 2004), Progression (Marques-Silva, Janota, and Belov 2013), or Inv-QX (Shchekotykhin et al. 2014). In case of JMP, these algorithms require a worst-case linear (in $|J|$) number of calls to a CP solver that decides the predicate p (or, equivalently, solves the decision version of JSSP).
(Obs4) JMP is easier than JOP. The reason is that JOP essentially requires finding a *best* JMP solution w.r.t. a utility function u over J. Thus, a linear number of calls to an NP oracle, as for JMP, generally does not even allow to verify a given JOP solution, let alone to compute one.
(Obs5) CP solvers typically do not support JMP solving.

Approach. Based on the analysis above, we propose a framework to address JOPs by solving multiple randomly modified JMP instances where the best JMP solution (w.r.t. utility u) is stored throughout the process. Essentially, the idea is to randomly sample elements from the JMP solution space which covers all JOP solutions. Our framework has three modules: a *CP solver* for solving decision versions of JSSP, cf. JMP criterion (i); an *MSMP unit* for finding subset-minimal job sets, cf. JMP criterion (ii); and, a *random number generator* enabling the optimization by generating *multiple random* JMP solutions, cf. JOP criterion (ii).

Compared to a direct CP approach, which can be seen as tackling two problems at once, i.e., the (implicit) subset-minimality and the optimal utility of the JOP solution, our framework achieves a disentanglement of these two problems by extracting the (efficient and well understood) MSMP reasoning from the solver and leaving to the solver the role of deciding a polynomial number of JSSP instances (for which state-of-the-art solvers are optimized).

The iterative solution process using the described framework (cf. Fig. 2(a)) is: *(1)* Given a JOP instance $\langle P, \kappa, u \rangle$, forward the JMP instance $\langle P, \kappa \rangle$ to the MSMP unit. *(2)* Depending on the particular used MSMP algorithm, use a suitable random modification $\langle P, \kappa \rangle_{rnd}$ of the JMP instance to obtain a random JMP solution in the next step. *(3)* Solve the JMP instance $\langle P, \kappa \rangle_{rnd}$ using the MSMP algorithm, which will involve $O(|J|)$ JSSP decision queries to a CP solver. *(4)* Based on the utility function u, update the best solution. *(5)* Continue Steps (2)–(4) until some stop criterion (e.g., a timeout or a required solution quality) is met.

Remarks:[1] *(a)* Solution quality w.r.t. utility will monotonically increase throughout the solving process. *(b)* Our approach is complete, i.e., it will yield a JOP solution given sufficient time and memory, and a full-cycle random number generator, cf. (Schrage 1979). *(c)* Each module in our framework is viewed as a black-box and can be realized by different algorithms; this allows our approach to profit from latest research advancements in the JSSP and MSMP fields. *(d)* No information exchange is required between different iterations; thus, our approach enables efficient multi-threaded implementations, which (over)compensates the potential generation of duplicate JMP solutions (cf. results in Sec. 3). *(e)* Our approach does not require to manually adapt the CP encoding of a given JSSP instance to a JOP encoding.

[1] Cf. online material: http://isbi.aau.at/ontodebug/evaluation

3 Evaluation

Implementation. To evaluate our approach, we developed a Java-based proof-of-concept implementation. As a CP solver, we rely on IBM's CP Optimizer (CPO), which is currently one of the most efficient solvers for scheduling problems, in particular JSSPs (Da Col and Teppan 2019a). For calculating JMP solutions, i.e., as an MSMP algorithm, we use Inv-QX (Shchekotykhin et al. 2014). Randomization is done by shuffling the jobs before feeding them to Inv-QX, so that random JMP solutions are produced (Narodytska et al. 2018; Rodler and Elichanova 2020). Inv-QX uses CPO as an oracle for JSSP decision problems, i.e., to test whether a given set of jobs can be accomplished within a deadline.

As a baseline in our experiments, we also use CPO, but this time to solve a *direct* encoding of the JOP, corresponding to a MaxCSP (Minoux and Mavrocordatos 2009) problem (cf. Fig. 2(c)). Specifically, we adapted the JSSP encoding provided in the documentation of IBM CPO in a way that a deadline constraint for a job j (stating that a delay for j is not allowed) is active when an associated integer variable v_j in the range of $[0..1]$ is set to 1. Correspondingly, the optimization criterion is formulated to maximize the sum over v_i where $i \in Jobs$. Calling CPO given this encoding will lead to the computation of a JOP solution.
Dataset. In our evaluations, we drew upon a dataset based on a subset of the widely-used benchmark problems of (Taillard 1993). A key plus of these benchmarks is that the optimal completion times are known, which allows us to systematically control how much a JOP problem instance is over-constrained. That is, given a problem instance P and the corresponding optimal completion time κ^*, a JOP instance can be created by setting $\kappa = r \cdot \kappa^*$ with $r < 1$. Clearly, the smaller r is, the smaller the set of jobs accomplishable given a maximum completion time κ will tend to be. To explore various scenarios that might arise for a company in case production deadlines prevent it to cover all orders, we specified five completion time levels $r \in \{0.95, 0.9, 0.85, 0.8, 0.75\}$. Thus, our test dataset consists of 20 (Taillard instances) $\times$ 5 (completion time levels r) = 100 JOP problem instances.
Experiment. We ran our experiments with timeouts of 1 and 2 hours, respectively. Such timeouts allow for frequent intra-day recalculations (re-scheduling) to react quickly to dynamics in industrial scenarios. Since the benchmark problems do not include individual job utilities, we set them uniformly in our tests, which amounts to maximizing the number of scheduled jobs. Our approach was configured to use 8 worker threads in parallel for independent solution search

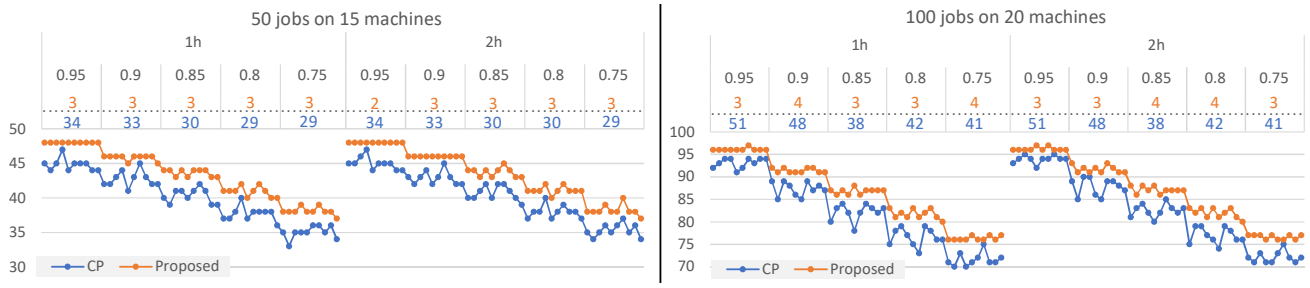

Figure 3: Evaluation results: Each data point indicates the number of accomplished jobs (y-axis) of the CP approach (blue) versus the proposed approach (orange) per benchmark problem instance, for different timeouts (1h, 2h) and different values $(0.95, \ldots, 0.75)$ of r (x-axis). The orange / blue numbers along the x-axis indicate the average number (per value of r) of (within-timeout) generated JMP solutions (proposed approach) / intermediate solutions towards JOP (CP approach), each of which improved the current best solution (w.r.t. utility).

(memorizing only the best solution). Also IBM's CPO was configured to instantiate the same number of worker threads.

Results. Fig. 3 shows our results.[2] For both timeouts, we can observe that our approach consistently yields better schedules with more finished jobs than the direct CP encoding. In numbers, our approach enables to schedule an avg. of 8 % and up to 15 % more jobs for the (50 jobs, 15 machines) instances, as well as 5 % (avg.) and 13 % (max.) more jobs for (100 jobs, 20 machines). Moreover, the proposed approach is faster as it always produces better results (up to 12 % more jobs) within 1 hour than the direct CP implementation is able to do in 2 hours. Notably, our approach needed one order of magnitude fewer internal solution improvement steps towards the final solution than the direct approach, by requiring each intermediate result to be a solution to JMP and thus ruling out obvious non-JOP-solutions.

4 Related Work

Scheduling is a well-studied topic, in terms of both complexity results and algorithmic approaches (Garey, Johnson, and Sethi 1976; Brucker, Sotskov, and Werner 2007; Barták, Salido, and Rossi 2010; Brucker, Jurisch, and Sievers 1994; Stecco, Cordeau, and Moretti 2008; Ku and Beck 2016; Bożejko et al. 2017; Danna and Perron 2003; Sadegheih 2006). See also (Potts and Strusevich 2009) for an overview.

Approaches related to our goal of relaxing over-constrained JSSPs are (weighted) MaxCSP methods, e.g., (Schiex et al. 1995; Domshlak et al. 2009; Bakker et al. 1993), which target the minimization of the number (weight) of violated constraints in a CSP. However, for this work we needed a state-of-the-art CP solver that is specialized on scheduling problems in particular. For that, IBM's CP Optimizer (CPO) proved to be the currently strongest available solver (Da Col and Teppan 2019a). An open source alternative, not as strong for scheduling as CPO but still competitive, are Google's OR Tools (Da Col and Teppan 2019b).

Due to the equal expressivity of CSP and SAT languages (Walsh 2000), problem instances from one class can be translated to the other (Argelich and Manya 2006; De Givry et al. 2003) using sophisticated encoding techniques (Argelich et al. 2008; Tamura et al. 2009). Hence, in principle also MaxSAT solvers, e.g., (Morgado et al. 2013; Alsinet, Manya, and Planes 2005; Pipatsrisawat and Darwiche 2007; Xing and Zhang 2005), could be used in place of the CP solver in our framework. However, this requires an initial non-trivial extra SAT encoding step.

JOP is related to the Throughput Maximization Problem (TMP) (Berman and DasGupta 2000; Bar-Noy et al. 2001) which also seeks to maximize the weight of jobs finished in time. However, the problems differ in that a JOP instance prescribes one *particular* machine for each job, while a TMP instance allows each job to be performed on *any* machine.

The direct and problem-decomposition approaches, as shown in Fig. 2, are conceptually similar to two types of MaxSAT solvers, ones that solve a problem instance directly versus others that iteratively call a SAT solver as inference engine (Koshimura et al. 2012; Morgado et al. 2013). Whereas our proposed framework views all involved algorithms as black-boxes, relies on a randomization of problem instances, and keeps iterations completely independent of one another, the latter types of MaxSAT approaches often assume that oracles provide information beyond the answer to a decision problem (Fu and Malik 2006), couple oracle calls with additional reasoning techniques (Martins, Manquinho, and Lynce 2012), or reuse information from earlier iterations (Ansótegui, Bonet, and Levy 2009).

Finally, there are works that approximate optimization problems such as MaxSAT by enumerating subset-minimal sets (Marques-Silva, Heras, et al. 2013; Mencia et al. 2015; Terra-Neves et al. 2018). Despite sharing the general idea with these works, our framework differs by using randomized, independent and parallel computations, requires no SAT encoding or SAT-specific techniques, and focuses on scheduling (different benchmarks, existing solvers, etc.).

5 Conclusion

We proposed a framework for resolving over-constrained scheduling problems based on a randomized computation of solutions to a relaxed problem. The framework is generic in that its modules are black-boxes that can be realized by different specialized algorithms from well-understood research fields. Evaluations on a well-known benchmark dataset attest that the suggested approach consistently outperforms a cutting-edge constraint solver for scheduling problems.

[2] Code and raw data: http://isbi.aau.at/ontodebug/evaluation

Acknowledgments

This work is an extension of (Rodler and Teppan 2020) and was supported by the Austrian Science Fund (FWF), contract P-32445-N38. We thank the anonymous referees for their valuable comments.

References

Alsinet, T.; Manya, F.; and Planes, J. 2005. Improved exact solvers for weighted Max-SAT. In *International Conference on Theory and Applications of Satisfiability Testing*, 371–377. Springer.

Ansótegui, C.; Bonet, M. L.; and Levy, J. 2009. Solving (weighted) partial MaxSAT through satisfiability testing. In *International Conference on Theory and Applications of Satisfiability Testing*, 427–440. Springer.

Argelich, J., and Manya, F. 2006. Exact Max-SAT solvers for over-constrained problems. *Journal of Heuristics* 12(4):375–392.

Argelich, J.; Cabiscol, A.; Lynce, I.; and Manya, F. 2008. Modelling Max-CSP as partial Max-SAT. In *International Conference on Theory and Applications of Satisfiability Testing*, 1–14. Springer.

Bakker, R. R.; Dikker, F.; Tempelman, F.; and Wognum, P. M. 1993. Diagnosing and solving over-determined constraint satisfaction problems. In *International Joint Conference on Artificial Intelligence*, volume 93, 276–281.

Baptiste, P. 2003. On minimizing the weighted number of late jobs in unit execution time open-shops. *European Journal of Operational Research* 149(2):344–354.

Bar-Noy, A.; Guha, S.; Naor, J.; and Schieber, B. 2001. Approximating the throughput of multiple machines in real-time scheduling. *SIAM Journal on Computing* 31(2):331–352.

Barták, R.; Salido, M.; and Rossi, F. 2010. New trends in constraint satisfaction, planning, and scheduling: a survey. *The Knowledge Engineering Review* 25(3):249—279.

Berman, P., and DasGupta, B. 2000. Multi-phase algorithms for throughput maximization for real-time scheduling. *Journal of Combinatorial Optimization* 4(3):307–323.

Blazewicz, J.; Ecker, K.; Pesch, E.; Schmidt, G.; and Weglarz, J. 2007. *Handbook on Scheduling: Models and Methods for Advanced Planning (International Handbooks on Information Systems)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Bożejko, W.; Gnatowski, A.; Pempera, J.; and Wodecki, M. 2017. Parallel tabu search for the cyclic job shop scheduling problem. *Computers and Industrial Engineering* 113:512 – 524.

Brucker, P.; Jurisch, B.; and Sievers, B. 1994. A branch and bound algorithm for the job-shop scheduling problem. *Discrete Applied Mathematics* 49(1):107 – 127.

Brucker, P.; Sotskov, Y. N.; and Werner, F. 2007. Complexity of shop-scheduling problems with fixed number of jobs: a survey. *Mathematical Methods of Operations Research* 65(3):461–481.

Da Col, G., and Teppan, E. C. 2019a. Industrial size job shop scheduling tackled by present day CP solvers. In *International Conference on Principles and Practice of Constraint Programming*, 144–160. Springer.

Da Col, G., and Teppan, E. C. 2019b. Google vs IBM: A constraint solving challenge on the job-shop scheduling problem. In *International Conference on Logic Programming*.

Danna, E., and Perron, L. 2003. Structured vs. unstructured large neighborhood search: A case study on job-shop scheduling problems with earliness and tardiness costs. In Rossi, F., ed., *Principles and Practice of Constraint Programming*, 817–821. Berlin, Heidelberg: Springer Berlin Heidelberg.

De Givry, S.; Larrosa, J.; Meseguer, P.; and Schiex, T. 2003. Solving Max-SAT as weighted CSP. In *International Conference on Principles and Practice of Constraint Programming*, 363–376. Springer.

Domshlak, C.; Rossi, F.; Venable, K. B.; and Walsh, T. 2009. Reasoning about soft constraints and conditional preferences: complexity results and approximation techniques. *arXiv preprint arXiv:0905.3766*.

Drótos, M.; Erdős, G.; and Kis, T. 2009. Computing lower and upper bounds for a large-scale industrial job shop scheduling problem. *European Journal of Operational Research* 197(1):296–306.

Fu, Z., and Malik, S. 2006. On solving the partial Max-SAT problem. In *International Conference on Theory and Applications of Satisfiability Testing*, 252–265. Springer.

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.

Junker, U. 2004. QuickXplain: Preferred Explanations and Relaxations for Over-Constrained Problems. In *National Conference on Artificial Intelligence*, volume 3, 167–172. AAAI Press / The MIT Press.

Koshimura, M.; Zhang, T.; Fujita, H.; and Hasegawa, R. 2012. QMaxSAT: A partial Max-SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation* 8(1-2):95–100.

Ku, W. Y., and Beck, J. C. 2016. Mixed integer programming models for job shop scheduling: A computational analysis. *Computers and Operations Research* 73:165 – 173.

Marques-Silva, J.; Janota, M.; and Belov, A. 2013. Minimal sets over monotone predicates in Boolean formulae. In *International Conference on Computer Aided Verification*, 592–607. Springer.

Marques-Silva, J.; Janota, M.; and Mencia, C. 2017. Minimal sets on propositional formulae: Problems and reductions. *Artificial Intelligence* 252:22–50.

Marques-Silva, J.; Heras, F.; Janota, M.; Previti, A.; and Belov, A. 2013. On Computing Minimal Correction Subsets. In *International Joint Conference on Artificial Intelligence* 615–622.

Martins, R.; Manquinho, V.; and Lynce, I. 2012. On parti-

tioning for maximum satisfiability. In *European Conference on Artificial Intelligence*. IOS Press. 913–914.

Mencia, C.; Previti, A.; and Marques-Silva, J. 2015. Literal-Based MCS Extraction. In *International Joint Conference on Artificial Intelligence* 1973–1979.

Minoux, M., and Mavrocordatos, P. 2009. *Maximum Constraint Satisfaction: Relaxations and Upper Bounds*. In *Encyclopedia of Optimization*. Boston, MA: Springer US. 1981–1991.

Morgado, A.; Heras, F.; Liffiton, M.; Planes, J.; and Marques-Silva, J. 2013. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints* 18(4):478–534.

Narodytska, N.; Bjørner, N.; Marinescu, M.; and Sagiv, M. 2018. Core-Guided Minimal Correction Set and Core Enumeration. In *International Joint Conference on Artificial Intelligence*. 1353–1361.

Pipatsrisawat, K., and Darwiche, A. 2007. Clone: Solving weighted Max-SAT in a reduced search space. In *Australasian Joint Conference on Artificial Intelligence*, 223–233. Springer.

Potts, C. N., and Strusevich, V. A. 2009. Fifty years of scheduling: a survey of milestones. *Journal of the Operational Research Society* 60(1):S41–S68.

Rodler, P., and Elichanova, F. 2020. Do we really sample right in model-based diagnosis? In *International Workshop on Principles of Diagnosis* (Corr abs/2009.12178).

Rodler, P., and Teppan, E. 2020. The Scheduling Job-Set Optimization Problem: A Model-based Diagnosis Approach In *International Workshop on Principles of Diagnosis* (Corr abs/2009.11142).

Rodler, P. 2020. Understanding the QuickXplain Algorithm: Simple Explanation and Formal Proof. *CoRR* abs/2001.01835.

Sadegheih, A. 2006. Scheduling problem using genetic algorithm, simulated annealing and the effects of parameter values on GA performance. *Applied Mathematical Modelling* 30(2):147 – 154.

Schiex, T.; Fargier, H.; Verfaillie, G.; et al. 1995. Valued constraint satisfaction problems: Hard and easy problems. *International Joint Conference on Artificial Intelligence*, 631–639.

Schrage, L. 1979. A more portable Fortran random number generator. *ACM Transactions on Mathematical Software* 5(2):132–138.

Shchekotykhin, K.; Friedrich, G.; Rodler, P.; and Fleiss, P. 2014. Sequential diagnosis of high cardinality faults in knowledge-bases by direct diagnosis generation. In *European Conference on Artificial Intelligence*, 813–818.

Stecco, G.; Cordeau, J.-F.; and Moretti, E. 2008. A branch-and-cut algorithm for a production scheduling problem with sequence-dependent and time-dependent setup times. *Computers and Operations Research* 35(8):2635–2655.

Sun, Y.; Chung, S.-H.; Wen, X.; and Ma, H.-L. 2021. Novel robotic job-shop scheduling models with deadlock and robot movement considerations. *Transportation Research Part E: Logistics and Transportation Review* 149:102273.

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

Tamura, N.; Taga, A.; Kitagawa, S.; and Banbara, M. 2009. Compiling finite linear CSP into SAT. *Constraints* 14(2):254–272.

Terra-Neves, M.; Lynce, I.; and Manquinho, V. 2018. Multi-Objective Optimization Through Pareto Minimal Correction Subsets. In *International Joint Conference on Artificial Intelligence* 5379–5383.

Walsh, T. 2000. SAT v CSP. In *International Conference on Principles and Practice of Constraint Programming*, 441–456. Springer.

Xing, Z., and Zhang, W. 2005. MaxSolver: An efficient exact algorithm for (weighted) maximum satisfiability. *Artificial intelligence* 164(1-2):47–80.