Automatic Generation of Dispatching Rules for Large Job Shops by Means of Genetic Algorithms

Erich C. Teppan[1] and Giacomo Da Col[1]

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

Abstract. Generating optimized large-scale production plans is an important open problem where even small improvements result in significant savings. Application scenarios in the semiconductor industry comprise thousands of machines and hundred thousands of job operations and are therefore among the most challenging scheduling problems regarding their size. In this paper we present a novel approach for automatically creating composite dispatching rules, i.e. heuristics for job sequencing, for makespan optimization in such large-scale job shops. The approach builds on the combination of event-based simulation and genetic algorithms. We introduce a new set of benchmark instances with proven optima that comprise up to 100000 operations to be scheduled on up to 1000 machines. With respect to this large-scale benchmark, we present the results of an experiment comparing well-known dispatching rules with automatically created composite dispatching rules produced by our system. It is shown that the proposed system is able to come up with highly effective dispatching rules such that makespan reductions of up to 38% can be achieved, and in fact, often near optimal or even optimal schedules can be produced.

Keywords: job shop scheduling · optimization · dispatching rules · simulation · large-scale

1 Introduction

The scheduling of jobs [2] is an important task in almost all production systems in order to optimize various objectives such as resource consumption, makespan (time to finish all products), tardiness (lateness of products), or flow time. Driven by the demands of the semiconductor industry, our general aim is the design of practically applicable algorithms for job shop scheduling problems for domains comprising thousands of machines and hundred thousands of job operations.

In real world domains like semiconductor manufacturing, common problem instances for a weekly workload are of the order of 10^4 operations on 10^2 machines in the back-end, i.e. where the products are made ready for shipping, and 10^5 operations on 10^3 machines in the front-end, i.e. where the chips are actually produced. To the best of our knowledge, the size of such large-scale job shop

scheduling problem instances go beyond existing benchmarks. For example, the well-known benchmark targeting on makespan optimization from [11] comprises up to 50 jobs and 20 machines, which results in 1000 job operations in total. The famous Taillard benchmark [24] contains job shop instances of sizes up to 100 jobs on 20 machines which results in 2000 job operations. When focusing on tardiness optimization, somewhat bigger instances can be found. For example, the benchmark described in [29] comprises up to 20000 job operations. In this paper we introduce a benchmark comprising instances with up to 100000 operations to be scheduled on up to 1000 machines.

Many approaches have been used to solve scheduling problems. Aiming at optimal solutions, constraint based approaches (e.g. [1, 22]), branch and bound (e.g. [4]), branch and cut (e.g. [23]) or mixed integer programming (e.g. [18]) have a long and successful history. Another declarative solving approach that was tried with so far limited success is answer set programming [12, 13]. However, hybridized approaches between answer set- and constraint programming (see e.g. [27, 25]) show some potential for dealing even with large-scale scheduling instances [26, 8]. Calculating optimal schedules is typically far out of reach for large-scale domains. For near optimum solutions, current state of the art approaches are based on tabu and large neighborhood search (e.g. [28, 3, 10]), simulated annealing or genetic algorithms (e.g. [21]). Such local search methods have shown to be effective whilst exhibiting low memory consumption. Though, for the successful application of local search methods on large-scale production environments the quality of the initial schedule that is iteratively optimized by some local search methodology is most important. Starting with, for example, a random schedule that is 200% off the optimum would not be sufficient since it would take much too long to reach a near optimum schedule that is acceptable. Note that because of the highly dynamic production regimes and the often occurring changes (e.g. machine break down) in nowadays production environments rescheduling is needed every few minutes. Yet, if starting with a schedule of good quality, local search methods can be applied to squeeze out the still remaining potential of the production environment.

One widely employed state-of-the-art technique for dealing with large and complex scheduling problems in nowadays manufacturing environments is the application of dispatching rules (e.g. [15, 17, 19]). Dispatching rules are greedy heuristics for step-wise deciding which is the operation to be processed next by a machine. One big advantage of dispatching rules is that they can be computed typically in linear time. There are basically four ways for using dispatching rules:

1. Dispatching rules are directly applied by operator staff for steering the production process. As soon as a machine becomes idle, the operator decides which of the runnable operations in the dispatch list of the machine is to be loaded next.
2. Schedules are built by doing simulations based on different dispatching rules. A schedule, which is found to be good enough, is then carried out.

3. An initial schedule of good quality is created by means of dispatching rules and iteratively optimized by local search methods. The optimized schedule is then carried out.
4. In a search framework dispatching rules can be used as search heuristics.

Hence, a deep understanding of dispatching rules is crucial for many different scheduling approaches and scenarios. Furthermore, the fully automatic generation of effective dispatching rules is a highly important open question. In this paper we focus on the automatic creation of dispatching rules for the optimization of the makespan (i.e. minimization of time needed for accomplishing all operations) in very large job shops.

The job shop scheduling problem (JSP) is among the most famous $\mathcal{NP}$-hard [14] combinatorial problems and can be defined as follows:

- Given is a set $M = \{machine_1, \ldots, machine_m\}$ of machines and a set $J = \{job_1, \ldots, job_j\}$ of jobs.
- Each job $j \in J$ consists of a sequence of operations $O_j = \{j_1, \ldots, j_{l_j}\}$ whereby j_{l_j} is the last operation of job j.
  Practically, jobs can be interpreted as products and operations can be interpreted as their production steps. With respect to 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.
- Each operation o has an operation length $length_o \in \mathbb{N}$.
- Each operation o is assigned to a machine $machine_o \in M$ by which it is processed.
- A (consistent and complete) schedule consists of a starting time $start_o$ for each operation o such that:
  - An operation's successor starts after the operation has been finished, i.e. with respect to a job j and the operations j_i and j_{i+1}:
    - $start_{j_{i+1}} \geq start_{j_i} + length_{j_i}$
  - Operations processed by the same machine are non-overlapping, i.e. with respect to two operations $o1 \neq o2$ with $machine_{o1} = machine_{o2}$:
    - $start_{o1} \neq start_{o2}$
    - $start_{o1} < start_{o2} \rightarrow start_{o1} + length_{o1} \leq start_{o2}$
- Makespan, i.e. the time period needed for processing all operations, is minimized. I.e.:
  - $\max\limits_{j \in J, o \in O_j} \{start_o + length_o\} \rightarrow min$

The JSP is a special case of scheduling in manufacturing lines, in particular wafer fab scheduling in the semiconductor domain (see Figure 1). The flexible JSP is a direct generalization of the JSP. In the flexible JSP for an operation type there can be many machines. Yet, for the flexible JSP it still holds that a machine can only perform one operation type. Allowing also that a machine can perform various operation types, as it is possible to change the setup of

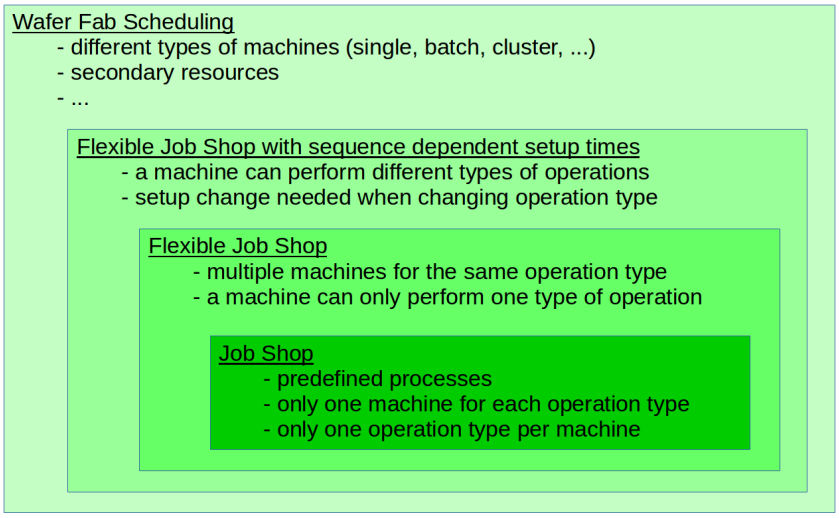

Fig. 1. How job shops relate to wafer fabs

machines, leads to the flexible JSP with sequence dependent setup times. Wafer fab scheduling introduces further concepts and constraints. For example, in wafer fabs it is common to have different types of machines. 'Single' machines only allow to perform one operation at a time. In (flexible) JSPs there are only 'single' machines. In wafer fabs there are also other types of machines, for example, batch machines that allow to perform a set of operations of the same type at the same time.

From the theoretical point of view, wafer fab scheduling is not more complex than the classic JSP. To see this, it is convenient to look at the decision problem versions of the optimization problems. The corresponding decision problems are about answering the question whether a solution below a certain makespan is existing. Having an oracle in $\mathcal{NP}$ to guess a solution it is obviously possible to check correctness of the solution in polynomial time. Thus, the corresponding decision problems are both in $\mathcal{NP}$[1]. Consequently, coping with the complexity of the JSP is the key for coping with large-scale wafer fab scheduling.

In the following, we present a novel approach for fully automatic generation of composite dispatching rules. Composite dispatching rules are based on underlying basic dispatching rules, like for example 'shortest-job first', and thus can be seen as hyper-heuristics [5]. The approach uses genetic algorithms, but in contrast to the approaches like [20] where genetic algorithms are used to directly find good schedules, in our approach the genetic algorithm is used to find a good dispatching rule, that in turn is used to produce the schedule. The proposed

[1] We can easily define an optimization procedure on top of the decision problems that calculates the optimal makespan by applying binary search that has only logarithmic complexity.

approach is conceptually similar to our previous work in [9], where we propose a genetic algorithm for learning hyper-heuristics for configuration problems.

After discussing the architecture of the proposed approach, we report on the design and the results of a large-scale job shop experiment in which we investigated the performance of our prototype system. The novel benchmark incorporates job shop problem instances that have proven optimal makespans and comprise up to 100000 operations to be scheduled on up to 1000 machines. Furthermore, they differ in the average number of operations per job, which has a big impact on the practical hardness of the problem instances. For all cases, we can show that the proposed approach outperforms any tested basic dispatching rule known from literature, and in fact, often (near-) optimal solutions can be found even for extremely large job shop problems.

2 Architecture

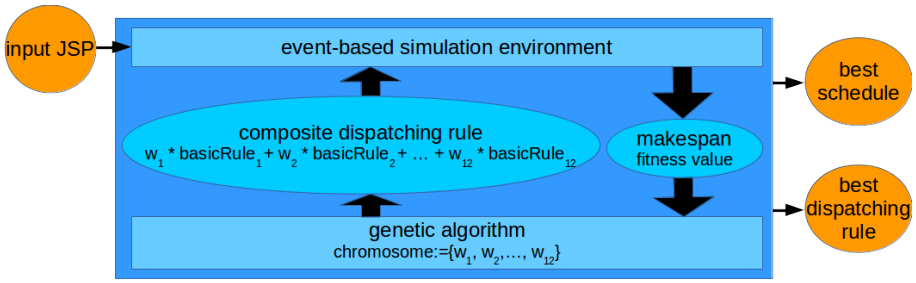

Fig. 2. System architecture

Figure 2 shows the overall architecture of our prototype system. The two main components are an event-based simulator and a genetic algorithm. For some input problem instance, the system produces two outcomes: the best schedule found and the corresponding dispatching rule.

2.1 Simulation Engine

The event-based simulation engine implements a simulation model in which each machine possesses one queue. Initially, only the starting operations, i.e. those without predecessors, are enqueued. After finishing an operation on a machine, the successor operation is enqueued on its machine. As soon as a machine becomes idle and its queue is not empty the next operation to be run is selected among the operations in the queue. This selection is based on a priority value calculated by a (composite) dispatching rule.

2.2 Basic Dispatching Rules

Composite dispatching rules in our system basically constitute weighted combinations of 12 basic rules. These basic dispatching rules are:

1. shortest processing time (SPT): Prefer the operation with the shortest processing time.
2. longest processing time (LPT): Prefer the operation with the longest processing time.
3. shortest job first (SJF): Prefer the operation of which the job possesses the shortest total processing time.
4. longest job first (LJF): Prefer the operation of which the job possesses the longest total processing time.
5. least total work remaining (LTWR): Prefer the operation for which the sum of the operation length and successor operations' lengths is the least.
6. most total work remaining (MTWR): Prefer the operation for which the sum of the operation length and successor operations' lengths is the most.
7. relative least total work remaining (RLTWR): As LTWR but normalized by total job length.
8. relative most total work remaining (RMTWR): As MTWR but normalized by total job length.
9. shortest waiting time (SWT): Prefer the operation that was enqueued as the last.
10. longest waiting time (LWT): Prefer the operation that was enqueued as the first.
11. urgency next (UN): Prefer the operation of which the successor operation's machine becomes idle as the first (based on the current queues).
12. urgency any (UA): Prefer the operation of which one of the successor operation's machine becomes idle as the first (based on the current queues).

SPT, LPT, SJF, LJF, LTWR, MTWR, SWT and LWT are well known (see, e.g. [16]). RLTWR and RMTWR are variants of LTWR and MTWR respectively. In RLTWR and RMTWR the total work remaining (TWR) is divided by the total job length. Hence, TWR is interpreted as the share of the remaining processing time of the operation's job relative to the total processing time of the job.

Also UN and UA are rule variants based on the well known WINQ rule [6, 7]. WINQ selects the operation of which the next successor operation will go on to the machine that has currently the least workload in the queue. The idea is to prevent idle times because of empty queues. UN follows the same idea but also takes into account the currently running operation of the successor's machine. Thus, with respect to some operation, UN calculates the nearest time point when the successor operation's machine can become idle.

UA is a generalization of UN calculating the earliest time point when one of the successor machines can become idle. I.e. whereas UN only considers the machine of the direct successor operation, UA takes all machines of the direct and indirect successor operations into account.

In order to purposefully combine the basic dispatching rules, every dispatching rule is implemented by a function that returns normalized priority values between 0 and 1. The semantic is: the higher the priority value the earlier an operation is processed by a machine.

2.3 Genetic Algorithm

For producing the different weight combinations the composite dispatching rules are based on, our system uses a genetic algorithm implemented with the Java Jenetics[2] library. The algorithm operates on genomes consisting of only a single chromosome that is composed of 12 integer genes. Each gene represents the weight for a basic dispatching rule and, thus, each genome represents the set of weights for a composite dispatching rule. The fitness value for a genome is determined by testing the resulting composite dispatching rule within the simulation engine with respect to some input problem instance. Hence, the simulation engine produces a schedule on the basis of the dispatching rule and the corresponding makespan represents the fitness value.

3 Experimental Setup

The goal of the experiment is to show the effectiveness of genetic algorithms in finding good composite dispatching rules for very large job shops. Hereby, we compare the composite dispatching rules generated by our system to each basic dispatching rule run on its own.

3.1 Algorithmic Configuration

The used genetic algorithm for producing the weights for the composite rules can be considered more or less standard. We use a 6-point crossover operator and a mutation rate of 10%. Genes (i.e. the weights) are restricted to integers between 0 and 10 (inclusive). The population size is set to 100 and the offspring fraction is 60%. For selecting genomes for recombination a tournament selector (tournament size = 10) is used. In the experiment we allowed a maximum of 100 generations.

3.2 Problem Instances

The problem instances used for the experiments described herein are on the one hand patterned on real world production scheduling problems, in particular in terms of size. On the other hand, the instances have known optimal makespans.

The creation process is as follows: First produce an optimal solution without (idle) holes for (1) a given number of job operations to be scheduled, (2) the

[2] jenetics.io

number of machines and (3) the desired optimal makespan. This is done by randomly partitioning the machines' time continuum into the predefined number of partitions. Each partition corresponds to the processing period of one operation. Consequently, the optimal makespan, average operation length (avg(length)), number of operations (#ops) and number of machines (#machines) relate conforming to $makespan = \frac{\#ops \times avg(opLength)}{\#machines}$. Based on such a partitioning, successor relations are randomly generated. Each partition has maximally one successor and/or predecessor such that the successor's starting time is greater than the predecessors finishing time. Figure 3 shows the principle.

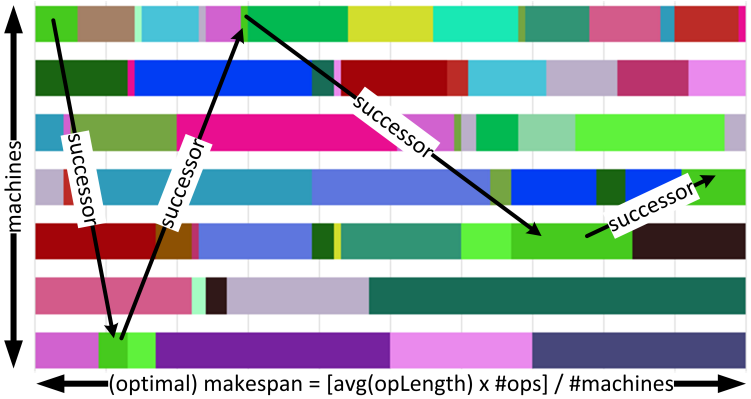

Fig. 3. Principle of instance generation

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

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

The two different generating approaches result in benchmark instances that are different in nature: (1) produces many jobs consisting of a small number of operations. We refer to this set of instances as the 'short-jobs' benchmark instances. In contrary, (2) produces fewer jobs but with a larger number of operations per job. We refer to this set of instances as the 'long-jobs' benchmark instances. Table 1 and 2 show the statistics for the long-jobs benchmark instances. Table 3 and 4 show the statistics for the short-jobs benchmark instances. All instances have a minimal makespan of 600000. This roughly constitutes one week in seconds, which is a common planning horizon in semiconductor manufacturing.

#ma-#ops-file	#jobs	min #ops	max #ops	avg #ops
100-10000-1	103	37	134	97,1
100-10000-2	103	54	125	97,1
100-10000-3	103	28	128	97,1
1000-10000-1	1002	1	22	10,0
1000-10000-2	1002	2	22	10,0
1000-10000-3	1002	2	24	10,0
100-100000-1	109	1	1021	917,4
100-100000-2	114	1	1019	877,2
100-100000-3	109	185	1011	917,4
1000-100000-1	1002	74	133	99,8
1000-100000-2	1002	69	131	99,8
1000-100000-3	1003	68	129	99,7

Table 1. Long-jobs benchmark: operations and job counts

#ma-#ops-file	min opLen	max opLen	avg opLen
100-10000-1	2	72196	6000
100-10000-2	1	49264	6000
100-10000-3	1	53090	6000
1000-10000-1	6	600000	60000
1000-10000-2	2	547505	60000
1000-10000-3	1	567640	60000
100-100000-1	1	6549	600
100-100000-2	1	7777	600
100-100000-3	1	6982	600
1000-100000-1	1	70829	6000
1000-100000-2	1	63685	6000
1000-100000-3	1	70263	6000

Table 2. Long-jobs benchmark: operation lengths

#ma-#ops-file	#jobs	min #ops	max #ops	avg #ops
100-10000-1	2162	2	12	4,6
100-10000-2	2192	1	13	4,6
100-10000-3	2169	1	13	4,6
1000-10000-1	2882	1	9	3,5
1000-10000-2	2863	1	10	3,5
1000-10000-3	2897	1	9	3,5
100-100000-1	20685	2	16	4,8
100-100000-2	20870	2	19	4,8
100-100000-3	20767	2	16	4,8
1000-100000-1	21280	2	16	4,7
1000-100000-2	21349	2	15	4,7
1000-100000-3	21338	2	16	4,7

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

#ma-#ops-file	min opLen	max opLen	avg opLen
100-10000-1	1	69723	6000
100-10000-2	1	53463	6000
100-10000-3	1	68936	6000
1000-10000-1	2	503525	60000
1000-10000-2	3	471671	60000
1000-10000-3	2	600000	60000
100-100000-1	1	9158	600
100-100000-2	1	8373	600
100-100000-3	1	6505	600
1000-100000-1	1	86811	6000
1000-100000-2	1	68044	6000
1000-100000-3	1	65721	6000

Table 4. Short-jobs benchmark: operation lengths

4 RESULTS AND DISCUSSION

Tables 5 and 6 summarize the the experimental results for the long-jobs and short-jobs benchmark respectively. The result tables list for each problem instance the best makespan achieved by one of the basic dispatching rules on its own, the makespan produced by the composite dispatching rules generated by our system and how much the generated rule worked better than the best basic rule.

Generally, it can be stated that the 'short-jobs' benchmark is easier, at least for dispatching rule based approaches. There is one exception: For job shops comprising 1000 machines and 10000 operations the best achieved makespans are better in the long-job benchmark. Concerning the basic dispatching rules tested in standalone manner, it can be stated that best performers are RMTWR and UA.

The composite dispatching rules generated by the approach presented in this paper outperform any tested basic dispatching rule on its own. With respect to the best basic dispatching rules, schedules produced with the automatically generated dispatching rules have up to 38% smaller makespans. Even more impressive is the fact that in many cases near optimal or even optimal schedules could be produced. For such cases, clearly the proposed approach can be perfectly used as a standalone system. In the worst case, that is for long-jobs instances comprising 100000 operations on only 100 machines, the makespans are less than 70% above the minimal makespans, which still is a very good starting value that can be further optimized by some local search method.

In order to get a more intuitive impression on what it means to be 70% above optimum, Figure 4 shows a visual representation of the schedule produced for the '100-10000-3' case (see Table 5) with the generated dispatching rule. Obviously, also such a schedule seems quite good. Though, the schedule for '100-10000-2' (see Figure 5), being roughly 25% above the optimum, is much denser.

5 CONCLUSIONS

In this paper we present an approach based on event-based simulation and genetic algorithms for automatically generating composite dispatching rules for job shop scheduling problems. Hereby, we focus on very large job shops like found in semiconductor industry where exact methods are totally out of reach. We furthermmore report on an experiment investigating the efficacy of the proposed approach for makespan optimization of very large job shop problem instances. To this end, we describe a new benchmark incorporating problem instances with proven optima that comprise up to 100000 operations to be scheduled on up to 1000 machines. To the best of our knowledge, these instances are bigger than any benchmark instances found in literature so far. Summarizing, we can state that for many problem instances we could find highly effective composite dispatching rules resulting in near optimum schedules.

#ma-#ops-file	best basic	generated	reduction
100-10000-1	1040217 (MTWR)	853879	18%
100-10000-2	1033087 (MTWR)	754355	27%
100-10000-3	1073028 (RMTWR)	1000130	7%
avg	1048777	869455	17%
1000-10000-1	600000 (LJF)	600000	0%
1000-10000-2	600000 (LJF)	600000	0%
1000-10000-3	893222 (UA)	600000	33%
avg	697741	600000	11%
100-100000-1	1077736 (RMTWR)	1013399	6%
100-100000-2	1040606 (UA)	1006303	3%
100-100000-3	1072846 (RMTWR)	1013414	6%
avg	1063729	1011039	5%
1000-100000-1	967557 (UN)	600000	38%
1000-100000-2	1046191 (MTWR)	679312	35%
1000-100000-3	1079303 (RMTWR)	823301	24%
avg	1031017	700871	32%

Table 5. Results for 'long-jobs' (optimal makespans = 600000)

#ma-#ops-file	best basic	generated	reduction
100-10000-1	639392 (RMTWR)	600610	6%
100-10000-2	634527 (UA)	602231	5%
100-10000-3	623094 (RMTWR)	600327	4%
avg	632338	601056	5%
1000-10000-1	814514 (UA)	687569	16%
1000-10000-2	808937 (UA)	754101	7%
1000-10000-3	843376 (UA)	726314	14%
avg	822276	722661	12%
100-100000-1	600893 (RMTWR)	600034	0.1%
100-100000-2	601111 (RMTWR)	600045	0.2%
100-100000-3	600671 (RMTWR)	600005	0.1%
avg	600892	600028	0.1%
1000-100000-1	655741 (RMTWR)	603058	8%
1000-100000-2	651725 (UA)	600383	8%
1000-100000-3	652523 (RMTWR)	601065	8%
avg	653330	601502	8%

Table 6. Results for 'short-jobs' (optimal makespans = 600000)

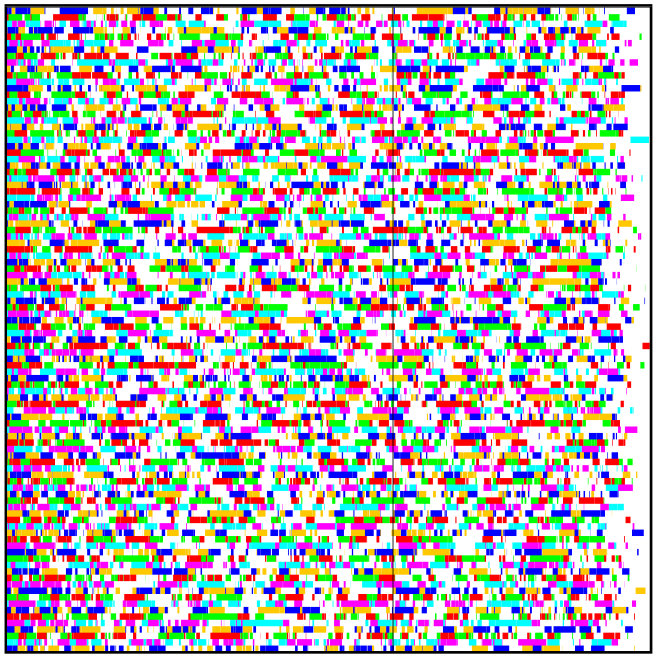

Fig. 4. Schedule for long-jobs: 100-10000-3 (generated dispatching rule)

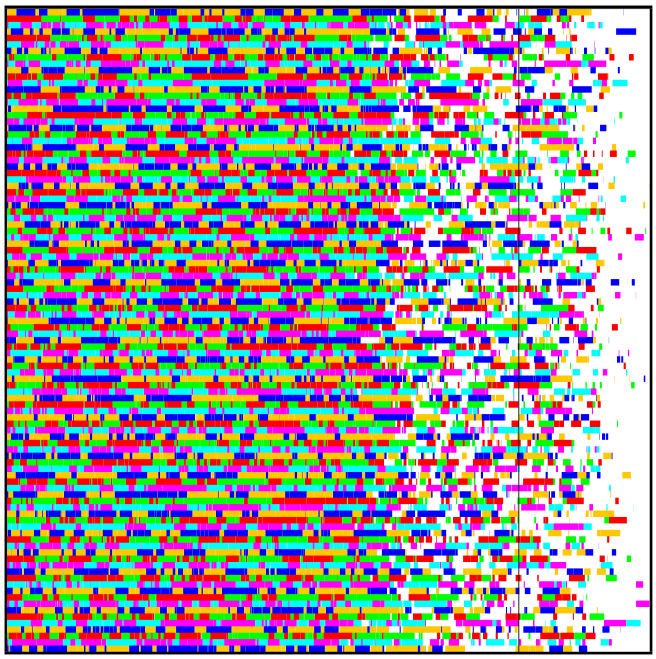

Fig. 5. Schedule for long-jobs: 100-10000-2 (generated dispatching rule)

References

1. Bartk, R., Salido, M., Rossi, F.: New trends in constraint satisfaction, planning, and scheduling: a survey. The Knowledge Engineering Review **25**(3), 249–279 (2010). https://doi.org/10.1017/S0269888910000202
2. Blazewicz, J., Ecker, K., Pesch, E., Schmidt, G., Weglarz, J.: Handbook on Scheduling: Models and Methods for Advanced Planning (International Handbooks on Information Systems). Springer-Verlag New York, Inc., Secaucus, NJ, USA (2007)
3. Boejko, W., Gnatowski, A., Pempera, J., Wodecki, M.: Parallel tabu search for the cyclic job shop scheduling problem. Computers and Industrial Engineering **113**, 512 – 524 (2017)
4. Brucker, P., Jurisch, B., Sievers, B.: A branch and bound algorithm for the job-shop scheduling problem. Discrete Applied Mathematics **49**(1), 107 – 127 (1994)
5. Burke, E.K., Gendreau, M., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., Qu, R.: Hyper-heuristics: a survey of the state of the art. J. of the Operational Research Society **64**(12), 1695–1724 (2013)
6. Conway, R.W.: An experimental investigation of priority assignment in a job shop. RM-3789-PR (1964)
7. Conway, R.W.: Priority dispatching and work-in-process inventory in a job shop. Journal of Industrial Engineering **16**, 228–237 (1965)
8. Da Col, G., Teppan, E.C.: Declarative decomposition and dispatching for large-scale job-shop scheduling. In: Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz). pp. 134–140. Springer, Cham (2016)
9. Da Col, G., Teppan, E.C.: Learning constraint satisfaction heuristics for configuration problems. In: 19th International Configuration Workshop. pp. 8–11 (2017)
10. Danna, E., Perron, L.: 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 – CP 2003. pp. 817–821. Springer Berlin Heidelberg (2003)
11. Demirkol, E., Mehta, S., Uzsoy, R.: Benchmarks for shop scheduling problems. European Journal of Operational Research **109**(1), 137 – 141 (1998)
12. Falkner, A., Friedrich, G., Schekotihin, K., Taupe, R., Teppan, E.C.: Industrial applications of answer set programming. KI-Künstliche Intelligenz pp. 1–12 (2018)
13. Friedrich, G., Frühstück, M., Mersheeva, V., Ryabokon, A., Sander, M., Starzacher, A., Teppan, E.: Representing production scheduling with constraint answer set programming. In: Operations Research Proceedings 2014, pp. 159–165. Springer, Cham (2016)
14. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1990)
15. Hildebrandt, T., Goswami, D., Freitag, M.: Large-scale simulation-based optimization of semiconductor dispatching rules. In: Proceedings of the 2014 Winter Simulation Conference. pp. 2580–2590. WSC '14, IEEE Press, Piscataway, NJ, USA (2014)
16. K. Kaban, A., Othman, Z., Rohmah, D.: Comparison of dispatching rules in job-shop scheduling problems using simulation: A case study. Int. J. of Simulation Modelling **11**, 129–140 (09 2012)
17. Kaban, A.K., Othman, Z., Rohmah, D.S.: Comparison of dispatching rules in job-shop scheduling problem using simulation: a case study. Int. Journal of Simulation Modelling **11**(3), 129–140 (2012)

18. Ku, W.Y., Beck, J.C.: Mixed integer programming models for job shop scheduling: A computational analysis. Computers and Operations Research **73**, 165 – 173 (2016)
19. Panwalkar, S.S., Iskander, W.: A survey of scheduling rules. Oper. Res. **25**(1), 45–61 (Feb 1977)
20. Pezzella, F., Morganti, G., Ciaschetti, G.: A genetic algorithm for the flexible job-shop scheduling problem. Computers and Operations Research **35**(10), 3202–3212 (2008)
21. Sadegheih, A.: Scheduling problem using genetic algorithm, simulated annealing and the effects of parameter values on ga performance. Applied Mathematical Modelling **30**(2), 147 – 154 (2006)
22. Sadeh, N.M., Fox, M.S.: Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. Artificial Intelligence **86**, 1–41 (1996)
23. Stecco, G., Cordeau, J.F., Moretti, E.: A branch-and-cut algorithm for a production scheduling problem with sequence-dependent and time-dependent setup times. Comput. Oper. Res. **35**(8), 2635–2655 (2008)
24. Taillard, E.: Benchmarks for basic scheduling problems. European Journal of Operational Research **64**(2), 278 – 285 (1993), project Management anf Scheduling
25. Teppan, E.C.: Solving the partner units configuration problem with heuristic constraint answer set programming. In: Configuration Workshop. pp. 61–68 (2016)
26. Teppan, E.C., Friedrich, G.: Heuristic constraint answer set programming for manufacturing problems. In: Advances in Hybridization of Intelligent Methods, pp. 119–147. Springer (2018)
27. Teppan, E.C., Friedrich, G.: Heuristic constraint answer set programming. In: ECAI. pp. 1692–1693 (2016)
28. Watson, J.P., Beck, J.C., Howe, A.E., Whitley, L.D.: Problem difficulty for tabu search in job-shop scheduling. Artif. Intell. **143**(2), 189–217 (Feb 2003)
29. Zhang, R., Wu, C.: A hybrid approach to large-scale job shop scheduling. Applied Intelligence **32**(1), 47–59 (Feb 2010)