2019 22nd Euromicro Conference on Digital System Design (DSD)

An Application of Hyper-Heuristics to Flexible Manufacturing Systems

Alexis Linard
Institute for Computing and Information Science
Radboud University Nijmegen, The Netherlands
Email: a.linard@cs.ru.nl

Joost van Pinxten
Department of Electrical Engineering
Eindhoven University of Technology, The Netherlands
Océ Technologies
Venlo, The Netherlands
Email: Joost.vanPinxten@oce.com

***Abstract*—Optimizing the productivity of Flexible Manufacturing Systems requires online scheduling to ensure that the timing constraints due to complex interactions between modules are satisfied. This work focuses on optimizing a ranking metric such that the online scheduler locally (i.e., per product) chooses an option that yields the highest productivity in the long term. In this paper, we focus on the scheduling of a re-entrant Flexible Manufacturing System, more specifically a Large Scale Printer capable of printing hundreds of sheets per minute. The system requires an online scheduler that determines for each sheet when it should enter the system, be printed for the first time, and when it should return for its second print. We have applied genetic programming, a hyper-heuristic, to heuristically find good ranking metrics that can be used in an online scheduling heuristic. The results show that metrics can be tuned for different job types, to increase the productivity of such systems. Our methods achieved a significant reduction in the jobs' makespan.**

***Keywords*-Embedded and Cyber-Physical Systems, Re-entrant Flow Shops, Flexible Manufacturing Systems, Hyper Heuristics, Genetic Programming**

I. Introduction

Flexible Manufacturing Systems (FMS) are composed of thousands of components, which work together to manufacture products. The timing between the different production steps can be complicated due to the wide range of products that the system can produce. Such systems require an online scheduler to ensure that the timing constraints for a single product, as well as for consecutive products, are satisfied. A Large Scale Printer (LSP, see Fig. 1) is such an industrial FMS that requires online scheduling. The combination of minimal and maximal time lags between different production steps, the re-entrant loop, as well as the variability of products makes this a challenging task. The design of scheduler components, as well as the design of ranking metrics, takes a significant amount of effort. In this paper, we use the LSP as a use case for automatically optimizing metrics during design time, to be used during online scheduling.

The paper path of an LSP [1] consists of a Paper Input Station, an Image Transfer Station, a Re-entrant Loop with a Turn Station, and a Finish Station. A scheduler must determine an efficient order of the first and second pass printing at the Image Transfer Station for a given sequence of sheets. This is a difficult problem to solve online due to the minimal and maximal re-entrance time, and the sequence-dependent set-up times between subsequent products [2]. The sheets of paper must not collide, and there should be sufficient time between operations on subsequent products. Leaving as little time as possible between the release of products may lead to high productivity in the short term, but if a future sheet requires large set-up times, then the Re-entrant Loop must first be emptied before that sheet can be printed.

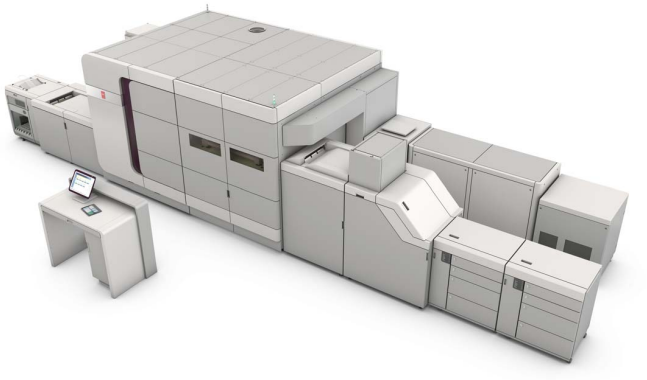

Fig. 1. An example of Océ's Large Scale Printer.

The scheduling decisions are therefore critical for optimizing the long-term productivity: for each newly entering sheet, a scheduling decision is made whether to first print a returning sheet or to release the new sheet. This decision is referred to as determining the interleaving of sheets. The second pass sheets re-enter the print section and are called re-entrant sheets. The goal is then to optimize the scheduling decisions of the print jobs, to maximize the productivity (i.e., minimize the printing time) for a document.

Print jobs can be grouped into different categories, based on structural similarities. Take, for instance, multiple copies of a booklet, which follow a specific pattern (a cover and several inside pages), then the number of pages only differs between print jobs. Finding ranking metrics that optimize for such families can yield improvements for the online scheduler.

In this work, we focus on discovering productive ranking metrics, that are used by an online scheduler to determine the most productive sequencing decision. Our contributions are: (1) an application of hyper-heuristics to Flexible Man-

978-1-7281-2862-7/19/$31.00 ©2019 IEEE
DOI 10.1109/DSD.2019.00057

ufacturing Systems, and particularly to Large-Scale Printers and (2) a modification of genetic programming approaches to ensure a fast convergence towards optimal solutions. We use genetic programming as hyper-heuristics to perform design-time optimization of the metrics that are used online to select the scheduling decisions [3]. Hyper-heuristics explore the search-space of parameters or more generally *programs*, to discover the best settings for the (scheduling) heuristic. The hyper-heuristic generates programs, representing ranking metrics, which are evaluated on a benchmark given the same scheduling framework like the one used for online scheduling. We evaluate the productivity of a given benchmark of jobs.

We have experimented with two different genetic programming approaches and two benchmark aggregations. Our first approach consists of running a Genetic Program (GP) for each group in the benchmark, using user-defined initialized metrics. Our second approach collects the metrics generated for all groups generated by the first approach and uses these as the initial population of metrics for a second run. We also report the results for different aggregations; our two approaches are applied to each print job individually, and groups of structurally similar benchmarks.

The remainder of this paper is as follows: in Sections II and III we define the re-entrant flow-shop scheduling problem and discuss its relevance for the productivity of FMSs, as well as related work. In Sections IV and V we present several versions of the hyper-heuristic genetic programming approach. We present then in Section VI the experimental evaluation of our methods on a Large-Scale Printer. Finally, we conclude and mention further work.

II. Problem Definition

The growing complexity of industrial systems leads to new issues related to scheduling and control. Solving these combinatorial problems can be difficult, and online solutions are generally based on heuristics, representing a good trade-off between computation time and solution quality. These heuristics are, in practice, built upon expert knowledge. However, the acquisition of such human expertise is costly (over the years) and may be biased towards particular solutions.

Therefore, there has been an increase in attention for automatically generating new heuristics for search problems [3], [4]. Approaches that heuristically find new heuristics are called hyper-heuristics [5]. Hyper-heuristic approaches, such as tabu search [6], genetic programming [4], and simulated annealing [7], have been successfully applied to finding better heuristics for bin-packing [8], vehicle routing problems [9], scheduling (constructively or with improvement steps) [10], as well as improving strategies for reinforcement learning [11].

A. Use Case

The use case of this paper, an LSP, is modelled as a re-entrant flow-shop scheduling problem. We repeat a shortened version of the problem definition in this section. For a comprehensive discussion of the problems components, we refer the reader to [12].

Definition 1 (re-entrant flow shop)**.** *A re-entrant flow shop with sequence-dependent set-up times and relative due dates is a tuple* $(M, J, r, O, \phi, P, S, SS, D)$. *$M$ is a set of machines that executes operations. J is a sequence of jobs. Every job j in J is a sequence of r operations $\langle o_{j,1}, \ldots, o_{j,r}\rangle$ that need to be executed. The operations O of the flow shop are the union of the operations of its jobs: $O = \{o_{j,k} | j \in J, k \in \{1, \ldots, r\}\}$. We write $o_{j,k}$ to denote the k-th operation of the j-th job. ϕ is the re-entrance vector $\langle \mu_1, \ldots, \mu_r\rangle$ with $\mu_i \in M$ of r machines. $\phi(i)$ denotes which machine executes the i-th operation of each job is mapped. The processing times of operations are $P : O \to \mathbb{R}_{>0}$. Set-up times that occur regardless of the sequence of operations on a machine are a partial function $S : O \times O \mapsto \mathbb{R}_{\geq 0}$, where $S(o_x, o_y)$ is the minimal time needed between completion of o_x and beginning of o_y. Sequence-dependent set-up times are $SS : O \times O \to \mathbb{R}_{\geq 0}$, where $SS(o_x, o_y)$ is the minimal time needed from completion of o_x to beginning of o_y. Relative due dates are a partial function $D : O \times O \mapsto \mathbb{R}_{>0}$.*

Definition 2. *A schedule $B : O \to \mathbb{R}_{\geq 0}$ for a flow shop describes begin times for all operations; $C(o) = B(o) + P(o)$ denotes the time that operations complete. The following constraints need to be satisfied: Machine $\mu \in M$ executes at most one operation at a time. The sequence-independent set-up times of S between each pair of operations o_x and o_y in the domain of S must be met: $B(o_y) \geq C(o_x) + S(o_x, o_y)$. If no other operation begins between o_x and o_y on the same machine, then the sequence-dependent set-up time between them must hold: $B(o_y) \geq C(o_x) + SS(o_x, o_y)$. If a due date $D(o_x, o_y)$ from o_x to o_y is defined, then it imposes a due date on o_y s.t. $B(o_y) \leq B(o_x) + D(o_x, o_y)$. A schedule is feasible only when all of these constraints are satisfied.*

The goal of the optimization problem is to minimize the makespan of a schedule B, which is the latest completion time of any operation: $makespan(B) = \max_{o \in O} B(o) + P(o)$

The scheduling freedom consists of choosing the ordering and timing of the products at the re-entrant machine (the Image Transfer Station) that satisfies the sequence-dependent set-up times and does not violate the relative due dates (i.e. the maximum travelling time between two prints). An LSP can be modelled as a 3-machine flow shop with four operations per job (i.e., a sheet), re-entrance vector $\langle \mu_1, \mu_2, \mu_2, \mu_3\rangle$, and one job for each sheet that needs to be printed [2].

The heuristic makes its decision on a per-sheet basis. For each subsequent sheet, its re-entrant (second pass) operation is inserted at several positions of a previously determined ordering of first and second passes. To determine whether a given sequencing option is (locally) feasible, a longest-path algorithm is used to determine the earliest possible realization times of the operations of a series of products, taking into account all active timing constraints. The scheduling heuristic then uses a ranking metric to select the scheduling option that lead to the best long-term productivity. The BHCS heuristic of [12] is currently the best on-line dispatching rule heuristic. This paper aims to improve the ranking metric used in the

dispatching rule framework of BHCS.

Typical ranking metrics for this problem include information such as the realization time for executing the second pass of the currently scheduled sheet, the number of operations that remain to be scheduled, the delay of an event due to a scheduling decision (adding the current sheet's second pass into the previously determined sequence).

B. Ranking metric components and structure

The re-entrant operations are sequenced one at a time. Such an operation is denoted by $O_{j,p}$, where j is the job id, and $p \in \{1, 2\}$ is the pass id of the operation. The operation immediately following $O_{x,y}$ is denoted by $O_{w,z} = NEXT(O_{x,y})$, which is influenced by the schedulers' decisions. The earliest realization time of $O_{x,y}$ (for a given sequence) is denoted as $t_{x,y}$. The earliest realization time without the scheduling option under consideration is denoted by $t'_{x,y}$. A limit job L is found for each scheduling option; any option for which the limit job's first pass precedes the current job's second pass is a priori infeasible [12].

TABLE I
RANKING METRIC'S VARIABLES. THE CURRENTLY SCHEDULED SECOND-PASS OPERATION IS $O_{j,p}$.

Variable name	Component
X0	# sheets in the print job
X1	# operations per sheet
X2	L
X3	j
X4	p
X5	$t_{j,p}$
X6	$t'_{j,p}$
X7	# operations between $O_{j,p-1}$ and $O_{j,p}$
X8	Maximum of X7 over all options of the current iteration
X9	x of $O_{x,y} = NEXT(O_{j,p})$
X10	y of $O_{x,y} = NEXT(O_{j,p})$
X11	$t_{x,y}$ for $O_{x,y} = NEXT(O_{j,p})$
X12	$t'_{x,y}$ for $O_{x,y} = NEXT(O_{j,p})$

Note that X0, X1, X3, X4, X6, and X8 have the same value for all options in a single scheduling iteration. These values do change per scheduling iteration, and may therefore still contain information regarding the distinction of scheduling options.

The ranking metric used in [12] has been determined empirically. It uses normalization of these components over the set of scheduling options P. The ranking metric for an option $p \in P$ is a mix of constants, min, max, multiplications, additions, subtractions and divisions:

$$\begin{aligned} rank(p) = 0.3 & \frac{X5_p - min_{p' \in P}(X5_{p'})}{max_{p' \in P}(X5_{p'}) - min_{p' \in P}(X5_{p'})} \\ + 0.6 & \frac{X11_p - min_{p' \in P}(X11_{p'})}{max_{p' \in P}(X11_{p'}) - min_{p' \in P}(X11_{p'})} \\ + 0.1 & \frac{X7_p - min_{p' \in P}(X7_{p'})}{max_{p' \in P}(X7_{p'}) - min_{p' \in P}(X7_{p'})} \end{aligned}$$

Besides the calculated parameters, the following functions are allowed: minimum, maximum, addition, subtraction, multiplication, `asapst`, square root, absolute value, logarithm and negation. The limitation to these functions is domain dependent. The `asapst` function is a domain specific function that takes a job j and a pass id p, and returns the earliest realization time of that operation, i.e. $t_{j,p}$.

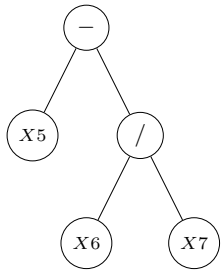

Fig. 2. Ranking metric as expression tree.

Note here that the structure of the ranking metric can be seen as an expression tree. Nodes correspond to the functions and leaves to either variables or constants. This is this structure that will, later, be manipulated by the genetic program. An example of such a tree-like structure is shown in Fig. 2.

III. Genetic Programming

The approach we use is characterized as an off-line heuristic learning approach for a constructive heuristic [5]. To that end, we focus on genetic programs that act as hyper-heuristics for generating heuristics. GPs are suitable for the generation of heuristics (we refer in this regard to the survey paper of [4]).

Evolutionary Algorithms [13] are Artificial Intelligence methods based on heuristics which are particularly suitable to automatically learn *models* (e.g., trees [14], graphs [15], equations [16]...). They are used in a broad variety of domains e.g. learn the optimal shape of tree-like NASA's antennas [17], robot designs [18] and even the internal structure of artificial neural networks [19]. While most evolutionary algorithms focus on solving single-objective problems, there are extensions to solve multi-objective problems [20]. Genetic Programs [21] are a specific type of Evolutionary Algorithms, where programs/heuristics compute solutions, while Evolutionary Algorithms in a broad sense directly evolve solutions to a problem.

Many different techniques have been developed in the past few years, among others, the combination of scheduling rules instead of considering a set of rules separately, in order to enlarge the search space [22], [23]; genetic algorithms to search a space of sequences of heuristic choices [24], [25]; genetic algorithm to solve real-world scheduling and delivery problems [26]; evolutionary algorithm to learn successful heuristics from previous ones [27], [28]. The latter caught our attention: previously learned heuristics (automatically or manually, using domain expertise) could be applied to instances from a given problem after a learning phase. In the context of GPs, this existing knowledge can, therefore, be used as an initial population of metrics.

IV. Learning Approach

One of the critical concepts of GP is to formalize what makes a solution better than another, i.e., writing a *fitness*

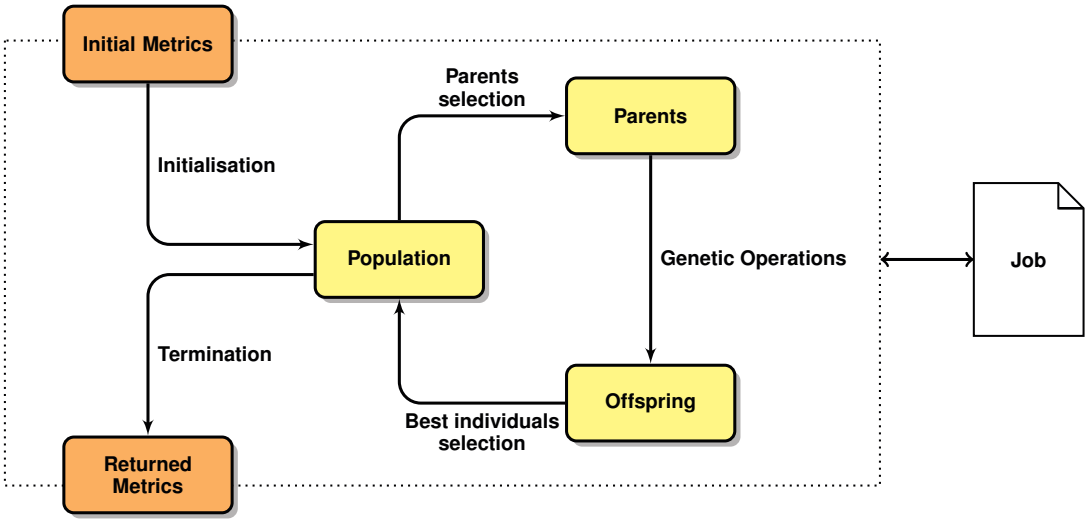

Fig. 3. Genetic Program optimizing an individual printjob (GP-i).

function which takes any proposed solution and returns a *score* for that solution. In our case, we want a ranking metric to lead to the highest productivity as possible, that is to say, to minimize the makespan of the corresponding print job.

Then, the GP iteratively optimizes a *population* of candidate solutions, by randomly mutating and combining (i.e., applying *genetic operators* to) solutions from a population, such that the fitness of the *best solution* per population improves in time:

1) *Initialization*: the initial population contains a set of possible ranking metrics;
2) *Selection*: the selection of the best ranking metrics in the population is based on their fitness values.
3) *Mutation and recombination*: genetic operations are performed on the selected ranking metrics with a fixed probability, and generate new ranking metrics.
4) *Evaluation*: the fitness of each new schedule is evaluated.
5) *Replacement*: high-fitness ranking metrics from the new generation replace low-fitness ranking metrics from the previous generation.

As shown in Fig. 3, the last four steps are repeated until a maximum number of allowed iterations is reached: then, the best solution in the population is returned.

Many variations or parameters can be tuned in GP, such as the definition of a good fitness measure to evaluate candidate programs; the termination criteria, standing for how a solution needs to be good enough to consider returning it; the terminal set, meaning what symbols are allowed in the leaves of the tree-like structure of the program; the function set, defining the functions at the non-terminal nodes of the program; the initial population, which can be user-defined or randomly generated; other GP execution parameters such as (but not limited to) selection method, population size, mutation methods, probability of occurrence of genetic operators, allowing elitist strategies, maximum allowed size/depth of the programs, etc.

A. The Initialization of the Genetic Program

We start with an initial population consisting of commonly used ranking metrics.

- $X5 - \frac{X6}{X7}$
- $|X5|$
- $\texttt{asapst}(X2, X1)$
- $X5 + X6$
- $X5 - X6$
- $X11 - X12$
- $\frac{X7}{X8}$
- $(X5 - X6 + X11 - X12) - \frac{X7}{X8}$

B. Selection

The selection of individuals that will undergo genetic operations is an important task: if the selection method overly selects fit individuals, the population may converge too quickly to a uniform solution, while not exploring the search space widely enough. On the contrary, a selection method not selecting enough fit individuals, will not correctly exploit the information present in the fittest individuals.

We used here a *tournament* selection. From the population, a smaller subset is selected at random to compete. The fittest individual in this subset is then selected to move on to the next generation. Depending on the size of the tournament subset, fitter programs can tend to be found quickly, and the evolution process tends to converge to a solution in less time (large tournament size) or maintain more diversity in the population and find a better solution but for a longer runtime (small tournament size).

C. Terminal and Function Sets

The terminal set is composed of all variables names as described in Table I, plus constants ranging over $\mathbb{R}$. The function set is composed of 5 chosen operations, which are addition, multiplication, subtraction, division and `asapst`.

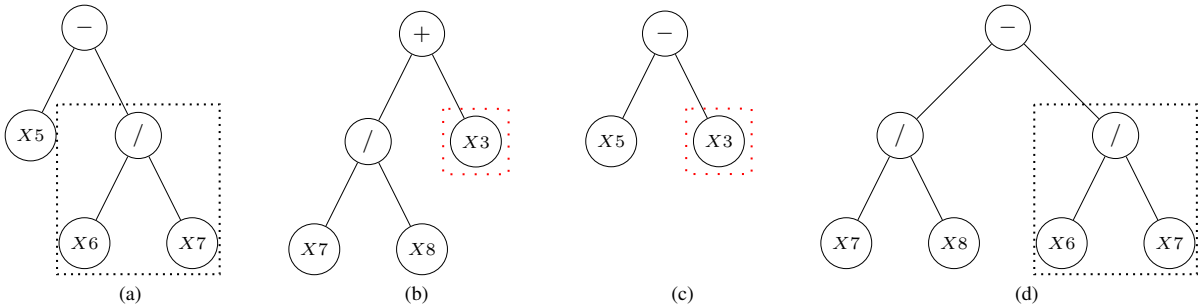

Fig. 4. Crossover between individuals (a) and (b) and resulting children (c) and (d).

D. Genetic Operators

We define stochastic genetic operators as follows:

1) *Crossover*: Crossover takes a metric M_1 and selects a random subtree from it to be replaced. A second metric M_2 also has a subtree selected at random, and this is inserted into the original parent to form an offspring in the next generation and vice versa. An example of this genetic operator is shown in Fig. 4.

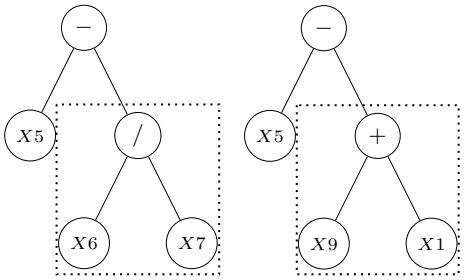

Fig. 5. Subtree mutation.

2) *Subtree Mutation*: Subtree mutation takes a metric M and selects a random subtree from it to be replaced. A donor subtree is generated at random, and this is inserted into the original parent to form an offspring in the next generation. An example of this genetic operator is shown in Fig. 5.

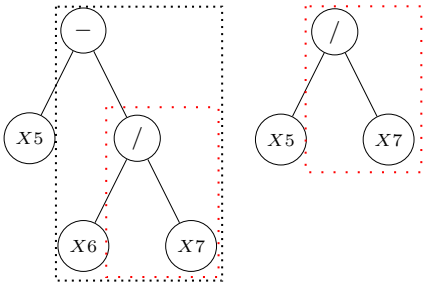

Fig. 6. Hoist mutation.

3) *Hoist Mutation*: Hoist mutation takes a metric M and selects a random subtree from it. A random subtree of that subtree is then selected, and this is *hoisted* into the original subtrees location to form an offspring in the next generation. An example of this genetic operator is shown in Fig. 6.

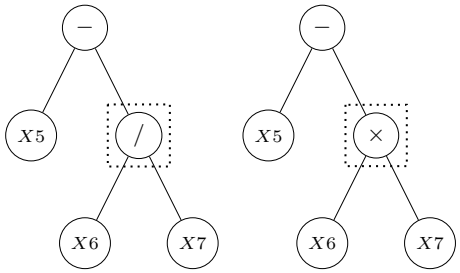

Fig. 7. Point mutation.

4) *Point Mutation*: Point mutation takes a metric M and selects random nodes from it to be replaced. Other terminals and functions replace terminals that require the same number of arguments as the original node. The resulting tree forms an offspring in the next generation. An example of this genetic operator is shown in Fig. 7.

E. Fitness Function

We use the makespan of a job J for a ranking metric r as its *fitness*. In other words,

$$\mathit{fitness}(r) = \mathit{makespan}_r(J) = \max_{o_{j,p} \in O} (t_{j,p})$$

V. Further Approaches

In this section, we describe three extensions of our genetic program. The first one consist of first running the GP on the initial population described in Section IV-A, then reusing the resulting metrics as the initial population for a second run of the GP. The second and third ones consist of evolving metrics such that the fitness is not a function of a given job, but a set of jobs to schedule.

A. Reseeding the Genetic Program

The Genetic Program, as depicted by Fig. 3 tries to optimize a schedule for a given job. The extension we propose is to first run this optimization process once, for the given job. We gather at the end of this process, a set of metrics for this job. The next step of the process is the following: a second GP is run, still in order to optimize this same job, with input population all the metrics of the job to optimize gathered during the first step. This operation of "Re-seed" allows us to reuse previously found good metrics for the job, so that good metrics will tend to be selected, bred and returned at the end of this second process. The whole process of GP-ir is described in Fig. 8.

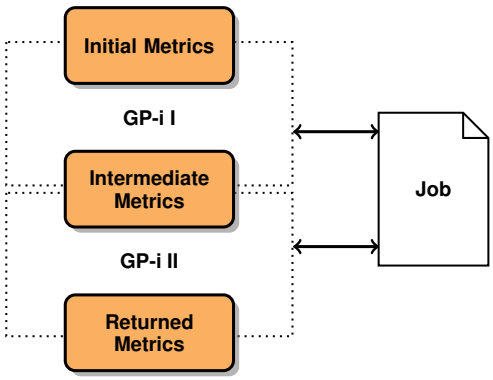

Fig. 8. Genetic Program optimizing an individual print job, using "re-seeding" (GP-ir).

B. Global Solutions

While the previous methods consider a fitness function of the GP as a function of 1 job to optimize, we propose two methods trying to optimize a set of jobs at once.

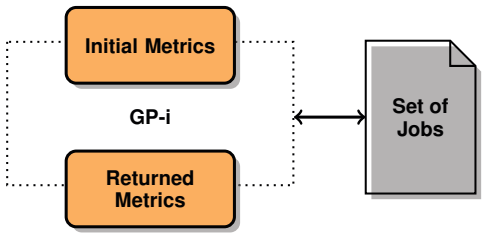

Fig. 9. Genetic Program optimizing a group of printjobs (GP-g).

The first one, Genetic Programming to solve a group G of printjobs (GP-g), starts with the initial population as described in Section IV-A. Then, unlike GP-i, the fitness function is redefined as the sum of all the makespans of the set of jobs to schedule, such that

$$fitness(r, G) = \Sigma_{J \in G} makespan_r(J)$$

The task becomes then to optimize a single metric common to a set of jobs. This is a particularly interesting approach in the context of similar jobs, i.e. sharing similar operations or patterns of operations. This process is described in Fig. 9.

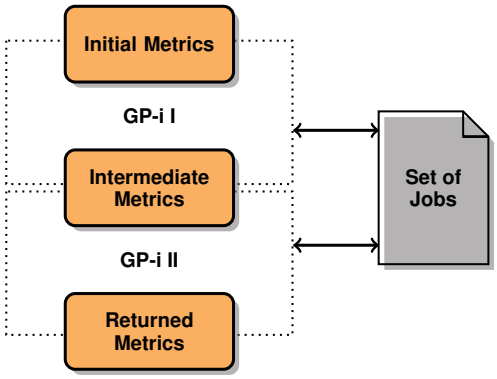

Fig. 10. Genetic Program optimizing a group of printjobs, using "re-seeding" (GP-gr).

The second one, Genetic Program to solve a group of printjobs using "re-seeding" (GP-gr), is a mixture of GP-ir and GP-g. It tries to optimize a ranking metric common to a set of jobs, going through 2 steps: GP-g then GP-g re-seeded with the results of the first iteration. This process is described in Fig. 10.

VI. Experimental Evaluation

In this section, we describe the experimental set-up and compare the makespan of the generated schedules through our different methods.

We consider in this paper a benchmark for Large Scale Printers (such as the one in Fig. 1). We have implemented our 4 genetic programs, GP-i, GP-ir, GP-g and GP-gr, using the python *gplearn* API[1]. BHCS is implemented as a single-thread program. We reused the benchmark used in [12] consisting of 85552 sheets in 684 print requests. Each print request is taken from one of the following categories:

H Homogeneous: repetition of one sheet type
RA Repeating A: repetition of one sheet type followed by another sheet type
RB Repeating B: repetition of one to three sheets of a type followed by another sheet type
BA Block A: 5 blocks of 5 different sheet types, each block contains 10 or 20 sheets of the same type
BB Block B: 5 blocks with two alternating sheet types, each block contains 5 to 25 sheets of the same type

The GPs were parametrized as follows:

- maximum number of iterations: 100
- population size: 100
- tournament size: 10
- crossover probability: 0.7
- subtree mutation probability: 0.1
- hoist mutation probability: 0.1
- point mutation probability: 0.1

Note here that the population size is voluntarily kept low: indeed, it has been shown that too big populations are not always helpful in order for the GP to converge faster towards an optimal solution [29]. Large population sizes also have deleterious effects on the time efficiency of the GP: indeed, in case of a too high calculation complexity of the fitness function, the run time may dramatically blow up. In order to speed up the process, results for given metrics were cached, so the recomputation of previously seen individuals is not required. For the methods GP-g and GP-gr, up to 100 processors were used in parallel for the computation of ranking metrics for a set of jobs.

We compare the makespans obtained with the makespans of schedules generated by BHCS in Fig. 11. For each job category, we show the median improvement (white dot), average improvement (red star), as well as the interquartile range (black box), and the probability density of the data at different values. We also calculated the runtime of the different algorithms, per benchmark, in Fig. 12.

[1]https://gplearn.readthedocs.io/en/stable/index.html

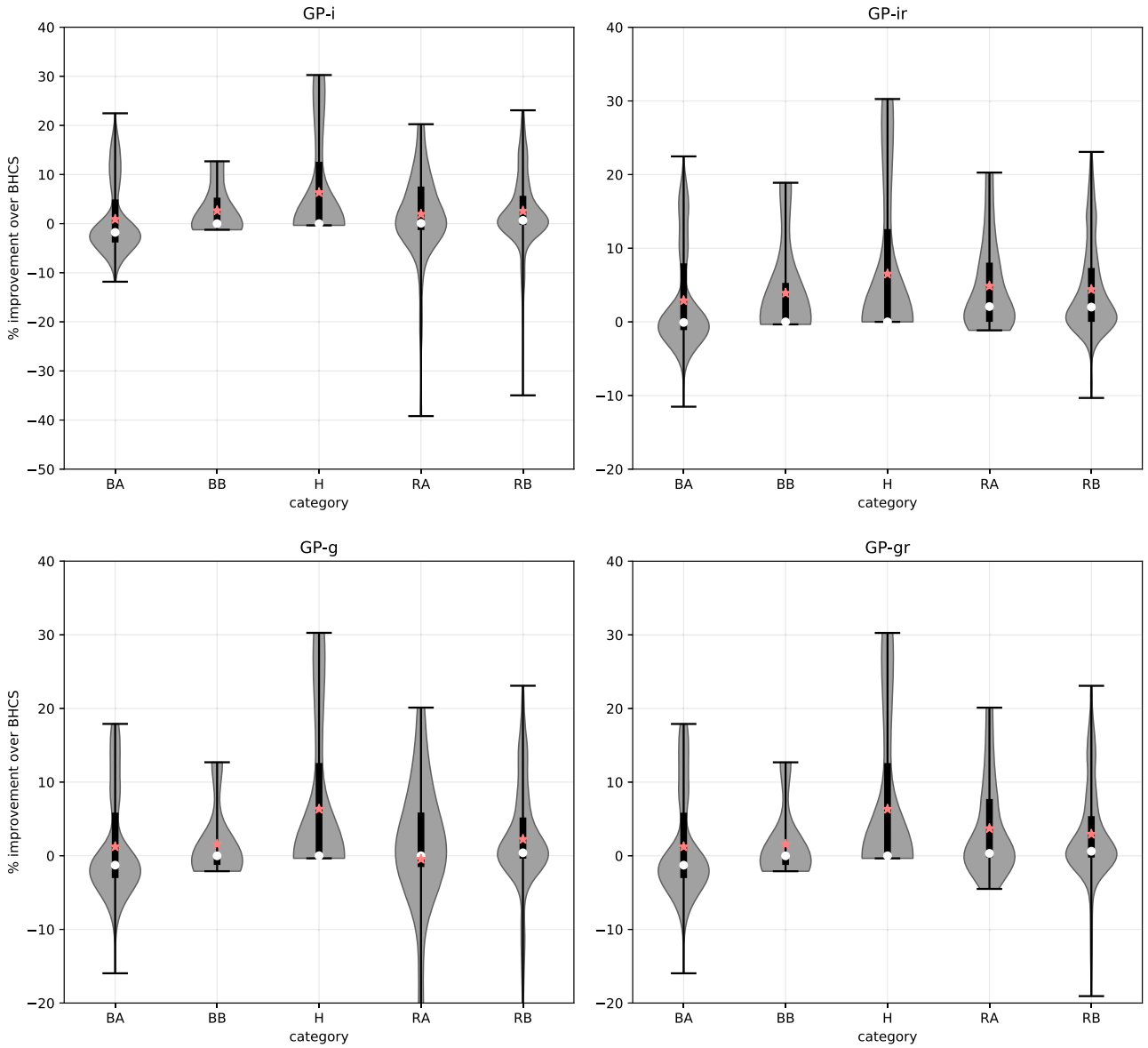

Fig. 11. Makespan improvements of our methods compared to BHCS (in %).

We can see here that for all methods and job types (except GP-g for RA), the average obtained makespan compared to BHCS is improved. In some cases (jobs H), the found heuristics improve the makespan for up to 30.26%. Concerning the median improvement close to 0%, it shows that our methods improve the printing time of the jobs in half of the cases in our benchmark. These are encouraging results: whenever a job is not improved, we can instead fall back to the metric provided by BHCS. So, improving the productivity of half of the jobs is a good result.

VII. Conclusion and Future Work

We presented a new method for the improvement of metrics used to make scheduling decisions for product sequences in a Flexible Manufacturing System. To that end, we employed hyper-heuristics to find these productive metrics. To the best of our knowledge, our work is the first contribution using hyper-heuristics for improving the productivity of FMSs and LSPs. In the case study we considered, our re-seeded genetic program achieved an average improvement of 3.10% in terms of schedule completion time. Moreover, out of 684 cases, we could improve 396 of the print job schedules. For some cases, the execution time of the print jobs has been reduced by up to 13.5 seconds (30.3%), which is a significant productivity improvement in an industrial context.

Our re-seeding of the initial population based on the schedules of a set of jobs shows that some jobs share common

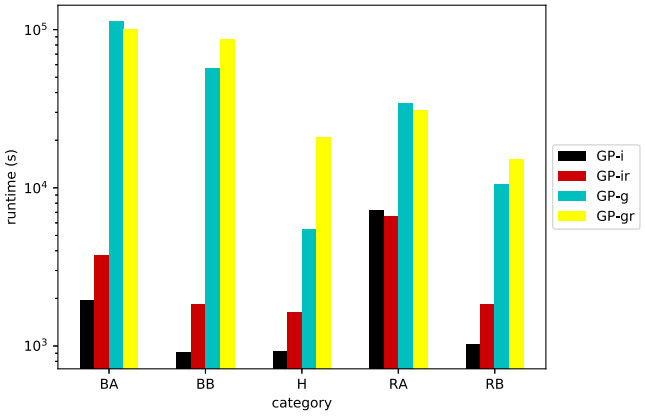

Fig. 12. Runtime of the different algorithms (in seconds).

productive metrics. However, the running time of a Genetic Program prevents its online use in practice. Nonetheless, the schedules obtained can be, once obtained, injected in the online scheduling of jobs as a knowledge base. The integration of such metrics will lead to an improvement in the productivity of FMSs. Indeed, whenever a print job has to be scheduled, the schedules of similar print jobs in the database in terms of job patterns [30], [31] can be reused. In further work, we will focus on the integration of good ranking metrics for similar print job patterns.

Acknowledgements

This work is part of the research program Robust CPS with project number 12693, which is (partly) financed by the Netherlands Organization for Scientific Research (NWO).

References

[1] L. Swartjes, L. Etman, J. van de Mortel-Fronczak, J. Rooda, and L. Somers, "Simultaneous analysis and design based optimization for paper path and timing design of a high-volume printer," *Mechatronics*, vol. 41, pp. 82 – 89, 2017.

[2] U. Waqas, M. Geilen, J. Kandelaars, L. Somers, T. Basten, S. Stuijk, P. Vestjens, and H. Corporaal, "A re-entrant flowshop heuristic for online scheduling of the paper path in a large scale printer," in *DATE, 2015*, 2015, pp. 573–578.

[3] C. D. Geiger, R. Uzsoy, and H. Aytuğ, "Rapid modeling and discovery of priority dispatching rules: An autonomous learning approach," *Journal of Scheduling*, vol. 9, no. 1, pp. 7–34, Feb 2006.

[4] E. K. Burke, M. R. Hyde, G. Kendall, G. Ochoa, E. Özcan, and J. R. Woodward, "Exploring hyper-heuristic methodologies with genetic programming," in *Computational intelligence*. Springer, 2009, pp. 177–201.

[5] E. K. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and R. Qu, "Hyper-heuristics: A survey of the state of the art," *Journal of the Operational Research Society*, vol. 64, no. 12, pp. 1695–1724, 2013.

[6] E. K. Burke, G. Kendall, and E. Soubeiga, "A tabu-search hyperheuristic for timetabling and rostering," *Journal of heuristics*, vol. 9, no. 6, pp. 451–470, 2003.

[7] R. Bai and G. Kendall, "An investigation of automated planograms using a simulated annealing based hyper-heuristic," in *Metaheuristics: Progress as real problem solvers*. Springer, 2005, pp. 87–108.

[8] P. Ross, S. Schulenburg, J. G. Marín-Bläzquez, and E. Hart, "Hyper-heuristics: learning to combine simple heuristics in bin-packing problems," in *Proc. of the 4th Conf. on Genetic and Evolutionary Computation*. Morgan Kaufmann Publishers Inc., 2002, pp. 942–948.

[9] J. D. Walker, G. Ochoa, M. Gendreau, and E. K. Burke, "Vehicle routing and adaptive iterated local search within the hyflex hyper-heuristic framework," in *Learning and Intelligent Optimization*. Springer, 2012, pp. 265–276.

[10] E. K. Burke, B. McCollum, A. Meisels, S. Petrovic, and R. Qu, "A graph-based hyper-heuristic for educational timetabling problems," *European Journal of Operational Research*, vol. 176, pp. 177–192, 2007.

[11] E. Özcan, M. Misir, G. Ochoa, and E. K. Burke, "A reinforcement learning: great-deluge hyper-heuristic for examination timetabling," in *Modeling, Analysis, and Applications in Metaheuristic Computing: Advancements and Trends*. IGI Global, 2012, pp. 34–55.

[12] J. van Pinxten, U. Waqas, M. Geilen, T. Basten, and L. Somers, "Online scheduling of 2-re-entrant flexible manufacturing systems," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5, pp. 160:1–160:20, 2017.

[13] T. Back, *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[14] D. Bucur, G. Iacca, G. Squillero, and A. Tonda, "The impact of topology on energy consumption for collection tree protocols: An experimental assessment through evolutionary computation," *Applied Soft Computing*, vol. 16, pp. 210 – 222, 2014.

[15] P. Dupont, "Regular grammatical inference from positive and negative samples by genetic search: the gig method," in *Grammatical Inference and Applications*. Springer Berlin Heidelberg, 1994, pp. 236–245.

[16] A. L. Oliveira and A. Sangiovanni-Vincentelli, "Learning complex boolean functions: Algorithms and applications," in *Advances in Neural Information Processing Systems*, 1994, pp. 911–918.

[17] G. Hornby, A. Globus, D. Linden, and J. Lohn, "Automated antenna design with evolutionary algorithms," in *Space 2006*, 2006, p. 7242.

[18] N. Cheney, R. MacCurdy, J. Clune, and H. Lipson, "Unshackling evolution: evolving soft robots with multiple materials and a powerful generative encoding," in *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. ACM, 2013, pp. 167–174.

[19] J. Lehman and R. Miikkulainen, "Neuroevolution," *Scholarpedia*, vol. 8, no. 6, p. 30977, 2013, revision #137053.

[20] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, Apr 2002.

[21] D. Simon, *Evolutionary optimization algorithms*. John Wiley & Sons, 2013.

[22] H. Fisher, "Probabilistic learning combinations of local job-shop scheduling rules," *Industrial scheduling*, pp. 225–251, 1963.

[23] W. B. Crowston, F. Glover, J. D. Trawick *et al.*, "Probabilistic and parametric learning combinations of local job shop scheduling rules," Carnegie Institute of tech Pittsburgh, Tech. Rep., 1963.

[24] H.-L. Fang, P. Ross, and D. Corne, *A promising genetic algorithm approach to job-shop scheduling, rescheduling, and open-shop scheduling problems*. University of Edinburgh, Department of Artificial Intelligence, 1993.

[25] H.-L. F. P. Ross and D. Corne, "A promising hybrid ga/heuristic approach for open-shop scheduling problems," in *Proc. 11th European Conference on Artificial Intelligence*, 1994, pp. 590–594.

[26] E. Hart, P. Ross, and J. Nelson, "Solving a real-world problem using an evolving heuristically driven schedule builder," *Evolutionary Computation*, vol. 6, no. 1, pp. 61–80, 1998.

[27] R. Drechsler and B. Becker, "Learning heuristics by genetic algorithms," in *Proceedings of ASP-DAC'95/CHDL'95/VLSI'95 with EDA Technofair*. IEEE, 1995, pp. 349–352.

[28] R. Drechsler, N. Göckel, and B. Becker, "Learning heuristics for obdd minimization by evolutionary algorithms," in *International Conference on Parallel Problem Solving from Nature*. Springer, 1996, pp. 730–739.

[29] T. Chen, K. Tang, G. Chen, and X. Yao, "A large population size can be unhelpful in evolutionary algorithms," *Theoretical Computer Science*, vol. 436, pp. 54–70, 2012.

[30] A. Linard, C. de la Higuera, and F. Vaandrager, "Learning unions of k-testable languages," in *Language and Automata Theory and Applications*, C. Martín-Vide, A. Okhotin, and D. Shapira, Eds. Cham: Springer International Publishing, 2019, pp. 328–339.

[31] U. Waqas, M. Geilen, S. Stuijk, J. v. Pinxten, T. Basten, L. Somers, and H. Corporaal, "A fast estimator of performance with respect to the design parameters of self re-entrant flowshops," in *2016 Euromicro Conference on Digital System Design (DSD)*, Aug 2016, pp. 215–221.