applied sciences

MDPI

Article

Applying Learning and Self-Adaptation to Dynamic Scheduling

Bernhard Werth [1,2,*,†], Johannes Karder [1,2,†], Michael Heckmann [1,†], Stefan Wagner [1,†] and Michael Affenzeller [1,2]

[1] Campus Hagenberg, University of Applied Sciences Upper Austria, 4232 Hagenberg im Mühlkreis, Austria; michael.heckmann@fh-hagenberg.at (M.H.); stefan.wagner@fh-ooe.at (S.W.); michael.affenzeller@fh-hagenberg.at (M.A.)
[2] Institute for Symbolic Artificial Intelligence, Johannes Kepler University Linz, 4040 Linz, Austria
* Correspondence: bernhard.werth@fh-hagenberg.at
† Josef Ressel Center adaptOp.

Abstract: Real-world production scheduling scenarios are often not discrete, separable, iterative tasks but rather dynamic processes where both external (e.g., new orders, delivery shortages) and internal (e.g., machine breakdown, timing uncertainties, human interaction) influencing factors gradually or abruptly impact the production system. Solutions to these problems are often very specific to the application case or rely on simple problem formulations with known and stable parameters. This work presents a dynamic scheduling scenario for a production setup where little information about the system is known a priori. Instead of fully specifying all relevant problem data, the timing and batching behavior of machines are learned by a machine learning ensemble during operation. We demonstrate how a meta-heuristic optimization algorithm can utilize these models to tackle this dynamic optimization problem, compare the dynamic performance of a set of established construction heuristics and meta-heuristics and showcase how models and optimizers interact. The results obtained through an empirical study indicate that the interaction between optimization algorithm and machine learning models, as well as the real-time performance of the overall optimization system, can impact the performance of the production system. Especially in high-load situations, the dynamic algorithms that utilize solutions from previous problem epochs outperform the restarting construction heuristics by up to ~24%.

Keywords: dynamic optimization; scheduling; online machine learning

check for updates

Citation: Werth, B.; Karder, J.; Heckmann, M.; Wagner, S.; Affenzeller, M. Applying Learning and Self-Adaptation to Dynamic Scheduling. *Appl. Sci.* **2024**, *14*, 49. https://doi.org/10.3390/app14010049

Academic Editor: Christos Bouras

Received: 10 November 2023
Revised: 13 December 2023
Accepted: 16 December 2023
Published: 20 December 2023

1. Introduction

Dynamic scheduling is a long-studied topic [1,2] that has gained significant traction in the research community and has found many practical applications:

- Georgiadis and Michaloudis [3] propose a real-time production planning and control system for job-shop manufacturing systems and apply it in the production of refrigeration bodies for commercial vehicles. They state that near-optimal control variables are achieved, leading to improved system behavior. However, they also state that the real-world operator was not always able to operate under near-optimal variables.
- Cafaro and Cerdá [4] successfully solve a real-world pipeline scheduling problem. They conduct a case study and distribute multiple refined petroleum products through a 955 km long pipeline to a handful of terminals over 1-week periods.
- Cowling et al. [5] present an integrated dynamic scheduling approach for steel casting and milling. Scheduling is handled by a multi-agent architecture, where each agent uses its own scheduling model to cooperatively develop global schedules for the overall system.

A core form of scheduling problems is the static scheduling problem, where a number of n jobs needs to be scheduled to a number of m machines with the goal of optimizing an objective function(e.g., the latest completion time of any job). Practical considerations introduced numerous extensions to this:

- The processing time $p_{i,j}$ of a job i may be dependent not only on the machine j where it is executed but also on any number of additional factors (including but not limited to: the previous job performed on machine j (setup times), degrading machine states or material properties).
- Precedence or grouping constraints can restrict possible solutions to jobs being performed only in specific orders, in direct succession, in parallel, on explicitly the same or explicitly different machines.
- Jobs sometimes may be terminated pre-emptively and either restarted or resumed at a later point in time with varying degrees of penalties and restrictions.
- Machines may be unique or identical, with some machines accepting more or other jobs than others.
- Both jobs and machines might require idle time for transportation, machine setups, maintenance or quality control.
- Timings might not be fully known in advance, transforming the problem from a deterministic to a *stochastic* variant.
- Jobs may have release or due times, further restricting the space of potential solutions, while machines might be only operational during certain time windows.

In the static formulation of the problem, all problem parameters are known a priori, the problem is solved once (often heuristically) and the best-found solution is then implemented. A major extension to the static formulation includes *dynamic* formulations, where a number of dynamic events can influence the optimization (e.g., the arrival of new jobs, machine delays or breakdowns, corrections of data that are going to influence constraints or predicted processing times, jobs might end in failure and might have to be restarted, etc.).

Solution approaches to dynamic scheduling range from exact solvers, construction heuristics, learned or handcrafted dispatching rules and deep (reinforcement) learning to meta-heuristic solvers like neighborhood searches and evolutionary algorithms. Mohan et al. [6] provide a recent review of the different classes of such optimization approaches.

One main class of approaches for this problem are variations of reinforcement learning [7,8]. An extensive review of reinforcement learning approaches for dynamic task scheduling has been published by Shyalika et al. [9]. The other popular field of research is (simulation-based) meta-heuristic algorithms, ranging from simple genetic algorithms [10] to more complex hybrid solutions [11], with a strong trend towards adaptive solutions in the recent years [12].

Another relevant distinction is not in the optimization algorithm but in how the dynamicity of the problem is handled. The purest of reactive approaches simply restarts the optimizer, which, despite its simplicity, can be a powerful strategy, especially when the machine times are very long compared to the time required by the optimizer to converge toward solutions of acceptable quality. Approaches more aware of the dynamic environment adapt existing solutions from previous problem configurations (a specific problem version and the time while it is valid are often referred to as an *epoch* [13]). Even more sophisticated setups use predictive models to anticipate some or all of the potential events produced by the dynamic problem. Anticipated events can then be used to quantify the *robustness* of solutions, inform the objective function and the expected schedules associated with generated solutions or influence the hyperparameters of the solver itself—especially in self-aware systems.

In recent years, the role of machine learning in scheduling applications has become more and more significant. An overview of such approaches can be found in various works published over the last three decades [14–19]. Approaches using ML models in recent years, including strategies that utilize models in order to evaluate potential schedules, range from Lee et al. [20], who describe a job shop scheduling system that incorporates a genetic algorithm and decision trees, to much more recent advances that allow the use of full-blown digital twins that combine real and simulated data [21]. A more direct way to utilize models in scheduling applications is the use of different variations of deep neural networks. Luo [22] uses reinforcement learning to directly generate the next relevant action(s) at any point in time. Tuli et al. [23] employ an additional "constraint satisfaction module" that interacts with the main reinforcement network in order to filter out unwanted solutions, while Hu et al. [24] build upon Baruwa et al.'s [25] formalization of scheduling problems as Petri-nets, together with a combination of a deep Q-network and specialized layer. Additionally, Hu et al.'s setup utilizes a pre-existing digital twin as a simulation to generate training data.

By utilizing a semi-realistic setup of a production system, this work demonstrates:

- The use of relatively simple machine learning models to obtain machine information during production.
- The incorporation of these models into the optimization.
- The benefits of dynamic optimization methods that update solution candidates over time.
- An analysis of the dynamic interaction between an optimization algorithm and the machine learning models it uses to evaluate its solutions.
- The impact of wall-clock time performance of the optimizer in an asynchronous setup on the overall system performance.

Both model–optimizer interactions and wall-clock time performance are aspects of real-world applications that can heavily influence the overall system performance but are often abstracted away in academic studies.

The remainder of this paper is structured as follows. First, Section 2 introduces the reader to the materials and methods used for the experimental study. It provides insights into the simulated production system and the implementation thereof, as well as details of the implemented optimizers and the used machine learning models. The results of the study are detailed in Section 3. Finally, the paper concludes with a summary of the presented study and possible future work in Section 4.

2. Materials and Methods

In this section, we emulate a production system that disassembles raw materials into useful parts, optionally refines them and then assembles more complex products. An optimizer is used to decide the production schedule in real time based on the information obtained during production.

Figure 1 summarizes the overall setup of the experiments to follow. The main component is a discrete event simulation that acts as the real world for experimentation purposes. It emits production events (i.e., a new batch of tasks has arrived, a machine has started, a machine has finished, the material was moved) to a world state that provides information to both the optimizer and the machine learning models in the form of new training data or updated problem definitions. While the models and some of the optimizers act reactively by performing an update step or by computing a new solution, the more involved optimizers run continuously, holding the last available problem–model combination until an opportune time for an update is achieved (i.e., the end of a generation for evolutionary algorithms, the end of a neighborhood batch for the local search). The optimizer computes a schedule for all production tasks of all current orders and publishes it as a list to the simulation whenever an update of the last solution becomes necessary or a new better solution is found. Whenever a machine in the simulation finishes or a new solution is sent from the optimizer, the simulation checks whether the first—not already dispatched—task of the last solution is currently dispatchable without waiting and removes it from the solution. This process is repeated until the first element of the schedule can not be immediately dispatched, at which point the simulation resumes its normal simulation behavior. Tasks are neither pre-empted nor prematurely dispatched. All components shown in Figure 1 perform asynchronously from each other and receive their respective updates batch-wise (as not to spawn multiple tasks performing model updates).

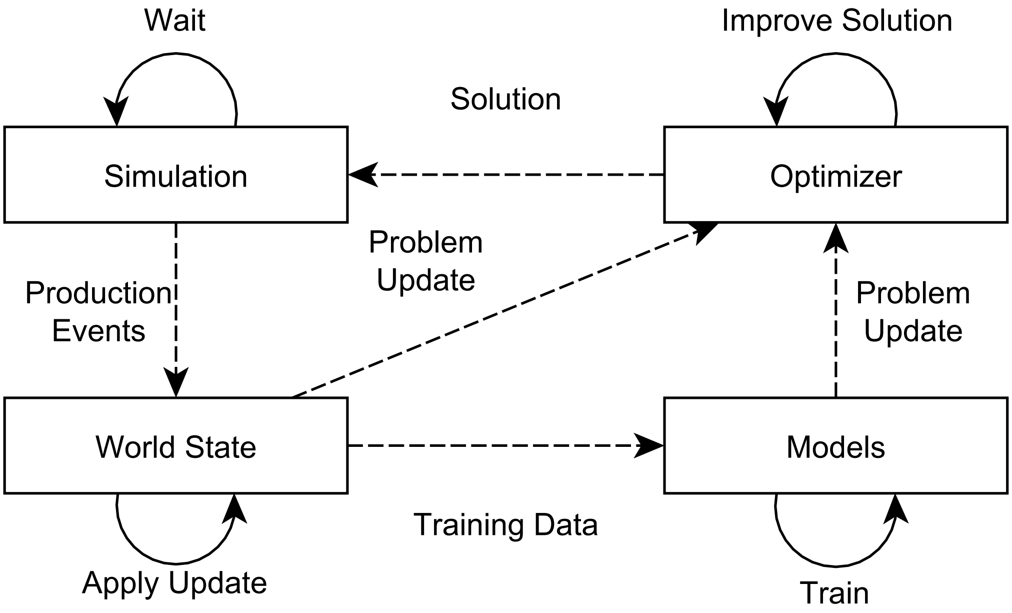

Figure 1. Asynchronous components of the experimental setup.

2.1. Scenario

Benchmarks for dynamic versions of the job shop scheduling problem can be roughly characterized into the following two categories:

(a) Established academic benchmark instances for *static* scheduling problems that have been ad hoc "dynamicized" (e.g., Shahrabi et al. [8] or Kundakcı and Kulak [11]);

(b) Full-blown libraries that essentially have to include the full simulation code and are often intrinsically combined with optimization problems (e.g., Lopez et al. [26] or Beham et al. [27])

The former variation appears to be suitable for comparing the performance of different optimizers because at least the original states equate to the static scheduling problem instances that are known to be challenging. To the best of the authors' knowledge, most work focuses on scenarios where processing times are known a priori and arrival rates of new jobs follow very simple distributions and are independent of the simulations' current state. Such a setup is often quite reasonable as it challenges the optimizer and disentangles its performance from the performance of any machine learning models used to predict system properties (setup times, process times, breakdowns). However, this assumes very well-established and verified knowledge about the process that gives rise to the dynamic scheduling problem, which is not always given in industry 4.0 settings. The second variation is often driven by an existing real-world use case and is so complex that the full application code, including proprietary business logic is needed to accurately re-perform the numerical experiments.

In this work, we opt for a simplified variation of the second approach, allowing us to introduce several different system behaviors that may influence the simulation–modeling–optimization composition. At the same time, we deliberately choose a smaller scenario in order to retain the explainability of the results.

Figure 2 displays a small production system comprising upstream and downstream processes, a disassembly, an assembly stage, a processing stage and two intermediary buffers for the materials used in the individual steps. Table 1 summarizes the parameters and behavior of the machines, including three disassembly machines with different task times, three intermediary processing machines and two assemblers. Two machines (21 and 32) require setup times under certain conditions, which should be avoided by the optimizer, while the task time of machine 23 depends on specific material properties ($x_{1,m} \cdot x_{2,m}$ is presented to the learning model as a precalculated feature to fit the linear model). Both assembly and disassembly actions are predefined in the orders and target random materials in the generated task batch (without repetition). Therefore, neither task type introduces additional decisions that the optimizer has to make. The main impact these actions have is the increased demands on the buffer elements and the precedence constraints implicit with an assembly task (i.e., all sub-materials have to be ready).

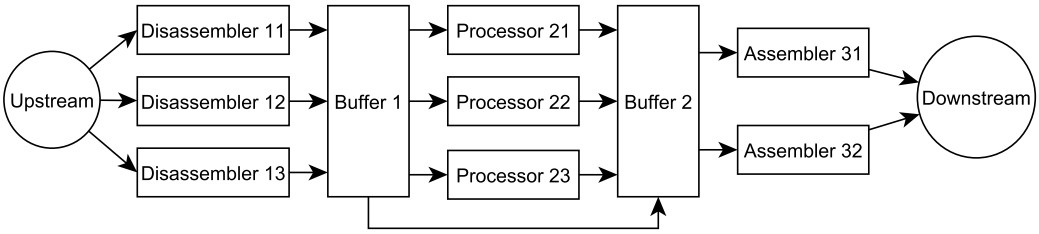

Figure 2. Scenario: production pipeline.

Table 1. Scenario parametrization.

Variable	Description	Distribution
$x_{1,m}$	Material property 1 of material m	(0.8, 25%), (1.0, 50%), (1.2, 0.15%), (14.0, 10%)
$x_{2,m}$	Material property 2 of material m	(0.8, 25%), (1.0, 70%), (1.2, 5%)
$x_{3,m}$	Material property 3 of material m	(8.0, 25%), (10.0, 50%), (12.0, 15%), (14.0, 10%)
d_{count}	Number of disassembly tasks per batch	3 (constant)
a_{count}	Number of assembly tasks per batch	5 (constant)
a_{size}	Number of materials per assembly task	(1, 15%), (2, 70%), (3, 15%)
d_{choice}	Disassembly machine distribution	(1, 52%), (1, 35%), (1, 13%)
i_{choice}	Intermediary machine distribution	(None, 50%), (21, 25%), (22, 20%), (23, 5%),
a_{choice}	Assembly machine distribution	(31, 30%), (32, 70%)
m_{11}	Machine 11 task time	$N(30,1)$
m_{12}	Machine 12 task time	$N(45,1)$
m_{13}	Machine 13 task time	$N(120,1)$
m_{21}	Machine 21 task time	$N(65,5)+s_{21}$
m_{22}	Machine 22 task time	$N(25,5)$
m_{23}	Machine 23 task time	$N(10+3x_1x_2,5)$
m_{31}	Machine 31 task time	$N(35,4)$
m_{32}	Machine 32 task time	$N(20,1)+s_{32}$
s_{21}	Machine 21 setup time	$N(60,5)$ if $x_{3,p}$ of the last material p is not equal to $x_{3,m}$ of the current material m else 0
s_{32}	Machine 32 setup time	$N(15,5)$ if the last material is more than 60 s in the past else 0
$b_{23,M}$	Whether a set of materials M can be batched on machine 23	$\sum_m x_{1,m} \cdot x_{2,m} < 15$ and $\max_{m \in M} x_{3,m} - \min_{m \in M} x_{3,m} = 0$

2.2. Simulation

The simulation is implemented in SimSharp (https://github.com/heal-research/SimSharp, accessed on 21 August 2023) [28], which allows for setting a targeted speedup factor in real time. As the simulation runs on pseudo-real time (real time sped up by a constant factor) asynchronously to the optimizer and machine learning [29], both components' runtime will impact the overall system performance. SimSharp allows for setting a target speedup factor; however, it can not guarantee it is achieved. All speedups reported in Section 3, therefore, are the real (measured) speedups for each experiment (all but ensuring that no two problem instances are perfectly aligned).

2.3. Optimizer

This section details the algorithms that are part of the experimental setup. On the one hand, two population-based approaches (the offspring selection genetic algorithm and the relevant alleles preserving genetic algorithms) are evaluated. The main benefits of choosing evolutionary heuristics in general and genetic algorithms, in particular, are their black-box approach with respect to the problem, which decouples the specifics of the optimizers from the features (e.g., batching and clustering of tasks, number and size of buffers, etc.) of the production system and their proven applicability to a wide spectrum of dynamic optimization problems [30,31], as well as their white-box approach with respect to the resulting schedule that could be reviewed by human operators. On the other hand, a trajectory-based algorithm (local search), as well as two construction heuristics (first-in-first-out and shortest-job-first) are used to optimize the same simulation scenarios for comparison. Both are implemented as open-ended optimization algorithms, which are executed in parallel to the system to be optimized and only ever terminate manually. The dynamic algorithms incorporate dynamic changes into the problem model during algorithm execution and need to adapt existing solutions to be valid for the current system state.

2.3.1. First-In-First-Out

First-in-first-out (FIFO) is the first and simplest algorithm to determine the schedule of pending jobs. It is implemented as a rule-based approach. For every event yielded by the simulated production system, FIFO returns a complete schedule of all jobs currently present in the system. The jobs are sorted with respect to their creation date, meaning that jobs that entered the system first are scheduled before jobs that entered the system later. FIFO scheduling is always executed immediately if there are pending system changes that require a new schedule to be created. If multiple changes accumulate over the course of one FIFO execution, FIFO is started again— taking into account the latest known system state—until no more changes are pending. A depiction of the FIFO algorithm can be seen in Figure 3.

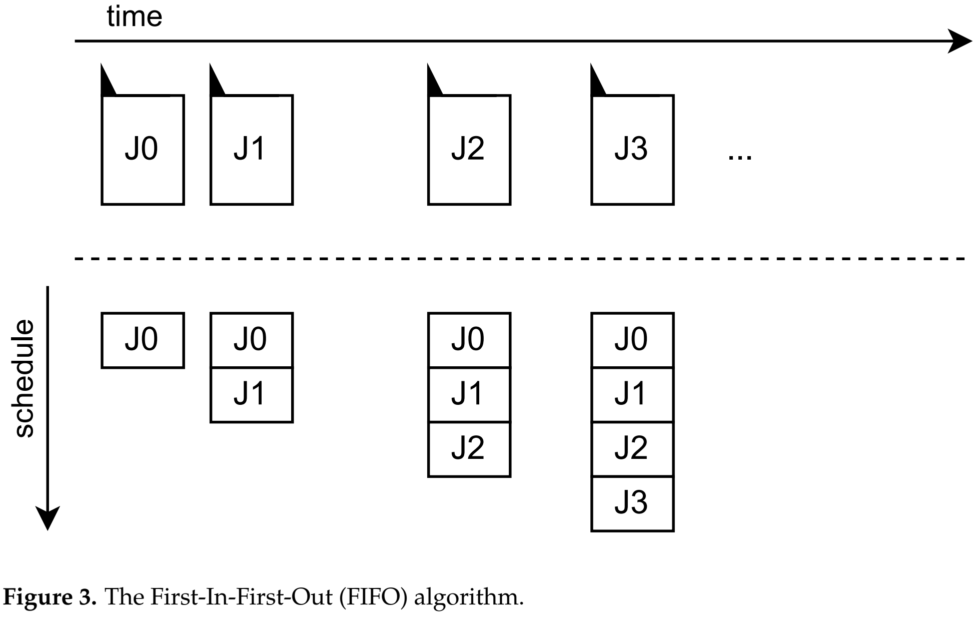

Figure 3. The First-In-First-Out (FIFO) algorithm.

2.3.2. Shortest-Job-First

The *Shortest-Job-First* (SJF) scheduling algorithm inherits some of its behavior from the FIFO scheduler. It executes repeatedly as long as pending changes are present to produce updated schedules that contain all currently known jobs. SJF processes all known jobs as follows. It sorts all n jobs with respect to their predicted processing time in ascending order and selects the job with the shortest predicted processing time that does not have unscheduled predecessors first. It then assumes this first job to be processed and sorts all remaining $n-1$ jobs again with respect to their shortest predicted processing time. Therefore, SJF splits the set of jobs into two lists $\mathcal{S}$ and $\mathcal{U}$, where $\mathcal{S}$ contains all scheduled jobs, and $\mathcal{U}$ contains all remaining jobs. Scheduling a job, i.e., moving it from $\mathcal{U}$ to $\mathcal{S}$, might lead to other jobs in $\mathcal{U}$ being processable and therefore schedulable. Figure 4 shows how jobs are sorted and selected. Job J1 depends on J0 and cannot be scheduled until J0 is scheduled. Jobs J4 and J5 both depend on J3. Jobs that can be scheduled are marked with a check mark, jobs that cannot be scheduled (because of precedence constraints) are marked with a cross. Already scheduled jobs are marked with a star. The fill level of each job corresponds to the predicted processing time, making J7 the fastest job and J6 the slowest. In the first step, all jobs are sorted with respect to their processing time. Then, the fastest schedulable job—here J7—is scheduled, i.e., moved from $\mathcal{U}$ to $\mathcal{S}$. As no jobs depend on J7, no precedence constraints are changed. SJF then virtually applies J7 and updates the timing models, as well as the predicted processing times for all remaining, schedulable jobs. SJF then selects the next fastest, schedulable job in the sorted set $\mathcal{U}$, now being J2, in which the predicted processing time was updated. Since both J4 and J5 depended on J2, their precedence constraints are now lifted and both jobs are schedulable. SJF would then select the next fastest, schedulable job in $\mathcal{U}$, being J4. This repeats until $\mathcal{U} = \emptyset$.

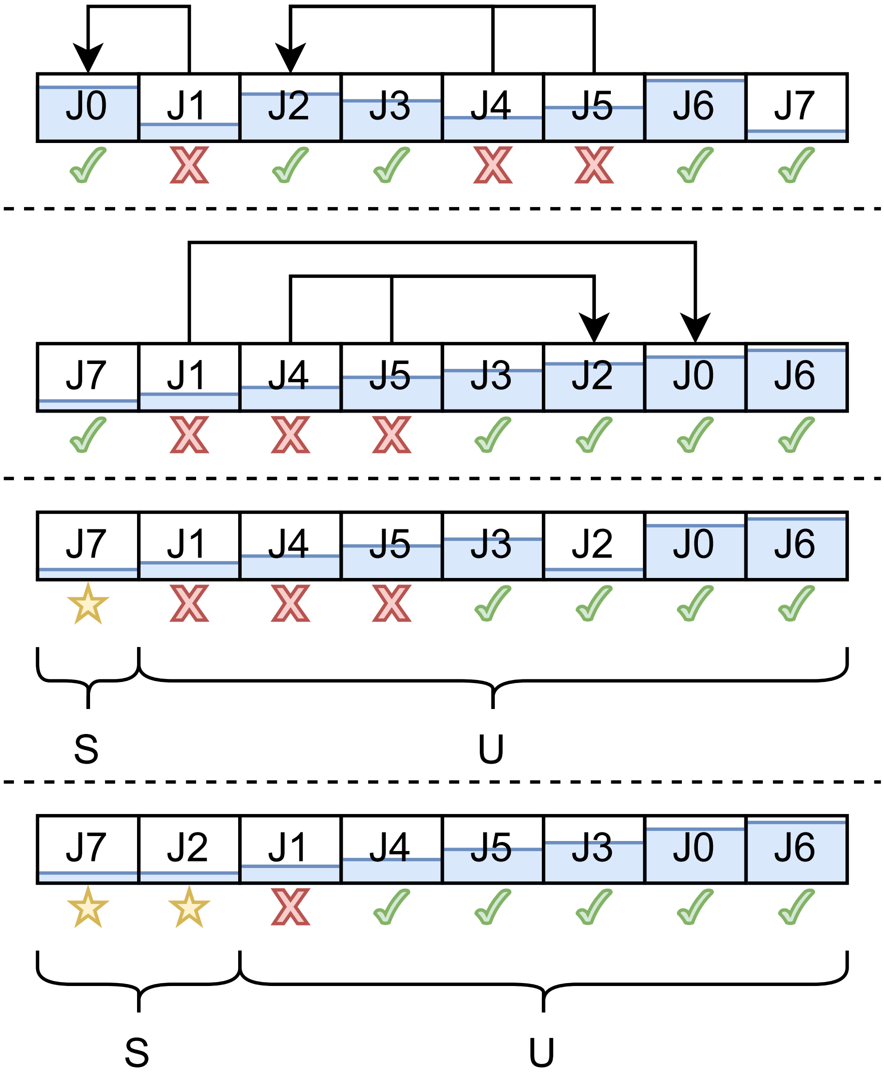

Figure 4. Shortest-Job-First (SJF) processing eight jobs.

2.3.3. Open-Ended Local Search

Furthermore, an open-ended version of a local search (LS) is created. The algorithm starts off by randomly creating an initial solution and conducts move operations to sample all neighbors of this solution in sequence until a better solution is found. It then replaces the current solution with the newly found (better) neighbor and starts to enumerate the new neighborhood. Once a local optimum has been found, i.e., no better neighbors can be discovered, the algorithm sets the current solution as an elite solution and restarts by replacing the current solution with a newly generated one. Furthermore, if dynamic problem changes occur, the problem model is updated accordingly and both the current and elite solutions are updated. The algorithm then reevaluates both solutions and publishes the best. The local search procedure is continued on the current solution and once a local optimum has been found, the elite solution is again replaced by the current solution.

2.3.4. Open-Ended Offspring Selection Genetic Algorithm

The first population-based approach is an open-ended variation on the *offspring selection genetic algorithm* (OSGA) [32]. Offspring selection (OS) is the single population version of the self-adaptive segregative genetic algorithm with simulated annealing aspects (SASEGASA) [33], which is a further development of the SEGA algorithm [34]. Offspring selection introduces self-termination to genetic algorithms by implicitly measuring the convergence of the overall population, and states where not enough genetic material is available anymore to produce better offspring, which can be alleviated by restarting with a new population. OS selects offspring by comparing the fitness of created children against the fitness of respective parents. A comparison factor ϕ is used to define the threshold of what is considered the minimum fitness value of successful offspring, which is computed as follows

$$f_s = f_w + (f_b - f_w) \cdot \phi \tag{1}$$

where f_s is the minimum required fitness to be successful, f_w is the fitness of the worse parent, f_b is the fitness of the better parent and ϕ is the comparison factor. As depicted in Figure 5, only the successful survive into the new population. Conversely, OSGA variations usually need not introduce convergence pressure via parent selection and thus parents are usually chosen uniformly and randomly from the old population. Weak offspring selection requires offspring to be fitter than the worse parent and is achieved with a comparison factor of 0.0, while strict offspring selection (used in the experiments in Section 3) uses a comparison factor of 1.0 and thus requires children to outperform their better parent. Finally, OSGA also requires a minimum fraction of offspring to be successful when creating a new generation, which is defined by the *success ratio*. OSGA also tracks the amount of generated children that are required to generate the necessary amount of successful offspring, which is defined as:

$$ActSelPress = \frac{GeneratedOffspring}{SuccessfulOffspring} \tag{2}$$

Preliminary results showed that the last generations before convergence and restart required significantly more run time than the faster speed up settings allowed while maintaining good qualities. In order to combat this phenomenon, we use the selection pressure after each parallel batch to parameterize a normal approximation of a binomial distribution and, if achieving the number of successful offspring still required lies outside the $2 \cdot \sigma$-interval, the population is reseeded prematurely.

This allows OSGA to track the current selection pressure within the population over generations. In the default implementation, OSGA will terminate once the selection pressure within the population rises to a certain amount. The open-ended version instead restarts with a newly created population to introduce new genetic material (except for the current best solution that is appended to the population as to not lose optimization progress).

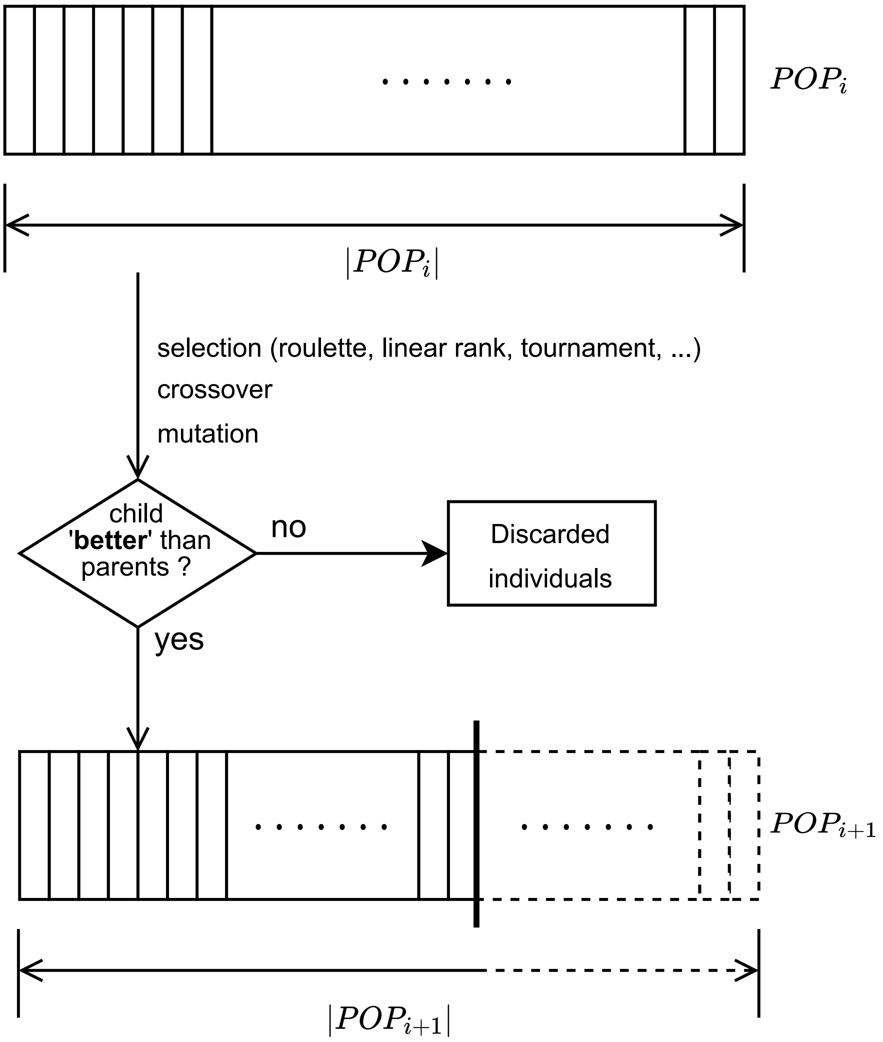

Figure 5. Strict offspring section.

2.3.5. Open-Ended Relevant Alleles Preserving Genetic Algorithm

An open-ended version of the *relevant alleles preserving genetic algorithm* (RAPGA) has been implemented, which is called OERAPGA in short and has proven applicable to discrete encoded dynamic problems in the past [35]. The original RAPGA [36] is a self-adaptive algorithm that automatically adjusts its population size based on offspring success. It features offspring selection similar to the OSGA, but compared to it, no pool of discarded offspring is used. Instead, RAPGA accepts all successful and only successful offspring into the next generation, and adds the elite solutions from the parent population. This results in varying population sizes, depending on how much the offspring outperform their parents. Most importantly, the computational effort decreases as the population loses diversity, making the algorithm faster at the end of the restart cycle.

At the end of each generational cycle, all individuals in the newly created generation are updated to be feasible for the new scenario and reevaluated in order to allow future parent–child comparisons.

2.3.6. Improvement Operators

To create a new offspring from the two parents, a *maximal preservative crossover* (MPX) [37] is used, while mutation is a consecutive application of $0.05 \cdot permutation_length$ two-swap operations with a mutation probability of 0.05 for each child. Note, that since the total number of tasks changes over time, the disruptiveness of the mutation operator stays constant on a per-task basis.

For all solutions of the aforementioned optimization approaches, two additional preprocessing steps are applied before the solution is evaluated. The first preprocessing step tries to keep the level of the first buffer low. This is done by moving all tasks that remove materials from the buffer to the front, while not violating the precedence constraints. This results in a task sequence that removes all possible materials from the buffer before adding the incoming materials to it, reducing the risk of overfilling the buffer.

The second preprocessing step batches tasks based on a predefined set of rules while not violating the precedence constraints. This helps the behavior of the machines regarding the processing of more than one material at a time. To create batches, the batching rules of each machine are used to batch as many materials into one task. Once a rule is broken, the current batch is complete and a new batch with the current material is started.

2.3.7. Optimization Objectives

For many variations of scheduling problems [38] and hybridizations [39], one of the most prominent objectives is the *makespan* of a schedule, which, in this scenario, can be defined as the last finishing time of any job. Since there are numerous solutions that exhibit the same makespan when one machine becomes the bottleneck of the system, we differentiate these solutions by minimizing the overall (sum) duration of materials in the buffers.

2.4. Machine Learning Models

As the focus of this work is to study the interaction of the optimizer and models and not to evaluate the power of any given ML model, we restrict ourselves to online linear regression, by performing a Sherman–Morrison update [40] to the inverted Hessian matrix every time a new data point is collected (i.e., a machine has finished a task). Should the model not yet have full rank, we opt to calculate the Moore–Penrose pseudo-inverse of the Hessian from scratch [41]. This results in a type of model that is fast to calculate and does not have any hyper-parameters that would require tuning and therefore impact results.

As the focus of this work is to study the interaction of the optimizer and models and not to evaluate the power of any given ML model, we restrict ourselves to online linear regression. This choice is mainly because linear regression requires no parameters whose tuning could impact the results. With the Sherman–Morrison update [40], a well established and fast technique exists with witch to update the model when new training data are available. New training data become available whenever a machine finishes a task. The generated input vector (independent variables) for the model consists of the aggregated material properties, the time at which the action started, as well as the material properties, and start and finishing times of the *previous* action performed on this machine (this allows the model to capture simple batching and pipe-lining behaviors). The dependent variable is then simply the time it took to perform the current action. Every machine is assigned an independent model, so transfer learning possibilities from machines of similar types are not in the scope of this work.

In this work, we deliberately assume "cold start" conditions. Only the production layout is known, as well as the buffer capacities b_1 and b_2. The machine learning models predicting machine/setup times are initialized with the same constant value $c = 45$ s that is within an order of magnitude for all actual processing times as not to intentionally mislead them and allow for the generation of initial solutions.

Additionally, when only a few data points are available, the number of training data points might be smaller than the degrees of freedom in the model or several input features might be co-linear to each other. In this case, the linear system is under-determined and we opt to calculate the Moore–Penrose pseudo-inverse of the Hessian matrix from scratch [41]. This essentially chooses from all possible linear models with equal quality, the model where the coefficients display the smallest Euclidean norm (i.e., the coefficients closest to zero). Since online updating the pseudo-matrix is more complex than the Sherman–Morrison formula, and this edge case is mainly relevant in stages where the number of data points is small and training is fast anyway, we recalculate this matrix whenever the calculation of the actual Hessian matrix is not possible.

A note is made that given enough data, the linear models should be able to learn the *true* behavior of all machines in this scenario ($x_1 \cdot x_2$ provides a singular input variable) and more complex structures (artificial neural networks, random forests, etc.) should not be required.

3. Results

In this section, we take a detailed look at the effects apparent in the optimizer–model–simulation system, starting by visualizing and comparing individual executions and leading into aggregated views on algorithm performance.

3.1. Individual Runs

Figure 6 shows the results of one application (a *run*) of the open-ended local search. The top section contains a utilization chart of the actions performed in the simulation. On the one hand, not all machines are equally utilized, as machine 23 is working considerably less then the others. On the other hand, no machine achieves full utilization, indicating that the critical path is not trivially located on a single machine. The bottom section juxtaposes the quality of the best found solution by the optimizer whenever it is published (solid, blue line) to the number of tasks that had to be scheduled in it (dashed, green line). As can be seen, the number of unfinished tasks in the system does not generally increase much over time, while also never reaching zero again after the start, indicating that the system is *stable* with regard to Little's theorem [42] (i.e., the average time it takes to work through a batch of jobs is not larger than the average time it takes for a new batch to arrive). A note is to be made, that comparisons between makespans of different times or worse between different runs or algorithms are often invalid as they depend on the predictions of learned models with different training data. An indication of this can be seen whenever the relation between the solution size and makespan changes significantly; this is especially visible at the "peak" of the makespan in the first minutes of the run, where the models overestimate the time required to perform tasks on certain machines.

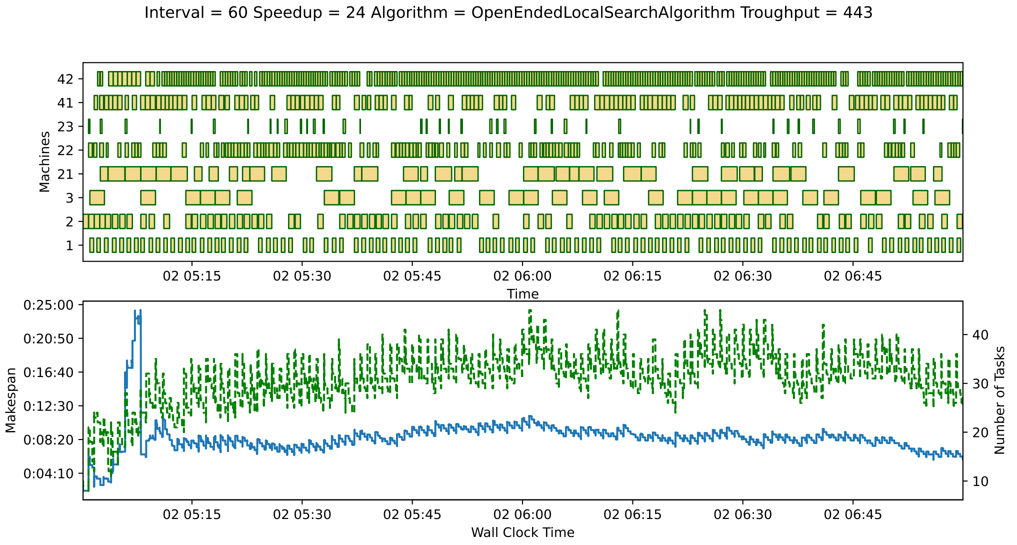

Figure 6. Gantt chart and quality curves of a single run (open-ended local search; OELS).

Figure 7 is the visualization of a run of the SJF optimizer on the same scenario with equal dynamic parameters. Both solution size and makespan increase steadily throughout the run, indicating that the performance of the SJF is not enough to keep the system stable, even though such an under-performance is not easily visible in the Gantt chart.

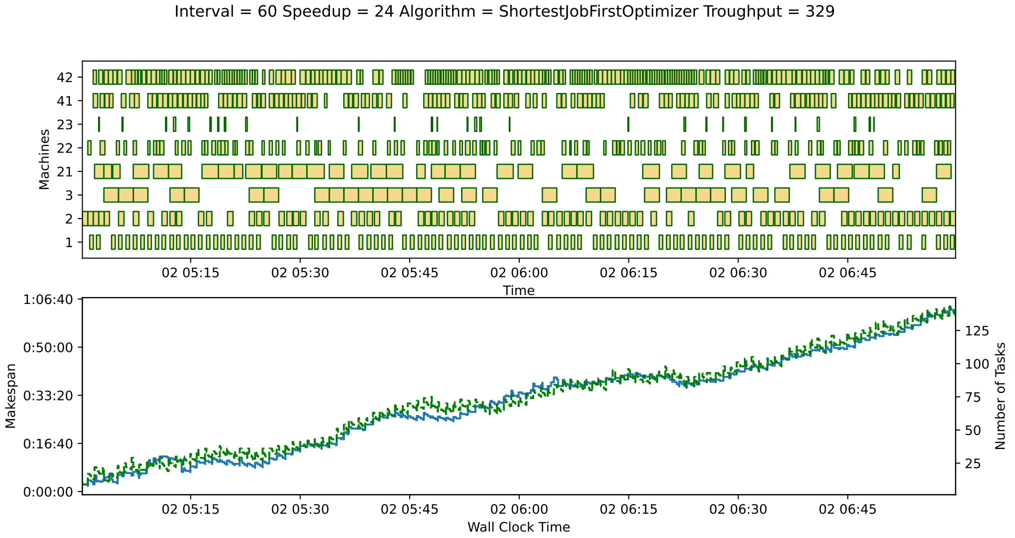

Figure 7. Gantt chart and quality curves of a single run (shortest-job-first; SJF).

3.2. Oversaturation

Figure 8 serves as a reminder that if the arrival rate of new tasks is too high, stable system behaviors can not be achieved by any algorithm. Similar to Figure 6, a spike in the perceived makespan is noticeable in the first few minutes of the run, which is again indicative of overestimated machine times. However, even after the models stabilize, the total number of tasks to schedule steadily increases.

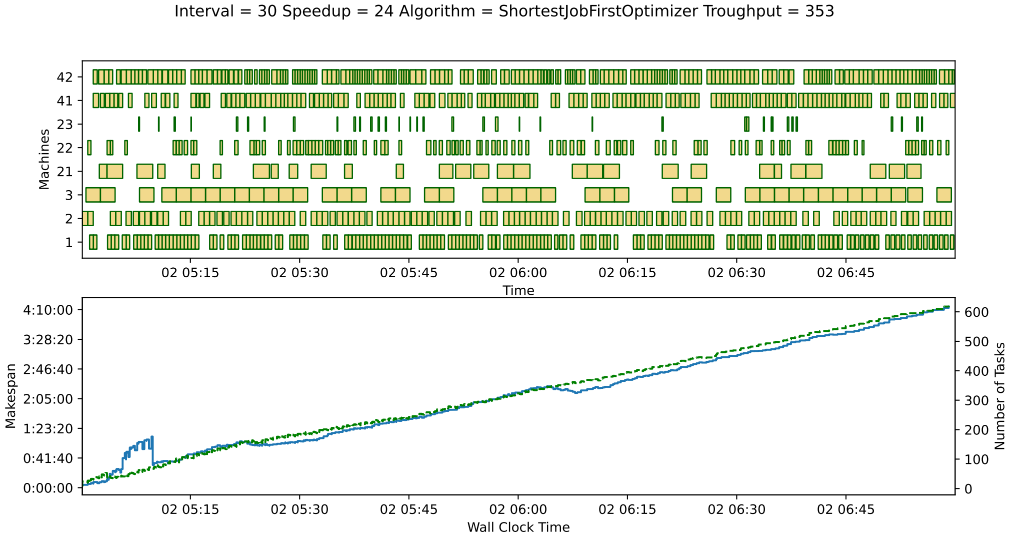

Figure 8. Typical algorithm performance for a problem where tasks arrive faster than they can be processed.

The effects of real-time computation costs can even compound this problem. Figure 9 shows the performance of the OERAPGA (artificially hindered by a high speedup factor). Note that the elongated flat steps in the graph are symptoms of the linearly increasing evaluation time it takes to calculate the quality of a new solution candidate. This makes the algorithm increasingly slow to react to the changes in the system and leads to underutilized machines, as the simulation has to work with outdated and incomplete schedules.

Simultaneously, this results in bad schedules, in general, as the algorithms can no longer converge its population before drastic changes invalidate much of the progress made.

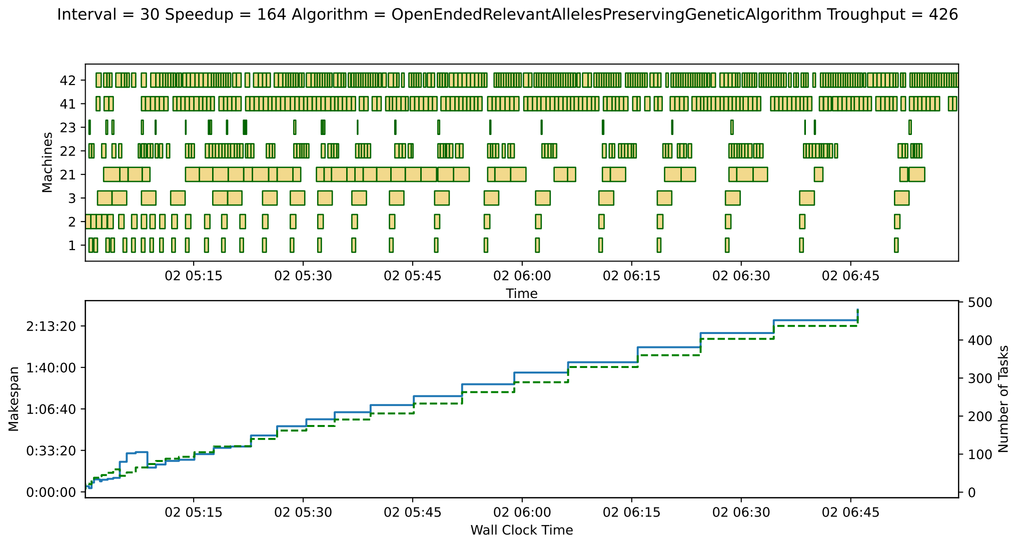

Figure 9. Algorithm performance for a problem instance with very high speedup.

3.3. Model Qualities

On the performance of the regression models, it should be noted that the model and optimizer can enter a feedback loop with the model steering the optimizer toward potentially good solutions and the optimizer reaffirming the model by selecting operations that have good predicted machine times and adding those back into the training data.

Figures 10 and 11 show the demonstrate part of this effect. Figure 10 relates to a single simulation run using the OSGA as an optimizer, while Figure 11 depicts a run with similar settings using the SJF. In both diagrams, the predicted (unmarked line end) and actual machine time (marked line end) for machines 21 and 32 (the machines with setup times) are compared along the simulation time. The predictions are taken just before the point in question is added to the respective model's training set via online update.

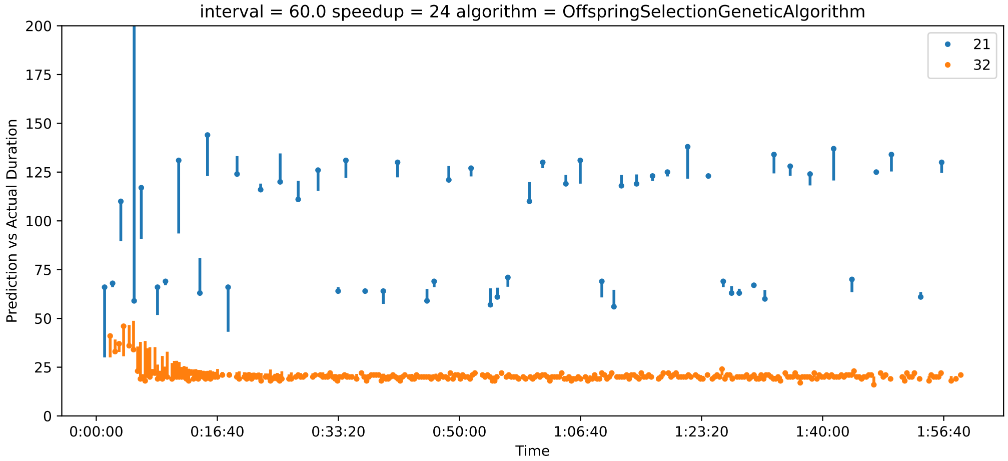

Figure 10. "Well"-performing models for machines with setup times.

At the beginning, both models for both runs display erratic behavior, as is expected from under-defined models with very little training data. Then, all models slowly improve their behavior with the model for machine 32 stabilizing faster as it is provided more training data per time than its machine 21 counter part. However, in the run shown in Figure 11, the model only manages to learn the machine time for operations that do not require a setup, while constantly underestimating situations that do. Whether the model for machine 21 in the run underpinning Figure 10 has overcome this problem, can not be discerned from the data alone; either the model is intentionally steering the optimizer away from requiring setup times or random chance was barely ever required to encounter this situation. For machine 32, conversely, the models manage to distinguish both situations (with the occasional mishap) even with considerably fewer training samples overall.

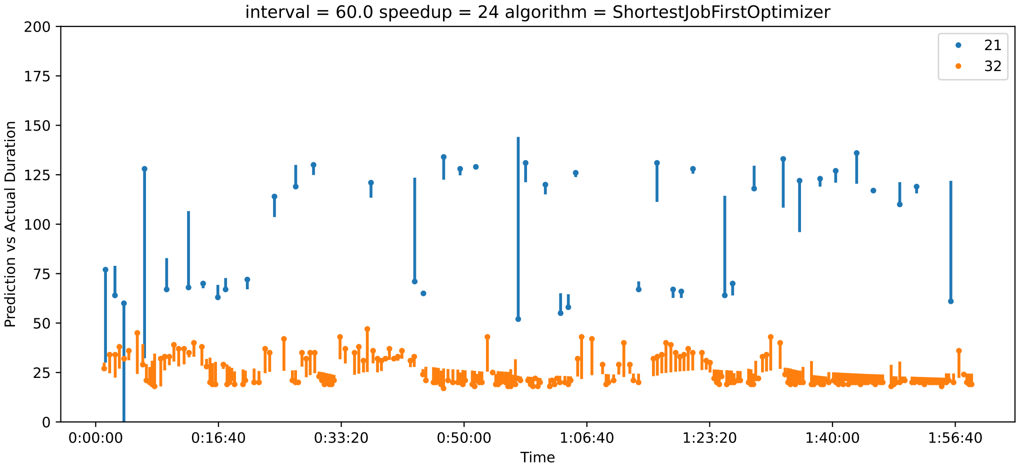

Figure 11. "Mediocre"-performing models for machines with setup times.

3.4. Optimizer Performance

In order to assess the dynamic performance of the optimizer, we compare the total number of completed assembly (Figure 12) and total tasks (Figure 13) as an estimation of the combined systems *throughput*. Both numbers could be exploited if selected as a direct optimization target. Maximizing the number of total finished tasks would lead to extremely full buffer zones, while focusing only on the number of finished products could lead the optimizer to ignore preparatory tasks. Both numbers should therefore be viewed in tandem.

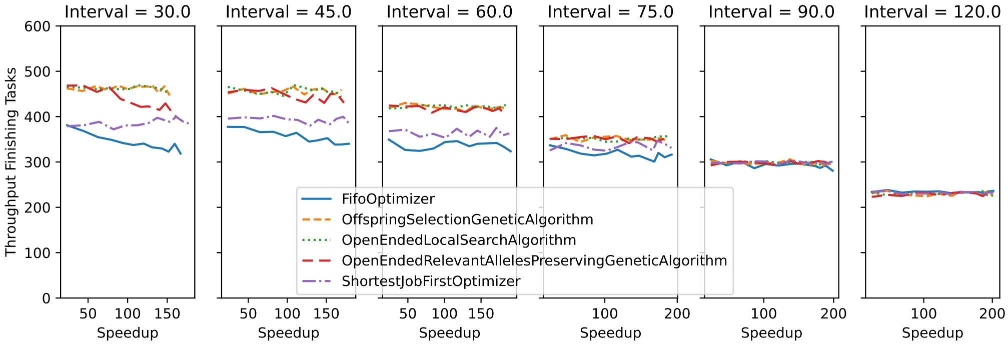

Figure 12. Overall performance of different optimizers measured by finished products.

Both figures plot the respective median performance measure over the speed of the simulation while grouping problem instances by the selected timing interval (in seconds) for the order generation, with larger generation intervals leading to easier problem instances. First, for both the throughput measures for intervals higher than 75 s, there is no noticeable difference between all algorithms, indicating that the inherent capacity of the system is larger than what is required for the slowly arriving orders.

Second, for the harder problem instances, the rankings between algorithms are relatively stable with the informed FIFO algorithm performing worst, and the SJF optimizer outperforming it in terms of finished orders while staying relatively comparable in terms of total tasks. The open-ended variations of the local search and RAPGA perform even better but do not distinguish much from each other, which points to the mutation operator being the main driving force during the search.

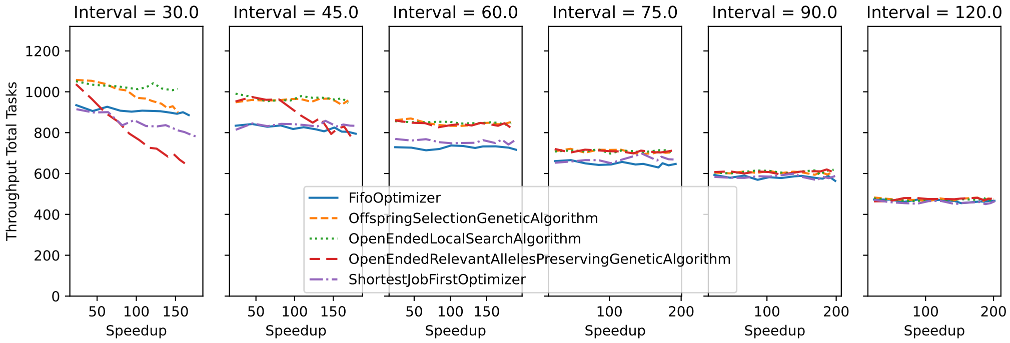

Figure 13. Overall performance of different optimizers measured by finished subtasks.

Third, the total tasks performed by the computationally more expensive OERAPGA seem to deteriorate in short interval problem instances with increasing speed up, which is in line with the observation in Section 3.2 that longer order queues, and the therefore longer evaluation times of solution candidates, can impact the algorithm significantly. Additionally, in the short interval setting, the performance of the SJF optimizer seems to slightly decrease with speedup, although the effect is much less pronounced.

4. Discussion

This work presents a setup for scheduling via dynamic heuristic optimization and integrated machine learning with a focus on asynchronous execution. Effects of real time considerations, algorithm behavior, the "cold start" of machine learning models and model–optimizer interactions were analyzed. The chosen scenario contains several real-world characteristics (i.e., noise, setup times, batching, assembly and disassembly operations, as well as resource/buffer constraints). While not all are simultaneously influential to the system's performance at hand, the described setup could, in principle, accommodate all of these features.

In the computational experiments, the more sophisticated algorithms could outperform the simpler constructions heuristics, but they were also more afflicted by run time issues.

The results in Section 3.3 indicate the need for careful consideration of the integrated machine learning models. Not only the questions of training and prediction speed, retention and un-learning of experienced training data will have to be answered in the future, but also the fact that the optimizer biases the generation of training data, potentially skewing classic error measures, like mean squared error, is an open question going forward. For purely practical considerations, not every model needs to start without an existing data set, which could alleviate the issue, but also quite often data will only be available in the form of knowledge by domain experts, requiring a method to integrate this knowledge into an otherwise automatically updating prediction model. All the while, errors or blind spots in the models could appear and be largely undetected simply because the optimizer avoids creating solutions that would expose them.

As a side note, the demonstrated approach of learning machine behaviors is not applicable to every production system, as it depends on fairly complete, accurate and up-to-date information about almost every aspect of the system. Especially sections of the system with timings that depend largely on human interaction or performance might be difficult to accurately measure and/or predict. Furthermore, in many situations, accurate historic data are scarce, but domain knowledge in the form of constrains, rule-of-thumbs or even physically proven equations might be available, generating a need to combine domain knowledge with machine learning in order to generate accurate assessments of potential solutions.

In order to fully gauge the potential of the considered optimization algorithms, the comparative study will need to be extended to reinforcement learning approaches, more complex neighborhood searches (e.g., variations of variable neighborhood search [43]) and, certainly, a considerably larger set of different machine setups and layouts. Hence, the observed good performance of the LS and OERAPGA are not generalizable as of yet. Here, the reader is referred to the numerous studies that compare algorithmic performance in more controlled setups where timings need not be learned and model and optimizer performance are thus disentangled.

Author Contributions: Conceptualization, B.W., M.H., J.K., S.W. and M.A.; methodology, B.W. and J.K.; software, B.W., M.H. and J.K.; validation, B.W., M.H. and J.K.; data curation, B.W. and M.H.; writing—original draft preparation, B.W., J.K. and M.H.; visualization, B.W., J.K. and M.H.; supervision, S.W.; project administration, S.W.; funding acquisition, S.W., B.W., J.K. and M.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded in part by the Austrian Science Fund (FWF) (I 5315-N). The financial supports by the Austrian Federal Ministry for Digital and Economic Affairs, The National Foundation for Research, Technology and Development and the Christian Doppler Research Association are gratefully acknowledged.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available on request from the corresponding author. The data are not publicly available due to select parts of the code being proprietary.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

DJSP	Dynamic Job Scheduling Problem
ML	Machine Learning
OERAPGA	Open-Ended Relevant Alleles Preserving Genetic Algorithm
OSGA	(open-ended) Offspring Selection Genetic Algorithm
FIFO	First In First Out
LS	Local Search
SJF	Shortest Job First
RAPGA	Relevant Alleles Preserving Genetic Algorithm
OS	Offspring Selection
SASEGASA	Self-Adaptive Segregative Genetic Algorithm with Simulated Annealing Aspects
SEGA	Segregative Genetic Algorithm
MPX	Maximal Preservative Crossover

References

1. Jackson, J.R. Simulation research on job shop production. *Nav. Res. Logist. Q.* **1957**, *4*, 287–295. [CrossRef]
2. Ramasesh, R. Dynamic job shop scheduling: A survey of simulation research. *Omega* **1990**, *18*, 43–57. [CrossRef]
3. Georgiadis, P.; Michaloudis, C. Real-time production planning and control system for job-shop manufacturing: A system dynamics analysis. *Eur. J. Oper. Res.* **2012**, *216*, 94–104. [CrossRef]
4. Cafaro, D.C.; Cerdá, J. Dynamic scheduling of multiproduct pipelines with multiple delivery due dates. *Comput. Chem. Eng.* **2008**, *32*, 728–753. [CrossRef]
5. Cowling, P.I.; Ouelhadj, D.; Petrovic, S. Dynamic scheduling of steel casting and milling using multi-agents. *Prod. Plan. Control* **2004**, *15*, 178–188. [CrossRef]
6. Mohan, J.; Lanka, K.; Rao, A.N. A Review of Dynamic Job Shop Scheduling Techniques. *Procedia Manuf.* **2019**, *30*, 34–39. [CrossRef]
7. Aydin, M.; Öztemel, E. Dynamic job-shop scheduling using reinforcement learning agents. *Robot. Auton. Syst.* **2000**, *33*, 169–178. [CrossRef]
8. Shahrabi, J.; Adibi, M.A.; Mahootchi, M. A reinforcement learning approach to parameter estimation in dynamic job shop scheduling. *Comput. Ind. Eng.* **2017**, *110*, 75–82. [CrossRef]

9. Shyalika, C.; Silva, T.; Karunananda, A. Reinforcement learning in dynamic task scheduling: A review. *SN Comput. Sci.* **2020**, *1*, 1–17. [CrossRef]
10. Lin, S.C.; Goodman, E.D.; Punch, W.F., III. A genetic algorithm approach to dynamic job shop scheduling problem. In Proceedings of the ICGA, East Lansing, MI, USA, 19–23 July 1997; pp. 481–488.
11. Kundakcı, N.; Kulak, O. Hybrid genetic algorithms for minimizing makespan in dynamic job shop scheduling problem. *Comput. Ind. Eng.* **2016**, *96*, 31–51. [CrossRef]
12. Cao, Z.; Zhou, L.; Hu, B.; Lin, C. An adaptive scheduling algorithm for dynamic jobs for dealing with the flexible job shop scheduling problem. *Bus. Inf. Syst. Eng.* **2019**, *61*, 299–309. [CrossRef]
13. Morales-Enciso, S.; Branke, J. Tracking global optima in dynamic environments with efficient global optimization. *Eur. J. Oper. Res.* **2015**, *242*, 744–755. [CrossRef]
14. Shaw, M.J.; Park, S.; Raman, N. Intelligent scheduling with machine learning capabilities: The induction of scheduling knowledge. *IIE Trans.* **1992**, *24*, 156–168. [CrossRef]
15. Aytug, H.; Bhattacharyya, S.; Koehler, G.J.; Snowdon, J.L. A review of machine learning in scheduling. *IEEE Trans. Eng. Manag.* **1994**, *41*, 165–171. [CrossRef]
16. Priore, P.; De La Fuente, D.; Gomez, A.; Puente, J. A review of machine learning in dynamic scheduling of flexible manufacturing systems. *Ai Edam* **2001**, *15*, 251–263. [CrossRef]
17. Priore, P.; de la Fuente, D.; Puente, J.; Parreño, J. A comparison of machine-learning algorithms for dynamic scheduling of flexible manufacturing systems. *Eng. Appl. Artif. Intell.* **2006**, *19*, 247–255. [CrossRef]
18. Priore, P.; Gomez, A.; Pino, R.; Rosillo, R. Dynamic scheduling of manufacturing systems using machine learning: An updated review. *Ai Edam* **2014**, *28*, 83–97. [CrossRef]
19. Li, S.; Yu, T.; Cao, X.; Pei, Z.; Yi, W.; Chen, Y.; Lv, R. Machine learning-based scheduling: A bibliometric perspective. *IET Collab. Intell. Manuf.* **2021**, *3*, 131–146. [CrossRef]
20. Lee, C.Y.; Piramuthu, S.; Tsai, Y.K. Job shop scheduling with a genetic algorithm and machine learning. *Int. J. Prod. Res.* **1997**, *35*, 1171–1191. [CrossRef]
21. Zhang, M.; Tao, F.; Nee, A. Digital twin enhanced dynamic job-shop scheduling. *J. Manuf. Syst.* **2021**, *58*, 146–156. [CrossRef]
22. Luo, S. Dynamic scheduling for flexible job shop with new job insertions by deep reinforcement learning. *Appl. Soft Comput.* **2020**, *91*, 106208. [CrossRef]
23. Tuli, S.; Ilager, S.; Ramamohanarao, K.; Buyya, R. Dynamic scheduling for stochastic edge-cloud computing environments using a3c learning and residual recurrent neural networks. *IEEE Trans. Mob. Comput.* **2020**, *21*, 940–954. [CrossRef]
24. Hu, L.; Liu, Z.; Hu, W.; Wang, Y.; Tan, J.; Wu, F. Petri-net-based dynamic scheduling of flexible manufacturing system via deep reinforcement learning with graph convolutional network. *J. Manuf. Syst.* **2020**, *55*, 1–14. [CrossRef]
25. Baruwa, O.T.; Piera, M.A.; Guasch, A. Deadlock-free scheduling method for flexible manufacturing systems based on timed colored Petri nets and anytime heuristic search. *IEEE Trans. Syst. Man Cybern. Syst.* **2014**, *45*, 831–846. [CrossRef]
26. Lopez, V.; Jokanovic, A.; D'Amico, M.; Garcia, M.; Sirvent, R.; Corbalan, J. Djsb: Dynamic job scheduling benchmark. In Proceedings of the Job Scheduling Strategies for Parallel Processing: 21st International Workshop, JSSPP 2017, Orlando, FL, USA, 2 June 2017; Springer: Berlin/Heidelberg, Germany, 2018; pp. 174–188.
27. Beham, A.; Leitner, S.; Karder, J.; Werth, B.; Wagner, S. Dynstack: A benchmarking framework for dynamic optimization problems in warehouse operations. In Proceedings of the Genetic and Evolutionary Computation Conference Companion, Boston, MA, USA, 9–13 July 2022; pp. 1984–1991.
28. Beham, A.; Kronberger, G.; Karder, J.; Kommenda, M.; Scheibenpflug, A.; Wagner, S.; Affenzeller, M. Integrated simulation and optimization in heuristiclab. In Proceedings of the 26th European Modeling and Simulation Symposium EMSS 2014, Bordeaux, France, 10–12 September 2014; pp. 418–423.
29. Karder, J.; Beham, A.; Werth, B.; Wagner, S.; Affenzeller, M. Integrated machine learning in open-ended crane scheduling: Learning movement sSpeeds and service times. *Procedia Comput. Sci.* **2022**, *200*, 1031–1040. [CrossRef]
30. Liang, J.; Ban, X.; Yu, K.; Qu, B.; Qiao, K.; Yue, C.; Chen, K.; Tan, K.C. A survey on evolutionary constrained multiobjective optimization. *IEEE Trans. Evol. Comput.* **2022**, *27*, 201–221. [CrossRef]
31. Zhan, Z.H.; Shi, L.; Tan, K.C.; Zhang, J. A survey on evolutionary computation for complex continuous optimization. *Artif. Intell. Rev.* **2022**, *55*, 59–110. [CrossRef]
32. Affenzeller, M.; Wagner, S. SASEGASA: An Evolutionary Algorithm for Retarding Premature Convergence by Self-Adaptive Selection Pressure Steering. In Proceedings of the Artificial Neural Nets Problem Solving Methods, 7th International Work-Conference on Artificial and Natural Neural Networks, IWANN 2003, Maó, Menorca, Spain, 3–6 June 2003; Mira, J., Alvarez, J.R., Eds.; Springer: Berlin/Heidelberg, Germany, 2003; Volume 2686, pp. 438–445.
33. Affenzeller, M.; Wagner, S. Offspring selection: A new self-adaptive selection scheme for genetic algorithms. In Proceedings of the Adaptive and Natural Computing Algorithms, 8th International Conference, ICANNGA 2007, Warsaw, Poland, 11–14 April 2007; Ribeiro, B., Albrecht, R.F., Dobnikar, A., Pearson, D.W., Steele, N.C., Eds.; Springer: Berlin/Heidelberg, Germany, 2005; pp. 218–221.
34. Affenzeller, M. Segregative genetic algorithms (SEGA): A hybrid superstructure upwards compatible to genetic algorithms for retarding premature convergence. *Int. J. Comput. Syst. Signals (IJCSS)* **2001**, *2*, 18–32.

35. Karder, J.; Werth, B.; Beham, A.; Wagner, S.; Affenzeller, M. Analysis and Handling of Dynamic Problem Changes in Open-Ended Optimization. In Proceedings of the International Conference on Computer Aided Systems Theory, 18th International Conference, Las Palmas de Gran Canaria, Spain, 20–25 February 2022; Springer: Berlin/Heidelberg, Germany, 2022; pp. 61–68.
36. Affenzeller, M.; Wagner, S.; Winkler, S. Self-adaptive population size adjustment for genetic algorithms. In Proceedings of the International Conference on Computer Aided Systems Theory, 1th International Conference on Computer Aided Systems Theory, Las Palmas de Gran Canaria, Spain, 12–16 February 2007; Springer: Berlin/Heidelberg, Germany, 2007; pp. 820–828.
37. Mühlenbein, H. Parallel genetic algorithms in combinatorial optimization. In *Computer Science and Operations Research*; Balci, O., Sharda, R., Zenios, S.A., Eds.; Pergamon: Amsterdam, The Netherlands, 1992; pp. 441–453. [CrossRef]
38. Ahmadian, M.M.; Khatami, M.; Salehipour, A.; Cheng, T. Four decades of research on the open-shop scheduling problem to minimize the makespan. *Eur. J. Oper. Res.* **2021**, *295*, 399–426. [CrossRef]
39. Fernandez-Viagas, V.; Perez-Gonzalez, P.; Framinan, J.M. Efficiency of the solution representations for the hybrid flow shop scheduling problem with makespan objective. *Comput. Oper. Res.* **2019**, *109*, 77–88. [CrossRef]
40. Sherman, J. Adjustment of an inverse matrix corresponding to changes in the elements of a given column or a given row of the original matrix. *Ann. Math. Stat.* **1949**, *20*, 621.
41. Penrose, R. A generalized inverse for matrices. *Proc. Math. Proc. Camb. Philos. Soc.* **1955**, *51*, 406–413. [CrossRef]
42. Whitt, W. A review of L = λW and extensions. *Queueing Syst.* **1991**, *9*, 235–268. [CrossRef]
43. Mladenović, N.; Hansen, P. Variable neighborhood search. *Comput. Oper. Res.* **1997**, *24*, 1097–1100. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.