A Reinforcement Learning Approach to Solving Hybrid Flexible Flowline Scheduling Problems

Bert Van Vreckem · Dmitriy Borodin · Wim De Bruyn · Ann Nowé

Abstract In this paper, we present a method based on Learning Automata to solve Hybrid Flexible Flowline Scheduling Problems (HFFSP) with additional constraints like sequence dependent setup times, precedence relations between jobs and machine eligibility. This category of production scheduling problems is noteworthy because it involves several types of constraints that occur in complex real-life production scheduling problems like those in process industry and batch production. In the proposed technique, Learning Automata play a dispersion game to determine the order of jobs to be processed in a way that makespan is minimized, and precedence constraint violations are avoided. Experiments on a set of benchmark problems indicate that this method can yield better results than the ones known until now.

1 Introduction

Within the vast research field of scheduling, the optimisation of production processes is an area that has known considerable progress in the last years. Originally, the only thing computers could solve to optimality, or even to a feasible solution, were toy problems. In some industries, where the production process is not a very complex one, automated scheduling solutions have been in use for some time. On the other hand, in industries like the process industry, where very hard constraints are imposed on production schedules, the planning and scheduling is still mainly a manual job, done
by highly experienced people that know the production line inside out. Recent advances
in computer hardware however (more specifically faster processors and algorithms) have
made it possible for production scheduling problems of more realistic size to be tackled.

The typical approach in the operational research community is to model the scheduling
problem as a Mixed Integer (Linear) Programming model (MIP/MILP) (Mendez et al, 2006) and then solve it to optimality with methods that are suitable for this
formalisation. Nevertheless, this method still has its limits. Setting up a MILP model
is hard, and involves a particular expertise. The way constraints are formulated can
have a serious impact on the efficiency of the solver. Additionally, this model is rarely
reusable for other, even similar production processes and is hard to adapt to changes in
the plant. Furthermore, in order to make the problem tractable, often simplifications
are introduced that no longer reflect reality (e.g. only allowing to allocate resources
for blocks of four hours, which would be useless in a real-life setting). This greatly
decrees the usefulness of the solutions found (Gicquel et al, 2012).

Because of the above reasons, heuristics and meta-heuristics are still being con-
sidered as an interesting alternative to exact methods like MILP, as they are able to
produce solutions of high quality in a reasonable time. The main disadvantage is that
they no longer guarantee that found solutions are optimal, and it is generally not pos-
sible to estimate how far they are from an optimal solution. However, plant managers
are usually neither looking for the optimum at all (especially if that solution turns
out to be unusable in practice), nor for a completely automated black box scheduling
algorithm. In many cases, a “better than manual” solution would already be a great
improvement over current practice.

One approach is to start with a MILP formulation and apply techniques from the
class of heuristics or meta-heuristics. The advantages include relatively fast response
time, tractability of the problem, scalability while it is still required to define a MILP
model and incorporate an appropriate heuristic within the defined MILP. An overview
is provided in the survey by Ribas et al (2010). For a real-life case-study of applying
genetic algorithm “on top” of a MILP model we refer to Borodin et al (2011).

2 Hybrid Flexible Flowline Scheduling Problems

In this paper, we introduce a learning approach, and apply it to Hybrid Flexible Flow
line Scheduling Problems (HFFSP) with a set of additional restrictions (Ruiz et al,
2008). This is a particularly interesting type of scheduling problems, as it involves
several hard constraints that occur in complex real-life production processes. In a Flow
line Scheduling Problem (FSP), a number of jobs have to be processed in a set of
consecutive stages. In each stage, there are a number of unrelated parallel machines
that can process jobs (this is the hybrid case of FSP). Jobs are processed at each stage
in the same order and by exactly one machine. Some stages can be skipped (this is the flexible case of FSP) and for some jobs, not all machines are eligible to process it. Jobs
may have precedence relations. There may be a time lag between the processing of a
job in two consecutive stages that can be either positive or negative. Machine-based
sequence-dependent setup times are also considered.

First, a MIP model for HFFSPs is introduced that is applied to a set of benchmark
problems first published in (Ruiz et al, 2008). Subsequent chapters of the thesis discuss
the solution strategies based on CPLEX, heuristics, genetic algorithms, and local search
techniques. Even in smaller problems with five jobs, given the extra real-life constraints, CPLEX was unable to find an optimal solution for all instances. In larger problems, it found the optimum in only a fraction of the cases. For example, in a subset of problem instances with 15 jobs, two stages and one machine per stage (which does not seem to be an overly large problem), only in 0.26% of the cases an optimal solution was found. A statistical analysis of the MIP model revealed that the difficulty of the problem instances is not caused by the number of jobs, but primarily by the precedence relations between jobs.

A typical approach for solving an FSP is to first determine the order in which the jobs are processed, i.e. a job permutation. In the next phase, machines in each production stage are assigned to jobs, and start times are determined according to machine availability. The job permutation and machine assignments will determine the quality of the schedule, e.g. the makespan. Heuristics or machine learning techniques can be used in either or both stages. For example, Kurz and Askin (2004) research the application of several different methods to flexible flow line scheduling problems with sequence-dependent setup times, namely greedy heuristic, flow line methods, Insertion Heuristic and Random Keys Genetic Algorithm (RKGA). Each of these heuristics assumes the job ordering first, and then assignment of jobs to machines. RKGA was found to be superior in the majority of cases in terms of solution quality and solution time. Tavakkoli-Moghaddam et al (2009) introduced a method where the job permutation is selected by tabu search and then the assignment/timing procedure uses a number of dispatching rules to determine the detailed schedule and resulting makespan. Zandieh et al (2009) proposed a robust genetic algorithm for solving real flow line scheduling problems. The solution representation of the genetic algorithm itself comprised both job sequencing and machine assignment parts, though a set of underlying heuristics was used in the sense of machine assignment rules. A bit different approach was demonstrated by Borodin et al (2011) where the authors first performed the job assignment to machines using genetic algorithm, and then determined the job order. The approach was based on the MILP formulation of the problem: once the job assignment to machines had been fixed, the original MILP resulted in an easy-to-solve LP, and its solution allowed to construct a resulting schedule.

This paper contributes to the research domain by introducing a Reinforcement Learning (RL) approach based on Learning Automata (LA) for solving complex real-life HFFSPs. Rather than starting with a MIP formulation of the problem, we used a straightforward model that is easily implemented in a mainstream programming language, without trying to represent constraints in a way that improves efficiency of a solver. We use Learning Automata to find job permutations that lead to high quality schedules. We currently do not apply learning in the machine assignment phase, but we use a heuristic that performed well in previous experiments (Urlings, 2010).

The rest of the paper comprises the following sections. In the Learning Automata section RL and LA are introduced, after which we describe the proposed algorithm in detail. Then, computational results are presented and compared to the best known ones for the set of benchmarks in Experiments. Finally, we discuss conclusions and propose ideas for future research.
3 Learning Automata

Reinforcement Learning (RL) is a technique that allows an agent to learn how to maximize a numerical reward signal. A learning agent must determine the actions that yield the most reward by a trial-and-error approach, in contrast to supervised learning where the best action would be known a priori. In real-life cases, the selected actions affect not only the immediate reward but also the reward of the next situation and, consequently, all following rewards. The trial-and-error search and delayed reward are the two most important distinguishing features of RL (Sutton and Barto, 1998).

In a RL setting, an agent perceives the state of its environment at instant \( t \) and based on this, chooses an action \( a(t) \) from a specific set \( A = a_1, \ldots, a_n \). This action may change the state of the environment and generates a reinforcement signal \( r(t) \) that is received by the agent. The learning task consists of finding a strategy that maximizes long term cumulative reward.

Learning Automata (LA) (Narendra and Thathachar, 1974) are RL agents that determine their actions according to a probability distribution \( p(t) = p_1, \ldots, p_n \), with \( p_i = \text{prob}[a(t) = a_i] \), and with the condition that \( \sum_{i=1}^{n} p_i = 1 \). Initially, all actions are chosen according to the same probability \( p_i = \frac{1}{n} \), and these are updated using the reinforcement signal from the environment, usually 0 or 1 for a penalty, resp. reward signal. The general update rule for Learning Automata is formulated as follows:

\[
\begin{align*}
p_i(t+1) &= p_i(t) + \alpha_{\text{rew}} r(t)(1 - p_i(t)) - \alpha_{\text{pen}} (1 - r(t))p_i(t) \\
& \quad \text{if } a_1 \text{ is the action taken at instant } t \\
p_j(t+1) &= p_j(t) - \alpha_{\text{rew}} r(t)p_j(t) + \alpha_{\text{pen}} (1 - r(t)) \left( \frac{1}{n-1} - p_j(t) \right) \\
& \quad \text{if } a_j \neq a_i
\end{align*}
\]

with \( \alpha_{\text{rew}} \) and \( \alpha_{\text{pen}} \in [0, 1] \) the reward and penalty rate. If \( \alpha_{\text{rew}} = \alpha_{\text{pen}} \), the update rule is called linear reward-penalty (\( LR-P \)), when \( \alpha_{\text{pen}} = 0 \), it is referred to as linear reward-inaction (\( LR-I \)). There are more variations to this update scheme, but those are beyond the scope of this paper. In a stationary environment, these update schemes can be proven to converge to an optimal strategy (Narendra and Thathachar, 1974).

4 Learning permutations with precedence constraints

Wauters (2012) proposes a method called Probabilistic Basic Simple Strategy (PBSS) that uses Learning Automata to learn a permutation, and he applies it a.o. to solve complex Resource Constrained Multi-Project Scheduling Problems (RCMPS). Every position in the permutation is assigned a Learning Automaton that chooses the job for that position in the permutation, using its probability distribution. Every time instant \( t \), the LAs play a dispersion game, i.e. each has to choose a unique job. Initially, since action probabilities are equal, some LAs will choose the same job. If such collisions occur, they will have to try again. Those LAs that did choose a unique job, will stick to their choice, or differently put, the action probability for that job is set to 1, and for
all other jobs to 0. In addition, other automatons will be forbidden to choose that job for
the following runs of the dispersion game, i.e. action probabilities for that job in those
other LAs is set to 0. For each agent, the probabilities that are still non-zero are then
renormalized so that their sum equals 1. Then, the dispersion game is played again,
until all LAs have chosen a unique job. The resulting permutation is then evaluated
according to some quality function. If the result is better than the best one found thus
far, \( r(t) = 1 \), if not, \( r(t) = 0 \). This signal is then used to update the probabilities from
the beginning of the time instant, before the dispersion game started, according to
\( L_{R-I} \). In the next time instant \( t+1 \), the LAs play the dispersion game again with the
updated probabilities. After a number of iterations, the action probabilities of the LAs
converge to pure policies where one specific, unique, action is chosen that, combined,
yield a permutation of high quality. This method can be used, and performs extremely
well, in many situations where learning permutations is involved.

Unfortunately, PBSS can not be readily applied to the HFFSP. In this class of
scheduling problems, precedence constraints between jobs may severely limit the num-
ber of valid permutations. Given the update scheme proposed in PBSS, agents don’t
learn to avoid violations of precedence constraints. This is a consequence of the \( L_{R-I} \)
update scheme that only learns from positive feedback. Indeed, in preliminary exper-
iments on instances with precedence constraints, the Learning Automata never came
up with a valid schedule, therefore no learning happened at all. In this paper, we pro-
pose a variation of PBSS where generating invalid schedules also results in a learning
experience:

- If the job permutation is valid, perform a \( L_{R-I} \) update in all agents, depending on
  the resulting makespan \( ms \) and best makespan until now \( ms_{best} \):
  - improved: \( r(t) = 1 \);
  - equally good: \( r(t) = 0.5 \);
  - worse: \( r(t) = ms_{best}/ms/2 \);
  - no valid schedule found: \( r(t) = 0 \);
- If the job permutation is invalid, perform a \( L_{R-P} \) update with \( r(t) = 0 \) for all
  agents that are involved in the violation of precedence constraints.

The main difference from PBSS is that we allow to learn explicitly from negative
experience by extending the \( L_{R-I} \) update with an \( L_{R-P} \) component.

In the following section, we present some results of experiments on HFFSP instances
with this update scheme.

5 Experiments

We used the problem set from Ruiz et al (2008)\(^1\) to evaluate the performance of Learn-
ing Automata for solving HFFSPs. The set is subdivided according to the number of
jobs, respectively 5, 7, 9, 11, 13, and 15. For each subset, there are 96 instances with
other specific characteristics (e.g. precedence constraints or not, skipping stages or not,
\ldots ).

We used 0.1 for the reward \( \alpha_{rew} \) rate and 0.5 for the penalty rate \( \alpha_{pen} \). We did not
perform careful parameter tuning, but these values seem to perform well. The algorithm
was run until the LAs converged to a pure strategy, where each agent chooses a unique

\(^1\) Available at http://soa.iti.es/problem-instances
Table 1 Summary of results. Columns denote the instance sets, according to the number of jobs. The top row are mean relative deviations (in percentages) w.r.t. the best known values. Next, best relative deviations for each set are given (also in percentages). Negative values are improvements over the best known result. The three bottom rows denote the number of instances where we got respectively improvements, the same, and worse results w.r.t. best known values.

<table>
<thead>
<tr>
<th>Instance set</th>
<th>5</th>
<th>7</th>
<th>9</th>
<th>11</th>
<th>13</th>
<th>15</th>
<th>overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean RD (%)</td>
<td>0.0697</td>
<td>2.0131</td>
<td>1.1568</td>
<td>1.6565</td>
<td>3.7294</td>
<td>7.9189</td>
<td>2.7484</td>
</tr>
<tr>
<td>best RD (%)</td>
<td>-35.70</td>
<td>-24.71</td>
<td>-26.92</td>
<td>-21.10</td>
<td>-43.34</td>
<td>-10.46</td>
<td>-43.34</td>
</tr>
<tr>
<td>instances improved</td>
<td>11</td>
<td>12</td>
<td>18</td>
<td>12</td>
<td>9</td>
<td>6</td>
<td>68</td>
</tr>
<tr>
<td>instances equal</td>
<td>62</td>
<td>40</td>
<td>19</td>
<td>18</td>
<td>8</td>
<td>7</td>
<td>154</td>
</tr>
<tr>
<td>instances worse</td>
<td>23</td>
<td>44</td>
<td>59</td>
<td>66</td>
<td>79</td>
<td>82</td>
<td>354</td>
</tr>
</tbody>
</table>

Fig. 1 Box plots for the relative deviations of makespan (in percent) from the best known results. Negative values indicate an improvement. One may expect that results will be worse as the number of jobs increases, but this is evidently not the case in the set of instances with 13 jobs that yields better results than the ones with 11 jobs. Remark that the number of jobs is not the only indicator of the difficulty of the problem instance. For example, precedence constraints between jobs may make the problem much more complex.

For each instance, the schedule with the shortest makespan that was encountered during a run was chosen as the result. In all subsets, we obtained schedules with a shorter makespan than the best known until now for some of the instances. Table 1 contains a summary of the results. In Figure 1, we show box plots for the relative deviations with regard to the best known makespans in each instance subset. Finally, Figure 2 shows an example of a schedule where substantial improvement was achieved.
6 Discussion

In this paper, we introduced a technique based on Learning Automata to solve highly constrained Hybrid Flexible Flowshop Scheduling Problems. These are not made difficult by the problem size (specifically, the number of jobs), but by other constraints that occur in real-life scheduling problems, in particular precedence relations. We started from a simple, natural model of the problem without trying to represent constraints in a way that could make the algorithm more efficient. We presented the results of experiments on a set of benchmark problems and showed that we could improve on the best known solutions in several cases. By tuning the experimental parameters, such as the reward and penalty rate, we should be able to get even better results.

At this stage, we only focused on determining the order in which jobs are scheduled, but did not look at machine assignments in great detail. This area should also provide opportunities for improving results.

It is interesting to note that the $LR-P$ update rule is considered to have weaker convergence guarantees than $LR-I$ (Narendra and Thathachar, 1974). However, in this setting with precedence constraints, there are only few or no positive learning experiences. The Learning Automata would often select an invalid job sequence, and are unable to avoid them. The $LR-P$ component of our solution strategy appears to help achieve convergence more quickly. On larger instances with precedence constraints, the algorithm has more difficulties to converge, so further investigation into its behaviour is definitely needed.

References
