Solving the Paintshop Scheduling Problem with Memetic Algorithms

Wolfgang Weintritt
wolfgang.weintritt@gmail.com
Databases and Artificial Intelligence Group
TU Wien
Vienna, Austria

Nysret Musliu
musliu@dbai.tuwien.ac.at
Christian Doppler Laboratory for Artificial Intelligence and Optimization for Planning and Scheduling, DBAI
TU Wien
Vienna, Austria

Felix Winter
winter@dbai.tuwien.ac.at
Christian Doppler Laboratory for Artificial Intelligence and Optimization for Planning and Scheduling, DBAI
TU Wien
Vienna, Austria

ABSTRACT

Finding efficient production schedules for automotive paint shops is a challenging task and several paint shop problem variants have been investigated in the past. In this work we focus on a recently introduced real-life paint shop scheduling problem appearing in the automotive supply industry where car parts, which need to be painted, are placed upon carrier devices. These carriers are placed on a conveyor belt and moved into painting cabins, where robots apply the paint. The aim is to find an optimized production schedule for the painting of car parts.

In this paper, we propose a memetic algorithm to solve this problem. An initial population is generated, followed by the constant evolution of generations. Selection, crossover, mutation, and local improvement operators are applied in each generation. We design three novel crossover operators that consider problem-specific knowledge. Finally, we carefully configure our algorithm, including automated and manual parameter tuning.

Using a set of available real-life benchmark instances from the literature, we perform an extensive experimental evaluation of our algorithm. The experimental results show that our memetic algorithm yields competitive results for small- and medium-sized instances and is able to set new upper bounds for some of the problem instances.

CCS CONCEPTS

• **Computing methodologies** → **Search methodologies**; **Randomized search**; **Planning and scheduling**;

KEYWORDS

Paint shop scheduling, memetic algorithms, production scheduling

ACM Reference Format:
Wolfgang Weintritt, Nysret Musliu, and Felix Winter. 2021. Solving the Paintshop Scheduling Problem with Memetic Algorithms. In *2021 Genetic and Evolutionary Computation Conference (GECCO '21), July 10–14, 2021, Lille, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3449639.3459375

GECCO '21, July 10–14, 2021, Lille, France

ACM ISBN 978-1-4503-8350-9/21/07…$15.00
https://doi.org/10.1145/3449639.3459375

1 INTRODUCTION

Each day, large amounts of synthetic material pieces need to be painted in the automotive supply industry's paint shops. As the process of painting is costly and time-consuming, the industry's paint shops are highly automated. Material pieces are placed on carrier devices, which in turn are placed on a conveyor belt. The carrier devices are moved to painting cabins, where several painting robots paint the materials.

Usually human planners are constructing the production schedules for such problems. However, due to the long planning horizons and the tight due dates as well as a lot of other constraints, it takes a lot of time for human planners to create the schedules, and they are normally not able to find optimal solutions in practice. Therefore, there is a strong need for automated approaches to generate effective schedules for this challenging problem.

In the literature this scheduling problem has recently been introduced in [22], and is called the *Paint Shop Scheduling Problem* (PSSP). It is a complex combinatorial problem, and the decision variant of the problem was shown to be NP-complete in [21].

The PSSP imposes a number of constraints in addition to due dates. Those include a limited amount of carrier devices, forbidden color and carrier sequences, limited capacities, and minimum and maximum block sizes for carriers of the same type. The combined complexity of these constraints separates the PSSP from other automotive scheduling problems. The PSSP has a cost function that combines two main objectives - the number of color changes in the production sequence and the number of carrier device changes between production cycles should both be minimized.

Recently, approaches to solve the PSSP have been proposed in [22] and [21]. An exact approach from [21] has shown to work well for small instances and could provide several optimal solutions. For large instances optimal solutions are still unknown. However, heuristics can generate feasible solutions in a reasonable time - [22] achieved good results with simulated annealing.

In this paper we investigate memetic algorithms [13] for the PSSP. Memetic algorithms have been successfully applied to other scheduling problems (e.g. [1, 10, 11, 20]). However, to the best of our knowledge, the PSSP has not been tackled with memetic algorithms.

The main contributions of this paper are:

- We design and implement a memetic algorithm for the PSSP and propose three novel crossover operators as well as different population construction strategies. The algorithm is highly parameterizable, which helps in evaluating the performance of the different construction strategies and memetic operators.
- We experimentally evaluate the algorithm's performance on problem instances from the literature. Via automated and manual parameter tuning we optimize our algorithm's performance and further analyze the impact of different algorithm parameters.
- We compare our algorithm's results with results produced by state-of-the-art approaches. An experimental evaluation shows that our method produces competitive results for many problem instances and can improve upper bounds for some of the real-life instances.

2 THE PAINT SHOP SCHEDULING PROBLEM

In the automotive supply industry's paint shops, a large number of car parts need to be painted each day. As a paint shop usually supplies different car manufacturers, many different car parts have to be painted, such as engine covers, bumpers, and wheel rims. Just-in-time manufacturing [17] is a commonly used concept in the car industry, which forces suppliers to adhere to tight due-dates. Therefore, the main goal of the PSSP is the construction of feasible production sequences which fulfill all due dates.

During production multiple car parts are transported on carrying devices that automatically move through the paint shop on a circular conveyor belt system. Because of the paint shop's circular layout, the schedule is organized in rounds (a round refers to the processing of a set of carriers on the full conveyor belt cycle). A carrier may transport well-defined combinations of raw material parts which are loaded by paint shop employees at the beginning of each production round. Then, carriers are moved to the painting cabins, where the raw material parts are painted by several painting robots. After the carriers return from the painting cabins, the finished parts are unloaded by paint shop employees. The empty carrier may then be removed from the conveyor belt system, or can be reused for the next production round where it is reloaded with new raw material parts.

A paint shop schedule can be represented in tabular form, where columns are rounds, and table cells represent carrier type, carrier configuration, and the proposed color. Figure 1 depicts a simple example schedule.

The example schedule shown in Figure 1 consists of three consecutive rounds (R1, R2, R3). Round R1 schedules 5 carriers of different type, color, and material configurations. The first and second carrier in round R1 are carriers of type A that use a light gray color and material configuration 1. The third carrier uses the same type and color, but assigns material configuration 2. Finally, the carriers at positions 4 and 5 in round R1 are of type B, use a dark gray color and material configurations 1 and 2. Similarly, rounds R2 and R3 also schedule a sequence of 5 carriers, but here different type, color, and material configurations are used.

The PSSP has a cost function that combines two main objectives - to minimize the number of color changes in the carrier sequence and the number of carrier device changes between two consecutive rounds. The practical benefit of these objectives is to reduce waste and save costs. A good schedule should group requests with similar colors to reduce costs. Besides, it is preferential to keep the number of carrier changes between rounds as low as possible, since they may lead to delays in the schedule. Therefore, as many carrier devices as possible should be reused, but if necessary carrier devices can also be inserted and removed between consecutive rounds. The maximum amount of reusable carriers between consecutive rounds is determined by the longest common subsequence [6] of the two associated carrier sequences, as exactly those carrier types which are appearing in a longest common subsequence can be reused.

	R1	*R2*	*R3*	...
1	A1	A2	C1	...
2	A1	A2	C2	...
3	A2	C1	C3	...
4	B1	B2	B1	...
5	B2	B3	B2	...

Figure 1: A PSSP schedule in tabular form for three rounds (see also [22]).

2.1 Hard Constraints

Feasible paint shop schedules have to fulfill several hard constraints:

- All demands must be satisfied within time (overproduction is allowed).
- Carrier availability must be respected in each round.
- The minimum and maximum number of carriers per round must be fulfilled.
- Minimum and maximum carrier block length restrictions must be fulfilled (a carrier block refers to a group of consecutively scheduled carriers that use the same carrier type).
- Forbidden carrier sequences must not appear in the schedule.
- Forbidden color sequences must not appear in the schedule.

2.2 Objective function

The PSSP's objective function (Equation 1) combines two minimization objectives which aim to minimize the number of color change costs (cc) and carrier changes (sc). Each round's color change costs (cc_r) and carrier changes (sc_r) are squared, to balance the required changes over the scheduling horizon. This is done to avoid peaks of such changes within a single round, which could lead to delays in the schedule.

$$minimize \sum_{r \in R} sc_r^2 + \sum_{r \in R} cc_r^2 \quad \text{with } R\text{: set of rounds.} \tag{1}$$

During the search we also consider infeasible solutions. However, for each violation a cost penalty equal to the upper bound of the objective value is added.

3 RELATED WORK

The PSSP was introduced in [22], where the authors additionally proposed a metaheuristic approach based on local search to solve the problem. Their approach utilizes a simulated annealing-based move acceptance function which takes the search progress into account and additionally uses a tabu list to prevent the selection of recently performed moves. The authors further proposed a greedy construction heuristic to generate initial solutions for the metaheuristic approach and provided a set of 24 real-life benchmark instances. Experiments showed that the overall best results were achieved with the following combination of techniques: The initial solution was greedily created and passed to simulated annealing, where the neighborhood was built via a min-conflicts heuristic. This approach was able to find feasible solutions within a time limit of one hour even for the largest benchmark instances. An exact method using constraint programming (CP) for the PSSP was proposed in [21]. This approach works well for small instances, and has been able to find optimal solutions for several benchmark instances. However, the proposed exact methods were not able to solve any of the larger instances within a runtime limit of 6 hours.

While the PSSP has just recently been introduced in the literature, many previous publications investigated other production scheduling problems from the automotive industry. The *Car Sequencing Problem (CSP)* is a related problem, where cars have to be sequenced along the production line, fulfilling a number of constraints. This problem also tackles the minimization of color changes. The CSP is a popular problem in the literature, and has been tackled with different approaches, like CP (e.g [3]), integer linear programming (e.g. [14]), ant colony optimization (e.g. [5]) and a follow-up sequencing algorithm (e.g. [2]). An overview of state-of-the-art methods for this problem is given in [15].

Authors of [16] also deal with the sequential ordering in automotive paint shops. The objective of the presented problem is to maximize the average color batch size. They present a branch&bound algorithm to solve the problem. Another work in [18] focuses on reducing energy consumption in the automotive production process by optimizing schedules. The proposed problem aims to reduce energy consumption over peak demand periods on a group of production lines of an automotive factory, while also obeying time and resource constraints. An evolutionary algorithm is proposed to solve the problem, which is starting from a population of random schedules and applies local transformations to create offspring. The best individuals of the union set of parents and children are then selected for the next generation.

In [4] the *Master Production Scheduling Problem (MPS)* is considered in the context of the automotive industry. The customization of final products in the automotive industry involves large numbers of optional parts, which leads to a huge variety of operation times at the various stations of the assembly line. The authors develop a mathematical model formulation for the MPS and propose several heuristic solution procedures to solve this problem, which focus on minimizing the workload variability.

The PSSP is different from the automotive scheduling problems discussed above since it does not deal with the manufacturing of complete cars, but rather with individual car parts which need to be grouped and placed on carrier devices. Furthermore, it uses a unique quadratic objective function, which takes the whole scheduling horizon into consideration and punishes peaks in color and carrier device changes.

4 A MEMETIC ALGORITHM FOR THE PAINT SHOP SCHEDULING PROBLEM

In this section we propose a memetic algorithm for the PSSP where we represent a solution by a sequence of rounds (see Figure 1). A solution's chromosome is symbolized by a vector for each round, with it's length equal to the number of carriers scheduled this round. Each element of the vector is an identifier, which represents a carrier configuration and color code combination.

4.1 Crossover Operators and Memetic Representations

Crossover operators are an essential part of population-based algorithms and may have a significant influence on the solution's quality (hereinafter called fitness). Finding effective crossover operators for the PSSP is not a trivial task, since partial solutions (memes) cannot be evaluated individually because of several reasons:

- Costs caused by a single round or a group of rounds cannot be calculated without also considering the neighboring rounds. This is because the costs depend on the differences between the carrier sequences from two consecutive rounds.
- Demands and due date constraints must be fulfilled as they are hard constraints. As the carrier devices associated to a demand may be distributed over the whole schedule, we cannot evaluate constraint violations for partial solutions.
- All constraints must be evaluated over the boundaries of the schedule's rounds. A round could, for example schedule a carrier of type A at its last position and thereby cause that the next round cannot start with a carrier of type B if AB is a forbidden carrier sequence. In other words, the rounds are just a way of dividing the whole schedule, but cannot be viewed independently.
- Rounds do not have a fixed size; their size only has to be within a certain range. When evaluating partial solutions, smaller rounds usually have fewer constraint violations and lower costs than larger rounds.

These points make it difficult to evaluate and improve memes. Thus, for the first two crossover operators (vertical and horizontal crossover), full solutions are evaluated and selected as parents. We also do not improve the memes during the local search phase, but instead improve solutions as a whole.

4.1.1 Vertical Crossover. For this crossover operator rounds are taken from two parent solutions to create a child solution. The solutions are then cut in *vertical* direction and a meme is one round of the solution. Whether a round is taken from parent A or parent B is decided randomly. Figure 2 depicts an example of such a crossover.

This crossover operator's main idea is to introduce an operator that - while as simple as possible - still performs well. The operator uses one of the problem's natural units - the round - as its memes. A round is neither too big nor too small to be a meme and carrier- as well as color sequences are preserved. Furthermore, the round capacity constraints (too few/too many carriers in a round) or the

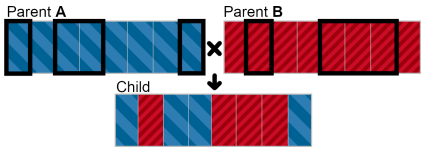

Figure 2: Vertical crossover.

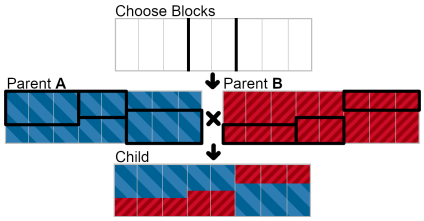

Figure 3: Horizontal crossover.

carrier device availability constraints (too many carriers of a type are used in a round) are not violated. Since we are merging rounds of different solutions together, some of the problems mentioned at the beginning of Section 4.1 can occur - such as higher costs due to carrier sequences of successive rounds or hard constraint violations (demands, carrier device sequences, etc.).

4.1.2 Horizontal Crossover. This crossover operator cuts the solutions in *horizontal* direction. Two parent solutions are taken, cut, and a child solution is created by merging the pieces together.

First, the solution is split into round blocks of length l: $l \in [\log_3 R, \log_2 R]$, where R is the number of rounds in the instance. We choose a logarithmic block size since the number of rounds can vary a lot between the instances. If we have an instance with just a few rounds, we still want to have a few cuts. However, for a problem instance with ten times as many rounds, the amount of cuts should not grow by a factor of ten. Instead, we want the number of blocks - and thus the number of memes - to be more consistent for different instances.

For each of these round blocks, a horizontal cutting point h is chosen randomly from the interval $h \in [r * 0.25, r * 0.75]$, where r is the minimum round length, i.e. the minimum number of carrier devices allowed per round. A meme is equivalent to one of the halves of such a block. The cutting point h is chosen from this interval to obtain blocks of various sizes, while at the same time avoiding blocks consisting of just a few carrier devices. Figure 3 illustrates a horizontal crossover.

This crossover operator aims to keep carrier change costs between consecutive rounds low, since those are vital for a solution's fitness. The blocks ensure that we have a longer sequence of consecutive "half" rounds, thus helping us to achieve this goal. Many constraints, like the min/max block constraint or the forbidden color/block sequence constraint, depend on the sequence of blocks. The color costs also depend on the block sequence. Those constraints/costs suffer from switching between solutions, which occurs now twice as often compared to the vertical crossover. Again, since we are merging different solutions together, some of the problems mentioned at the start of Section 4.1 may occur.

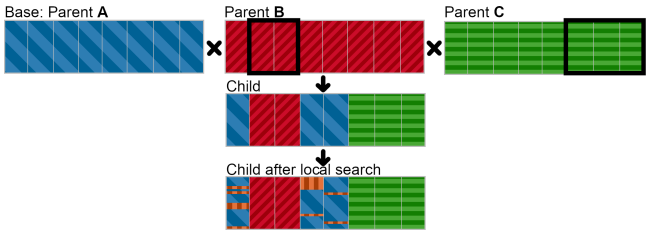

Figure 4: Cost and demand crossover.

4.1.3 Costs and Demand Crossover. This crossover operator is more complex than the two crossover operators described above. The first difference is that the number of parents is not fixed at two, but is arbitrary (> 2). The second difference is that we conduct local search to repair the solution at the merging point.

There is one base parent solution. Blocks of rounds are chosen from different other parent solutions replacing those rounds in the base solution - those are the memes. Between the selected blocks must be a buffer zone. This buffer zone consists of rounds of the base solution and has a minimum size of one round. The size of the blocks is randomly chosen from the interval $[\log_3 l, \log_2 l]$, where l is the instance length. It's the same block size as in the horizontal crossover, chosen for the same reasons stated there. The crossover's name comes from the fact that those blocks are chosen by their costs (color + carrier change costs) and by a score of demand fulfillment.

The calculation of a block's costs is done in two steps. In a first step a scarcity score is calculated for each material. The scarcity is calculated by contrasting the demanded material amount with the estimated amount, which can be held by the available carrier devices. Eventually, we have a score for each material, which is then normalized in the interval $[0.3, 1]$. This score is independent of the solution and depends only on the problem instance. Therefore it is only calculated once.

The second step is the calculation of the demand fulfillment and the final score for the block. The materials' scarcity calculated in step 1 is used here. For each of the carrier devices in the block's rounds, we look at the demands that this carrier device can fulfill. A carrier device gets a higher score if it fulfills urgent demands. This score gets divided by the material scarcity calculated in step 1. The score for the whole block is the average of all the carrier devices' scores. The block's *final costs* are the block's costs (color and carrier device change costs) multiplied with the score from step 2.

After creating the child solution by merging the base solution and the other parent solutions' blocks together, a special local search operator is conducted. The operator is only allowed to manipulate the buffer zones. There is a higher chance in these zones than in the blocks for possible improvement since the blocks have been chosen for their low cost. Also, when merging parts of different solutions, constraints can easily be violated. The local search operator is stopped if no improvement is found for 6 cycles, or if the time limit of 1 second is passed.

5 OVERALL ALGORITHM

In this section we present all components of our algorithm including selection, crossover and mutation, local search as well as the

construction heuristics. Pseudo code and a detailed description of the algorithm is given in [19].

5.1 Selection

Selection of solutions is done at different stages of the algorithm.

Selection of the elitist. The elitist is selected at the start of each generation, by choosing the individual with the best fitness. It is selected to survive the current round. The elitist can still be selected as a parent for crossover operators or for mutation. Those operators are immutable, i.e. they do not modify the parents. Thus, the elitist will never be changed during this phase.

K-tournament selection of the individuals for crossover operators (see [12]). Crossover operators use memes of parents to create child individuals. To select the parents, the operator takes $k < |\mathcal{P}|$ individuals from the current generation and performs a k-tournament. The fittest individual out of the k competitors wins, and thus is selected. There are different fitness functions for the various crossover operators - since for the cost and demand crossover blocks are competing to be selected instead of full individuals. The number of times a k-tournament is run likewise depends on the crossover operator. For binary crossover operators, two parents are selected, while more parents are selected for the cost and demand crossover. The k-tournament operator is also used for the mutation operator - where it only needs to be executed once.

Selection of the current generation's survivors (shrinking). This selection is only performed if the algorithm is configured to execute more crossover and mutation operators and thus create more offspring than there are members in the initial population ($\mathcal{N} > |\mathcal{P}|-1$). In our implementation, the population size always stays the same. So only a certain number of created children can survive the current generation. The best $p = |\mathcal{P}| - 1$, which are non-duplicates, are selected for survival. If there are too many duplicates in the new generation ($> 10\%$ of the population), we include individuals of the old generation for survival (i.e. the parents). This is done to prevent the population from converging towards the elitist, which would frequently happen before introducing this change. Eventually, the elitist is added to the new population.

5.2 Crossover and Mutation

Offspring is created by applying memetic operators to the population, like crossover and mutation operators. Mutation is done with probability m, while crossover operators are executed with probability $1 - m$. The parents for the crossover operators, as well as the individuals to be mutated, are selected via a k-tournament operator. The value m and the probabilities for the different crossover operators are passed as parameters to the algorithm (see the experimental section for a list of all parameters).

Our mutation operator chooses and executes a random local search move. The randomness of the move is on purpose - it is a move the local search algorithm would probably not make - to promote diversity in the population.

5.3 Local Search

The local search algorithm for improving the selected individuals is based on the simulated annealing approach proposed in [22].

Local search is conducted on a few selected individuals $l \leq |\mathcal{P}|$ each generation. The number of individuals selected for local search depends on the parameter ilf (individual learning frequency): $l = |\mathcal{P}| * ilf$. Child solutions are selected randomly for local search - fitness is not taken into account. The time reserved for local search is controlled by the parameter ili (individual learning intensity), which is denoted in seconds.

Local search specific parameters like the simulated annealing temperature are saved and passed to the next local search call. We save those local search parameters since the population converges together. For a population consisting of rather fit individuals, the temperature thus will already be lower. If no improvements are found for a certain amount of time, reheating is used to escape local optima. The local search algorithm eliminates violated hard constraints first, as for a single hard constraint violation a cost penalty which is an upper bound to the objective value is added.

There are three neighborhood moves: insertion, deletion, and swapping of carriers. All these moves can be done as block moves which simultaneously modify successive carriers. The neighborhood is generated via a min-conflicts heuristic where positions involved in constraint violations are tracked. Some of those positions are then randomly selected to generate fitting moves. A simulated annealing move acceptance function decides whether a move should be accepted (see [8]). The temperature function also supports reheating - a technique to escape local maxima in case no improvements can be found for a certain amount of iterations.

5.4 Initialization

The fitness and diversity of the initial population have a big influence on genetic and memetic algorithms' performance. There is a trade-off between execution time, quality, and diversity when creating the initial population. We implemented three construction strategies which differ regarding the focus on these aims.

5.4.1 Random Construction. The random construction is the fastest one of the three construction strategies. This strategy yields completely random solutions. Due to the fact that carriers and colors are randomly chosen, population diversity is very high.

5.4.2 Random Greedy Construction. This is another very fast construction strategy. Carriers and colors are again chosen randomly, but they cannot violate some of the constraints. Carrier block size constraints are always met, and only permitted color/carrier sequences are used. The diversity is again very high since we choose carriers and colors randomly. The difference to a purely random construction is that we try to avoid violations of minimum/maximum block size and forbidden color/carrier sequences.

5.4.3 Greedy Construction. The third construction strategy is taken from the literature ([22]). It is a lot slower than the first two - and much more sophisticated.

The greedy construction heuristic is a two-phase algorithm. In the first phase, the round layout is constructed. In this step, carrier configurations and colors are assigned to the rounds, but not yet put in sequence, while considering the problem's hard constraints. In the second phase, the carrier sequence for each round is determined. Sequence constraints are taken into consideration while trying to keep the amount of carrier and color changes low.

Once the greedy construction terminates and yields a single solution, we have to generate a full population from this single solution because of time constraints - for the largest instances, the construction can take up to 20 minutes. Therefore, for each additional population member, we modify the yielded solution by performing random local search moves on 3% of the carriers. Afterwards, local search is conducted for 5 seconds to repair some of the conflicts introduced by the modifications.

5.5 Crossover Delta Evaluation

Evaluating solutions is a task which takes a lot of computation power. The costliest part of a local search algorithm is often the evaluation of the new solutions. A technique to speed up solution evaluation is *delta evaluation*. We can calculate the fitness of a child solution s by using the fitness of its parent solutions $p_1, ..., p_n$, some cached data structures, and information about the genetic differences caused by the crossover or mutation operator.

Suppose we have a solution and its fitness. When applying a move, we would normally evaluate the modified solution from scratch. With delta evaluation, we can just evaluate the differences in the part of the solution that has changed. If our move adds additional carriers, we just need to consider how those carriers affect the various constraints - which the PSSP has many of. For example, for the delta evaluation of the demand constraint, we cache (among others) the number of scheduled pieces until a round. The move's impact on the demand constraint can easily be calculated by using this cache and information about the performed move, thus avoiding iterating through all the solution's rounds.

5.5.1 Full delta evaluation for crossover operators. For the local search moves (insert/delete/swap blocks), [22] already implemented delta evaluation. Based on their implementation, we add delta evaluation logic for each of our crossover operators.

One problem with this approach is that a lot of large data structures have to be cached. We also have to create a deep copy of those data structures for new population members. This is due to the fact that we cannot just modify one of the parent solutions and create the child by applying the crossover operator. Parents need to be immutable since they could be

(1) selected as a parent for another crossover operator.
(2) selected for mutation.
(3) taken over into the next round (if there are too many duplicates).

For each new child solution, all the data structures need to be copied. Creating offspring is an essential task of a memetic algorithm, making this copy process very costly.

To further improve the performance of the delta evaluation, via profiling we identified the most time-consuming constraint evaluators. The constraint which takes (depending on the instance) 60% to 95% of the evaluation's runtime is the carrier change constraint. This is because, for this constraint, the *longest common sub-sequence* between each pair of consecutive rounds has to be calculated. Instead of deep copying the large data structures for each new child solution, we just cache the costs of carrier changes per round directly in the solution which simplifies the cost calculation.

6 COMPUTATIONAL RESULTS

In this section, we evaluate our memetic algorithm presented in the previous section. We used a computing cluster that has 10 nodes, each having 24 cores, with an Intel Xeon E5-2650 v4 2.20GHz CPU and 252 GB RAM for all experiments.

6.1 Instances

The problem instances we use for benchmarking our algorithm are taken from [22]. They provide 24 problem instances based on real life planning scenarios of the automotive supply industry. These instances have recently been used as benchmark instances for this problem and are publicly available for download.[1]

The instances have six different planning horizons of 7, 20, 50, 70, 100, and 200 rounds. For each of the planning horizons, there are two instances - only one of them imposes forbidden color and carrier sequences. There are 12 big instances and 12 small instances.

6.2 Parameter tuning

As memetic algorithms are often highly parameterizable, we configure many parameters for our algorithm, which may have an influence on the solutions' fitness. Since our algorithm is non-deterministic, 10 stochastic runs are executed for each parameter configuration. The result of those runs is used to calculate the mean, best, and worst fitness, and the standard deviation for the parameter configuration.

We applied SMAC (sequential model-based algorithm configuration) ([7, 9]) for configuration of our algorithm. Furthermore, we use the *Wilcoxon signed-rank test* as statistical method to assess whether the means of solution costs of parameter settings provided by SMAC differ significantly (see [19] for detailed results). The range of values tested for each parameter is given below (in format Parameter, Value sequence, Default value):

- Population size p, [2, 5, 10, 15, 30], 10
- Crossover population size cp^*, [1.0, 1.2, 1.4, 2], 1.4
- K-tournament competitors k^*, [0.0, 0.15, 0.35, 0.5], 0.15
- Individual learning frequency ilf, [0.0, 0.15, 0.35, 0.5], 0.15
- Individual learning intensity ili, [3, 8, 20, 45], 20
- Mutation frequency mf, [0.0, 0.1, 0.2, 0.4], 0.2
- Vertical crossover freq. vcf, [0.0, 0.4, 0.5, 1.0], 0.5
- Horizontal crossover freq. hcf, [0.0, 0.3, 0.5, 1.0], 0.0
- Costs and demand crossover freq. $cdcf$, [0.0, 0.4, 0.5, 1.0], 0.5

Those values, as well as the default value, were selected by manual tuning. Note that some parameters (k^*, cp^*) had to be modeled as multipliers for p, since no dependencies between parameters can be modeled in SMAC. Thus: $k = max(1, p * k^*)$.

As a runtime limit, we defined 30 minutes for each call of our algorithm. Further, we conducted the tuning process separately for the small and big instance sets.

6.3 Crossover operators

We evaluated several combinations of the crossover operators mixed with different population sizes. For the small instance set we observed that the horizontal crossover operator performed the worst, especially for larger populations. The vertical crossover operator

[1] https://dbai.tuwien.ac.at/staff/winter/ps_instances.zip

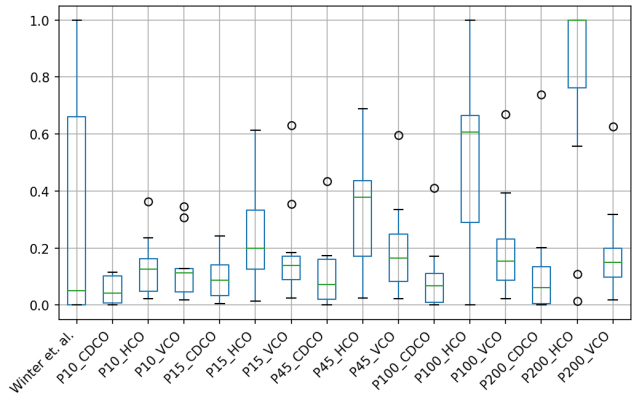

Figure 5: RDI values of parameter configurations with various crossover combinations for the small instance set.

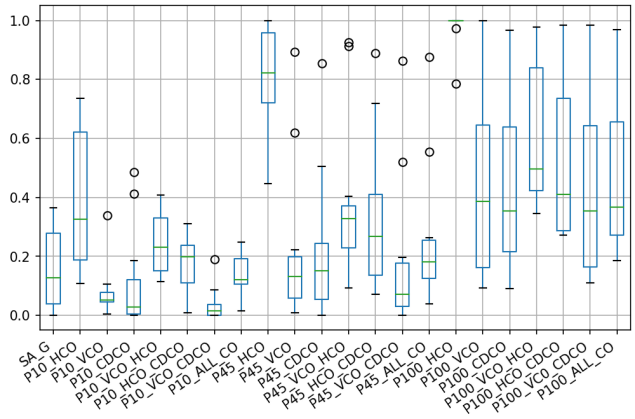

Figure 6: RDI values of parameter configurations with various crossover combinations for the big instance set.

yields competitive results, while the best results are generated by the cost and demand crossover operator. Figure 5 shows the corresponding RDI ($RDI_{I,S} = \frac{cost_{I,S} - best_I}{worst_I - best_I}$) values. Configuration P10_CDCO has the lowest median of all configurations (configuration identifiers denote the population size and used crossover operator, i.e. in this case *P*10 means population size 10 and *CDCO* means cost and demand crossover). Interestingly, a bigger population can lead to improved solution quality for the small instance set's larger instances, as can be seen by configuration P45_CDCO and P100_CDCO.

For the large instance set, all configurations used the greedy construction strategy - which is needed to achieve feasible solutions within time constraints. Looking at the RDI values in Figure 6, three combinations yield the best results: vertical crossover only, cost and demand crossover only, and a combination of both. We further observed that a population larger than 45 yields worse results for this instance set.

We also evaluated our different construction strategies, and the algorithm without local search, making it a genetic algorithm. Results showed that the genetic algorithm could find competitive results for some of instances, but was in general outperformed by the memetic algorithm. We refer the reader to [19] for detailed a detailed comparison.

6.4 Comparison to Literature Results

This section compares the results obtained by runs of our algorithm with the best parameter configurations to literature results. We take our best result out of 10 runs for each instance for the comparison. The time limit for each run is set to 1 hour to match the time limit used in literature. Table 1 depicts parameter values of our best configurations. The configurations were obtained using SMAC and further manual tuning. A *Wilcoxon signed-rank test* showed that there was no significant difference between the best literature results and the memetic algorithm results.

Configuration	*p*	*cp*	*k*	*m*	*vcf*	*hcf*	*cdcf*	*ilf*	*ili*	*c*
P100_G	100	1.0	7	0.0	0.5	0	0.5	0.01	3	Greedy
P200_RG	200	1.0	7	0.0	0.5	0	0.5	0.02	3	Random Greedy
P10_WS	10	1.0	5	0.1	0.5	0	0.5	0.15	8	Greedy
P45_G	45	1.0	7	0.0	0.5	0	0.5	0.05	3	Greedy

Table 1: Best parameter configurations.

6.5 Small Instance Set

The results from literature approaches and our best configurations for the small instance set are shown in Table 2.

Instance	LS	LS/G	LS/G/T	CP	P100_G	P200_RG
7R-small	1028	844	882	**775***	781	776
7R-HC-small	868	932	927	**842***	842	844
20R-small	990	992	994	**961***	995	976
20R-HC-small	1016	975	1050	**918***	962	937
50R-small	616	593	599	**530***	655	672
50R-HC-small	887	891	895	**842***	909	906
70R-small	1084	1088	1137	**844***	1272	1353
70R-HC-small	1871	1834	2553	**1237***	1683	1657
100R-small	1767	1735	2421	**975***	1500	2230
100R-HC-small	1262	1243	1269	**964**	1137	1113
200R-small	6298	**5476**	6439	-	2240	**2070**
200R-HC-small	**5723**	7916	8274	-	3172	**2069**

Table 2: Comparison of literature results with results from our algorithm for the small instance set.

The best result achieved out of 10 runs is shown in columns *P100_G* and *P200_RG*. The literature results shown in columns LS, LS/G, LS/G/T show the best results from [22] that have been achieved under a time limit of 1 hour with Simulated Annealing (LS), Simulated Annealing starting from a greedily created solution (LS/G), and Simulated Annealing starting from a greedy solution with incorporation of a tabu list (LS/G/T). The CPU used by the authors of [22] was an Intel Xeon E5345 2.33GHz with 48GB RAM. Results shown in column CP show the best results achieved with a

constraint programming approach from [21] under a time limit of 6 hours, where results with a * denote proven optimal solutions. Bold values indicate upper bounds from [22]. Underlined values are improved bounds compared to [22] found by our memetic algorithm.

Our algorithm achieves new upper bounds for the instances 200R-small and 200R-HC-small. When comparing our results to the simulated annealing approaches, our algorithm achieves similar results for most instances apart from those two. We suspect that the large population sizes are beneficial for the small instance set, especially for those two problem instances.

Figure 7 illustrates RDI values for this comparison. We can see that the medians of the configurations *P200_RG* and *P100_G* are smaller than the other approaches' medians.

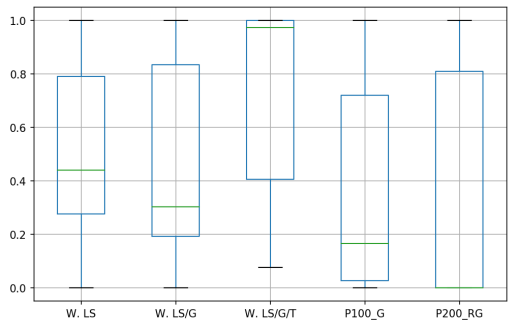

Figure 7: RDI values of the best configurations and literature results for the small instance set.

6.6 Big Instance Set

For the big instance set, we compare the results in Table 3.

Instance	LS	LS/G	LS/G/T	P10_WS	P45_G
7R	2097235	**116235**	123830	80121	**79526**
7R-HC	1985513	**118628**	130552	**116504**	120946
20R	8159361	180863	**172679**	**162012**	178251
20R-HC	8621490	**262252**	262897	**254127**	266444
50R	23320626	**421777**	455321	509188	538880
50R-HC	23947097	**581021**	606917	**535031**	546846
70R	34294393	**555829**	576225	576450	595859
70R-HC	34713814	930564	**927822**	**870889**	916365
100R	-	**917955**	957854	1031698	1087649
100R-HC	-	**1128716**	1142530	1450234	1680627
200R	-	1889804	**1884125**	2209354	2711918
200R-HC	-	**2086450**	-	2312213	2822476

Table 3: Comparison of literature results with results from our algorithm for the big instance set.

The best result achieved out of 10 runs is shown in columns *P10_WS* and *P45_G*. Columns LS, LS/G, LS/G/T show the best results from [22]. Bold values are upper bounds from [22]. Underlined values are improved bounds compared to [22].

We are able to improve upper bounds compared to [22] for six of the instances: 7R, 7R-HC, 20R, 20R-HC, 50R-HC, and 70R-HC. Analyzing the cost differences between our two best configurations, P45_G only performs better for the smallest instance. The gap between the two configurations gets bigger for increasing instance sizes. For the largest four instances, our algorithm performed worse than the local search approaches.

RDI values of the various methods for the big instance set are shown in Figure 8. The median for our configuration P10_WS is smaller than the literature approaches' median, which indicates a good performance of our approach.

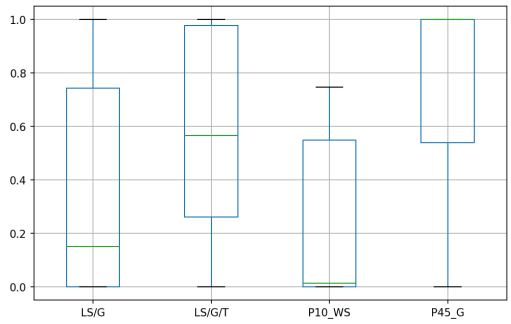

Figure 8: RDI values of the best configurations and literature results for the big instance set.

7 CONCLUSIONS

In this paper, we proposed a memetic algorithm to solve the recently introduced Paint Shop Scheduling Problem. We designed a memetic representation and proposed different population construction strategies. Further, we introduced memetic operators for selection, mutation, and three novel crossover operators, which consider problem-specific knowledge.

We observed that a large population yields good results for the small instance set, while smaller population sizes are preferred for the big instance set. A combination of the vertical crossover- and cost and demand crossover operators is preferential for most instances. Construction strategies for the initial population have a massive influence on the result. For the big instance set, sophisticated construction strategies are needed to generate feasible results within time constraints. We compared the results obtained by the best parameter configuration to the best literature results using a set of publicly available real-life instances. Our memetic algorithm's results are competitive for the small instance set, as well as for small instances of the big instance set and provides new upper bounds for some instances.

For future work it would be interesting to investigate other crossover operators and local search strategies, where a variable population size that starts with a smaller population and increases its size once solutions reach a better quality could be beneficial.

ACKNOWLEDGMENTS

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association is gratefully acknowledged.

REFERENCES

[1] Edmund K. Burke, James P. Newall, and Rupert F. Weare. 1995. A Memetic Algorithm for University Exam Timetabling. In *PATAT (Lecture Notes in Computer Science, Vol. 1153)*. Springer, Edinburgh, UK, 241–250.

[2] Sara Bysko and Jolanta Krystek. 2019. Follow-Up Sequencing Algorithm for Car Sequencing Problem 4.0. In *Automation 2019 - Progress in Automation, Robotics and Measurement Techniques, outcomes of the international conference AUTOMATION 2019, 27-29 March, 2019 (Advances in Intelligent Systems and Computing, Vol. 920)*, Roman Szewczyk, Cezary Zielinski, and Malgorzata Kaliczynska (Eds.). Springer, Warsaw, Poland, 145–154. https://doi.org/10.1007/978-3-030-13273-6_15

[3] Mehmet Dincbas, Helmut Simonis, and Pascal Van Hentenryck. 1988. Solving the Car-Sequencing Problem in Constraint Logic Programming. In *ECAI*. Pitmann Publishing, London, Munich, Germany, 290–295.

[4] Jan Dörmer, Hans-Otto Günther, and Rico Gujjula. 2015. Master production scheduling and sequencing at mixed-model assembly lines in the automotive industry. *Flexible Services and Manufacturing Journal* 27, 1 (2015), 1–29.

[5] Caroline Gagné, Marc Gravel, and Wilson L. Price. 2006. Solving real car sequencing problems with ant colony optimization. *Eur. J. Oper. Res.* 174, 3 (2006), 1427–1448.

[6] Daniel S. Hirschberg. 1977. Algorithms for the Longest Common Subsequence Problem. *J. ACM* 24, 4 (1977), 664–675.

[7] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In *LION (Lecture Notes in Computer Science, Vol. 6683)*. Springer, Rome, Italy, 507–523.

[8] Scott Kirkpatrick, D. Gelatt Jr., and Mario P. Vecchi. 1983. Optimization by Simmulated Annealing. *Sci.* 220, 4598 (1983), 671–680.

[9] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, Stefan Falkner, André Biedenkapp, and Frank Hutter. 2017. SMAC v3: Algorithm Configuration in Python. https://github.com/automl/SMAC3.

[10] Bo Liu, Ling Wang, and Yihui Jin. 2007. An Effective PSO-Based Memetic Algorithm for Flow Shop Scheduling. *IEEE Trans. Syst. Man Cybern. Part B* 37, 1 (2007), 18–27.

[11] Bo Liu, Juan-Juan Xu, Bin Qian, Jian-Rong Wang, and Yan-Bin Chu. 2013. Probabilistic memetic algorithm for flowshop scheduling. In *Memetic Computing*. IEEE, Singapore, Singapore, 60–64.

[12] Brad L. Miller and David E. Goldberg. 1996. Genetic Algorithms, Selection Schemes, and the Varying Effects of Noise. *Evol. Comput.* 4, 2 (1996), 113–131.

[13] Pablo Moscato. 1989. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. *Caltech concurrent computation program, C3P Report* 826 (1989), 1989.

[14] Matthias Prandtstetter and Günther R. Raidl. 2008. An integer linear programming approach and a hybrid variable neighborhood search for the car sequencing problem. *Eur. J. Oper. Res.* 191, 3 (2008), 1004–1022.

[15] Christine Solnon, Van-Dat Cung, Alain Nguyen, and Christian Artigues. 2008. The car sequencing problem: Overview of state-of-the-art methods and industrial case-study of the ROADEF'2005 challenge problem. *Eur. J. Oper. Res.* 191, 3 (2008), 912–927.

[16] Sven Spieckermann, Kai Gutenschwager, and Stefan Voß. 2004. A sequential ordering problem in automotive paint shops. *International journal of production research* 42, 9 (2004), 1865–1878. Publisher: Taylor & Francis.

[17] Y Sugimori, K Kusunoki, F Cho, and SJTIJOPR UCHIKAWA. 1977. Toyota production system and kanban system materialization of just-in-time and respect-for-human system. *The international journal of production research* 15, 6 (1977), 553–564.

[18] Junwen Wang, Jingshan Li, and Ningjian Huang. 2011. Optimal vehicle batching and sequencing to reduce energy consumption and atmospheric emissions in automotive paint shops. *International Journal of Sustainable Manufacturing* 2, 2-3 (2011), 141–160.

[19] Wolfgang Weintritt. 2020. *Solving the Paintshop Scheduling Problem with Memetic Algorithms*. Thesis. TU Wien. https://repositum.tuwien.at/handle/20.500.12708/16215

[20] Magdalena Widl and Nysret Musliu. 2014. The break scheduling problem: complexity results and practical algorithms. *Memetic Comput.* 6, 2 (2014), 97–112.

[21] Felix Winter and Nysret Musliu. 2021. Constraint-Based Scheduling for Paint Shops in the Automotive Supply Industry. *ACM Trans. Intell. Syst. Technol.* 12, 2, Article 17 (Jan. 2021), 25 pages.

[22] Felix Winter, Nysret Musliu, Emir Demirovic, and Christoph Mrkvicka. 2019. Solution Approaches for an Automotive Paint Shop Scheduling Problem. In *ICAPS*. AAAI Press, Berkeley, CA, USA, 573–581.