Paintshop: Constructive, Improving and Meta-heuristics applied to a Linear Assignment Problem

Justification of research and implementation

Oliver Mulder and Sebastiaan Craens, 7/10/2024

Introduction

As a part of course *Operations Research 5: Discrete Improving Search* of the core phase of *Applied Mathematics* at *Fontys University of Applied Sciences*, students were tasked with designing and implementing heuristics for a linear assignment problem called 'Paintshop'. The problem concerns a set of orders to be processed on a set of machines. A machine can only process one order at a time and each machine has a specified processing speed. Each order has its own processing load, deadline and penalty in terms of cost unit per time unit over its deadline. Each order moreover corresponds to a color, hence the name 'Paintshop', and when a machine has to switch colors (to process an order with a different color than the one before it), it takes some time to clean and configure the machine - specified for each combination of colors.

Student were asked to create a Python script that produces a high-quality schedule for this problem. The program should be generic to the extent that it would work for any set of orders and machines. The Python script also had to be efficient and readable. Student were to apply some of what was taught in the curriculum in the form of "

1. a constructive heuristic that constructs a feasible schedule from scratch,
2. a simple first improvement or best improvement heuristic based on Discrete Improving Search, that will improve a given feasible schedule to a local optimum, and
3. a more complex meta-heuristic, such as Tabu Search or Simulated Annealing, that will provide a robust solution method for generating a high-quality schedule." (Fontys University of Applied Sciences, 2024)

This document presents justifications for design- and implementation decisions that were made during the creation the said scripts.

Table of Contents

1 Problem definition

The problem is presented as the following mathematical model:

Sets & indices:

M	$\triangleq$ *Set of machines*	$m \in M$
O	$\triangleq$ *Set of orders*	$o \in O$
C	$\triangleq$ *Set of colors*	$c \in C$

Parameters:

l_o	$\triangleq$ *Processing load of order* $o \in O$
c_o	$\triangleq$ *Color of order* $o \in O$
d_o	$\triangleq$ *Deadline of order* $o \in O$
p_o	$\triangleq$ *Penalty cost of order* $o \in O$
s_m	$\triangleq$ *Processing speed of machine* $m \in M$
$t^{color}_{c_1,c_2}$	$\triangleq$ *Time required to change configuration of a machine from color* $c_1 \in C$ *to* $c_2 \in C$

Derived parameters:

$t^{order}_{o_1,o_2}$	$\triangleq$ *Time required to change configuration of a machine from order* $o_1 \in O$ *to* $o_2 \in O$	$= t^{color}_{c_{o_1},c_{o_2}}$
$v_{o,m}$	$\triangleq$ *Processing time for order* $o \in O$ *on machine* $m \in M$	$= \frac{l_o}{s_m}$

Decision variables:

$machine_o$	$\triangleq$ *Machine of which order* $o \in O$ *is processed*
$index_o$	$\triangleq$ *Index in processing queue of order* $o \in O$
$queue_m$	$\triangleq$ *Sequence of orders in the processing queue of machine* $m \in M$

Derived variables:

Commented [SC1]: Should these be decision variables with constraints perhaps?

$pred_o$	$\triangleq$ *The order previous to order* $o \in O$ *in the processing queue*	$= \begin{cases} -, & index_o = 0 \\ queue_m(index_o - 1), & index_o > 0 \end{cases}$
e_o	$\triangleq$ *The ending time of order* $o \in O$	$= \begin{cases} v_{o,m}, & index_o = 0 \\ e_{pred_o} + t^{order}_{pred_o,o} + v_{o,m}, & index_o > 0 \end{cases}$
$penalty_o$	$\triangleq$ *The penalty* $o \in O$	$= \max\{0, d_o - e_o\} * p_o$

Math program:

min	$\sum_{o \in O} penalty_o$
$st.$	...

2 Overall design approach

Initially, the team was ambitious. Not only did we want our code to be efficient, but we also wanted to dive deeper into optimizing and comparing the different search strategies taught to us. By implementing multiple heuristics, we wanted to see which one could produce the best solution using a set amount of computer processing resources. Every step of the way, we wanted to be meticulous in our design decisions.

To keep things clean, an object-oriented approach to code-architecture was taken. Created classes include:

- A static 'Paintshop' class for the problem itself, containing the given- and derived parameters as shown in the mathematical program.
- A 'Schedule' class, instances of which represent a solution to the Paintshop problem. This class simplifies interactions with a schedule such as initialization, cost calculation, printing or plotting and even feasibility checking.
- A 'Move' class, instances of which represent a move that mutates a schedule into another schedule. This class has subclasses for the different types of moves that can be made. The abstract superclass serves as the interface and has a static method for getting a list of every move that can be made on a given schedule.
- A 'MoveSelectionStrategy' class, containing functions to select a move from a list moves according to a certain strategy and criteria.
- A 'ConstructiveHeuristic' class, instances of which can generate a schedule from scratch.
- A 'ImprovementHeuristic' class, instances of which can improve a given schedule.

In further chapters, we will provide the designs of the implemented constructive- and improvement heuristics along with justifications for the decisions made.

Commented [SC2]: Perhaps there should be chapters for each of the classes.

Commented [SC3R2]: In this case, I would call this chapter 'code architecture, with sub-headers for each of the classes.'

Commented [SC4]: Chapters or sections?

3 Constructive heuristics

When it came to constructive heuristics, the team didn't want to spend too much time coming up with clever strategies. Three constructive heuristics were made:

- A simple constructive heuristic, in which a feasible schedule is constructed in the simplest way.
- A greedy constructive heuristic, which constructs a schedule in a greedy way based on the problem parameters.
- A random constructive heuristic, which construct a schedule randomly.

The first are stable; they create the same schedule every time. The last is not, meaning that it can be used in multistart-heuristics.

3.1 Simple constructive heuristic

The first constructive heuristic we made which we named 'Simple' went from order 1 through 30 and placed each order at the machine with the least items on it, with the lowest numbered machine being prioritized as a tie breaking rule. The schedule this produced was quite good. The greedy constructive heuristic was not able to beat it. Through solution space analysis, we were later able to estimate the percentile of the resultant schedule to be 0.2507 (see chapter *Solution space*).

The pseudocode is as follows:

- **Step 0: initial**
  Start with a completely empty solution and set index r to 1
- **Step 1: stopping**
  If index r is greater than the number of orders, stop
- **Step 2: step**
  assign order(r) to the machine with the fewest jobs assigned to it, if there is a tie assign the job to the machine with the fewest jobs and the lowest machine number
- **Step 3: increment**
  Set order index r to r + 1

3.2 Greedy constructive heuristic

This constructive heuristic is greedy. It makes informed decisions in the process of constructing a schedule. The pseudocode is as follows:

0. Initialize an empty schedule, let $i = 0$, let O be the list of orders sorted by their deadline ascending.
1. If $i \geq |O|$, stop.
2. Assign order O_i to the machine with the lowest ending time (when the last order is done processing).
3. Set $i = i + 1$
4. Return to step 1.

3.3 Random constructive heuristic

This constructive heuristic creates a random schedule. By utilizing the code used for calculating the size of the solution space (see chapter *Solution space*), we were able to ensure that the probability of creating any schedule is evenly distributed over the entire solution space. One should note that the solution space is skewed towards bad solutions, meaning that a function that generates random solutions that are generally greedy, might improve multistart performance. The pseudocode is as follows:

0. Let O be the list of orders. Let n be the number of orders. Let k be the number of machines.
1. Randomly shuffle O.
2. Calculate j: the number of ways of partitioning n items into k subsets (empty subsets allowed).
3. Generate a random number i, in the range $[0, j-1]$.
4. Return the i^{th} k-way partition of the shuffled list O.
5. Stop.

4 Improving heuristics

In this chapter, we elucidate the decisions made regarding the neighborhood structure, move-selection-strategies, as well as the different discrete improving search algorithms implemented. We are attempting to tackle both simulated annealing and taboo search. We would like to find as good a solution as we can get so we are exploring all our options. If time allows it, we would like to attempt a genetic algorithm as well.

4.1 Neighborhood structure.

We decided that we wanted to create a fully connected neighborhood structure. We started off with implementing 'swaps'; a type of move which swaps the schedule positions of two orders. We realized that this move-set does not imply a fully connected neighborhood, proven by the fact that the length of the individual processing-queues would always stay the same. Next, we therefore implemented 'moves'; a move type with which any order could be inserted at any other position in the schedule. Now, the neighborhood was fully connected since any schedule could be turned into any other schedule by the application of the moves. One other set of moves was added to the total move set: The team realized that swapping the processing queues of two machines, while taking many moves using the current move set, would be worth trying on a decently optimized schedule since the carefully crafted processing-queue sequences would remain the same. The move type 'swap-queues' was therefore implemented, consisting of moves that swap the processing queues of two machines.

Each kind of move is implemented as a subclass of the abstract superclass 'Move' which serves as the interface. This superclass has a method that returns a list of all possible moves that can be made on a given schedule. This was done to keep all the logic related to moves under a single class.

Validation was performed to determine whether our rather large total move-set was composed on moves that have no overlap in terms of their moved schedules, tests were performed where moved schedules were compared. Initially, it was discovered that there was some overlap between the swap-moves and the move-moves. First, a random schedule was constructed. Next, all moves were applied to this schedule to produce a list of moved schedules. A set of moved schedules was then created, showing that, in the supplied case, the list of 927 moved schedules contained only 900 unique ones. All moves were then grouped by their moved schedules resulting in the output shown in figure 1.

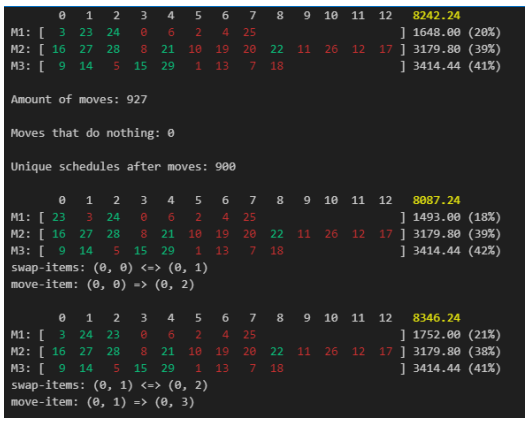


Figure 1: Output from validating the move-set showing overlap between moves in terms of their produced schedules.

The validation showed that swapping an order with the order that comes after it, results in the same moved schedule as moving the former behind the latter. In other words, swapping the order at queue-index n with the order at index $n + 1$ is effectively the same operation as inserting order n at index $n + 2$ (deleting the original). It was therefore decided that move-moves where an order was moved back one spot in the queue would be excluded from the list of moves. The validation now shows that all moves produce a unique schedule. It was furthermore determined that no moves produce the original schedule and that the size of the move-set was independent of the moved schedule – it is always 900 (see figure 2).

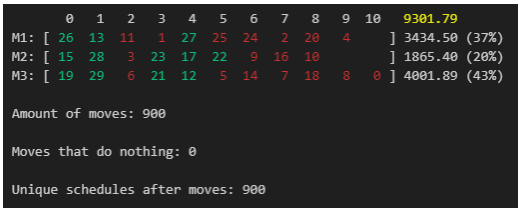


Figure 2: Output from the move-set validation after excluding some of the swap-moves.

4.2 Move selection strategies

Since most improving heuristics need a way of selecting a move from a set of moves, a class of move-selection-strategies (MSS) was created. The associates subclasses implement a function that returns a move, taking into account a given criteria such as 'must-improve', or nothing if no valid move was found. The first two MSS that were implemented were *First* and *Best*, in which the first, and best valid move are returned, respectively. We later noticed that *First* was biased toward moves in the beginning of the move-set, prompting us to implement the MSS *Random*, which operates similar to *First* but on a shuffled move-set so that the results are independent of the way the move-set is implemented.

4.3 Basic improving heuristic

The 'basic' type of improving heuristic takes a move-selection-strategy to improve a given schedule to a local optimum.

4.3.1 Improved complex heuristic:

To improve the complex heuristics, additions were made to it. On top of checking every possible swap, the algorithm will now also check every possible combination of swapping full ques between machines as well every combination of moving a single order to every other spot it could be for every individual order. This has caused the improvement heuristic to reach more better local optimums than it did before. Since it is a rather complex heuristic we have done our best to optimize the process as best we can. For one thing, made it such that the best solution for any loop of the algorithms tracks which values actually changed and only recalculates those instead of going over every order again to calculate the total delay penalty cost of that schedule.

4.4 simulated annealing:

the simulated annealing algorithm work via every iteration randomly choosing a feasible neighbor and accepting this neighbor as the next step if it is improving or, if it is not improving, with a probability of exp(obj gain/temperature).

The main 2 design decisions here are what move set to use and what starting solution to apply. To find a possible neighbor, a move from the move set defined in the complex heuristic will be chosen at random as to explore the solution thoroughly as possible. The starting solution will be chosen at random as we would like for this algorithm to remain generic. Both for the November schedule we will receive later and in order to allow multi-start to be usable.

As for what values to set the parameter to will need to be found via trial and error.

4.5 taboo search:

In designing the taboo-search algorithm, thought was given to the nature of the taboo-list. What exactly is a move? We determined… that using a list of previously tried solutions, which cannot be moved to, would constitute a decent attempt at implementing taboo-search. The schedules are compared by the hash-codes they produce to save on resources.

Commented [SC5]: I will elaborate later.

In order to be able to test different MSS used in the taboo-search algorithm, the implementation is a class that can be instantiated with the MSS in the case of improving moves, as well as the MSS in the case of non-improving moves as parameters.

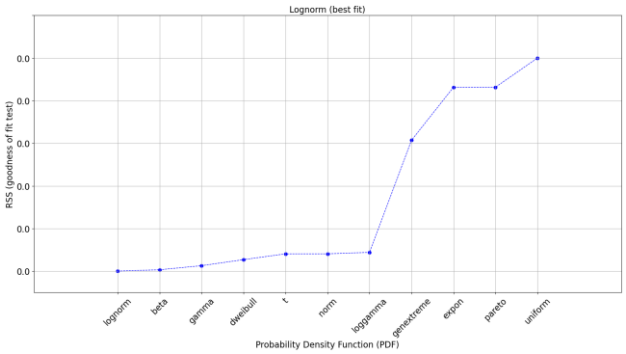
The pseudocode for this is as follows:

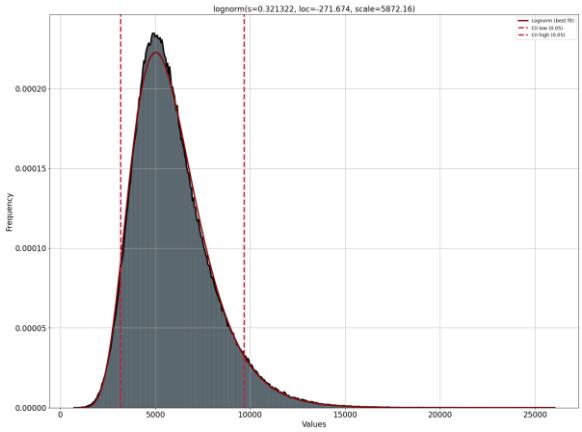
0. Let O be the initial schedule. Let $MSS^{improving}$ be a given MSS. Let $MSS^{non-improving}$ be a given MSS. Let T be an empty list of schedules. Let I be the criteria for a schedule to have to improve upon O.
1. Add the current schedule to T. Let C be the criteria for a schedule not to be contained in T.
2. Try to get an improving move for O using $MSS^{improving}$ and C. If an improving move was found, apply the move and return to step 1.

3. Try to get a move according to $MSS^{non-improving}$ and C
4. Stop.

5 Solution space

[to be continued]





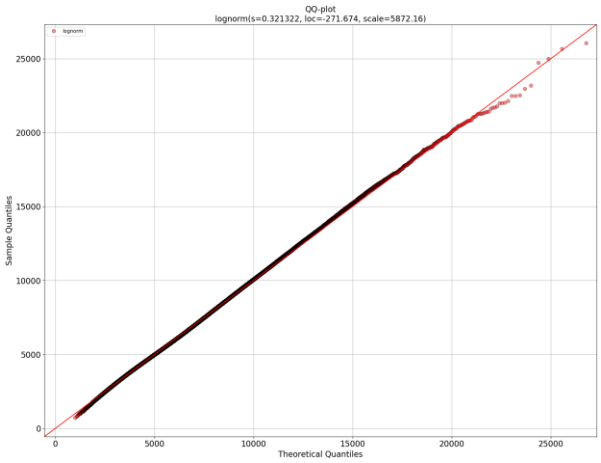

```
# Chance of getting a solution with 0 cost:
model.cdf(0)
```

✓ 0.0s

```
5.6218610531626O7e-22
```

```
# Number of estimated solutions with 0 cost:
PS.solution_space_size * model.cdf(0)
```
✓ 0.0s

```
5.117131454522534e+24
```



6 Visualization:

In order to visualize a solution for debugging and progress-tracking purposes, the Schedule class implements the '__str__' function to produce a string representation of the schedule. In order to provide a better view of the processing times in a schedule, the Schedule class also implements a function named 'draw', which produces a gantt-chart of the schedule using Pyplot (see figure 3).

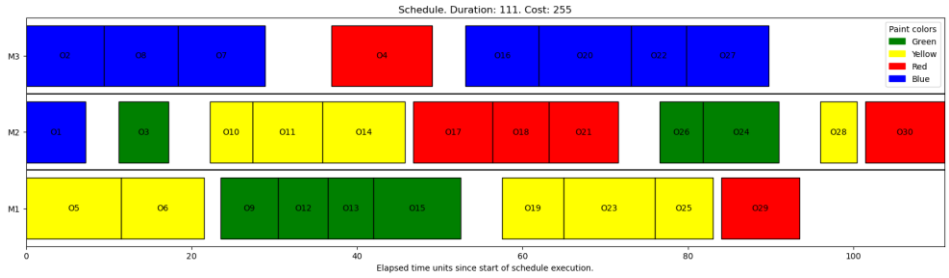


Figure 3: A gantt-chart of a schedule.