Genetic Algorithms for Solving the Pigment Sequencing Problem

Vinasetan Ratheil HOUNDJI[1] **and Tafsir GNA**[1]

[1]Institut de Formation et de Recherche en Informatique, University of Abomey-Calavi, Benin

*E-mail : ratheil.houndji@uac.bj

DOI : 10.46298/arima.11382
Submitted on May 28, 2023 - Published on March 6, 2024
Volume : **40** - Year : **2024**

Editors : *Mathieu Roche, Nabil Gmati, Clémentin Tayou Djamegni*

Abstract
Lot sizing is important in production planning. It consists of determining a production plan that meets the orders and other constraints while minimizing the production cost. Here, we consider a Discrete Lot Sizing and Scheduling Problem (DLSP), specifically the Pigment Sequencing Problem (PSP). We have developed a solution that uses Genetic Algorithms to tackle the PSP. Our approach introduces adaptive techniques for each step of the genetic algorithm, including initialization, selection, crossover, and mutation. We conducted a series of experiments to assess the performance of our approach across multiple trials using publicly available instances of the PSP. Our experimental results demonstrate that Genetic Algorithms are practical and effective approaches for solving DLSP.

Keywords
Genetic algorithm; production planning; pigment sequencing problem; lot sizing.

I INTRODUCTION

Lot sizing problems involve determining which items to produce, when to produce them, and which machine to use in order to meet customer demand while also achieving financial goals. These problems are complex, as they often require producing multiple types of items while balancing conflicting objectives, such as minimizing production and stocking costs while also meeting customer needs. Different types of lot-sizing problems have been studied in the literature. In recent years, researchers such as Houndji et al. [20] and Ceschia et al. [24] have focused on an NP-Hard variant known as the Pigment Sequencing Problem. This problem is included in the CSPlib repository [8] and involves producing multiple items on a single machine with limited capacity (one item per period). The planning horizon is discrete and finite, with predefined stocking and setup costs for each item.

Like many Discrete Lot Sizing and Scheduling Problems, the Pigment Sequencing Problem can be formalized and solved with Genetic Algorithms. Genetic Algorithms are heuristic search methods inspired by the natural evolution of living species. Based upon the concept of the survival of the fittest, genetic algorithms are able, over multiple generations, to find the best solution to a problem. Several studies [14] [30] [16] have shown how efficient they could be in solving optimization problems. This paper presents a search method that relies on genetic algorithms and experiments with this approach. The results indicate that Genetic Algorithm-based methods are a promising solution for addressing Discrete Lot Sizing and Scheduling Problems like the Pigment Sequencing Problem.

The rest of the paper is organized as follows: Section 2 exposes some background on the Discrete Lot Sizing and Scheduling Problems and Genetic Algorithms (GAs); Section 3 presents the problem to be solved (the Pigment Sequencing Problem) and shows an instance of the problem; Section 4 gives details on our method based on genetic algorithms; Section 5 presents some experimental results obtained from the implementation of our approach; and Section 6 concludes this paper and provides some perspectives.

II BACKGROUND

2.1 Discrete Lot Sizing and Scheduling Problems (DLSP)

The PSP belongs to the Discrete Lot Sizing and Scheduling Problems (DLSP) category. The PSP is a problem in which the total capacity available for a period is used to produce one item. The origin of the multi-item DLSP traces back to Fleischmann (1990) [2], in which a branch-and-bound procedure is presented using Lagrangian relaxation for determining lower bounds and feasible solutions. The relaxed problems are solved by dynamic programming, yielding optimal solutions or at least feasible solutions with tight lower bounds in a few minutes. Cattrysse et al. [5] introduced a dual ascent and column generation heuristic to solve a DLSP with setup times formulated as a Set Partitioning Problem (SPP). Later, Van Hoesel et al. [6] formulated DLSP as an integer programming problem and presented two solution procedures: the first procedure based on a reformulation of DLSP as a linear programming assignment problem, with additional restrictions to reflect the specific (setup) cost structure; the second procedure based on dynamic programming (DP).

Besides, Jordan et al. [7] solved a Discrete Lot Sizing and Scheduling Problem for one machine with sequence-dependent setup times and setup costs as a single machine scheduling problem, which they named the batch sequencing problem. This batch sequencing problem is solved with a branch-and-bound algorithm accelerated by some bounding and dominance rules. Later, Miller and Wolsey [12] formulated the DLSP with setup costs not dependent on the sequence as a network flow problem. They exposed some MIP formulations for various modifications (with backlogging, safety stock, and initial supply). Moreover, several more MIP formulations and variants have been proposed and discussed by Pochet and Wolsey [15]. Gicquel et al. [18] exposed a formulation in which they derived valid inequations for the DLSP with several items and sequential setup costs and periods. This formulation is a modification of the problem proposed by Wolsey [10]. Another approach proposed by Gicquel et al. [17] consists of modeling the DLSP with several items and sequential setup costs and periods. It considers relevant physical attributes such as color, dimension, and quality. This allowed them to effectively reduce the number of variables and constraints in the MIP models.
Houndji et al. [20] introduced a new global constraint they named `StockingCost` to solve the PSP with Constraint Programming. They tested it on new instances and published it on

CSPlib (Gent and Walsh [8]). The experimental results showed that *stocking cost* is effective in filtering compared to other constraints in Constraint Programming. Later, Ceschia et al. [23] used the Simulated Annealing (SA) to solve the PSP. They introduced an approach along with a statistically-principled tuning procedure that guides the local search and used it to solve new instances available in the Opthub repository. Their solver could find near-optimal solutions in a short time for all instances, including those not solved by state-of-the-art methods [24]. More recently, Park et al. [36] proposed a framework for solving the DLSP using reinforcement learning in which they formalized the scheduling process as a sequential decision-making problem with the Markov decision process.

2.2 Genetic Algorithms and Optimization problems

Genetic Algorithms are stochastic search algorithms that mimic living species' natural evolution and reproduction mechanisms. They were proposed for the first time by John Holland [3] in 1970. One of the main principles of these algorithms is the concept of the "*survival of the fittest*". It states that one individual whose features fit the best in the environment is more likely to survive. Goldberg et al. [1] introduced the concept of Messy Genetic Algorithms. They solved the problems by combining relatively short, well-tested building blocks to form longer, more complex strings that increasingly cover all problem features. This approach stood in contrast to the usual fixed-length, fixed-coding genetic algorithm. By emulating natural mechanisms, Genetic Algorithms assure the evolution of a population over several generations with concepts such as *Initialization* [21], *Selection* [27], *Crossover* [26], or *mutation* [33] as shown in Figure 1.

Several studies explored the application of genetic algorithms in the context of optimization problems. A. Kimms [9] introduced a mixed-integer programming formulation for the multi-level, multi-machine proportional lot sizing and scheduling problem and presented a genetic algorithm to solve that problem. Later, J. Duda [13] presented a study of genetic algorithms for a lot-sizing problem formulated for operational production planning in a foundry. Three variants of the genetic algorithm were considered, each using special crossover and mutation operators and repair functions. Moreover, Xie et al. [11] proposed heuristic genetic algorithms for lot-sizing problems by designing a domain-specific encoding scheme for the lot sizes and by providing a heuristic shifting procedure as the decoding schedule. More recently, Laroche et al. [35] dealt with a complex production planning problem with lost sales, overtime, safety stock, and sequence-dependent setup times on parallel and unrelated machines by developing a genetic algorithm that combines several operations already defined in the literature to solve the problem.

III PROBLEM DEFINITION

Several studies addressed the Pigment Sequencing Problem - PSP (see, for example, [20, 23, 32]). It can be described as a problem that requires producing various items on one machine with predefined setup costs. Setup costs are necessary for the transition from an item i to another item j so that $i \neq j$. Often, the production planning needs to meet the customer orders while:

- not exceeding the production capacity of the machine;
- minimizing the setup and stocking costs.

Without loss of generality, it is assumed that only one item is produced per period and all orders are normalized i.e., the machine's production capacity is restricted to one item per period and $d(i, t) \in \{0, 1\}$ with i the item and t the period. The PSP is a production planning problem with

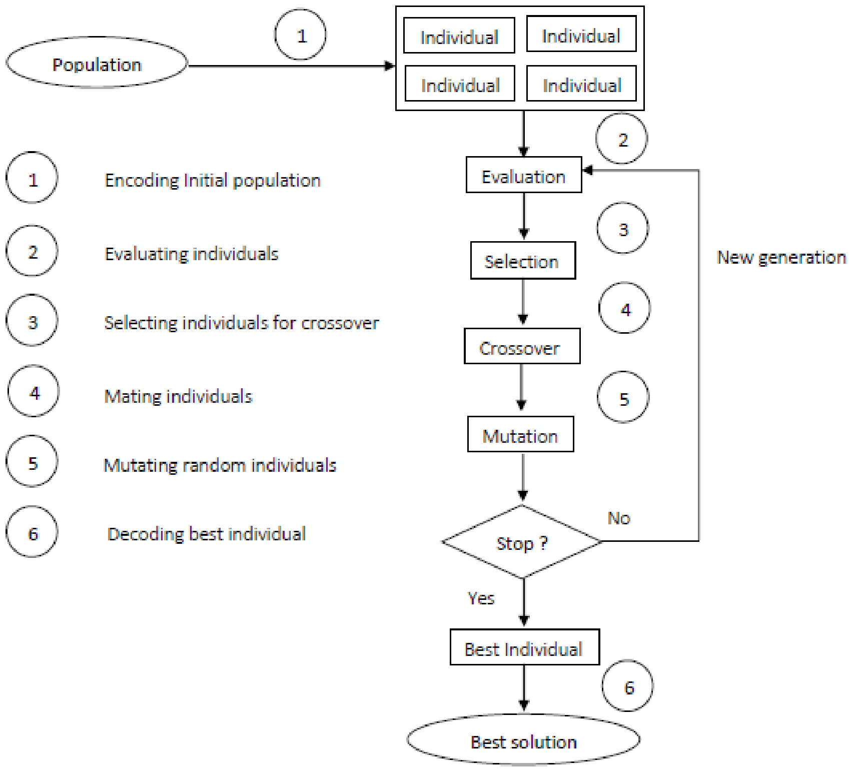


Figure 1: Flow chart of Genetic Algorithms.

the following specifications: a discrete and finite planning horizon, some capacity constraints, a deterministic and static order, several items, small buckets, setup costs, only one level, and without shortage.

Formally, the problem can be formulated as [15] :

$$min \sum_{i,j=0}^{M} \sum_{t=0}^{P} q^{i,j} X_t^{i,j} + \sum_{i=0}^{M} \sum_{t=1}^{P} h^i s_t^i \tag{1}$$

$$s_0^i = 0, \forall i \in M \tag{2}$$

$$x_t^i + s_{t-1}^i = d_t^i + s_t^i, \forall i \in M, t \in P \tag{3}$$

$$x_t^i \leq y_t^i, \forall i \in M, t \in P \tag{4}$$

$$\sum_i y_t^i = 1, \forall i, j \in M, t \in P \tag{5}$$

$$X_t^{i,j} \geq y_{t-1}^i + y_t^j - 1, \forall i, j \in M, t \in P \tag{6}$$

with the following indices and index sets:

- M: set of item indices, $i, j \in M$ and $M \subseteq \mathbb{N}$;
- P: set of period indices, $t \in P$ and $P \subseteq \mathbb{N}$;

the parameters:

- h_i: the holding cost of the item *i* with $i \in M$;
- $q^{i,j}$: the changeover cost from item *i* to item *j* with $i, j \in M$;

and the following variables:

- x_t^i: binary production variable that is 1 if item *i* is produced in period *t*, 0 otherwise;

- y_t^i: binary setup variable that is 1 if the machine is set for the production of item *i* in period *t*, 0 otherwise;
- d_t^i: binary variable that is 1 if item *i* is ordered in period *t*, 0 otherwise;
- $X_t^{i,j}$: binary changeover variable that is 1 if in period *t*, we transitioned from the production of item *i* to the one of item *j*, 0 otherwise;
- s_t^i: integer variable that represents the number of item *i* stored in the period *t*, $s_t^i \in \mathbb{R}_+$.

The goal is to minimize the overall stocking and setup costs, as expressed by (1). Constraint (2) clearly states that there is no initial stock. Constraint (3) expresses the rule of flow conservation. Constraint (4) aims to get the setup variable y_t^i to equal 1 if the item *i* is produced in the period *t*. Constraint (5) ensures the machine is always set to produce an item. Therefore, y_t^i is bound to take the value that minimizes the changeover cost. Furthermore, if there is no production in the period *t*, $y_t^i = y_{t-1}^i$ or $y_t^i = y_{t+1}^i$. Thus, it is interesting to set up the machine for production even if there is no item to produce. Constraint (6) sets values to changeover variables. If y_{t-1}^i and y_t^i equal 1, then $X_t^{i,j}$ is bound to equal 1; otherwise, $X_t^{i,j}$ would equal 0 thanks to the goal function that minimizes the changeover cost.

Example: Consider the following tiny problem:

- Number of items: $NI = 2$;
- Number of periods: $NT = 5$;
- Order per period. Be d(*i*, *t*) the order of item *i* in the period *t*: $d(1,t) = (0,1,0,0,1)$ and $d(2,t) = (1,0,0,0,1)$;
- Stocking cost. Be *h(i)* the stocking cost of the item i, $h(1) = h(2) = 2$

Let *xT* be the production planning representing a potential solution to the problem. It is an array of size *NT*. A possible solution to the problem is $xT = (2,1,2,0,1)$ with a cost of $q(2,1) + q(1,2) + q(2,1) + 2h(2) = 15$. The optimal solution is $xT = (2,1,0,1,2)$ with a cost of $q(2,1) + q(1,2) + h(1) = 10$.

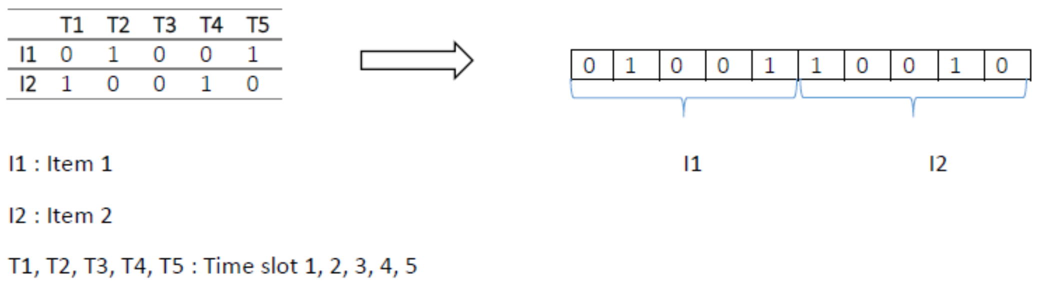
IV OUR APPROACH

This section presents each aspect of implementing genetic algorithms to solve the PSP.

4.1 Genetic representation

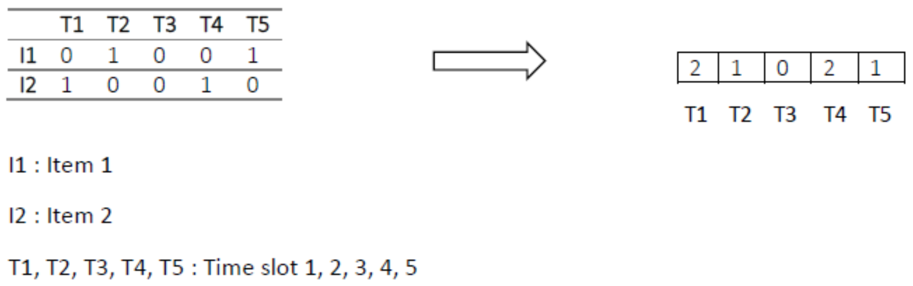
When implementing genetic algorithms to solve a problem, finding the proper representation for the individual is important and influences the efficiency of the whole algorithm. One of the most straightforward representations used in genetic algorithms is the one used by John Holland [4]: the bit-array representation where a chromosome is represented by a string of bits containing 0 and 1 to express if an item i has been produced at a given period t as pictured on Figure 2.

Although correct, this representation significantly increases the complexity of the whole algorithm, forcing us to go through a list of $nT * nI$ items with ***nT**: the number of periods* and ***nI**: the number of items*. All of this prompted the emergence of another representation, as used by Mirshekarian et al. [31], in which the chromosome is represented by a string of integers of the length of the planning horizon (*nT*). In this string, each period corresponds to the produced item's index or 0 otherwise (as shown in Figure 3). Thus, the complexity is considerably reduced.



	T1	T2	T3	T4	T5
I1	0	1	0	0	1
I2	1	0	0	1	0

Figure 2: Chromosome bit-array representation.



	T1	T2	T3	T4	T5
I1	0	1	0	0	1
I2	1	0	0	1	0

Figure 3: Chromosome final representation.

4.2 Initialization

As stated earlier, the initialization process consists of generating the initial population. We have opted for the heuristic algorithm based on the breadth-first search technique described in Algorithm IV.1. The process starts at the end of the planning horizon and backtracks to the first production period. The goal is to seed the best possible individuals for the initial population. At every step of the process, the algorithm determines which subsequent child nodes are the best to expand. This process produces better individuals than random seeding [21], helping bootstrap the overall search process.

4.3 Evaluation

Each chromosome is evaluated before proceeding to the selection. The cost estimation is a key input to the selection process and the element the genetic-based algorithm seeks to minimize. In our study, the cost function is based on the aforementioned MIP formulation (1) and can be stated as follows:

$$F(x) = \sum_{i,j=0}^{M} \sum_{t=0}^{P} q^{i,j} X_t^{i,j} + \sum_{i=0}^{M} \sum_{t=1}^{P} h^i s_t^i \tag{7}$$

It consists of two costs:

- the setup cost $\sum_{i,j=0}^{M} \sum_{t=0}^{P} q^{i,j} X_t^{i,j}$: the sum of the setup costs for all periods;
- the stocking cost $\sum_{i=0}^{M} \sum_{t=1}^{P} h^i s_t^i$: the sum of the stocking time slots of all items multiplied by the stocking cost h^i of each item i

Algorithm IV.1 Population initialization algorithm.

```
BEGIN
    READ Expected_Population_Size, PSP_Instance
    SET population to []
    SET queue to firstNode
    SET popCounter to queue length
    WHILE population length is less than Expected_Population_Size
        IF queue is empty THEN
            BREAK
        SET node to popFirst (queue)
        DECREMENT popCounter
        IF node is leafNode THEN
            ADD node chromosome to population
            CONTINUE
        ENDIF
        FOR child in node children (PSP_Instance)
            APPEND child to queue
            INCREMENT popCounter
            IF popCounter is greater than Expected_Population_Size THEN
                BREAK
        ENDFOR
    ENDWHILE
END
```

4.4 Selection

The selection operator we chose to implement is based on the process commonly known as the "Roulette wheel" [27]. Hence, each chromosome is given a probability of being selected based on its fitness. Therefore, the fittest chromosome is given the highest chance. Then, a selector is used to pick two chromosomes based on their probability. Those chromosomes will mate and produce offspring. We evaluate each chromosome based on the data provided by each instance and for each item (stocking cost and setup cost) (7). The higher the cost, the less fit the chromosome is, and the lower the probability of being chosen. In practice, the fitness of each chromosome in a population is computed (8) relative to the cost of the fittest chromosome of this population (9)

$$M = max(c), \forall c \in P \tag{8}$$

$$p_i = ((M+1) - B_i) / \sum_c ((M+1) - B_c) \tag{9}$$

.

along with the following variables:

- M: the cost of the fittest chromosome of the population P;
- p_i: the "Roulette wheel" probability of the chromosome *i*;
- B_i: the production cost of the chromosome *i*;

4.5 Crossover

In the crossover, both chromosomes obtained from the selection process are mated only if it has been randomly decided so. A random number is drawn, and the crossover occurs if it is below the crossover rate. In the implementation (Algorithm IV.2), we mate two chromosomes to produce one offspring, which consists of iteratively moving Chromosome 1 towards Chromosome 2 while reducing its production cost and, therefore, improving its fitness. This method is inspired by the principle of the heuristic crossover as described by Umbarkar et al. [22]. We ensure the generated offspring is a new chromosome i.e. it has never been encountered before. This crossover implementation is interesting because it improves the overall fitness score of the population over the generations. The process is best illustrated by the following example (each chromosome is represented with its cost):
Parent 1: (2, 2, 1, 1, 3, 0, 2, 0): 592 -> the one chosen for yielding the offspring
Parent 2: (0, 2, 2, 2, 3, 1, 0, 1): 375
Offspring (Step 1): (2, 2, 1, 1, 0, 3, 2, 0): 580
Offspring (Step 2): (2, 2, 1, 0, 1, 3, 2, 0): 570
Offspring (Step 3): (2, 2, 0, 1, 1, 3, 2, 0): 560
Offspring (Final Step): (2, 0, 2, 1, 1, 3, 2, 0): 545

Algorithm IV.2 Crossover operator algorithm.

```
BEGIN
    READ chromosome1, chromosome2, crossoverRate, PSP_instance
    SET randomValue to random()
    SET distanceD to distance (chromosome1, chromosome2)
    IF randomValue is less than crossoverRate THEN
        FOR neighborChromosome in random shuffle(chromosome.neighbors(PSP_instance))
            IF distance (neighborChromosome, chromosome2) is less than distanceD and
            neighborChromosome is new THEN
                IF neighborChromosome.cost is less than chromosome.cost THEN
                    CALL crossover with neighborChromosome, chromosome2, crossoverRate
                        and PSP_instance
            ENDIF
        ENDFOR
    ENDIF
    CALL localSearch with chromosome1, chromosome2 and PSP_instance
END
```

4.6 Mutation

Once the crossover is performed, the random process of mutation takes place. For each offspring obtained from the crossover, it is randomly decided whether or not this chromosome should undergo a mutation. A mutation occurs if the randomly drawn number is below the mutation rate. The algorithm checks whether switching place with another nearby gene is possible for each randomly picked chromosome gene. It is about checking if it is possible to produce an item at another period other than the one it is currently produced without violating the instance's constraints as described by Algorithm IV.3. Not only does it have to respect the constraints, but it also has to ensure the generated chromosome is a new chromosome in the sense that it has never been encountered before. This condition allows for the exploration of new areas of the

search space. The process is best illustrated by the following example:
Input Chromosome: (2, 2, 1, 0, 1, 3, 2, 0) -> with a possible mutation (switch) of periods 2 and 3
Result of mutation: (2, 2, 0, 1, 1, 3, 2, 0)

Algorithm IV.3 Mutation operator algorithm.

```
BEGIN
    READ chromosome, mutationRate, PSP_instance
    SET randomValue to random()
    IF randomValue is less than mutationRate THEN
        FOR neighborChromosome in random shuffle(chromosome neighbors(PSP_instance))
            IF neighborChromosome is new THEN
                RETURN neighborChromosome
        ENDFOR
    ENDIF
    RETURN None
END
```

4.7 Application of the Hybridization concept

The hybridization concept suggests combining two search methods to produce better results. As shown by Gopal et al. [19], local search and Genetic Algorithms are two complement solutions. On the one hand, Genetic Algorithms perform well on the global scale because they can quickly find promising regions, but they take a relatively long time to find the optima in those regions. On the other hand, local search algorithms can, despite their well-known pitfalls, find the local optima with high accuracy. This entices the implementation of a local search in our study. This local search (Algorithm IV.4) is performed every time the crossover cannot generate a new offspring. It is fairly based on the *Hill climbing* method which is one of the most straightforward heuristics used in local search algorithms. Its fast convergence and memory-efficiency characteristics have repeatedly proven critical. Therefore, our local search algorithm searches in a large neighborhood of Chromosome 1 towards Chromosome 2 to see if a better result can be found. This algorithm is also helpful as it prevents getting stuck at some local optima. Figure 4 provides some measurements backing our use of a local search algorithm to refine individuals and improve the quality of the solutions. On the axis x are displayed the labels of the CSPlib repository's instances [5.2] used to prove this, and on the axis y are shown the average gap between the found solutions and the optimal ones. The blue bars (AverageGap1) picture this average gap while using the local search algorithm, and the red ones present this average gap (AverageGap2) without using a local search algorithm. Hence, the hybridization improves the quality of the found solution on average by over 81.5%.

4.8 Termination

We define that the algorithm stops once it cannot improve the best solution found so far over a given number of generations. In our case, this number is 5. We call these generations "idle" generations.

Algorithm IV.4 Local search algorithm.

```
BEGIN
    READ chromosome1, chromosome2, PSP_instance
    SET distanceD to distance (chromosome1, chromosome2)
    FOR neighborChromosome in random shuffle(chromosome neighbors(PSP_instance))
        IF distance (neighborChromosome, chromosome2) is less than distanceD
            and neighborChromosome is new THEN
            IF neighborChromosome cost is less than chromosome1 cost THEN
                RETURN neighborChromosome
            CALL localSearch with neighborChromosome, chromosome2 and PSP_instance
        ENDIF
    ENDFOR
    RETURN None
END
```

V EXPERIMENTAL RESULTS

In this section, we first present the tools used in the implementation and tests. We then describe the instances in which we performed our approach to Genetic Algorithms and the parameters we defined. Finally, we expose the experimental results obtained from the tests.

5.1 Tools

Our approach (available at [28]) is implemented using Python, specifically version 3.6, and on a computer with the following specifications:

- Operating system: Linux Ubuntu 18.04.6 LTS ;
- Processor: Intel® Core TM i5-8250U CPU @ 1.60GHz * 8 ;
- Memory: 11.6 GiB ;
- Type of the operating system: 64 bits ;
- Graphics: Intel® UHD Graphics 620 (KBL GT2) ;

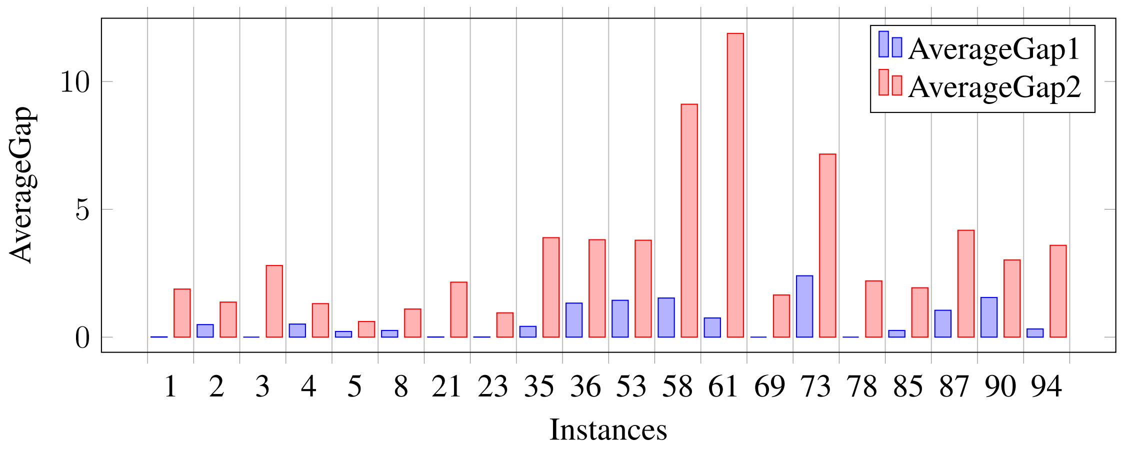


Figure 4: Chart of the average gaps with (AverageGap1) and without (AverageGap2) local search.

5.2 Benchmarks

To our knowledge, the Pigment Sequencing Problem has two publicly available benchmarks. Houndji et al. proposed publicly available instances (and their corresponding best solutions) in the CSPlib. Some of these instances are characterized by a number of periods of *NT=20*, a number of items of *NI=5*, and a number of orders of *ND=20*. Others are characterized by a much higher number of periods (100 or 200) and a higher number of items (10 or 15): pigment100a, pigment100b, pigment200a. Later, in their study of the PSP and seeking to apply their Simulated Annealing approach to some more complex instances of the problem, Ceschia et al. [23] developed a parameterized generator that receives as input the number of items m, the number of periods n, and the density of requests δ (i.e., total request divided by n) and produces a random instance with those features.

For our tests, and considering we are in an early phase of our application of Genetic Algorithms to the Pigment Sequencing Problem, we resolve the instances available in the CSPlib repository. We test our approach on this benchmark with 20 instances picked for these first experiments.

5.3 Parameter tuning

The performance of genetic algorithms is greatly affected by the settings of their parameters. These parameters and the population size are specifically the probabilities of crossover and mutation. Several studies [34] have explored the impact of these different parameters on the quality of the solutions. The following notions can be derived from these studies:

- Crossover is made hoping that new chromosomes will have good parts of old chromosomes. Hence, the crossover probability, the controlling parameter here, is expected to be a high value but not too high to let some of the population survive to the next generation.
- The mutation probability, which is the parameter that determines the likelihood that an individual will undergo the mutation, is expected to be a low value. A high value of mutation probability tends to prevent the population from converging to an optimum solution.
- The population size, the number of individuals in the population, tends to slow the algorithm when it is too high and shrink the exploration space otherwise.

All these parameters are dependent on the problem being solved. However, for the sake of our study, we randomly pick some instances from the CSPlib repository and draw from the state of the art to set the range of each of these parameters for our tuning exercise as follows: the mutation probability [0.05, ..., 0.15] with a step of 0.01, the crossover probability [0.75, ..., 0.9] with a step of 0.1, the population size [25, ..., 40] with a step of 1 with 10 trials over each instance. The only performance characteristic is the accuracy of the solution. Figure 5 pictures the distribution of the error rate, symbolizing the performance of the algorithm throughout the parameter tuning. This distribution is left-skewed, showing that, for most of the values of the population size, the mutation rate, and the crossover rate, the error rate is fairly low (lower than 0.001). However, the computation of the correlation coefficient of the error rate with these same variables (the population size, the mutation rate, and the crossover rate, respectively 0.063, -0.052, and -0.19) exposes a weak correlation of these parameters with the error rate that we attribute to the fairly stochastic nature of the overall scheme. Nevertheless, after multiple iterations, a recurring set of values emerged and can be represented as follows:

- Size of the population L_p: 30;
- Probability of mutation P_m: 0.05;
- Probability of crossover P_c: 0.9.

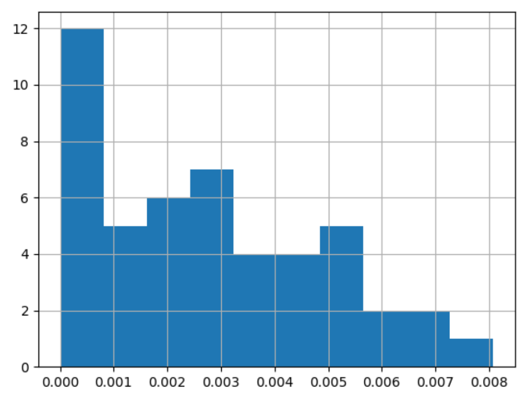


Figure 5: Histogram of the error rate representing the algorithm's performance during the parameter tuning.

5.4 Results

Once our parameter values are set and to test our approach, we draw a very miscellaneous set of 20 instances from the CSPlib repository and run it ten times over each instance using the aforementioned parameter values [5.3] to configure every run.

For each instance, after ten runs, we write down the solutions found and determine the best solution among them and the time spent searching for it. Table 1 compiles the experiment's results on the instances from the CSPlib repository. For each instance (represented as Instance NI-NT), we note the optimal solution, the time spent by the CP algorithm [29] to reach it, the best solution found by our approach over ten runs, and the corresponding time, along with the gap between the global optimum and the best solution, the coefficient of variation of the solutions and the meantime of the search.

When analyzing these results, it appeared essential to proceed with a statistical analysis due to the stochastic nature of Genetic Algorithms. From the results of Table 1, we notice that our approach of Genetic Algorithms has successfully spotted the global optimum for most instances (3/4 of the tested instances). For the remaining 1/4 of the instances, our approach identified a solution close to the global optimum (on average 2.008% close) . We suspect these instances to have a little bit more convoluted search space. On all the instances, our approach finds the global optimum or a solution close to this one quite easily with a gap between the global optimum and the found solution not exceeding 7.3% and an average of 0.502%. Moreover, the results from the instances p100a, p100b, p100c, and p200a show a trend similar to those with fewer periods (15, 20, 30 periods). On these larger instances, our approach succeeds in finding the global optimum or a solution close to it (0.611% close on average) .
Given that Genetic Algorithms are stochastic methods and having tested each instance 10 times, we analyze the coefficient of variation of all the solutions found for each instance. We note

[1]the global optimum as available in Csplib repository
[2]the time (in seconds) spent by CP to find the global optimum [32]
[3]the best solution found by our approach
[4]the time (in seconds) spent by our approach to finding its best solution
[5]the gap between the global optimum and the best solution found by our approach
[6]the coefficient of variation of all the solutions found over 10 trials
[7]the meantime (in seconds) of all the 10 trials

Instance	Opt[1]	CP time [2]	GA Best [3]	GA time [4]	Gap [5]	Coef var. [6]	Meantime [7]
1 5-20	1377	9.14	1377	1.838	0%	0.031	2.305
3 5-20	1107	2.946	1107	1.294	0%	0	1.765
5 5-20	1471	0.235	1471	0.949	0%	0.294	0.858
8 5-20	3117	25.352	3117	2.815	0%	0.285	2.583
23 5-20	1473	15.039	1473	1.418	0%	0.021	1.798
36 5-20	1493	121.909	1502	2.98	0.6%	0.756	2.495
58 5-20	1384	2.347	1386	2.767	0.1%	1.508	2.462
69 5-20	1619	1.223	1619	1.487	0%	0	1.757
78 5-20	1297	16.187	1297	1.173	0%	0	1.434
85 5-20	2113	9.404	2113	2.766	0%	0.242	2.954
90 5-20	2449	23.811	2449	1.861	0%	1.36	2.288
94 5-20	1403	11.726	1403	1.683	0%	0.763	2.207
p15b 10-15	1486	12	1486	6.819	0%	0.558	3.521
p15c 10-15	1583	16	1583	1.675	0%	0.133	2.03
p30a 5-30	1119	124	1201	1.51	7.327%	0.414	1.817
p30c 10-30	1707	156	1731	1.741	1.405%	0	2.351
p100a 10-100	1323	60	1323	4.863	0%	0.239	7.853
p100b 10-100	1962	10	1974	8.004	0.611%	2.863	9.067
p100c 15-100	1982	143	1982	10.358	0%	0.182	15.749
p200a 15-200	2324	854	2324	28.61	0%	0	32.848
Average	-	80.715	-	4.33	0.502%	0.482	5.007

Table 1: Experimental results on 20 CSPlib instances.

that this metric, which measures the dispersion of the found solutions around a mean, doesn't exceed a maximum value of 1.508. This helps us infer that over the 10 trials for each instance, our approach has consistently found a solution quite close to the mean. These results are to be put in perspective with the ones of the CP implementation shown in Table 1. Note that the CP approach (Houndji et al. [25]), whose results are reported here, has successfully found the optimum and proved the optimality for these instances.

Overall, these results suggest that our approach of Genetic Algorithms can easily find a solution quite close to the global optimum for this type of instance of the PSP.

VI CONCLUSION AND PERSPECTIVES

In this paper, we have solved the Pigment Sequencing Problem (PSP), a Discrete Lot Sizing and Scheduling Problem (DLSP), using Genetic Algorithms. We have presented the basic concepts supporting the implementation of Genetic Algorithms. Solving a Discrete Lot Sizing and Scheduling Problem with Genetic Algorithms is met with some exciting challenges, including the good design of the chromosome and the right choice in implementing aspects such as the selection, the initialization, the crossover, and the mutation. We have experimentally shown that using Genetic Algorithms' approaches to solving a DLSP can be a promising research area.

As further works, we would like to dive deeper into designing and experimenting with new approaches of crossover and mutation. It would also be interesting to test our approach on more complex instances or variants of DLSP.

REFERENCES

Publications

[1] D. Goldberg, B. Korb, and K. Deb. “Messy Genetic Algorithms: Motivation, Analysis, and First Results”. In: *Complex Systems* 3 (1989), pages 493–530.

[2] B. Fleischmann. “The discrete lot-sizing and scheduling problem”. In: *European Journal of Operational Research* 44 (Feb. 1990), pages 337–348.

[3] J. H. Holland. “Adaptation in natural and artificial systems : an introductory analysis with applications to biology, control, and artificial intelligence”. In: *MIT press* (1992).

[4] J. H. Holland. “Computer programs that "evolve" in ways that resemble natural selection can solve complex problems even their creators do not fully understand”. 1992.

[5] D. Cattrysse, M. Salomon, R. Kuik, and L. N. V. Wassenhove. “A Dual Ascent and Column Generation Heuristic for the Discrete Lotsizing and Scheduling Problem with Setup Times”. In: *Management Science* 39 (1993), pages 477–486.

[6] S. V. Hoesel, R. Kuik, M. Salomon, and L. N. V. Wassenhove. “The single-item discrete lot-sizing and scheduling problem: Optimization by linear and dynamic programming”. In: *Discrete Applied Mathematics* 48 (Feb. 1994), pages 289–303.

[7] C. Jordan and A. Drexl. “Discrete Lotsizing and Scheduling by Batch Sequencing”. In: *Management Science* 44 (May 1998), pages 698–713.

[8] I. P. Gent and T. Walsh. “CSPlib: A Benchmark Library for Constraints”. In: (1999). Edited by J. Jaffar, pages 480–481.

[9] A. Kimms. “A Genetic Algorithm for Multi-Level, Multi-Machine Lot Sizing and Scheduling”. In: *Computers Operations Research* 26 (July 1999), pages 829–848.

[10] L. A. Wolsey. “Solving multi-item lot-sizing problems with a mip solver using classification and reformulation”. In: *Management Science* 48 (2002), pages 1587–1602.

[11] J. Xie and J. Dong. “Heuristic genetic algorithms for general capacitated lot-sizing problems”. In: *Computers Mathematics with Applications* 44 (July 2002), pages 263–276.

[12] A. J. Miller and L. A. Wolsey. “Tight mip formulation for multi-item discrete lot-sizing problems”. In: *Operations Research* (2003), pages 557–565.

[13] J. Duda. “Lot-Sizing in a Foundry Using Genetic Algorithm and Repair Functions”. In: (2005). Edited by G. R. Raidl and J. Gottlieb, pages 101–111.

[14] J. F. Gonçalves, J. J. de Magalhaes Mendes, G. Mauricio, and C. Resende. “A hybrid genetic algorithm for the job shop scheduling problem”. In: *European journal of operational research* (2005), pages 77–95.

[15] Y. Pochet and L. A. Wolsey. *Production planning by mixed integer programming*. Springer Science and Business Media, 2006.

[16] H. G. Goren, S. Tunali, and R. Jans. “A review of applications of genetic algorithms in lot sizing”. In: *Journal of Intelligent Manufacturing* 21 (May 2008), pages 575–590.

[17] C. Gicquel, N. Miègeville, M. Minoux, and Y. Dallery. “Discrete lot sizing and scheduling using product decomposition into attributes”. In: *Computers and Operations Research* (2009), pages 2690–2698.

[18] C. Gicquel, M. Minoux, and Y. Dallery. “On the discrete lot-sizing and scheduling problem with sequence-dependent changeover times”. In: *Operations Research Letters* (Jan. 2009), pages 32–36.

[19] G. Gopal. “Hybridization in Genetic Algorithms”. In: *International Journal of Advanced Research in Computer Science and Software Engineering* 3 (Apr. 2013), pages 403–409.

[20] V. R. Houndji, P. Schaus, L. A. Wolsey, and Y. Deville. "The StockingCost constraint". In: *International conference on principles and practice of constraint programming* (Sept. 2014), pages 382–397.

[21] B. Kazimipour, X. Li, and K. Qin. "A Review of Population Initialization Techniques for Evolutionary Algorithms". In: *Proceedings of the 2014 IEEE Congress on Evolutionary Computation, CEC 2014* (July 2014).

[22] A. Umbarkar and P. Sheth. "Crossover operators in genetic algorithms: a review". In: *ICTACT Journal on soft computing* 6 (Oct. 2015).

[23] S. Ceschia, L. D. Gaspero, and A. Schaerf. "Solving discrete lot-sizing and scheduling by simulated annealing". In: *Computers Industrial Engineering* 114 (2016), pages 235–243.

[25] V. R. Houndji. "Cost-based filtering algorithms for a capacitated lot sizing problem and the constrained arborescence problem". PhD thesis. Université Catholique de Louvain, 2017.

[26] P. Kora and P. Yadlapalli. "Crossover Operators in Genetic Algorithms: A Review". In: *International Journal of Computer Applications* 162 (Mar. 2017), pages 34–36.

[27] N. Saini. "Review of Selection Methods in Genetic Algorithms". In: *International Journal Of Engineering And Computer Science* 6 (Dec. 2017), pages 22261–22263.

[30] P. Koken, V. A. Raghavan, and S. W. Yoon. "A genetic algorithm based heuristic for dynamic lot sizing problem with returns and hybrid products". In: *Computers Industrial Engineering* 119 (May 2018), pages 453–464.

[31] S. Mirshekarian and G. Süer. "Experimental study of seeding in genetic algorithms with non-binary genetic representation". In: *Journal of Intelligent Manufacturing* 29 (Oct. 2018).

[32] V. R. Houndji, P. Schaus, and L. Wolsey. "The item dependent stockingcost constraint". In: *Constraints* 24 (2019), pages 183–209.

[33] S. Katoch, S. S. Chauhan, and V. Kumar. "A review on genetic algorithm: past, present, and future". In: *Multimedia Tools and Applications* 80 (Feb. 2021), pages 8091–8126.

[34] V. Vlasov, A. Khomchenko, A. Faizliev, S. Mironov, and A. Grigoriev. "Parameter tuning of a genetic algorithm for finding central vertices in graphs". In: *Journal of Physics: Conference Series* (2021).

[35] B. Badezet, F. Larroche, O. Bellenguez, and G. Massonnet. "A Genetic Algorithm for a Capacitated Lot-Sizing Problem with Lost Sales, Overtimes and Safety Stock Constraints". In: (Feb. 2022), pages 170–181.

[36] T. J. Park and Y. J. Jang. "Discrete Lot-Sizing Problem of Single Machine based on Reinforcement Learning Approach". In: *International Symposium on Semiconductor Manufacturing Intelligence* (2022).

Software Project

[24] [SOFTWARE] S. Ceschia, *Bibliothèque d'instances PSP*, 2017.

[28] [SOFTWARE] T. GNA, *PspSolver*, Feb. 1, 2017.

[29] [SOFTWARE] V.R. Houndji and P. Schaus, *CP4PP : Constraint Programming for Production Planning*, 2017.