Genetic Algorithms for a Discrete Lot Sizing Problem

Vinasetan Ratheil Houndji, Tafsir Gna

Genetic Algorithms for a Discrete Lot Sizing Problem

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

[1]University of Abomey-Calavi, Republic of Benin

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

Abstract

Lot sizing is essential 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 Problem (DLSP), notably the Pigment Sequencing Problem (PSP). We have implemented an approach based on genetic algorithms to solve the PSP. We have proposed adaptive ways for each step of the genetic approach: initialization, selection, crossover, and mutation. We experimentally evaluate the performance of the approach over multiple trials on some publicly available instances of the PSP. These experimental results show that genetic algorithms are promising and practical approaches to solving DLSP.

Keywords

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

I INTRODUCTION

Solving Lot Sizing Problems consists of identifying items to produce, when to produce, and on which machine to meet the demands while seeking to achieve some financial goals. Such problems have been studied lately. Not only are several types of items required to be produced, but the production planning must often meet opposite goals, such as satisfying customer needs and minimizing production and stocking costs. Several versions of Lot Sizing Problems have been explored in the literature. Lately, Houndji et al. [12] and Ceschia et al. [17] have worked on an NP-Hard variant known as the *Pigment Sequencing Problem* (Pochet and Wolsey [7]), included in the CSPlib library (Gent and Walsh, [3]). It requires producing several items on a single machine whose capacity is restricted to 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 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 [6] [21] have shown how efficient they could be in solving optimization problems. In this paper, we expose a search method based on genetic algorithms, then experiment with this approach. The results show that genetic algorithms are a promising method for solving Discrete Lot Sizing Problems such as the Pigment Sequencing Problem.

This paper is organized as follows: Section 2 exposes some background on the Pigment Sequencing Problem and Genetics Algorithms (GAs), Section 3 gives details on our method based

on genetic algorithms, Section 4 presents some experimental results obtained from the implementation of our approach, and Section 5 concludes this paper and provides some perspectives.

II BACKGROUND

2.1 Overview of the state of the art

The PSP belongs to the Discrete Lot Sizing Problems (DLSP) category. The PSP is a problem in which the full capacity available for a period is used to produce one item. Miller and Wolsey [5] 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). In addition, several more MIP formulations and variants have been proposed and discussed by Pochet and Wolsey [7]. Gicquel et al. [8] exposed a formulation. They derived valid inequations for the DLSP with several items and sequential setup costs and periods, which is a modification of the problem proposed by Wolsey [4]. Furthermore, Gicquel et al. [9] proposed a new approach to the modelization of the DLSP with several items and sequential setup costs and periods that considers relevant physical attributes such as color, dimension, and level of quality. This allowed them to effectively reduced the number of variables and constraints in the MIP models. Houndji et al. [12] introduced a new global constraint they named *stocking cost* to solve the PSP with constraint programming. They tested it on new instances and published it on CSPlib (Gent and Walsh [3]). The experimental results showed that *stocking cost* is effective in filtering compared to other constraints used mainly in the community of Constraint Programming. Lately, Ceschia et al. [16] used the Simulated Annealing (SA) to solve the PSP. They introduced an approach that guides the local search and used it to solve new instances available on the Opthub library [17].

2.2 The Pigment Sequencing Problem

Several studies addressed the PSP (see, for example, [11, 16]). 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 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 [7] :

$$min(\sum_{i,j,t} q^{i,j} X_t^{i,j} + \sum_{i,t} h^i s_t^i) \tag{1}$$

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

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

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

$$\sum_i y_t^i = 1, \forall t \tag{5}$$

$$X_t^{i,j} = y_{t-1}^i + y_t^j - 1, \forall i, i, t \tag{6}$$

$$x, y, X \in 0, 1, s \in N, i \in 0, ..., NI, t \in 1, ..., NT \tag{7}$$

along with the following variables:

- x_t^i: binary production variable that is 1 if the item *i* is produced in the period *t* and 0 otherwise;
- y_t^i: binary setup variable that is 1 if the machine is set for the production of the item *i* and 0 otherwise;
- s_t^i: integer variable that represents the number of item *i* stored in the period *t*;
- $q^{i,j}$: the changeover cost from item *i* to item *j* with $i, j \in \{0, ..., NI\}$;
- d_t^i: binary variable that is 1 if the item *i* is ordered in the period *t* and 0 otherwise;
- h_i: the holding cost of the item *i* with $i \in \{1, ..., NI\}$;
- $X_t^{i,j}$: binary variable that is 1 if in the period *t*, we transitioned from the production of item *i* to the one of item *j* and 0 otherwise.

The goal is to minimize the overall stocking and setup costs as expressed by the first constraint (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 relatively easy 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 $q(2, 1) + q(1, 2) + h(1) = 10$.

2.3 Genetic Algorithms

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 [2] in 1970. One of the main principles of these algorithms is the concept of the "*survival of the fittest*", which states that one individual whose features fit the best with the environment is more likely to survive. Emulating the process of natural evolution, Genetic Algorithms induce the random exchange of genetic material among individuals of the same population. This section exposes its implementation in the context of optimization problems with concepts such as *Initialization*, *Selection*, *Crossover*, or *mutation*.

Initialization. It consists of creating the first population, a set of individuals (potential solutions). There are several strategies when making the initial population. The initialization can be stochastic or deterministic [13]. Stochastic initialization means that solutions, usually a sequence of numbers, are "seeded" without any logic driving the process while respecting all constraints. Deterministic initialization, on the opposite, is a deterministic method of initializing the population using heuristics to determine the best potential solutions close enough to optimal solutions. A good design of the initialization process is essential when implementing genetic algorithms.

Selection. It consists in choosing individuals from a given population for later breeding. Individuals are picked based on their fitness. Individuals with better genetic material are more likely to be selected. There are various selection methods, among which the "*Roulette wheel*" method [15] is the most common. It consists in assigning a probability to each individual in the population based on their fitness.

Crossover. It occurs after two or more individuals have been selected for breeding. The process produces offspring combining the genetic material of the selected individuals. Several strategies can be used to generate offspring. The crossover process can be single-point, two-point, or k-point or be uniform [19].

Mutation. It is a genetic operator that randomly alters the genetic material of randomly selected individuals [22]. In doing so, mutation strives to infuse diversity into the population over generations. Ultimately, the mutation is designed to avoid premature convergence in the population and the trap of a local optimum.

Termination criterion. Several criteria can be used to stop or terminate genetic algorithms. It can, for example, be decided to stop a genetic algorithm once a convergence occurs, meaning that all the individuals in the population are similar or after a set number of generations. Figure 1 summarizes all these concepts.

III OUR APPROACH

In this section, we present each aspect of our implementation of genetic algorithms to solve the PSP.

3.1 Genetic representation

When implementing genetic algorithms to 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 [1]: 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.

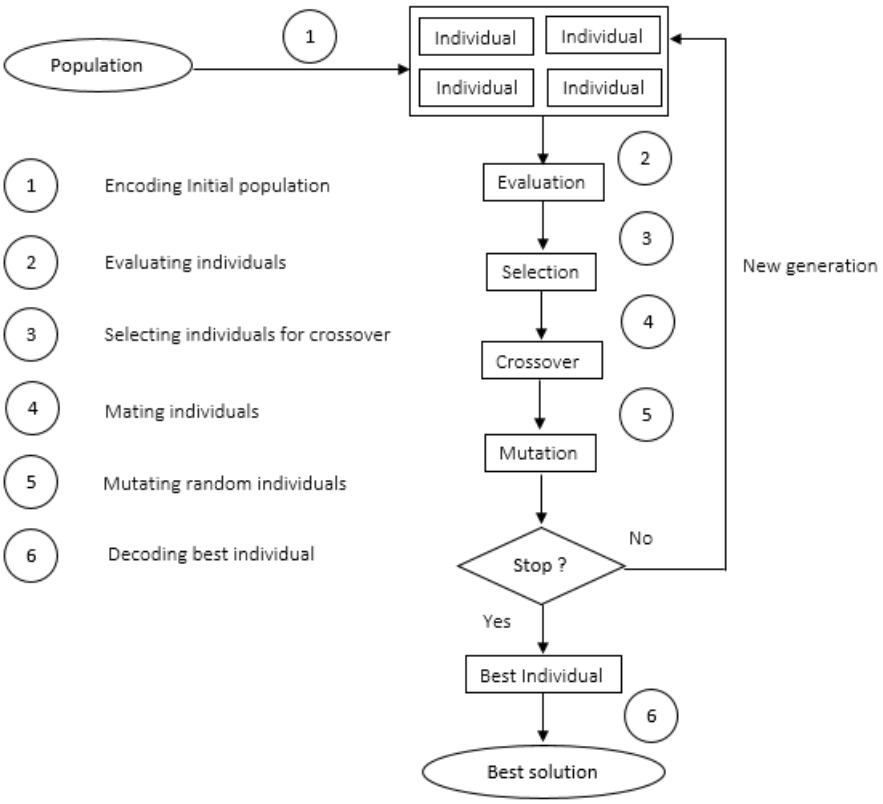

Figure 1: Flow chart of genetic algorithms

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. [21], in which the chromosome is represented by a string of integers of the length of the planning horizon (*nT*). In this string, each integer corresponds to the item's index produced at the exact period and 0 otherwise. Thus, the complexity is considerably reduced.

3.2 Initialization

As stated earlier, the initialization process consists in generating the initial population. We have opted for the heuristic algorithm based on the bread-first search technique described in Algorithm III.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 of the subsequent children nodes are the best to expand. This process produces better individuals than random seeding, helping bootstrap the overall search process.

3.3 Selection

The selection operator we chose to implement is based on the process commonly known as the "Roulette wheel". 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). The higher the cost, the less fit the chromosome

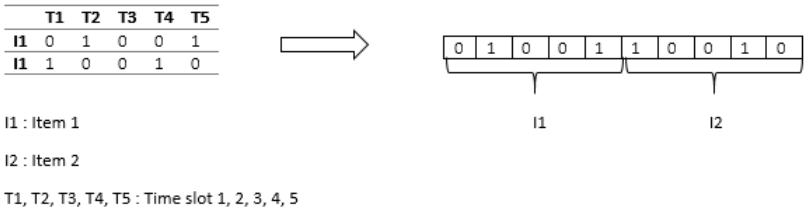

Figure 2: chromosome bit-array representation

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 chromosome with the highest cost in 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 chromosome with the highest cost in the population P;
- p_i: the "Roulette wheel" probability of the chromosome *i*;
- B_i: the production cost of the chromosome *i*;

3.4 Crossover

In the crossover, the two chromosomes obtained from the selection process are mated only if it has been randomly decided so. A random number is drawn, and if it is below the crossover

Algorithm III.1 Initial population algorithm

```
1   initialize (Expected_Population_Size, PSP_Instance)
2     population := []
3     queue := firstNode (PSP_Instance)
4     popCounter := queue.length
5     while population.length < Expected_Population_Size
6       if queue.empty
7         break
8       node := queue.popFirst ()
9       popCounter := popCounter - 1
10      if node is leafNode
11        population.add(node.chromosome)
12        continue
13      for child in node.children (PSP_Instance)
14        queue.append(child)
15        popCounter := popCounter + 1
16        if popCounter > Expected_Population_Size
17          break
```

rate, the crossover occurs. In the implementation (Algorithm III.2), we mate two chromosomes to produce one offspring, which consists in 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. [14]. The process produces offspring. We ensure the generated offspring is a new chromosome in the sense that it has never been encountered before. This crossover implementation is fundamental because it boots the improvement of the overall fitness score of the population over the generations.

Algorithm III.2 Crossover operator algorithm

```
1   crossover (chromosome1, chromosome2, crossoverRate, PSP_instance)
2       randomValue := random()
3       distanceD  :=  distance (chromosome1, chromosome2)
4       if  randomValue < crossoverRate
5           for  neighborChromosome in random.shuffle(chromosome.neighbors(PSP_instance))
6               if   distance (neighborChromosome, chromosome2) < distanceD and
7               isNewChromosome(neighborChromosome)
8                   if  neighborChromosome.cost < chromosome.cost
9                       return   crossover (neighborChromosome, chromosome2, crossoverRate, PSP_instance)
10
11      return   localSearch (chromosome1, chromosome2, PSP_instance)
```

3.5 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 a chromosome should undergo a mutation. A mutation occurs if the randomly drawn number is below the mutation rate. The algorithm checks, for each randomly picked chromosome gene, if it is possible to switch place with another nearby gene. Plainly, 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 constraints of the instance as described by Algorithm III.3. Not only does it have to respect the constraints, but this process also has to ensure the generated chromosome is a new chromosome in the sense that it has never been encountered before. This condition is essential to our approach to exploring new areas of the search space.

Algorithm III.3 Mutation operator algorithm

```
1   mutate(chromosome, mutationRate,  PSP_instance)
2     randomValue := random()
3     if  randomValue < mutationRate
4         for  neighborChromosome in random.shuffle(chromosome.neighbors(PSP_instance))
5         if  isNewChromosome(neighborChromosome)
6             return  neighborChromosome
7       return  None
```

3.6 Hybridization

The hybridization concept suggests combining two search methods to produce better results. Genetic algorithms are reasonably suitable methods for finding promising areas (exploration).

When exploiting the located regions, it is better to rely on a local search method to yield a better individual [10]. In our case, a local search (Algorithm III.4) is performed every time the crossover cannot generate a new offspring. The 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 and allows one to search beyond any encountered local optimum.

Algorithm III.4 Local search algorithm

```
localSearch (chromosome1, chromosome2, PSP_instance)
    distanceD := distance (chromosome1, chromosome2)
    for neighborChromosome in random.shuffle(chromosome.neighbors(PSP_instance))
        if distance (neighborChromosome, chromosome2) < distanceD and isNewChromosome(neighborChromos
            if neighborChromosome.cost < chromosome1.cost
                return neighborChromosome
            return localSearch (neighborChromosome, chromosome2, PSP_instance)
    return None
```

3.7 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.

IV EXPERIMENTAL RESULTS

In this section, we first present the tools used in the implementation and tests, then the instances on which we performed our approach of genetic algorithms and the hyperparameters we defined. Finally, we expose the experimental results obtained from the tests.

4.1 Tools

Our approach (available at [20]) is implemented using Python, specifically version 3.6. Python is well suited for this kind of implementation thanks to the vast amount of packages available for handling such data. We implemented the tests 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) ;

4.2 Instances

To test our approach, we use a set of 20 instances out of the 100 proposed by Houndji et al. [18] in the CSPlib library. We have chosen the first five ones and fifteen random others. These instances are characterized by the number of periods *NT=20*, the number of items *NI=5*, and the number of orders *ND=20*.

Instance	Opt[1]	CP time [2]	GA Best	GA time	GA worst	coef var.	mean time	bst occ. [3]
1	1377	9.14	1377	1.518	1381	0.121	2.042	70
2	1447	7.292	1447	1.737	1471	0.708	1.690	70
3	1107	2.946	1107	1.604	1107	0	1.992	100
4	1182	1.784	1182	1.759	1189	0.306	1.864	60
5	1471	0.235	1471	1.549	1480	0.321	1.397	50
8	3117	25.352	3117	3.065	3141	0.290	2.958	40
21	2774	11.177	2774	1.763	2793	0.223	1.914	70
23	1473	15.039	1473	1.821	1476	0.085	1.988	70
35	2655	12.846	2655	2.516	2674	0.275	2.644	20
36	1493	121.909	1505	2.552	1543	0.760	2.730	0
53	1108	0.935	1108	2.368	1128	0.748	2.330	20
58	1384	2.347	1384	4.374	1496	2.704	3.257	10
61	977	0.711	977	1.541	1053	2.546	1.970	60
69	1619	1.223	1619	1.755	1635	0.313	1.905	90
73	1104	12.508	1130	3.618	1172	1.326	2.775	0
78	1297	16.187	1297	1.276	1297	0	1.631	100
85	2113	9.404	2113	3.112	2136	0.386	2.817	30
87	1152	1.589	1152	2.586	1182	1.065	2.430	30
90	2449	23.811	2449	2.412	2520	1.098	2.380	20
94	1403	11.726	1403	1.866	1415	0.270	1.960	90

Table 1: Experiment results

4.3 Results

We have tested our approach by running it ten times over each instance and used the following parameters to configure every run:

- size of the population: 25 individuals;
- mutation rate: 0.05 ;
- crossover rate: 0.8.

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 contains, for each instance, the optimal solution, the time used by the CP algorithm to reach it, the best solution found by our approach over ten runs, and the corresponding time. Table 1 also shows, for each instance, the worst result obtained and some computed data, such as the coefficient of variation of the solutions, the mean time of the search, and the percentage of occurrences of global optimum in these ten runs.

When analyzing the 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 to genetic algorithms has successfully spotted the global optimum for all the instances except for two (Instances 36 and 73). Furthermore, for most instances (16 out of 20), the global optimum has been spotted much more quickly (on average, 75% faster) than the CP exact approach. Besides, even though our approach has underperformed the CP implementation in time on some instances (5, 53, 61, 69), it has found the global optimum in each of these instances. When focusing on the overall quality of the results found for the 20 instances, the mean time for solving the 20 instances confirms that the gain in time noticed earlier is consistent. The improvements are noticeable (on average, 79%). The compute of the coefficient of variation, as displayed in

Table 1, which measures the dispersion of the results around an expected value, shows that for all the 20 instances (with ten runs each), the results found tend to be close to the global optimum (the coefficient of variation for 15 of the instances is less than 1%) with 2 of the instances for which the algorithm is consistently able to find the global optimum (instances 3 and 78). These results suggest that our approach of genetic algorithms can find, over multiple trials, the global optimum or a solution quite close to the global optimum for this type of instance of the PSP (the ones proposed by Houndji et al. [18]).

V CONCLUSION AND PERSPECTIVES

In this paper, we have solved the Pigment Sequencing Problem (PSP), a Discrete Lot Sizing Problem (DLSP), using Genetic Algorithms. Then, we have exposed the basic concepts supporting the implementation of Genetic Algorithms. Solving Discrete Lot Sizing 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 demonstrated that the Genetic based approaches to solving a DLSP are 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 other variants of DLSP.

REFERENCES

Publications

[1] J. Holland. “Computer programs that "evolve" in ways that resemble natural selection can solve complex problems even their creators do not fully understand”. See also URL `http://www.abc.edu+`. 1992.

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

[3] I. P. Gent and T. Walsh. “Csplib : a benchmark library fr constraints”. In: *Springer* (1999), pages 480–481.

[4] L. A. Wolsey. “Solving multi-item lot-sizing problems with an mip solver using classification and reformulation”. In: *Management Science* 48(12) (2002), pages 1587–1602.

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

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

[7] L. A. W. Yves Pochet. “Production planning by mixed integer programming”. In: *Springer Science and Business Media* (2006).

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

[3]Global Optimum

[3]time spent by CP to find the global optimum

[3]the percentage of occurrences of the global optimum in all the results

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

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

[11] V. R. Houndji. “Deux problèmes de planification de production : Formulations et résolution par programmation en nombres entiers et par programmation par contraintes”. Master’s thesis. Université Catholique de Louvain, 2013.

[12] V. R. Houndji, L. W. Pierre Schaus, and Y. Deville. “The stockingcost constraint”. In: *International conference on principles and practice of constraint programming* (2014), pages 382–397.

[13] B. Kazimipour, X. Li, and K. Qin. “A Review of Population Initialization Techniques for Evolutionary Algorithms”. In: July 2014.

[14] P. S. A.J Umbarkar. “Crossover operators in genetic algorithms: a review”. In: *ICTACT Journal on soft computing,* 29 (Oct. 2015).

[15] N. Saini. “Review of Selection Methods in Genetic Algorithms”. In: *International Journal Of Engineering And Computer Science* (2015).

[16] L. D. G. S. Ceschia and A. Schaerf. “Solving discrete lot-sizing and scheduling by simulated annealing”. In: (2016).

[18] V. R. Houndji. “Cost-based filtering algorithms for a capacitated lot sizing problem and the constrai- ned arborescence problem”. PhD thesis. Université Catholique de Louvain, 2017.

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

[21] 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).

[22] Katoch, C. Sourabh, and V. Sumit Singh Kumar. “A review on genetic algorithm: past, present, and future”. In: *Multimedia Tools and Applications* 80 (Feb. 2021), pages 8091–8126.

Software Project

[17] [SOFTWARE] S. Ceschia, *Bibliothèque d’instances PSP*, 2017.

[20] [SOFTWARE] Tafsir GNA, *PspSolver* version 0.1, Feb. 1, 2017.