Piotr ŚWITALSKI[1],
Arkadiusz BOLESTA[1]

[1] Siedlce University of Natural Sciences and Humanities
Faculty of Exact and Natural Sciences
Institute of Computer Science
ul. 3 Maja 54, 08-110 Siedlce, Poland

Firefly algorithm applied to the job-shop scheduling problem

DOI: 10.34739/si.2021.25.05

Abstract. The job shop scheduling problem (JSSP) is one of the most researched scheduling problems. This problem belongs to the NP-hard class. An optimal solution for this category of problems is rarely possible. We try to find suboptimal solutions using heuristics or metaheuristics. The firefly algorithm is a great example of a metaheuristic. In this paper, this algorithm is used to solve JSSP. We used some benchmarking JSSP datasets for experiments. The experimental program was implemented in the aitoa library. We investigated the optimal parameter settings of this algorithm in terms of JSSP. Analysis of the experimental results shows that the algorithm is useful to solve scheduling problems.

Keywords. scheduling, job shop, firefly algorithm, aitoa library

1. Introduction

1.1. Scheduling problem

Scheduling in computer science is often referred to as NP-hard problems. The nature of this problem is exponential with an increasing number of tasks and machines. In scheduling, we decide about allocation of a set of tasks (activities) to available resources (e.g. machines or processors). The main problem of scheduling is to find the best possible solution in an

acceptable time. This solution must fulfill user criteria, e.g. order of execution of tasks or demand to execute tasks on dedicated machines.

In job shop scheduling problem (JSSP) we consider a set of independent $n \in \mathbb{N}$ jobs $J = \{1, 2, \dots n\}$, and factory which has $m \in \mathbb{N}$ machines $M = \{1, 2, \dots m\}$. Each job is composed of operations $O = \{1, 2, \dots, m\}$. There is a sequence of operations in job j. The single o_{ji} operation of the job j must be executed on the machine i. This assignment needs $t_{ji} \in \mathbb{N}$ time units for completion. We assume that operation is performed on machine without interruption during execution. Operations can be run on m machines in a different order. The main purpose of the problem is to find the best solution, a schedule that consists of assigning all n jobs to m machines fulfilling the optimisation criterion(s). We accept only feasible solutions which must meet the following conditions:

- operations of each job must be assigned to an appropriate machine and executed completely,
- each operation must be executed by an uninterrupted time on a assigned machine,
- each machine must execute only one operation at a time,
- the precedence constraints must be respected [16].

The size of the search space (a number of schedules) $\mathbb{Z}$ is directly dependent on the number of jobs n and a number of machines m:

$$\mathbb{Z} = (n!)^m \tag{1}$$

For $n = 2$ jobs and $m = 4$ machines we have $(2!)^4 = 16$ possible solutions (schedules). The number of solutions grows drastically even as the number of machines or jobs increases slightly. When an instance consists $n = 5$ jobs and $m = 5$ machines, a number of solutions come up to 207 360 000.

A solution can be represented as a Gantt chart. Below (see Fig. 1) an instance of JSSP with $n = 4$ jobs and $m = 4$ machines is presented on the Gantt chart.

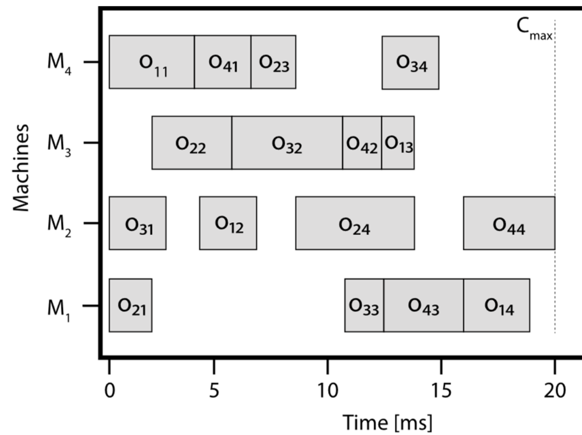

Figure 1. Gantt chart for an instance of JSSP with $n = 4$ jobs and $m = 4$ machines. Source: [16]

Let us suppose that operations are processed in the following order:

- job 1: $o_{11}, o_{12}, o_{13}, o_{14}$
- job 2: $o_{21}, o_{22}, o_{23}, o_{24}$
- job 3: $o_{31}, o_{32}, o_{33}, o_{34}$
- job 4: $o_{41}, o_{42}, o_{43}, o_{44}$

Operations must respect precedence constraints. For example, the operation o_{j1} must be processed before o_{j2} for the job j. We assume that machines have the same performance, which means that processing time of the given operation is equal on each machine. In this example, the operations need a given time (milliseconds) to be processed:

- $t_{11} = 3, t_{12} = 2, t_{13} = 1, t_{14} = 2$
- $t_{21} = 1, t_{22} = 3, t_{23} = 1, t_{24} = 5$
- $t_{31} = 2, t_{32} = 4, t_{33} = 1, t_{34} = 2$
- $t_{41} = 2, t_{42} = 1, t_{43} = 3, t_{44} = 3$

e.g. operation o_{11} needs $t_{11} = 3$ milliseconds to process by machine. As we can see (see Fig. 1), operations are assigned to machines. This assignment is feasible because we met all the conditions defined above. The objective of the JSSP is to find the correct permutation of all operations where time is minimized. In this case we minimize the makespan C_{max}.

Makespan is defined below:

$$C_{max} = \max_i(O(S_i)). \quad (2)$$

Let us denote by S as a schedule. By S_i we denote the schedule on machine M. The completion time of the operations on machine M_m in schedule S_i is denoted by $O(S_i)$. We consider minimizing the maximum completion time on each machine M_m across the system. In this example, the makespan is equal to 20 (see Fig. 1), because the completion time of the last executed operation occurred in machine M_2 – this is the maximal time of all machines' schedules.

1.2. Implementations of swarm algorithms in JSSP

The JSSP is extensively studied by many researchers. This problem is also considered in the categories of swarm algorithms. Pongchairerks and Kachitvichyanukul in their work [11] presented the particle swarm optimization (PSO) algorithm applied to JSSP. They considered an optimization algorithm for JSSP with multipurpose machines. This is a modification of the JSSP – operation has to be processed by exactly one machine from a set of machines. Lin and at. [7] also proposed PSO to solve JSSP. However, they used a different way of representing a particle. Instead of a simple conversion of values, they introduced several operators to modify and improve the solution.

FA was also used in the optimization of JSSP. In work [5] the authors used this algorithm. They encoded a firefly as a set of operations in the JSSP instance. Each operation was assigned a real value. After that they sort ascendingly, thus the permutation of operations changes. Finally, they checked the sequence of operations and repaired when the precedence constraints were violated. Some researchers use hybrid methods. An example of this approach is presented in [9]. The authors combined the Simulated Annealing and Firefly Algorithm for the JSSP. Simulated annealing was used to identify optimal and near-optimal makespans for the JSSP and FA as an optimization algorithm. The FA was used as well as for multi-objective JSSP. In [14] the authors considered the combined makespan, mean flow time, and tardiness objective for problems of various sizes. FA was likewise studied in different optimization problems. In the work [15] the authors analyze different modifications made by the researchers. They were concerned with parameter modification, modified search strategy, and change the solution space. They also analyzed and compared the performance of the standard and modified versions of the FA.

2. Firefly algorithm

The Firefly algorithm (FA) is a metaheuristic that is driven by the behavior of species in nature. FA is a swarm-based algorithm introduced by Yang [17]. In this algorithm, a swarm of fireflies communicate with each other by the light produced in a biochemical process called bioluminescence. Communication is used to attract other fireflies. The firefly is lighter, the attraction is higher. We must also take into account that the intensity of light decreases with distance [15].

In terms of optimization problem we assume that each firefly codes a solution in a search space. Fireflies are unisex, so the attractiveness of any firefly is not influenced in a different way. Each firefly moves through the search space toward a brighter firefly in the neighborhood. The brightness of the firefly is absorbed into the environment. The absorption is defined in Eq. 3:

$$\beta = \beta_0 e^{-\gamma r_{ij}^2}, \tag{3}$$

where:

- β_0 – attractiveness of the firefly at $r = 0$,
- γ – light absorption coefficient,
- r_{ij} – distance between firefly i and firefly j.

Each firefly updates its own position in space. Let us assume that firefly i headed to firefly j. The update of i-th firefly is defined as follow (Eq. 4):

$$x_i^{t+1} = x_i^t + \beta_0 e^{-\gamma r_{ij}^2}\left(x_j^t - x_i^t\right) + \alpha_t \varepsilon_i^t, \tag{4}$$

where:

- x_i^{t+1} – next position of i-th firefly,
- x_i^t , x_j^t – actual position of i-th firefly and j-th firefly accordingly,
- α_t – randomness strength,
- ε_i^t – a random vector.

Fireflies are updated until a termination criterion is met. Usually, this criterion is determined by maximum number of iterations $ITER$. Below the algorithm is presented.

Listing 1. The pseudocode of the Firefly Algorithm. Source: [17]

```
Set parameters of the algorithm: α_t, β_0, γ, N - a number of fireflies,
ITER - maximum number of iterations
Randomly generate N fireflies in search space
For t = 1 to ITER
   Compute the brightness I of the firefly
   Sort fireflies by their brightness
   For i = 1 to n-1
        For j = i+1 to n
         If I_j > I_i
            Move the firefly i in the direction of firefly j
         End if
        End for
   End for
End for
Calculate the best firefly and display the solution
```

In the first step of the algorithm (see Listing 1) initial parameters α_t, β_0, γ, N, and $ITER$ are set. Afterwards the algorithm randomly set of N fireflies. From this moment on, the algorithm computes the brightness of each firefly and compares its brightness. The firefly with less brightness is moved to the firefly with higher brightness in their neighborhood. The algorithm is terminated when the maximum number of iterations is achieved. Then the best solution is presented to the user.

3. Proposed solution

The original form of FA uses continuous values to code the solution. In JSSP we consider discrete values. The main issue is to find the proper method to represent the individual (particle in PSO, firefly in FA) that can represent a JSSP schedule. In work [10] the authors compared three methods to representation particles in PSO applied to JSSP:

- *Operation and Particle Position Sequence* – the particle position is joined with the position sequence in JSSP. When the positions of the particles are being sorted, the sequence of operations changes accordingly. In the next step, a new sequence of

positions is assigned to the machines according to precedence constraints in JSSP [8].

- *Random Keys Representation* - in this method we also sort particles' positions (ascending order). Then, the new order of particles is joined with a sequence of jobs. In the next step, we bring back the original order of particles, which also the sequence of jobs changes accordingly. At the end, a new sequence of positions is assigned to the machines according to precedence constraints in JSSP [11].
- *Multiple-type individual enhancement scheme* – this scheme is composed of swapping operation, inversion operation, and long-distance movement operation. These operations are applied to the representation of real numbers. The authors compare the makespan obtained before the selected scheme and that obtained after the selected scheme. If the selected scheme is better, they update the real vector of the individual by the selected operation scheme [7].

In our approach we used the form of random-key (RK) encoding. A vector in RK consists of real numbers. A firefly represented by real values can act out an operation permutation expressed by integer values. For n jobs on m machines, the firefly consists of a vector composed of $n \times m$ dimensions, thus the firefly is represented by $\{r_1, r_2, \dots, r_j\}$, where $1 \leq j \leq n \times m$. Single r_j corresponds to the job operation order of the job in a schedule.

In RK each real value has assigned an integer number $\{\pi_1, \pi_2, \dots, \pi_k\}$, where $1 \leq k \leq n \times m$, which π_k represents indirectly an operation order of a job. After that, the real values are sorted in ascending order. The assigned integer numbers change their own position in an RK vector. We assume that each job must consist of m operations – individual job must be processed by every machine in a set of M. Therefore, we need a transformation that could convert integer values $\{\pi_1, \pi_2, \dots, \pi_k\}$ to job indexes. To do this, we use the following formula (Eq. 5):

$$\text{job index} = (\pi_k \bmod n) + 1. \tag{5}$$

Using this formula the integer values $\{\pi_1, \pi_2, \dots, \pi_k\}$ are transformed into an operation order sequence $\{\gamma_1, \gamma_2, \dots, \gamma_k\}$, where $1 \leq k \leq n \times m$. Single γ_k represents a job index, $1 \leq \gamma_k \leq n$. Next, when we search values from γ_1 to γ_k we could find $i-$th occurrences of each job index.

Let us see an example of RK encoding. To encode the JSSP solution (schedule) presented in Fig. 1 we need 16 positions in each firefly ($n = 4$ jobs * $m = 4$ machines). Suppose that the firefly is represented by the following values: [8.4, 1.5, 2.8, 0.2, 2.0, 0.9, 3.5, 1.3, 4.9, 0.8, 6.0, 0.5, 2.2, 3.6, 6.3, 4.6] (see Fig. 2). Now we need to assign integer values $\{1,2, \dots, 16\}$ to values

in RK vector starting from the smallest value. In our example, the smallest value 0.2 has 4th element of the RK vector. For this position, we assign an integer value equal to 1. The next value (bigger than 0.2) is 0.5 (at position 12). We set the next integer value equal to 2. We continue this process until all integer values are assigned.

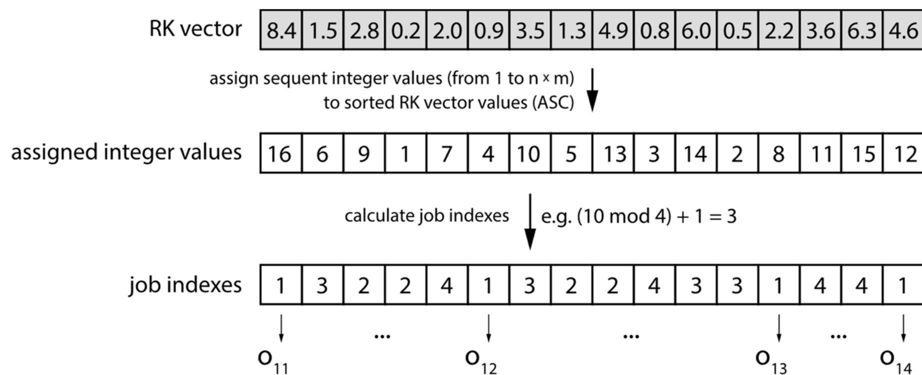

Figure 2. RK encoding for the schedule presented in Fig. 1. Source: own study

In the next step, we calculate job indexes. We use a formula from Eq. 5. Integers 4, 8, 12, and 16 indicate the operations belonging to job 1, because (4 mod 4) + 1 = 1, (8 mod 4) + 1 = 1, etc. The solution will always be feasible because the operation order will never violate the precedence constraints. These constraints are attributed after RK encoding. In this example, the operation sequence is $[o_{11}, o_{31}, o_{21}, o_{22}, o_{41}, o_{12}, o_{32}, o_{23}, o_{24}, o_{42}, o_{33}, o_{34}, o_{13}, o_{43}, o_{44}, o_{14}]$. From this moment on, operations can be assigned to machines according to the conditions described in Chapter 1.

4. Experimental results

The FA algorithm was implemented in Java language with the support of the *aitoa* library developed by Thomas Weise. This library is described in [16]. The experiments were carried out on the Dell Precision 7920 Tower workstation equipped with Intel Xeon Gold 6242 CPU 2.80GHz (16 cores, 32 threads), 64 GB RAM, GeForce RTX 3090, Windows 10 Pro for Workstations 21H1.

The code of the *aitoa* library was designed as a versatile and general implementation of metaheuristics in Java and provides the JSSP. An objective function, a search, and a solution space, as well as a mapping in between them, and search operators, can be composed and provided to a black-box optimization algorithm. They are encapsulated in an `IBlackBoxProcess` instance that can automatically remember the best solution and create comprehensive log files during an experiment run.

JSSP instances were written in the following scheme (see Listing 2):

Listing 2. The coding scheme of the example JSSP instance. Source: [16]

```
number of jobs      number of machines     index of machine

         +++++++++++++++++++++++++++
         4 5
Job 0:   1 10 2 20 3 20 4 40 5 10   <-  processing time
Job 1:   2 20 1 10 4 30 3 50 5 30       in milliseconds for
Job 2:   3 30 2 20 5 12 4 40 1 10       a given operation
Job 3:   5 50 4 30 3 15 1 20 2 15
         +++++++++++++++++++++++++++
```

where in the second line, the number $n = 4$ of jobs is specified, followed by the number of $m = 5$ machines. The following lines describe jobs and operations. Each operation is specified as a pair of two numbers: machine index and the number of time units (milliseconds) used for the process of the operation.

The `JSSPCandidateSolution` class provided by the library was used as a solution. To evaluate the solution, we implemented `JSSPMakespanObjectiveFunction`. In the *aitoa* library, there is a set of search operators. The zero-argument operator (`FireflyNullaryOperator`) is used to create a random population. The one-argument operator (`FireflyUnaryOperator`) modifies the best individual based on his current

position. The principal two-argument operator (`FireflyBinaryOperator`) was used to modify fireflies based on the position of a brighter firefly and their own.

Our model was tested on selected instances from the OR-library [2] and other libraries [18, 19] as test benchmarks. The original data of these instances come from:

- abz5-abz9 [1],
- dmu01-dmu80 [3],
- ft06-ft20 [4],
- la01-la40 [6],
- swv01-swv20 [12],
- ta01-ta80 [13].

We experimentally set the FA parameters, including the number of fireflies (n), the number of iterations (ITER), the light absorption coefficient (γ), the randomization parameter (α) and the attractiveness parameter (β_0). We tested each parameter in the range of 0.0 to 1.0 (for γ, α and β_0) and amount of fireflies in range 10 – 50. We assumed 1000 iterations of the algorithm for all experiments.

Each instance of the experiment was repeated 50 times for the assigned value of the parameter. After that we compared the best and averaged results and set the parameter value for optimal results. In the Tab. 1 the optimal values of parameters are given.

Table 1. Optimal FA parameters set in the experiments. Source: own study

Instance	Amount of fireflies	Number of iterations (ITER)	Light absorption coefficient (γ)	Randomization parameter (α)	Attractiveness parameter (β_0)
abz5	40	1000	1.0	0.6	1.0

In the next part of the experiments, we used JSSP test instances. These instances contain various number of jobs (from 6 to 100 jobs) and number of machines (from 5 to 20 machines). The most complex cases come from [13]. In the Tab. 2 we show results for selected instances used in the experiments.

Table 2. Results for selected instances derived from the OR-library and other libraries used in the experiments. Source: own study

Instance	Number of jobs (n)	Number of machines (m)	The best result C_{max} [ms]	Averaged result C_{max} [ms]	Lower bound C_{max} [ms]
abz5	10	10	1418	1488	1234
abz6	10	10	1079	1135	943
abz7	20	15	959	1006	656
abz8	20	15	975	1041	648
dmu01	20	15	4009	4148	2501
dmu02	20	15	4141	4297	2651
dmu06	20	20	5141	5304	3042
dmu10	20	20	4644	4923	2858
dmu16	30	20	6342	6601	3734
ft06	6	6	55	57	55
ft10	10	10	1180	1232	930
ft20	20	5	1485	1554	1165
la01	10	5	674	713	666
la02	10	5	719	755	655
la08	15	5	914	958	863
la22	15	10	1257	1305	927
la28	20	10	1665	1736	1216
swv01	20	10	2213	2284	1407
swv06	20	15	2778	2868	1630
swv11	50	10	5154	5378	2983
ta50	50	15	3219	3414	2723
ta72	100	20	7880	8164	5181

As we can see, the FA can solve both simple and most complex instances. *Lower bound* column in the Tab. 2 (the last column) shows the optimal of the makespan C_{max} values for each instance. FA gives only suboptimal results for these instances. For instances with small amount of jobs and machines (e.g. instance *ft06*, *la01*, *la02*) the results are close to optimal values. When the number of jobs and the number of machines increases, the algorithm tries to find suboptimal solutions. Even for the biggest instance (see instance *ta72* in the Tab. 2) algorithm found a relatively good schedule (7880 ms) vs. optimal (5181 ms).

As we mentioned, the obtaining optimal results is difficult for most metaheuristics due to complex search space and the NP-hard nature of JSSP. Finding an optimal solution by evolutionary algorithms has been the subject of many works. Our work confirmed that FA is useful in the JSSP problem. This task was supported by the *aitoa* library. This library lets us focus on the algorithm rather than on implementing the whole JSSP scheduler. In our opinion, this library is very useful in terms of scheduling problems.

5. Conclusions

The FA is one of the many known metaheuristics. Our task was focused on implementation of the FA in the *aitoa* library and providing test results. The implementation was successful. This library is useful for any category of metaheuristics. The structure of the library is universal. It can represent each form of the operators used in the metaheuristic and make it simpler to implement the JSSP.

We plan to modify the FA algorithm to improve the results. It can be done by modifying the standard operators in the FA. We can also modify the parameters of the FA by modification of the formulas. These parameters would be the subject of another metaheuristic. This metaheuristic could be a change of the parameters during scheduling. This hybrid solution could be more fitting to the JSSP problem.

References

1. Adams J., Balas E., Zawack D., The Shifting Bottleneck Procedure for Job Shop Scheduling. Management Science, 34(3), pp. 391-401, 1988. DOI: 10.1287/mnsc.34.3.391

2. Beasley, J.E., OR-Library: Distributing Test Problems by Electronic Mail. Journal of the Operations Research Society. 41, 11, pp. 1069-1072, 1990. DOI: 10.2307/2582903

3. Demirkol E., Mehta S.V., Uzsoy R., Benchmarks for Shop Scheduling Problems. European Journal of Operational Research (EJOR), 109(1), pp. 137-141, 1988. DOI: 10.1016/S0377-2217(97)00019-2

4. Fisher H., Thompson G.L., Probabilistic Learning Combinations of Local Job-Shop Scheduling Rules. In Muth JF, Thompson GL (eds.), Industrial Scheduling, pp. 225-251, 1963.

5. Khadwilard A., Chansombat S., Thepphakorn T., Thapatsuwan P., Thapatsuwan W., Pongcharoen P., Application of Firefly Algorithm and Its Parameter Setting for Job Shop Scheduling. The Journal of Industrial Technology, 2012.

6. Lawrence S.R., Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques (Supplement). PhD thesis, Graduate School of Industrial Administration (GSIA), 1984. Carnegie-Mellon University, Pittsburgh, PA, USA.

7. Lin T.L., Horng S.J., Kao T.W., Chen Y.H., Run R.S., Chen R.J., Lai J.L., Kuo I.H., An efficient job-shop scheduling algorithm based on particle swarm optimization. Expert Syst. Appl. 37 pp. 2629–36, 2010

8. Liu Z., Investigation of Particle Swarm Optimization for Job Shop Scheduling Problem 3rd Int. Conf. Nat. Comput. (ICNC 2007) vol 3 (Haikou: IEEE) pp. 799–803, 2007

9. Miller T.J., Steinhöfel K., Veenstra P., Firefly-inspired Algorithm for Job Shop Scheduling, 2018. DOI: 10.1007/978-3-319-98355-4_24

10. Nurul I.A., Adi S., Performance evaluation of different types of particle representation procedures of Particle Swarm Optimization in Job-shop Scheduling Problems IOP Conf. Ser.: Mater. Sci. Eng. 114, 2016

11. Pongchairerks P., Kachitvichyanukul V., A Particle Swarm Optimization algorithm on Job-Shop Scheduling Problems with multi-purpose machines Asia-Pacific J. Oper. Res. 26, pp. 161–84, 2009.

12. Storer R.H., Wu S.D., Vaccari R., New Search Spaces for Sequencing Problems with Application to Job Shop Scheduling. Management Science, 38(10), pp. 1495-1509, 1992. DOI: 10.1287/mnsc.38.10.1495

13. Taillard É.D., Benchmarks for Basic Scheduling Problems. European Journal of Operational Research (EJOR), 64(2), pp. 278-285, 1993. DOI: 10.1016/0377-2217(93)90182-M

14. Udaiyakumar K., Chandrasekaran, M., Optimization of Multi Objective Job Shop Scheduling Problems Using Firefly Algorithm. Applied Mechanics and Materials. 591, pp. 157-162, 2014. DOI: 10.4028/www.scientific.net/AMM.591.157.

15. Waqar A.K., Nawaf N.H., Surafel L.T., Jean M.T.N., A Review and Comparative Study of Firefly Algorithm and its Modified Versions, Optimization Algorithms – Methods and Applications, InTech, pp. 281-313, 2016. DOI: 10.5772/62472

16. Weise T., An Introduction to Optimization Algorithms. Hefei, Anhui, China: Institute of Applied Optimization (IAO), School of Artificial Intelligence and Big Data, Hefei University, 2018-2019. Available at: http://thomasweise.github.io/aitoa/

17. Yang X.S., Nature-inspired metaheuristic algorithms. Luniver Press, UK, 2010.

Internet sources:

18. Oleg V. Shylo's page (http://optimizizer.com/DMU.php), accessed 13.11.2021

19. Éric Taillard's page (http://mistic.heig-vd.ch/taillard/problemes.dir/ordonnancement.dir/ordonnancement.html), accessed 13.11.2021