Submitted to *Operations Research*
manuscript (Please, provide the manuscript number!)

Balancing the average weighted completion times in a scheduling problem with two classes of jobs: a genetic algorithm-based heuristics.

Matteo Avolio
Department of Mathematics and Computer Science, University of Calabria, Rende (CS), Italy, matteo.avolio@unical.it

Balancing the average weighted completion times among two classes of jobs (BAWCT) could be interpreted as a cooperative multiagent scheduling problem. BAWCT was introduced in Avolio and Fuduli (2020) and it was explored in [2] where the authors proved its NP-hardness, providing a Lagrangian relaxation based approach for solving it heuristically. Since their approach requires to solve, at each iteration, a Linear Programming problem that it's not a very difficult task but however requires some computational time, in this work we propose a genetic algorithm-based heuristics to speed up the resolution process.
Numerical results are presented on the same datasets used in literature to empirically show the efficiency of the proposed approach.

Key words: Scheduling; single-machine; multiagent; genetic algorithm
History:

1. Introduction

Allocating resources to tasks over given time periods with the aim of optimizing one or more objectives is what the theory of scheduling focus on. Scheduling, as a decision making process, plays a very important role in a lot of contexts such as manufacturing, production, transportation, and distribution (see as references Pinedo (2009, 2016)). Unfortunately solving this kind of problems often is not easy and, however, it depends on the assumptions made on the problems themselves.
In the first years of the century, with publication of Agnetis et al. (2004) and Baker and Smith (2003), a new scheduling model was introduced to take into account the presence of two or more agents. In multiagent scheduling problems (see Agnetis et al. (2014)) each agent has its own set of tasks to be processed and its own objective function to be optimized, then the difficulty of this kind of problems arises from the fact that the agents share the processing resources.
Agnetis et al. (2004), in a two-agent context, addresses the problem by considering as objective functions to optimize the maximum of regular functions (associated with each job), number of late jobs, and total weighted completion times. Additionally, the authors studied the complexity of each problem with respect to the different objective function and to the different structure of the processing system (single machine or shop).
Baker and Smith (2003), instead, in a multiagent context, faces the problem of minimizing three different objective functions: makespan, maximum lateness, and total weighted completion time. From a complexity view point, the authors proved the problem to be polynomial solvable according to any one of the three mentioned criteria but, additionally, they shown that when minimizing a mix of them, the problem becomes NP-hard.
The multiagent assumptions find also application in a lot of practical contexts. For example, Peha (1995) addresses the problem of minimizing the weighted number of tardy jobs in a real-time systems and integrated-services networks, while Arbib et al. (2004), in a telecommunication system with two users, focuses on the maximization of on-time data packets transmitted to one user, while guaranteeing a certain amount of on-time data packets to the other.
The majority of multiagent scheduling problems is of competitive type since agents purely compete with each other to allocate resources

to their tasks that only contribute to their own objective function.

In this work, instead, we tackle a scheduling problem which could be interpreted as a two agents problem of cooperative type since each job contributes to the same objective function aimed at balancing the average weighted completion times of the two agents.

This problem finds application in a lot of scenarios characterized by a decision maker who, for different reasons (e.g. economic, of efficiency, of reliability), wishes to balance the average completion times of two different classes of jobs. Let's consider, for example, a transportation firm using a drone for delivering packages and suppose that the drone is able to deliver only one package at time, i.e. it has to come back to the depot after each delivery in order to take the next package. Suppose, finally, that the transportation firm has to stock two different companies. In this scenario, the processing times represent the shipping times whose depend on the depot-company distance (fixed) and on the load of the corresponding cargo (variable), the latter acting on the speed of the drone, while the weight associated with each job could be, for example, the urgency specified by a company for that package.

From the point of view of the transportation firm, it's reasonable to schedule deliveries in order to balance the average weighted time of the two companies.

This problem was introduced in Avolio and Fuduli (2020) where the authors considered its basic version, by assuming all the jobs having the same processing time and unitary weight. In [speriamo] the problem was generalized by considering all the jobs having different processing time and weight. Unfortunately, for the generalized version of the problem, the authors proved its NP-hardness, providing a Lagrangian relaxation based algorithm to solve it heuristically. Since their approach requires, at each iteration, to solve a Linear Programming problem that it's not too hard but it requires some computational time, in this work we propose a genetic algorithm based heuristics to speed up the resolution process.

This work is organized as follows. In the next section we formally state the problem, reporting a nonsmooth formulation as a variant of the quadratic assignment problem, and we provide the state-of-the-art results for it. In Section 3 we propose a genetic algorithm based approach to heuristically solve the problem, describing in detail all the phases of our approach. In Section 4 we present some numerical results obtained on the same datasets considered in literature to empirically show the efficiency of the proposed algorithm and we conclude with Section 6 in which some conclusions are drawn.

2. Problem Definition and Literature Review

Let A and B be two different classes of jobs with n_A and n_B being their cardinalities. Let $J_A = \{1, \ldots, n_A\}$ and $J_B = \{n_A + 1, \ldots, n_A + n_B\}$ be the index sets of A and B, respectively. For each job $j \in J_A \cup J_B$, let p_j be its processing time and w_j its weight.

The problem of balancing the average weighted completion times (BAWCT)[2] of class A and B can be stated as follows:

$$\min \left| \frac{\sum_{j \in J_A} C_j w_j}{n_A} - \frac{\sum_{j \in J_B} C_j w_j}{n_B} \right|, \qquad (1)$$

where C_j represents the completion time of job j. From a mathematical programming point of view, in order to formulate BAWCT as an optimization problem, we proceed as follows. We define the following decision variables:

$$x_{jt} \triangleq \begin{cases} 1 & \text{if job } j \text{ is assigned to position } t \\ 0 & \text{otherwise,} \end{cases}$$

for $j \in J_A \cup J_B$ and $t = 1 \ldots, n_A + n_B$.

Taking into account that the completion time of a job j scheduled in position t is

$$p_j + \sum_{l \in J_A \cup J_B} \sum_{k=1}^{t-1} p_l x_{lk},$$

problem (1) can be formulated as follows:

$$\begin{cases} \min\limits_{x} \left| \dfrac{1}{n_A} \left(\sum\limits_{j\in J_A} \sum\limits_{t=1}^{n} w_j \left[p_j + \sum\limits_{l\in J} \sum\limits_{k=1}^{t-1} p_l x_{lk} \right] x_{jt} \right) \right. \\ \left. - \dfrac{1}{n_B} \left(\sum\limits_{j\in J_B} \sum\limits_{t=1}^{n} w_j \left[p_j + \sum\limits_{l\in J} \sum\limits_{k=1}^{t-1} p_l x_{lk} \right] x_{jt} \right) \right| \\ \sum\limits_{t=1}^{n} x_{jt} = 1 \quad j \in J \\ \sum\limits_{j\in J} x_{jt} = 1 \quad t = 1 \dots n_A + n_B \\ x_{jt} \in \{0,1\} \quad j \in J, \quad t = 1 \dots n_A + n_B, \end{cases} \tag{2}$$

where the constraints are the classical assignment constraints, which impose a bijection between jobs and positions. Problem (2) is a non-smooth integer optimization problem and represents a sort of quadratic assignment problem due to the quadratic terms in the objective function. In [2] the authors, after having proved its NP-hardness, provided an integer linearization of (2) based on the well known Glover Linearization and then they heuristically solved the linearized version using a Lagrangian relaxation based approach. The limit of their technique, even if the obtained results are valuable, consists on the fact that, at each iteration, a Linear Programming problem has to be solved. It's well known that it can be done in polynomial time but, when the size of the problem increases (i.e. n_A and n_B become very large), its resolution has a not negligible impact on the overall computational time. For this reason, in the next section, we propose a genetic algorithm-based heuristics that, from the empirical results, has proved to be able to speed up the resolution process.
It's worth noting that, in Avolio and Fuduli (2020), the authors tackled a simplified version of BAWCT in which they assumed, for $j = 1, \dots, n_A + n_B$, $w_j = 1$ and $p_j = p$. They proved that, by mean of these assumptions, the problem reduces to a particular instance of the well known subset sum and it becomes solvable in linear time, or constant time if the job-position assignment is not explicitly considered.

3. A Genetic Algorithm Heuristics

Genetic Algorithms (GAs) are heuristic search approaches successfully applied to many NP-hard optimization problems (see Kramer (2017)). GAs follow the evolution paradigm, i.e. starting from an initial population, they apply genetic operators in order to produce offsprings, trying to make them fitter than their ancestors, hopefully increasing the overall fitness of the population from one generation to another.
In GAs setting, a genotype representing a possible solution to the optimization problem and a fitness value representing its "goodness" correspond to each individual ; this is the reason why, from one generation to another, GAs wish to generate new individuals with higher fitness values.
The basic schema of GAs is reported in Algorithm 1.
Even if in literature there are plenty of sugges-

Algorithm 1: Genetic Algorithm

```
initialize population
while not end conditions do
    while new population uncomplete do
        selection
        crossover
        mutation
    end
    population replacement
end
```

tions about how to implement all the phases of Algorithm 1, in general each of them can be adapted and tailored to the problem dealing with; this flexibility makes GAs attractive for many optimization problems in practice. For example in Potvin (1996), Larranaga et al. (1999), Deep and Mebrahtu (2011) the well-known traveling salesman problem is faced, while in Murata et al. (1996), Pezzella et al. (2008) the authors, in a scheduling context, applied this technique to flow-shop and job-shop problems with the aim of minimizing the makespan. Before to describe in detail how we implemented each phase of Algorithm 1, its worth to specify some additional steps we performed. In particular, after having mutated

the current child, we apply a local search to improve its fitness value. Additionally, at each generation, we completely replace the old population with the new one, just mantaining the fittest current individual in order to have a monotonic behaviour of the overall fitness. These steps are used quite always in GAs in order to have a faster convergence to optimal solutions.
Finally, as exit conditions, we used a time limit of 1800 seconds like in [2]. Of course, we stop the execution also if a null upper bound is found since, by the definition of (1), it trivially corresponds to an optimal solution for BAWCT.

3.1. Encoding and Fitness Evaluation

The first step in the implementation of a GA regards the choice of a suitable encoding for solutions. Since we are dealing with a scheduling problem, the standard way to represent a solution is an ordered vector of values.
Given a solution S, we encode it in a vector V_S of size $n_A + n_B$ such that $V_S[t] = j$ if and only if job j is scheduled in position t within the solution S. Then, from now on, when the referenced solution is clear, we denote by $[t]$ the job processed in position t.
Example. Let's assume $n_A = 1$ and $n_B = 3$.
The following vector of size $n_A + n_B = 4$

3	1	4	2

encodes the solution in which job 1, belonging to class A, is scheduled in second position, while jobs 2, 3 and 4, belonging to class B, are scheduled in fourth, first, and third position, respectively. Then, following our notation, we have $[1] = 3$, $[2] = 1$, $[3] = 4$, and $[4] = 2$.
Concerning with fitness evaluation, since individuals with higher fitness are preferred by GAs but we are dealing with a minimization problem, we proceed as follows.
Let S be a solution for BAWCT. We compute $f(S)$, i.e. its corresponding objective function value, by using (1) and then we set $fitness_S = \frac{1}{f(S)}$.
In this way, when $f(S) \to 0$ (that is a trivial lower bound for (1)) follows that $fitness_S \to \infty$.

3.2. Initial population

Once the encoding strategy has been defined, the next step to be performed regards the generation of the initial population. A review of the main approaches used in this phase of GAs is done in Kazimipour et al. (2014).
In our approach, we tried three different generation techniques: *Random*, *Alternated*, and *Bidirectional*. Since the first strategy works randomly and we risk to create an initial population of completely unbalanced schedules, we defined the other two greedy techniques, in order to provide possibly the algorithm with a better starting point.
Let $N = \{1, \dots, n_A + n_B\}$, $N_A = \{1, \dots, n_A\}$, and $N_B = \{n_A + 1, \dots, n_A + n_B\}$. The three techniques we used can be described as follows:

- *Random generation.* The genotype of each individual is given by a random permutation of N trivially representing a feasible solution for BAWCT. Formally, let $P(N)$ be a random permutation of N, we set $[i] = P(N)_i$ for $i = 1, \dots, n_A + n_B$.
- *Alternated generation.* The genotype of each individual is given by alternating jobs of the two classes, until one of the two sets is completely scheduled. Afterwards, potentially remaining jobs are accommodated at the end of the sequence.

Let $P(N_A)$ and $P(N_B)$ be two permutations of N_A and N_B respectively. Then, for each $i = 1, \dots, \min\{n_A, n_B\}$, we set

$$[2i-1] = P(N_A)_i \text{ and } [2i] = P(N_B)_i.$$

Finally, if $n_A \neq n_B$, the remaining jobs for the biggest class are inserted at the end of the sequence.

- *Bidirectional generation.* This technique is inspired by the exact algorithm proposed in Avolio and Fuduli (2020) to solve the simplified version of BAWCT. The genotype of each individual is given by scheduling, at each step of the process, two jobs for one of the two classes, inserting one of them in the first remaining available position, while scheduling the other in the last one.

Let $P(N_A)$ and $P(N_B)$ be two permutations of N_A and N_B, respectively. At each step i we set either

$$[i] = P(N_C)_{t_C} \text{ and } [n_A + n_B - i] = P(N_C)_{t_C+1} \tag{3}$$

or

$$[i] = P(N_C)_{t_C+1} \text{ and } [n_A + n_B - i] = P(N_C)_{t_C}, \tag{4}$$

where the class C is swapped between A and B at each step, while the index t_C, initialized to 1, is increased by 2 after that two jobs of class C are scheduled. Following a greedy approach, among (3) and (4), the assignment currently minimizing (1) is chosen.
It's worth to specify that, if during the process one of the two classes is completely scheduled, C remains fixed to the other class. Also, eventually remaining jobs due to the **oddness** of n_A and n_B, are scheduled in the middle of the sequence in a greedy way taking into account (1).
It's worth noting that, for all the proposed techniques, the probability of getting two times the same schedule as output of the generation process is very low, since they are defined in function of permutations. This is crucial, in particular for the last two techniques, since it guarantees a *diversification* in the initial population, even if the same greedy procedure is used for generating each individual.

3.3. Selection

Selection (see for example Yadav and Sohal (2017)) is a crucial step in designing GAs since it defines the chance of a given individual to participate in the reproduction process. Since a convergence to optimal solutions is desired, it's recommended to give an higher chance to individuals having high fitness values.
In this work, we tested three different well-known selection techniques: *Roulette wheel*, *Binary tournment*, and *k-tournment*.

- *Roulette wheel*. Following this technique, the selection is performed by simulating a roulette wheel in which each individual has a chance to be selected for reproduction represented by a portion of the roulette. Since the size of this portion depends on its fitness value, each individual I has a probability p_I to be selected that is directly proportional to its fitness. In particular, letting P be the population of individuals, we set

$$p_I = \frac{fitness_I}{\sum_{i \in P} fitness_i}.$$

- *Binary tournament*. From two randomly selected individuals, the one having highest fitness value is chosen for reproduction.
- *k-tournament*. It's the same as the previous with the only difference that k individuals are involved in the tournament.

3.4. Crossover

Once parents are selected, they have to reproduce in order to generate new children. The reproduction process is simulated using crossover operators, aimed at creating a new individual whose features depend on both the parents by mixing their genetic properties. From an algorithmic point of view, this phase is crucial since it allows to navigate the solution space.
In literature a lot of crossover operators were proposed for scheduling problems and, in general, for sequencing problems like the well-known traveling salesman problem (for a general overview see Umbarkar and Sheth (2015)). However, in Murata et al. (1996), the authors noticed that crossover operators mainly proposed for traveling salesman problems don't perform very well in a scheduling context. Then, on the basis of their results, we implemented the following crossover operators: *One-point crossover*, *Two-point crossover Ver. I*, *Two-point crossover Ver. II*, *Position based crossover*, and, additionally, we defined a new crossover operator named *k-AlternateCrossover*. (which it turned out to be very effective o eventualmente toglierlo(?)) (inserire figure?)
Given two parents I_1 and I_2, the previous crossover operators can be described as follows:

- *One-point crossover*. One point $i \in [1, n_A + n_B]$ is randomly selected. Then, with the same probability, either the first i jobs or the last i ones are inherited from I_1, while the remaining $n_A + n_B - i$ are inserted in the sequence preserving the order of appearance they have in I_2.
- *Two-point crossover Ver. I*. Two points $i, j \in [1, n_A + n_B]$ are randomly selected. Suppose $i \leq j$, the first i jobs and the last $n_A + n_B - j$ ones are inherited from I_1, while the remaining $j - i$, exactly as in one-point crossover, are inserted in the sequence preserving the order they have in I_2.

• *Two-point crossover Ver. II.* Two points $i, j \in [1, n_A + n_B]$ are randomly selected. Suppose $i \leq j$, from position i to j the jobs are inherited from I_1, while the remaining jobs, following the same policy of one-point crossover, are inserted preserving the order they have in I_2.

• *Position based crossover.* First of all, a number of positions $k \in [1, n_A + n_B]$ is randomly selected, then k different values v_i, $i = 1, \ldots, k$, are sampled in the interval $[1, n_A + n_B]$. At this point the jobs in positions v_i, for $i = 1, \ldots, k$, are inherited from I_1, while the others, following the same idea of one-point crossover, are inserted preserving the order they have in I_2.

3.5. Mutation

Another important operator in GAs is mutation, strictly related to crossover since, even mutation, it allows to navigate the solution space. In particular, mutation operators perturb the current solution by applying random changes. Inspired by Murata et al. (1996), in this work we tested the following four mutation operators:

• *Adjacent two-job change.* A position $i \in [1, n_A + n_B - 1]$ is randomly selected and jobs in position i and $i + 1$ are swapped.

• *Arbitrary two-job change.* Two different positions $i, j \in [1, n_A + n_B]$ are randomly selected and jobs in position i and j are swapped.

• *Arbitrary three-job change.* Three different positions $i, j, k \in [1, n_A + n_B]$ are randomly selected and the corresponding jobs are swapped by setting at the same time: $[j] = [i]$, $[k] = [j]$, and $[i] = [k]$.

• *Shift change.* Two different positions $i, j \in [1, n_A + n_B]$ are randomly selected and job in position i is moved in position j by shifting all the intermediate jobs.

Additionaly, by considering a set of adjacent jobs as an (atomic) *batch*, we defined other two mutation operators:

• *Adjacent batch change.* A position $i \in [1, n_A + n_B - 1]$ and a batch size $s \in [1, \lfloor \frac{n_A + n_B - i + 1}{2} \rfloor]$ are randomly selected. Then, for $k = 0, \ldots, s - 1$, jobs in position $i + k$ and $i + s + k$ are swapped.

• *Arbitrary batch change.* Two positions $i, j \in [1, n_A + n_B]$ are randomly selected. Suppose $i \leq j$, also the batch size $s \in [1, \min\{j - i, n_A + n_B + 1 - j\}]$ is randomly determined. Then, for $k = 0, \ldots, s - 1$, jobs in position $i + k$ and $j + k$ are swapped.

3.6. Local Search

As already specified in the introduction to our heuristics, in order to improve the fitness of new children, a local search (see for example Aarts et al. (2003)) is applied immediately after mutation. The introduction of local search in our approach turned out to be very effective in practice since, in a lot of scenarios, starting from a "good" solution produced by the previous phases, it's able to reach an optimal one lying in the neighborhood.

In particular, given an individual I representing a solution S, for each of the $\frac{(n_A + n_B)(n_A + n_B - 1)}{2}$ couples of indices (i, j), with $i < j$, we check if by swapping jobs in position i and j we obtain a decrease in the current value of (1). If this is the case, we execute the exchange, we update the current value of (1) and we iterate the process.

It's worth to specify that, in order to evaluate the variation of (1) for each possible exchange (i, j), it's not necessary to compute afresh its value, since it can be done by a simple procedure iterating in the sequence from position i to position j (see [2]). Even if the overall worst-case complexity for computing the variation of (1) is however linear ($\mathcal{O}(n_A + n_B)$) also in this case, avoid its recalculation has a not negligible impact on the overall performance.

Concluding, since the number of possible exchanges are quadratics, while the variation of the objective function can be calculated in linear time, the overall complexity of the local search approach is $\mathcal{O}((n_A + n_B)^3)$.

4. Numerical Results

The proposed algorithm has been implemented in Java (version 14.02) and run on a Windows 10 system, characterized by 16 GB of RAM and a 2.30 GHz Intel Core i7 processor. As in [2], for each run, we have fixed a time limit equal to 1800 seconds and a maximum number k_{max} of iterations equal to ??. Note that, except for the local search, each phase of the proposed approach can be implemented in linear time, making the overall complexity of the algorithm equal to $\mathcal{O}(k_{max}(n_A + n_B)^3)$.

It's well known that, in general, each GA, in addition to the choice of one among the proposed techniques for implementing each phase, needs also values for other three parameters: *population size*, *crossover probability*, and *mutation probability.*
Our idea was to take these values in the following domains:

- population size $P_s \in \{10, 25, 50\}$;
- crossover probability $P_c \in \{0.2, 0.4, 0.6, 0.8, 1\}$;
- mutation probability $P_m \in \{0.2, 0.4, 0.6, 0.8, 1\}$.

Due to the huge number of possible configurations, we have decided to split our experiments in two phases: in the first one we have done preliminary approximated experiments in order to estimate, in reasonable time, approximately the best configuration for our parameters, while in the second one we have tested our algorithm on the datasets proposed in literature, by setting the parameters on the values obtained in the previous phase.
Preliminary experiments revealed that a good setting for our algorithm is the following:

- population size: $P_s = 10$;
- crossover probability: $P_c = 1$;
- mutation probability: $P_m = 0.8$;
- initial generation: *Bidirectional*;
- selection: *Binary tournment*;
- crossover:
- mutation:
- local search: Yes.

In order to have a fair comparison with the state of the art results, after the algorithm has been tested on the same datasets used in [2] that mainly consist of three sets of problems, namely *Small*, *Medium*, and *Large*, and of a sets of more challenging instances.
For each set of problems, the processing times and the weights are all integers and they are drawn from the uniform distribution in the interval $[1, P_{max}]$, where $P_{max} = 25$ for the small problems, $P_{max} = 50$ for the medium problems, $P_{max} = 100$ for the large problems, and $P_{max} = 3(n_A + n_B)$ for the more challenging set of problems. Note that, when P_{max} grows, the chances of having two isomorphic jobs (same processing time and weight) decrease, reducing the number of optimal solutions, and making the overall search process harder. This motivates the hardness of the last set of problems.

For each set of problems, different scenarios have been defined by varying the pair (n_A, n_B), then, for each of that, 50 instances have been proposed.
For each scenario, in Table 1,2, 3 and 4, we report:

- the value of the pair (n_A, n_B).
- the average time required to find an optimal solution;
- the average time require by the algorithm proposed in [2] to find an optimal solution.

Numerical experiments shown the efficiency of our approach, that turned out to be always able to find an optimal solution within the time limit and within a maximum number of iterations.
It's worth to underline the importance of the *bidirectional generation* we introduced to produce the initial population. It turned out to be able to provide the algorithm with a good starting point, making the convergence towards an optimal solution easier. In order to show this result, in Table 5,6, 7 and 8, we report, for each scenario and for each generation technique, the average objective function of the initial population.
Note that, since P_{max} is fixed within each set of problems, we report it only once as caption of the corresponding table.

INSERIRE LE ITERAZIONI E FAR NOTARE CHE IL TEMPO MEDIO é MINORE RISPETTO ALL'ALTRO APPROCCIO O TOGLIERLE DEL TUTTO?

n	n_A	n_B	Exit(s)	Iter
20		10	0,028	19,1
30	10	20	0,017	6,4
40		30	0,028	5,6
40		20	0,028	5,7
50	20	30	0,08	9,6
60		40	0,068	3,7
60		30	0,067	3,6
70	30	40	0,19	7
80		50	0,32	7,7
80		40	0,091	2,2
90	40	50	0,748	11,6
100		60	0,347	4,4

Table 1 Small test problems, $P_{max} = 25$.

n	n_A	n_B	Exit (s)	Iter
100	50	50	0,416	5,5
150		100	0,863	3,6
200	100	100	1,705	2,8
250		150	6,121	4,4

Table 2 Medium test problems, $P_{max} = 50$.

n	n_A	n_B	Exit (s)	Iter
300	150	150	14,566	5,3
350		200	62,988	11,7
400	200	200	24,737	3,2
450		250	135,138	11,3
500	250	250	49,91	2,8

Table 3 Large test problems, $P_{max} = 100$.

5. Conclusions

In this paper, we have proposed a genetic algorithm for balancing the average weighted completion times of two classes of jobs in a single machine scheduling problem.
For implementing each phase, we have tested different techniques: some of them taken from the literature, others introduced by us in this work. Experimental results have shown the importance, for generating the initial population, of using our technique that we have named *bidirectional generation*.
Numerical results have also proved the efficiency of our algorithm that is able to speed up the resolution process with respect to the algorithm proposed in literature in [2].

References

Aarts E, Aarts EH, Lenstra JK (2003) *Local search in combinatorial optimization* (Princeton University Press).

Agnetis A, Billaut JC, Gawiejnowicz S, Pacciarelli D, Soukhal A (2014) *Multiagent scheduling: Models and algorithms* (Springer).

Agnetis A, de Pascale G, Pacciarelli D (2009) A lagrangian approach to single-machine scheduling problems with two competing agents. *Journal of Scheduling* 12(4):401–415.

Agnetis A, Mirchandani PB, Pacciarelli D, Pacifici A (2004) Scheduling problems with two competing agents. *Operations Research* 52(2):229–242.

Arbib C, Smriglio S, Servilio M (2004) A competitive scheduling problem and its relevance to UMTS channel assignment. *Networks* 44(2):132–141.

Avolio M, Fuduli A (2020) A subset-sum type formulation of a two-agent single-machine scheduling problem. *Information Processing Letters* 155:105886.

Baker K, Smith J (2003) A multiple-criterion model for machine scheduling. *Journal of Scheduling* 6(1):7–16.

Deep K, Mebrahtu H (2011) New variations of order crossover for travelling salesman problem. *International Journal of Combinatorial Optimization Problems and Informatics* 2(1):2–13.

Kazimipour B, Li X, Qin AK (2014) A review of population initialization techniques for evolutionary algorithms. *2014 IEEE Congress on Evolutionary Computation*, 2585–2592.

Kramer O (2017) *Genetic Algorithm Essentials*, volume 679 of *Studies in Computational Intelligence* (Springer).

Larranaga P, Kuijpers CMH, Murga RH, Inza I, Dizdarevic S (1999) Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review* 13(2):129–170.

Murata T, Ishibuchi H, Tanaka H (1996) Genetic algorithms for flowshop scheduling problems. *Computers & Industrial Engineering* 30(4):1061–1071.

Peha JM (1995) Heterogeneous-criteria scheduling: Minimizing weighted number of tardy jobs and weighted completion time. *Computers & Operations Research* 22:1089–1100.

Pezzella F, Morganti G, Ciaschetti G (2008) A genetic algorithm for the flexible job-shop scheduling problem. *Computers & Operations Research* 35(10):3202–3212.

Pinedo ML (2009) *Planning and Scheduling in Manufacturing and Services* (Springer-Verlag).

Pinedo ML (2016) *Scheduling: Theory, algorithms, and systems, fifth edition* (Springer International Publishing).

Potvin JY (1996) Genetic algorithms for the traveling salesman problem. *Annals of Operations Research* 63(3):337–370.

Umbarkar AJ, Sheth PD (2015) Crossover operators in genetic algorithms: a review. *ICTACT journal on soft computing* 6(1).

Yadav SL, Sohal A (2017) Comparative study of different selection techniques in genetic algorithm. *International Journal of Engineering, Science and Mathematics* 6(3):174–180.

$p \in [1, 3n]$	n	n_A	n_B	Time for Incumbent (s)	Iter	Absolute Error (%)	Solved to Opt
$p \in [1, 180]$	60	30	30	1,83	119,8	0	20/20
$p \in [1, 300]$	100	50	50	6,207	81,9	0	20/20
$p \in [1, 600]$	200	100	100	202,428	322,5	0	20/20
$p \in [1, 900]$	300	150	150	782,097	339,4	0,002	16/20
$p \in [1, 1200]$	400	200	200	1087,752	183	0,004	11/20
$p \in [1, 1500]$	500	250	250	960,657	91,5	0,008	2/20

Table 4 Test problems with large ranges

n	n_A	n_B	Exit(s)	Iter
20	10	10	$0,028$	$19,1$
30		20	$0,017$	$6,4$
40		30	$0,028$	$5,6$
40	20	20	$0,028$	$5,7$
50		30	$0,08$	$9,6$
60		40	$0,068$	$3,7$
60	30	30	$0,067$	$3,6$
70		40	$0,19$	7
80		50	$0,32$	$7,7$
80	40	40	$0,091$	$2,2$
90		50	$0,748$	$11,6$
100		60	$0,347$	$4,4$

Table 5 Small test problems, $P_{max} = 25$.

n	n_A	n_B	Exit (s)	Iter
100	50	50	$0,416$	$5,5$
150		100	$0,863$	$3,6$
200	100	100	$1,705$	$2,8$
250		150	$6,121$	$4,4$

Table 6 Medium test problems, $P_{max} = 50$.

n	n_A	n_B	Exit (s)	Iter
300	150	150	$14,566$	$5,3$
350		200	$62,988$	$11,7$
400	200	200	$24,737$	$3,2$
450		250	$135,138$	$11,3$
500	250	250	$49,91$	$2,8$

Table 7 Large test problems, $P_{max} = 100$.

$p \in [1, 3n]$	n	n_A	n_B	Time for Incumbent (s)	Iter	Absolute Error (%)	Solved to Opt
$p \in [1, 180]$	60	30	30	1,83	119,8	0	20/20
$p \in [1, 300]$	100	50	50	6,207	81,9	0	20/20
$p \in [1, 600]$	200	100	100	202,428	322,5	0	20/20
$p \in [1, 900]$	300	150	150	782,097	339,4	0,002	16/20
$p \in [1, 1200]$	400	200	200	1087,752	183	0,004	11/20
$p \in [1, 1500]$	500	250	250	960,657	91,5	0,008	2/20

Table 8 Test problems with large ranges