Order Acceptance and Scheduling with Sequence-Dependent Setup Times
A New Memetic Algorithm and Benchmark of the State of the Art

He, Lei; Guijt, Arthur; de Weerdt, Mathijs; Xing, Lining; Yorke-Smith, Neil

DOI
10.1016/j.cie.2019.106102

Publication date
2019

Document Version
Final published version

Published in
Computers and Industrial Engineering

Citation (APA)
He, L., Guijt, A., de Weerdt, M., Xing, L., & Yorke-Smith, N. (2019). Order Acceptance and Scheduling with Sequence-Dependent Setup Times: A New Memetic Algorithm and Benchmark of the State of the Art. *Computers and Industrial Engineering*, *138*, 1-15. Article 106102. https://doi.org/10.1016/j.cie.2019.106102

Computers & Industrial Engineering 138 (2019) 106102

Contents lists available at ScienceDirect

Computers & Industrial Engineering

journal homepage: www.elsevier.com/locate/caie

Order acceptance and scheduling with sequence-dependent setup times: A new memetic algorithm and benchmark of the state of the art

Lei He[a,b,*], Arthur Guijt[b], Mathijs de Weerdt[b], Lining Xing[a], Neil Yorke-Smith[b]

[a] College of Systems Engineering, National University of Defense Technology, Changsha 410073, China
[b] Delft University of Technology, P.O. Box 5031, 2600 GA Delft, the Netherlands

ARTICLE INFO

Keywords:
Memetic algorithm
Biased random key genetic algorithm
Adaptive large neighbourhood search
Order acceptance and scheduling
Sequence-dependent setup times

ABSTRACT

The Order Acceptance and Scheduling (OAS) problem describes a class of real-world problems such as in smart manufacturing and satellite scheduling. This problem consists of simultaneously selecting a subset of orders to be processed as well as determining the associated schedule. A common generalization includes sequence-dependent setup times and time windows. We propose a novel memetic algorithm for this problem, called Sparrow. It comprises a hybridization of biased random key genetic algorithm (BRKGA) and adaptive large neighbourhood search (ALNS). Sparrow integrates the exploration ability of BRKGA and the exploitation ability of ALNS. On a set of standard benchmark instances, this algorithm obtains better-quality solutions with runtimes comparable to state-of-the-art algorithms. To further understand the strengths and weaknesses of these algorithms, their performance is also compared on a set of new benchmark instances with more realistic properties. We conclude that Sparrow is distinguished by its ability to solve difficult instances from the OAS literature, and that the hybrid steady-state genetic algorithm (HSSGA) performs well on large instances in terms of optimality gap, although taking more time than Sparrow.

1. Introduction

The Order Acceptance and Scheduling (OAS) problem consists of simultaneously selecting a subset of orders to be processed as well as determining the associated schedule. This problem is important because it represents a class of real-world industrial problems. For example, when a smart manufacturing system does not have the capacity to meet the demand, it has to reject some orders in favour of others. Other typical instances of the OAS problem are for example satellite observation scheduling, where the number of observations requests is usually higher than the number of observation a satellite can take (Wang, Reinelt, Gao, & Tan, 2011), and in industrial and commercial logistics, such as deciding the cities to visit within a day to maximize the total profit (Verbeeck, Vansteenwegen, & Aghezzaf, 2017).

Many real-world order acceptance and scheduling problems in smart manufacturing and other domains have setup times and time windows. The setup time is the time needed for the preparation of the next order, such as the time to prepare batches of products in the factory domain and the observation angle transition time in the satellite domain. The setup time is usually sequence-dependent, which means the setup between every two orders depends on the specific pair of orders. The time windows specify a time period for each order when it can be processed. These problems can be generalized as a typical type of OAS problem: the OAS problem with sequence-dependent setup times and time windows (Oğuz, Salman, & Yalçın, 2010). This problem has been proven to be NP-hard (Ghosh, 1997).

The current state-of-the-art for the OAS problem with sequence-dependent setup times and time windows limits the capability of smart industry and operations. Due to the NP-hardness of the problem, realistically-sized problems cannot be solved exactly within acceptable running time: instead, solution quality must be compromised to deliver timely solutions. In addition, since current research focuses on improving the performance of algorithms, few contributions try to understand the problem further by studying how the problem properties correlate with its difficulty, how different algorithms perform on problem instances with varying properties, and how the instances in the standard benchmark set by Cesaret, Oğuz, and Salman (2012) correspond to real-life scenarios. These research gaps motivate our contribution.

In this article we propose a novel memetic algorithm applied for the

* Corresponding author at: College of Systems Engineering, National University of Defense Technology, Changsha 410073, China.
E-mail address: helei@nudt.edu.cn (L. He).

https://doi.org/10.1016/j.cie.2019.106102
Received 3 January 2019; Received in revised form 12 June 2019; Accepted 28 September 2019
Available online 01 October 2019

OAS problem with sequence-dependent setup times and time windows, called Sparrow.[1] The main contributions of this article are summarized as follows:

1. Sparrow is a hybridization of the biased random key genetic algorithm (BRKGA) and the adaptive large neighbourhood search algorithm (ALNS). To the best of our knowledge, this is the first hybridization of these two algorithms. Several new strategies are introduced to make the hybridization efficient.
2. Sparrow is compared with state-of-the-art algorithms on a set of standard benchmark instances from the literature. The proposed algorithm obtains better-quality solutions with comparable running time.
3. We study the correlation of the problem properties and the algorithm performance and find that the congestion ratio, the length of time windows, and the correlation of processing time and revenue of orders are highly related to the difficulty of the problem.[2]
4. We further generate new OAS instances with three realistic properties that are more representative of real problem instances in satellite scheduling (more congested), commerce (high correlation between revenue and processing time), and the travelling repairman problem (short processing times and long time windows), and compare the performance of multiple state-of-the-art algorithms on these new instances.
5. Too few open source implementations and datasets for benchmarking for this problem are available to allow for fair comparison of approaches. We publish the source code of Sparrow and all the datasets used in the paper.[3]

The remainder of this article is summarized as follows: Section 2 provides background information; Section 3 introduces Sparrow; Section 4 provides the empirical study of the proposed algorithm; the conclusions are summarized in Section 5.

2. Background

This section first introduces the mathematical formation of the OAS problem. Then related work is reviewed. Finally the standard BRKGA and ALNS algorithms that form the basis of Sparrow are described.

2.1. Mathematical formulation

In this section we provide a compact mathematical formulation of the OAS problem with time windows and sequence-dependent setup times to precisely define the problem at hand. A linear formulation which is more efficient but more difficult to follow is provided by Oğuz et al. (2010).

Consider a set of orders $O = \{o_1, ..., o_n\}$ that can be potentially scheduled. The sequence of orders is not fixed. Each order has a revenue r_i, a processing duration time t_i, a due time d_i, a penalty weight of tardiness w_i and a time window $[b_i, e_i]$ indicating the release time and deadline of the order. Let x_i be a binary decision variable representing whether order o_i is selected and p_i be a decision variable representing the start time of o_i. Let T_i be the tardiness of o_i, $T_i = \max\{p_i + t_i - d_i, 0\}$. The problem can be formulated as a mixed integer programming (MIP) model:

$$\text{Maximize} \quad \sum_{i=1}^{n} x_i(r_i - w_i T_i) \tag{1}$$

subject to

$$\max\{b_j, p_i + t_i\} + s_{ij} \leqslant p_j \quad \forall\, i, j \in \{1, ..., n\} \quad \text{if} \quad x_i = 1, x_j = 1, p_i < p_j \tag{2}$$

$$T_i = \max\{p_i + t_i - d_i, 0\} \quad \forall\, i \in \{1, ..., n\} \tag{3}$$

$$b_i \leqslant p_i \leqslant e_i - t_i \quad \forall\, i \in \{1, ..., n\} \quad \text{if} \quad x_i = 1 \tag{4}$$

$$x_i \in \{0, 1\} \quad \forall\, i \in \{1, ..., n\} \tag{5}$$

The objective function (1) maximizes the total reward of scheduled orders. The reward of an order equals its revenue minus the penalty of tardiness. Constraints (2) ensure the time between every two orders o_i and o_j should be longer than the setup time s_{ij}, and this setup can only be started after the release of o_j. The value of s_{ij} depends on i and j and is always non-negative. Constraints (3) define the domains of the tardiness of orders. Constraints (4) and (5) define the domains of the decision variables p_i and x_i respectively.

A feasible solution of this problem is a sequence of orders with scheduled start times which meet all the constraints above. It can be represented as $S = \langle s_1, s_2, ..., s_{|S|} \rangle$, where $s_i \in O$ is a scheduled order, $i \in \{1, ..., |S|\}$, $p_{s_1} < p_{s_2} < \cdots < p_{s_{|S|}}$, and $|S| \leqslant n$.

2.2. Related works

Many OAS variants can be found in the literature. The following review only includes studies of the OAS problem with sequence-dependent setup times and time windows. Readers are referred to Slotnick (2011) for a comprehensive survey of the OAS problem.

The OAS problem with sequence-dependent setup times and time windows was first proposed by Oğuz et al. (2010). They proposed a mixed integer linear programming (MILP) method for this problem, two simple greedy constructive heuristics, and a heuristic algorithm called iterative sequence first-accept next (ISFAN), which is based on the simulated annealing (SA) algorithm. Cesaret et al. (2012) proposed a tabu search (TS) algorithm to solve this problem, which achieved better results than ISFAN. Another major contribution of this article is that it provided the benchmark set and the upper bounds of this problem. Nguyen, Zhang, and Johnston (2014b) proposed an exact branch-and-bound (B&B) with genetic programming (GP) to discover good ordering rules. Their method could find optimal solutions for instances with up to 20 orders. Silva, Subramanian, and Pessoa (2018) proposed a new arc-time-indexed formulation and two exact methods based on Lagrangian relaxation and column generation respectively. They computed tighter upper bounds compared with those by Cesaret et al. (2012). They also proposed an iterated local search (ILS) method to solve the problem.

The hybridization of population-based methods with local search (LS) has been used a lot to solve this problem. Lin and Ying (2013) proposed an artificial bee colony (ABC) algorithm to solve this problem. They used a permutation based representation and generated new solutions through standard order crossover or a LS algorithm (removing and inserting orders). Their methods achieved better results than those of TS (Cesaret et al., 2012). Chen, Yang, Tan, and He (2014) proposed a diversity controlling genetic algorithm (DCGA), which selects individual chromosomes not only depending on the fitness, but also on the diversity of the whole population. In their algorithm, they also adopted similar LS methods as the method by Lin and Ying (2013). Nguyen, Zhang, and Tan (2015) proposed a dispatching-rule-based genetic algorithm (DRGA). The dispatching rule is a strategy to calculate some heuristic priorities for orders and these priorities are used in the decoding process. They also used a simple greedy LS method to improve the solutions. Following Nguyen et al., Park, Nguyen, Johnston, and Zhang (2013) proposed a GP method with stochastic

[1] This algorithm combines a population-based genetic algorithm with adaptive large neighbourhood search. Each individual in a "swarm" thus has a bird's eye view of the search space – hence Sparrow.

[2] These terms are defined in Section 4.3.

[3] All the source codes and datasets used in this paper are available http://doi.org/10.4121/uuid:c3623076-a1ac-4103-ad31-3068a28312f9.

dispatching rules, which could generate and evolve rules to find good solutions. They later proposed a hybrid particle swarm optimization (PSO) method with TS using dispatching rules learned by GP (Park, Nguyen, Zhang, & Johnston, 2014). Similar ideas were also used by Nguyen, Zhang, and Johnston (2014a). Nguyen (2016) proposed a hyper-heuristic algorithm, which is a learning and optimizing system (LOS). The method trains a set of optimizing rules in the learning phase, then the rules are used by a genetic algorithm (GA) to find good solutions in the optimizing phase. More recently, Chaurasia and Singh (2017) proposed a hybrid steady-state genetic algorithm (HSSGA) and an evolutionary algorithm with guided mutation (EA/G-LS). The authors also proposed an ABC-based hyper-heuristic. However they only published the results on small instances (Chaurasia & Kim, 2019).

The above review shows that the hybridization of population-based methods with LS methods is a promising method for the OAS problem with sequence-dependent setup times and time windows. The evolutionary algorithm (EA) is an important population-based algorithm. The hybridization of EA and LS is called memetic algorithm (MA) (Moscato, 1989) and has also been successfully applied to problems where all orders are scheduled including the travelling salesman problem (Bontoux, Artigues, & Feillet, 2010), the flowshop scheduling problem (Wang, Fu, Huang, Huang, & Wang, 2017) and the location routing problem (Asgari, Rajabi, Jamshidi, Khatami, & Farahani, 2017). It is recognized that the superior performance of MA comes from the integration of the exploration ability of the EA and the exploitation ability of the LS (Krasnogor & Smith, 2005). A detailed description and comprehensive survey of MA could be found in Neri and Cotta (2012).

Next, we review the two components of Sparrow: BRKGA and ALNS.

The BRKGA algorithm is a variant of the EA method proposed by Gonçalves and De Almeida (2002). Compared with standard GA methods, BRKGA offers more flexibility in encoding solutions (Chaves, Gonçalves, & Lorena, 2018) and produces as good or better solutions (Gonçalves & Resende, 2011). BRKGA has shown competitive performance on a series of optimization problems (Prasetyo, Fauza, Amer, & Lee, 2015), including the satellite scheduling problem (Tangpattanakul, Jozefowiez, & Lopez, 2015), which is very similar to the problem studied in this article. The ALNS algorithm was first proposed by Pisinger and Ropke (2007). It performs particularly well on problems with order acceptance features, such as the orienteering problem (Palomo-Martínez, Salazar-Aguilar, & Laporte, 2017), the satellite scheduling problem (Liu, Laporte, Chen, & He, 2017) and the pickup and delivery problem with selective requests (Li, Chen, & Prins, 2016).

Comparing the two approaches, the advantage of BRKGA is that it can search the large solution space efficiently. However, the problem-independent nature of BRKGA makes it difficult to find high-quality solutions. For the second approach, ALNS, multiple neighbourhood operators can be defined according to the characteristics of the problem. Therefore ALNS has a good exploitation ability and can adapt itself according the different properties of problem instances. However, its search efficiency can founder due to the entrapment in a local optimum because of the single-point search. Considering the fact that it is promising to hybridize population-based methods with LS methods for the OAS problem, and the good performance BRKGA and ALNS have shown on similar problems, we select these two representatives, one is population-based and the other one is LS, as the two components of Sparrow. We are interested to see if a hybridization could be made such that the advantages of the two algorithms can be integrated and achieve better performances.

2.3. The standard BRKGA and ALNS algorithms

This section introduces the standard BRKGA (Gonçalves & Resende, 2011) and ALNS (Pisinger & Ropke, 2007).

In BRKGA, a population of p solutions are represented as p chromosomes, consisting of genes which are encoded by real values randomly generated in the interval [0,1], i.e., random keys. The number of genes in a chromosome equals the number of orders. Chromosomes are then decoded to obtain solutions and calculate fitness, which is a problem-dependent process. Example 1 shows a simple decoding method for the problem studied in this article. The best p_e chromosomes are recognized as elite individuals. Unlike standard genetic algorithms which generate mutants by changing parts of individuals from the current population, BRKGA generates p_m chromosomes randomly in the mutation operation (i.e., the same as the initial population). This mutation strategy helps BRKGA to avoid the entrapment in a local optimum. In the crossover operation, $p - p_e - p_m$ chromosomes are generated by inheriting each gene from an elite parent with the probability ρ_e and a non-elite parent with the probability $1 - \rho_e$. The encoding strategy of BRKGA ensures that there are no duplicate orders and all solutions are feasible in the crossover operation.

Example 1. All the orders are first sorted according to the ascending gene values. Then the orders are started as early as possible. Any order that can not complete before the deadline or can not achieve a positive reward, will be rejected. Consider a set of five orders {1, 2, 3, 4, 5} with the corresponding gene values {0.6, 0.2, 0.3, 0.5, 0.7}. The decoder tries to start each order as early as possible, following the sequence of {2, 3, 4, 1, 5}. Assume that the end time of order 3 is larger than the deadline of order 4, which means that order 4 cannot be inserted in the solution, the resulting solution would be {2, 3, 1, 5}.

ALNS starts from an initial solution usually generated by a simple heuristic, because it is less sensitive to the initial solution than general local search (Demir, Bektaş, & Laporte, 2012). ALNS proceeds to generate new solutions through destroying and repairing. In the destroying process, p_d orders are removed from the current solution by removal operators. The unscheduled and removed orders are then inserted into the destroyed solution in the repairing process by insertion operators. There are multiple removal and insertion operators. At each iteration, a pair of removal and insertion operators is selected by a roulette wheel mechanism according to their weights. After a certain number of iterations, the weight of the operator w_i is updated adaptively according to its accumulated score π_i in the previous iterations, $w_i = (1 - \lambda)w_i + \lambda\pi_i / \sum_j \pi_j$, where $\lambda \in [0, 1]$ is a reaction factor which controls how sensitive the weights are to changes in the performance of operators. A simulated annealing (SA) criterion is used to control the acceptance of new solutions by a temperature parameter T. Let $f(S)$ and $f(S')$ be the reward of current solution S and new solution S' respectively. The new solution S' is accepted if $f(S') > f(S)$; otherwise, it is accepted with probability: $\rho = \exp\left(\frac{100}{T}\left(\frac{f(S') - f(S)}{f(S)}\right)\right)$.

3. Sparrow

In this section, the main framework of Sparrow is introduced first. The tight hybridization aims to avoid the possible high running time of the combination of the two complex algorithms and aims to integrate the advantages of them. Then the BRKGA part is introduced, where three new algorithmic features, a new bounded-width gene encoding strategy for the problem with time windows, a hybrid decoding method for the OAS problem and an intelligent crossover operator are proposed. Finally the ALNS part is introduced, where several neighbourhood operators are defined and a fast insertion algorithm considering sequence-dependent setup times is proposed.

3.1. The main framework

The main framework of Sparrow is shown in Algorithm 1. The main structure of this algorithm is derived from BRKGA. ALNS is used in each iteration to improve the quality of the population. Fig. 1 shows how the algorithm evolves generations of solutions.

A problem of integrating ALNS in BRKGA is the complexity, because ALNS itself needs multiple iterations to improve a single solution. In

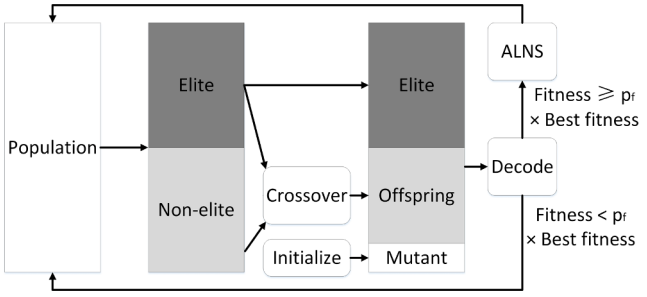

Fig. 1. The main framework of Sparrow.

order to hybridize these two algorithms without increasing the computation time too much, the destroying and repairing process of ALNS is run only once for each solution in the population and only on relatively good solutions with at least p_f of the best fitness (i.e., the value of the objective function Eq. (1)) found so far (Algorithm 1, Lines 6–8).

In standard ALNS, the algorithm updates the temperature in each iteration and updates the weights of operators after a certain number of iterations according to their performances. In this population-based ALNS, these two processes happen at the end of each segment of BRKGA (Algorithm 1, Lines 19–22). The segment is defined as $\lfloor \varphi/p \rfloor$ BRKGA iterations, where φ is a constant parameter and p is the number of individuals in the population. The length of the segment is defined to balance the frequency of the update of temperature and weights with the population size. The weights of operators are updated according to their performances in improving the individuals in the population.

There are three terminal conditions of Sparrow: when the maximum iteration is reached; when the best fitness is not improved for a maximum number of consecutive iterations; when all orders in the instance are scheduled and receive full revenue.

Algorithm 1. Sparrow

1: Generate an initial population P, consisting of p chromosomes;
2: Let S^* be the best solution and f^* be its fitness, $S^* \leftarrow \emptyset$, $f^* \leftarrow 0$;
3: **repeat**
4: **for** each chromosome c in P **do**
5: $S \leftarrow Decode(c)$
6: **if** $f(S) \geqslant p_f f^*$ **then**
7: $S \leftarrow ALNS(S, f^*)$
8: **end if**
9: **if** $f(S) > f^*$ **then**
10: $f^* \leftarrow f(S)$, $S^* \leftarrow S$
11: **end if**
12: **end for**
13: Sort chromosomes in P according to a descending order of fitness;
14: Add the top p_e chromosomes to Elite set E;
15: Add the remaining chromosomes to Non-elite set N;
16: Generate a mutation set of p_m chromosomes: $M \leftarrow Mutation()$;
17: Generate a crossover set of $p - p_e - p_m$ chromosomes: $C \leftarrow Crossover(E, N)$;
18: $P \leftarrow E \cup M \cup C$;
19: **if** The end of a segment **then**
20: Update the temperature of ALNS with the coefficient of annealing c_a: $T \leftarrow c_a T$;
21: Update the weights of different operators of ALNS;
22: **end if**
23: **until** Terminal condition is met;
24: **return** S^*, f^*;

3.2. The BRKGA

3.2.1. Encoding and decoding

In standard BRKGA, the initial random key is generated randomly in the interval [0,1] and then decoded according to the simple decoding method in Example 1. However, for this problem with time windows, if an order with early time windows receives a large random key, this order may not be scheduled because its time window is occupied by other orders with smaller random keys. To solve this problem, we propose a bounded-width gene encoding method and a hybrid decoding method.

Bounded-width gene encoding. When assigning the initial gene value to orders, the proportion of the window size in the whole scheduling horizon is used to bound the width of the interval of the random gene value. For example, if an order has the time window $[0, 10]$ and the whole scheduling horizon is [0,100], the interval for the gene value of this order is $[0, 0.1]$. This strategy helps to reduce the solution space, especially for the instances with short time windows.

One problem of this strategy is that orders with earlier time windows are preferred than those with later time windows, because those with later time windows receive larger random keys and may not be scheduled because of the early ones. This problem is solved by the ALNS algorithm in the removal process. The details are shown in Section 3.3.1.

Hybrid decoding method. Our hybrid decoding method consists of the simple decoding method as in Example 1 and a complex decoding method with order insertion. For the complex decoding, all the orders are also sorted according to the ascending gene values. Instead of starting each order following this sequence, this decoder tries to insert each order into the current partial solution at the position which increases minimum setup time. When inserting an order, the orders in the partial solution might be postponed but will not be canceled. The detailed insertion strategy is introduced in Section 3.3.2.

The time complexity of the simple decoding is $O(n\log n)$ (i.e., the time complexity for the sorting process), while the time complexity of the complex decoding is $O(n^2\log n)$. Although the complexity of the complex decoding is higher than the simple decoding, it helps to increase the probability of orders with short time windows being successfully scheduled. An adaptive strategy is used to hybridize these two strategies according to the instance. Let the value of the average length of time windows of all orders divided by the scheduling horizon be r_d. For each chromosome, the algorithm chooses the simple decoding with the probability of r_d; otherwise it uses the complex decoding.

After a number of iterations, the random keys of different orders may squeeze together due to the following crossover and ALNS operations. In order to avoid this, the keys are normalized to be distributed evenly in the interval of [0,1] in each iteration. Note that in the standard BRKGA, since the values of the genes are not changed, the normalization is not used.

3.2.2. Intelligent crossover

In the standard BRKGA, the offspring inherits each gene from the elite parent with the probability ρ_e; otherwise it inherits the gene from the non-elite parent. This strategy ensures that the offspring has more genes from the elite parent. However, in this OAS problem with sequence-dependent setup times, the quality of a sequence of orders is important. The setup time between any two orders can differ much. The standard crossover operation might break good sequences when it inherits genes from different parents, thus producing low-quality offspring.

We propose an intelligent crossover strategy to keep good order pairs together in the offspring. Before the crossover, the algorithm identifies good order pairs in the current elite and non-elite parents. A pair of orders is recognized as *good* if it has a relatively high unit reward (i.e., the total reward of the two orders divided by the time between the start time of the preceding order and the end time of the following order is higher than a constant parameter f_g).

When a good pair of orders is found, the gene value of the following

order will be changed to the gene value of the preceding order plus a small enough positive number ϵ, to ensure they will have a high probability of being together in the decoding phase. The crossover operator selects genes from one of the parents one-by-one. Normally there is a constant ρ_e determining the probability whether the gene comes from the elite or the non-elite parent. For the intelligent crossover, when one gene of a good pair is selected, the probability that the other gene in the good pair will be selected is enhanced to be p_g, a value close to 1. Note that the idea of the intelligent crossover is not to keep all pairs of good orders in the offspring, but to increase the probability that good pairs can stay together in the offspring. If the first gene of a good pair is given up, it does not matter whether the second gene is selected or not, because the good pair has already been broken.

The detailed process is shown in Algorithm 2.

Algorithm 2. Intelligent crossover

Input: Elite set E; Non-elite set N;
Let $C \leftarrow \emptyset$ be the crossover offspring set;
repeat
  Select an elite parent chromosome C_e from E randomly;
  Select a non-elite parent chromosome C_n from N randomly;
  Initialize a child chromosome C_c with an empty gene list;
  Identify and label *good* pairs of orders in C_e and C_n;
  for $i \leftarrow 1$, $i \leqslant n$, $i++$ **do**
    if the i^{th} gene in C_e is *good* $\wedge$ its pair is in C_c **then**
      Generate a random value r in [0,1];
      if $r < p_g$ **then**
        Add the i^{th} gene in C_e into C_c;
      else
        Add the i^{th} gene in C_n into C_c;
      end if
    else
      if the i^{th} gene in C_n is *good* $\wedge$ its pair is in C_c **then**
        Generate a random value r in [0,1];
        if $r < p_g$ **then**
          Add the i^{th} gene in C_n into C_c;
        else
          Add the i^{th} gene in C_e into C_c;
        end if
      else
        Generate a random value r in [0,1];
        if $r < \rho_e$ **then**
          Add the i^{th} gene in C_e into C_c;
        else
          Add the i^{th} gene in C_n into C_c;
        end if
      end if
    end if
  end for
until $|C| = p - p_e - p_m$
return C;

3.3. The ALNS

As mentioned above, if a solution has a relatively good fitness, ALNS is used to improve the solution further. In the following sections, the neighbourhood operators are first introduced; then a fast insertion algorithm used in the insertion operator to insert orders into the current solution is proposed. The full ALNS algorithm is shown in Algorithm 3. In Algorithm 3, $\sigma_1 > \sigma_2 > \sigma_3$ are three score increment parameters, used to increase the scores of different neighbourhood operators depending on their performances.

Algorithm 3. The ALNS algorithm

Input: Current solution S_C; best fitness f^*;
Add all unscheduled orders in order bank B according to S_C;
Select a removal operator O_R according to the weights;
Sort the scheduled orders in S_C according to O_R;
$S_N \leftarrow$ Remove the top p_d orders from S_C and add them into B;
Update the start times and time slacks of orders in S_N;
Select an insertion operator O_I according to the weights;
Sort the unscheduled orders in B according to O_I;
for each candidate order o_c in B **do**
  $S_N \leftarrow FastInsertionAlgorithm(S_N, o_c)$
end for
if $f(S_N) > f^*$ **then**
  $\pi_{O_R} \leftarrow \pi_{O_R} + \sigma_1$, $\pi_{O_I} \leftarrow \pi_{O_I} + \sigma_1$, $S_C \leftarrow S_N$;
else
  if $f(S_N) > f(S_C)$ **then**
    $\pi_{O_R} \leftarrow \pi_{O_R} + \sigma_2$, $\pi_{O_I} \leftarrow \pi_{O_I} + \sigma_2$, $S_C \leftarrow S_N$;
  else
    if The SA criterion accepts S_N **then**
      $\pi_{O_R} \leftarrow \pi_{O_R} + \sigma_3$, $\pi_{O_I} \leftarrow \pi_{O_I} + \sigma_3$, $S_C \leftarrow S_N$;
    end if
  end if
end if
return S_C;

3.3.1. Neighbourhood operators

In order to ensure ALNS is suitable for a diverse range of problem instances with varying characteristics, the following five removal operators and two insertion operators are used. These operators are adapted from those in the literature (Demir et al., 2012; Pisinger & Ropke, 2007) to fit our problem.

The five removal operators are: random (p_d orders are removed randomly); min revenue (p_d orders with lower revenue are removed); min unit revenue (p_d orders with lower unit revenue are removed: the unit revenue is the revenue of the order divided by its processing time); max setup time (p_d orders with longer setup time are removed); sequence removal (the worst sequence of p_d orders is removed: the quality of a sequence is evaluated by the unit revenue).

The two insertion operators are: max revenue; max unit revenue. The insertion operator first sorts the orders according to the descending revenue or unit revenue, then tries to insert orders in the current solution using the fast insertion algorithm in Section 3.3.2.

In order to make sure that the solution operated by ALNS can be encoded correctly for the following BRKGA operations, when removing an order, the gene of this order will be assigned a random real value larger than the largest gene in the current solution and smaller than 1; when inserting an order, the gene of this order will be assigned a random real value between the gene values of its preceding and following orders. This strategy helps to solve the problem brought by the bounded-width gene generation strategy. Although the orders with earlier time windows have smaller gene values in the initialization, they can be removed and get a large gene value in ALNS.

3.3.2. Fast insertion algorithm

Our last innovation is a fast insertion algorithm, which first evaluates the feasibility and the cost of all the positions rapidly by a concept called *time slack*. Then the best position is selected to insert the order. The insertion algorithm is used in the repairing process when inserting orders back to the solution. The detailed process of the fast insertion algorithm is shown in Algorithm 4.

Algorithm 4. Fast insertion algorithm

```
1: Input: Current solution S_C, candidate order o_c;
2: for each scheduled order o_i in S_C do
3:   if e_c > b_i then
4:     t_1← Calculate the end time of o_c if it is inserted after o_i;
5:     t_Temp← Calculate the temporary start time of o_{i+1} after o_c;
6:     t_2 ← t_Temp − p_{i+1};
7:     if t_1 > e_c ∨ t_2 >time slack of o_{i+1} then
8:       Continue;
9:     else
10:      if t_1 ⩽ d_c ∧ t_2 ⩽due time slack of o_{i+1} then
11:        Position_i. SetupIncrease ← s_ic + s_c(i+1) − s_i(i+1);
12:        Add Position_i into position list PL1;
13:      else
14:        Position_i. Fitness← Calculate the total fitness if o_c is inserted;
15:        if Position_i. Fitness > f(S_C) then
16:          Add Position_i into position list PL2;
17:        end if
18:      end if
19:    end if
20:  end if
21: end for
22: if PL1 ≠ ∅ then
23:   BestPosition← Select the position increasing minimum setup time;
24:   Insert o_c into S_C at BestPosition;
25:   Update start times and time slacks;
26: else
27:   if PL2 ≠ ∅ then
28:     BestPosition← Select the position increasing maximum fitness;
29:     Insert o_c into S_C at BestPosition;
30:     Update start times and time slacks;
31:   end if
32: end if
33: return S_C;
```

Time slack and due time slack. In the algorithm, all the scheduled orders start as early as possible. Therefore when inserting one order into the current solution at a position, it is possible to create more space for the candidate order by postponing some orders in the solution. In order to determine how much one order can be postponed, we adopt the *time slack* idea from Verbeeck et al. (2017). We further propose the *due time slack* heuristic for this problem with tardiness penalty.

The time slack is defined as the maximum amount of time an order can be postponed before the solution becomes infeasible. The time slack of each order depends on the latest start time of its succeeding order. Thus it is calculated from the last order to the first one in a back-propagation manner. Let o_i be the i^{th} order in the current solution, o_{i+1} be its immediate successor and $|S|$ be the number of orders in the current solution S. The latest start time of o_i is calculated by:

$$p_i^{Late} = \begin{cases} \max\{\min\{p_{i+1}^{Late} - s_{i(i+1)} - t_i,\ e_i - t_i\},\ b_i\} & \text{if} \quad 1 \leqslant i < |S| \\ e_i - t_i & \text{if} \quad i = |S| \end{cases} \tag{6}$$

The latest start time of the last order in the solution is the latest start time defined by its time window. Then the time slack of o_i can be calculated by $p_i^{Late} - p_i$.

The due time slack is the maximum amount of time an order can be postponed without adding penalty to any order. Similarly, to calculate the due time slack of an order, the latest start time of the order without receiving any penalty should be calculated. The latest start time of the order without receiving any penalty is

$$p_i^{DueLate} = \begin{cases} \max\{\min\{p_{i+1}^{DueLate} - s_{i(i+1)} - t_i,\ d_i - t_i\},\ b_i\} & \text{if} \quad 1 \leqslant i < |S| \\ d_i - t_i & \text{if} \quad i = |S| \end{cases} \tag{7}$$

Then the due time slack of o_i can be calculated by $p_i^{DueLate} - p_i$. Note that if some orders in the current solutions have some penalties, then their due time slacks will be negative.

These heuristics facilitate determining the feasibility and the cost of one insertion only by comparing the time needed with the corresponding slack.

Select the best position to insert. For every candidate order, all possible insertion positions can be found by comparing its time window with the current solution. Let o_c be the candidate order. For the possible position between order o_i and o_{i+1}, the insertion algorithm does the following evaluation: It first calculates the end time of o_c if the o_c is inserted after o_i (denoted as t_1) and the time o_{i+1} needed to be postponed (denoted as t_2). If t_1 is bigger than the deadline of o_c or t_2 is bigger than the time slack of o_{i+1}, the position is given up; if t_1 is smaller than the due time of o_c and t_2 is smaller than the due time slack (note that the due time slack is always smaller than the time slack), the algorithm calculates the increase of setup time if inserting o_c (i.e., $s_{ic} + s_{c(i+1)} - s_{i(i+1)}$) and adds the position to candidate position list 1, PL_1; otherwise (i.e., if t_1 is larger than the due time of o_c or t_2 is larger than the due time slack), the algorithm calculates the total fitness of the solution if the order is inserted at this position and if it increases the fitness, the position is added to the candidate position list 2, PL_2. Finally, if PL_1 is not empty, the position with the smallest value of the increase of setup time in PL_1 is selected to insert the order; otherwise, the position with the highest total fitness in PL_2 is selected to insert the order. If both PL_1 and PL_2 are empty, the candidate order is given up.

The position is selected according to the above strategy because when PL_1 is not empty, the candidate order can be inserted without receiving any penalty, which means the total fitness can be increased with the revenue of the candidate order. In this case the position increasing the minimum setup time is selected. The rationale is that it is better to use the time more for processing orders instead of setting up. If PL_1 is empty, some orders will receive penalty. In this case it is necessary to compute the fitness to find the best insertion position.

When an order is inserted, the start times of all its succeeding orders are updated according to the constraints until one whose start time does not change. The solution after the insertion process is always feasible. The time slacks of all the orders before the candidate order and the orders whose start time is changed are also updated.

4. Experiments

The goal of the experiments is to understand the strengths and weaknesses of the different algorithms for the OAS problem with sequence-dependent setup times and time windows. In particular, for the new algorithm, Sparrow, also to find out good parameter settings. For this purpose we start by running Sparrow for a range of parameter settings on a standard benchmark set. Then Sparrow is compared with state-of-the-art algorithms. Next we study how different properties of the problem correlate with its difficulty. Finally new problem instances with more realistic and harder properties are generated and used to compare the performances of different algorithms on the harder cases.

4.1. Parameter settings for Sparrow

In this section, we aim to understand how some important parameters of Sparrow influence its performance and answer the following questions: Is this hybridization better than the two standalone algorithms; for instances with different properties, is it better to have a larger population size or a larger number of iterations?

The standard benchmark set used is that by Cesaret et al. (2012), which is the most common benchmark set used by many articles (Chaurasia & Kim, 2019; Chaurasia & Singh, 2017; Chen et al., 2014; Cesaret et al., 2012; Lin & Ying, 2013; Nguyen, 2016; Nguyen et al., 2014a, 2014b, 2015; Park et al., 2013, 2014; Silva et al., 2018). This benchmark set has n, τ and r as parameters, where n is the number of orders n = 25, 50, 100, and τ and R are the tardiness factor and the due time range factor, having the values of 0.1, 0.3, 0.5, 0.7, and 0.9. The orders are generated by random values from a discrete random

Table 1
Five sets of experiments with different parameters.

Parameter name	Set 1	Set 2	Set 3	Set 4	Set 5
Size of population p	1	10	20	50	1000
Maximum iteration of no improvement	$200n$	$20n$	$10n$	$4n$	n
Maximum iteration	1,000,000	100,000	50,000	20,000	2000
Whether ALNS is used	Yes	Yes	Yes	Yes	No

distribution [1, 20] for processing time t_i and revenue r_i, similarly the range [1, 10] is used for the sequence-dependent setup times s_{ij}. Release times b_i of orders are from $[0, \tau \cdot t_T]$, where $t_T = \sum_{i=1}^{n} t_i$. The due time for an order is calculated by $d_i = b_i + s_{\max} + \max\{v, t_i\}$, where $s_{\max}$ is the largest sequence-dependent setup time over any order, and v is a random value generated from $[t_T(1 - \tau - R/2), t_T(1 - \tau + R/2)]$. Finally the deadlines and weights are calculated by $e_i = d_i + Rt_i$ and $w_i = r_i/(e_i - d_i)$. For each set of parameters, 10 instances are generated.

Sparrow has a lot of parameters. In this section we focus on three parameters: the size of population p, the maximum number of consecutive iterations of no improvement and the maximum iteration, because these three parameters influence the importance of the two components in the hybridization. When the population size is small and the number of iterations is large, solutions are improved by more ALNS operations, hence the algorithm performance relies more on ALNS; when the population size is large and the number of iterations is small, the algorithm explores more solutions in the space, hence the algorithm performance relies more on BRKGA. Five sets of parameters are compared, which are shown in Table 1. These multiple parameters are changed simultaneously and set as in Table 1 because the Sparrow algorithm with these sets of parameters have relatively equal CPU time. We hope to see with equal CPU time, whether it is better to have larger population or more iterations for the Sparrow algorithm. In Table 1, Set 1 refers to the case of standard ALNS and Set 5 refers to the case of standard BRKGA.

Other parameters of Sparrow are set as follows: number of elite individuals $p_e = 0.5p$; number of mutation individuals $p_m = 0.1p$; probability for a gene coming from the elite parent $\rho_e = 0.7$; the unit reward for a good pair $f_g = 2$; probability of keeping the good pair $p_g = 0.95$; number of orders to remove by ALNS is $0.4 \cdot |S|$, where $|S|$ is the number of orders in the current solution; the length of a segment $\lfloor 50/p \rfloor$; weight update parameter $\lambda = 0.5$; coefficient of annealing $c_a = 0.9975$; score increment according to the performance of operators: $\sigma_1 = 30$, $\sigma_2 = 20$, $\sigma_3 = 10$; parameter for ALNS improvement $p_f = 0.9$. These parameters are set empirically through some exploratory experiments. Although these parameters are not optimal for all the instances, they provide relatively good solutions.

In this section, the upper bounds provided by Silva et al. (2018) are used to calculate the gaps of the results of Sparrow with the different parameter settings, because these upper bounds are tight and more suitable for analyzing the performance of an algorithm. We received the detailed upper bounds from the authors of Silva et al. (2018).

The gaps of the Sparrow algorithm are calculated according to the different numbers of orders, τ and R and shown in Fig. 2. Only one point of results for Set 5 can be observed with the current scale, because results for Set 5 are significantly worse (e.g., 4.04% for order size 50 and 6.00% for order size 100). From all of the above results, it can be concluded that the performance of the two standalone algorithms perform much worse than the hybrid algorithm, which proves the effectiveness of this hybridization. Comparing these three sub-figures, we can find that the Sparrow algorithm works better when the problem size is smaller, and when τ and R are larger. The problem becomes harder for the Sparrow algorithm when the problem grows in size, and when τ and R become smaller.

Then we compare Sets 2–4 within each sub-figure. Regarding the number of orders, Set 2 performs better when the problem size is larger, while Set 4 performs better when the problem size is smaller. The performance of Set 3 is between the performances of Set 2 and Set 4. It shows that the ALNS component is more suitable for larger instances, while the BRKGA component works better for smaller instances. Regarding the values of τ and R, the pattern is not very obvious. Set 4 tends to work best when τ and R are larger.

As a summary, in this section the following conclusions can be derived: (1) the hybrid Sparrow algorithm outperforms the two standalone algorithms; (2) the problem becomes harder for Sparrow when the problem grows in size and when τ and R are smaller; (3) When the problem size is small, it is better to have a large population, while when the problem size is large, it is better to have a large iteration time; (4) Sparrow with a large population tends to work better when τ and R are larger.

4.2. Comparison with state-of-the-art algorithms

In this section, the same benchmark set as mentioned in Section 4.1 is used. Sparrow is compared with TS (Cesaret et al., 2012), DRGA (Nguyen et al., 2015), GA (Nguyen, 2016), HH (Nguyen, 2016), LOS (Nguyen, 2016), ILS (Silva et al., 2018), ABC (Lin & Ying, 2013), HSSGA (Chaurasia & Singh, 2017), and EA/G-LS (Chaurasia & Singh, 2017). Note that the performance of MIP solved by ILOG CPLEX has been tested by Cesaret et al. (2012) and its performance is bad for large instances with more than 25 orders. Therefore, we do not compare Sparrow to CPLEX in this article.

The benchmark set has instances with the number of orders ranging from 25 to 100, and τ and R ranging from 0.1 to 0.9. According to the conclusions derived in Section 4.1, a moderate set of parameters is selected: population size: $p = 20$; maximum number of consecutive iterations of no improvement: $10n$; maximum iteration: 50,000. Other parameters are set as mentioned in Section 4.1.

Sparrow is run on Intel Core i5-3470 3.20 GHz CPU with 8 GB memory, using a single core. The results of other methods are from the respective articles, and they were obtained using machines with Intel Core i5, i7 and Xeon CPU, 3.00–3.40 GHz, 4–16 GB memory. Runtime

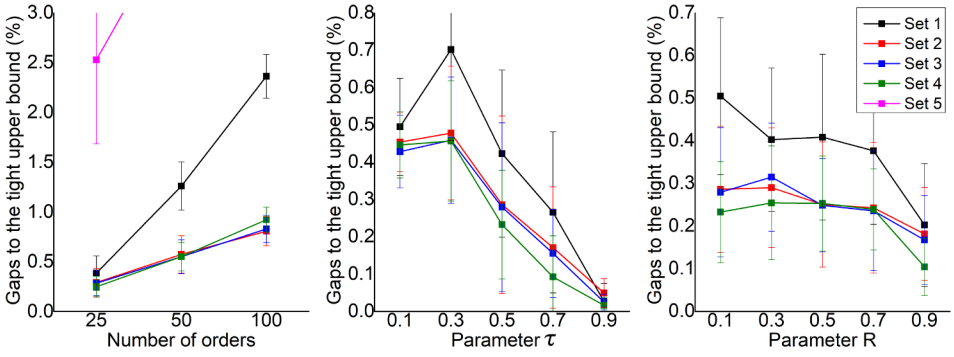

Fig. 2. The gap between the upper bound and Sparrow with different parameter settings. In figure left, each point is the average of 10 runs of all instances with the same number of orders in the standard benchmark set. In figure middle and right, it is noticed that the upper bounds from Silva et al. (2018) are tighter for instances with 25 orders. Therefore, to better analyze the influence of τ and r, we calculate the average of 10 runs of instances with 25 orders and the same τ and r. Set 5 is quite literally "off the charts".

Table 2
The CPU time (s) for instances with 100 orders by different algorithms.

τ	R	TS	DRGA	LOS	ILS	ABC	HSSGA	EA/G-LS	Sparrow
0.10	0.10	16.76	15	11	24.9	**3.98**	17.91	13.72	10.55
	0.30	17.26	15	11	25.3	**8.22**	16.68	13.76	11.22
	0.50	10.87	15	11	21.9	**7.61**	17.07	12.82	7.88
	0.70	6.21	15	11	13.9	6.25	15.08	11.06	**1.38**
	0.90	3.53	15	11	9.6	9.08	14.73	10.02	**0.48**
0.30	0.10	22.37	15	11	38.4	**4.89**	16.13	13.53	10.83
	0.30	20.10	15	11	32.4	**5.29**	19.70	14.33	10.75
	0.50	16.77	15	11	33.7	**4.88**	21.00	13.31	13.34
	0.70	9.48	15	11	28.3	**6.16**	16.60	14.04	9.16
	0.90	7.58	15	11	25.1	8.59	18.33	12.10	**6.75**
0.50	0.10	25.96	15	12	51.9	**5.27**	19.30	12.24	11.05
	0.30	28.87	15	12	57.9	**6.94**	20.96	12.87	12.21
	0.50	20.56	15	12	52.5	**5.86**	21.52	13.75	15.05
	0.70	15.57	15	12	42.7	**6.57**	17.27	11.79	15.49
	0.90	12.15	15	12	46.0	**6.76**	18.66	13.72	16.89
0.70	0.10	33.60	15	13	63.9	**6.04**	23.40	11.75	16.46
	0.30	26.62	15	12	73.9	**6.25**	24.90	12.79	17.98
	0.50	22.30	15	12	72.6	**6.77**	19.89	12.79	18.86
	0.70	26.36	16	12	77.5	**6.78**	24.24	14.33	20.53
	0.90	17.84	16	12	78.4	**7.42**	21.90	15.45	18.78
0.90	0.10	29.46	16	12	80.6	13.63	19.59	**10.65**	13.92
	0.30	26.32	16	12	79.1	**9.82**	19.64	11.75	13.75
	0.50	21.51	16	12	81.1	**6.46**	19.06	11.62	15.48
	0.70	22.70	16	12	91.5	**7.52**	17.61	14.30	16.69
	0.90	17.48	16	12	92.0	**8.61**	17.92	11.04	17.38
Avg.		19.13	15	11	51.8	**7.03**	19.16	12.78	12.91

results from such different machines are incomparable, even unmentioned details such as cache size can have a significant effect. However, we still include the runtime results for order size 100 in Table 2, to be able to draw some first conclusions about runtimes anyway.

On average, all the methods except ILS have comparable performance in terms of CPU time. ILS is slower than others. Regarding the solution quality, the above articles reported the gaps between their methods and the upper bounds provided by Cesaret et al. (2012). In order to compare with these methods directly, in this section the upper bounds by Cesaret et al. (2012) are used to calculate the gaps of Sparrow.

Sparrow, TS, DRGA, GA, HH, LOS and ILS are run ten times on each of the instances. The average gaps of the ten runs for each instance are calculated first. Then for each group of instances with the same parameters, the minimum, average and maximum gaps of the ten instances in this group are calculated. The results of instances with 25–100 orders are shown in Tables 3–5. The other three algorithms, ABC, HSSGA and EA/G-LS are run only once for each instance. In order to compare with these three algorithms, the minimum, average and maximum results of the best, the average, and the worst of the ten runs (denoted as Sparrow-Best, Sparrow-Average and Sparrow-Worst respectively) are calculated as a reference of the performance of Sparrow. The results are shown in Tables 6–8. Note that in Tables 6–8, only the best results among ABC, HSSGA, EA/G-LS and Sparrow-Average are highlighted.

In Tables 3–5, TS, DRGA, GA, HH and LOS only reported rounded-down integer values. But it is still obvious that Sparrow produces the best solutions on nearly all the instances. In Tables 6–8, the average results of Sparrow are better than those of ABC, HSSGA and EA/G-LS on most of the instances. Even the worst results of the ten runs of Sparrow are better than those of ABC, HSSGA and EA/G-LS on around half of the instances. Therefore it can be concluded that on average Sparrow produces the best solutions among all the algorithms.

Then we study how the performance gaps between Sparrow and other algorithms change with the different parameters of the instances. The performance gap between an algorithm A and Sparrow is evaluated by the following formula:

$$GapToSparrow_A = \frac{Gap_A - Gap_{Sparrow}}{Gap_{Sparrow}} \tag{8}$$

where Gap_A is the gap between the algorithm A and the upper bound and $Gap_{Sparrow}$ is the gap between Sparrow and the upper bound.

The performance gaps of different algorithms are calculated according to the different numbers of orders, τ and R and shown in Fig. 3.

Table 3
Gaps (%) of various algorithms on instances with 25 orders (multiple runs for each instance).[a]

n = 25		TS			DRGA			GA			HH			LOS			Sparrow		
τ	R	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
0.10	0.10	1	4	6	2	3	4	1	2	3	3	5	8	1	2	3	**0.72**	**1.82**	**2.69**
	0.30	2	3	6	1	2	6	1	2	5	2	4	9	**0**	2	5	**0.68**	**1.53**	**3.19**
	0.50	1	2	4	1	1	3	**0**	1	2	1	4	7	**0**	1	3	**0.00**	**0.71**	**1.60**
	0.70	**0**	1	4	**0**	1	3	**0**	**0**	2	**0**	2	5	**0**	**0**	2	**0.00**	**0.33**	**1.27**
	0.90	**0**	1	**2**	**0**	1	**2**	**0**	**0**	**2**	**0**	2	5	**0**	**0**	**2**	**0.00**	**0.34**	**2.08**
0.30	0.10	2	4	5	2	3	5	1	3	**4**	3	7	10	1	3	**4**	**0.89**	**2.31**	**4.19**
	0.30	3	5	7	2	3	5	**1**	3	**4**	4	7	10	2	3	5	**1.11**	**2.94**	5.06
	0.50	2	3	6	2	2	**3**	**0**	2	**3**	3	4	9	**0**	2	**3**	**0.86**	**1.65**	**3.49**
	0.70	1	2	6	1	2	5	**0**	**1**	5	2	4	10	**0**	**1**	5	**0.00**	**1.69**	**4.66**
	0.90	**0**	2	4	**0**	**1**	3	**0**	**1**	**2**	**0**	4	7	**0**	**1**	**2**	**0.00**	**1.06**	**2.79**
0.50	0.10	3	6	7	2	4	7	**1**	4	6	4	8	11	**1**	4	**5**	**1.68**	**3.87**	6.14
	0.30	3	5	9	2	5	8	2	4	**7**	4	8	13	**1**	4	**7**	**1.79**	**3.99**	**7.16**
	0.50	2	5	8	1	5	**6**	1	**4**	**6**	5	8	11	1	**4**	**6**	**0.97**	**4.32**	**6.31**
	0.70	2	6	11	2	**4**	**7**	**0**	**4**	**7**	2	7	11	**0**	**4**	**7**	**0.63**	**4.26**	**7.29**
	0.90	1	4	7	1	**3**	7	1	**3**	7	3	6	9	1	**3**	**6**	**0.85**	**3.10**	7.00
0.70	0.10	3	9	18	1	8	16	**0**	**7**	**14**	6	11	19	**0**	**7**	15	**0.92**	**7.46**	15.81
	0.30	7	10	14	**5**	9	13	**5**	**8**	**12**	10	12	16	**5**	**8**	**12**	**5.34**	**8.43**	**12.39**
	0.50	7	12	15	6	**10**	**14**	**5**	**10**	**14**	10	14	18	**5**	**10**	**14**	**5.41**	**10.11**	**14.02**
	0.70	2	8	14	2	7	**12**	**1**	**6**	**12**	**1**	9	14	**1**	**6**	**12**	**1.69**	**6.80**	**12.08**
	0.90	3	10	15	1	**8**	14	**0**	**8**	**13**	2	11	15	**0**	**8**	**13**	**0.65**	**8.21**	**13.63**
0.90	0.10	**0**	1	6	**0**	1	5	**0**	1	5	**0**	1	6	**0**	1	5	**0.00**	**0.49**	**4.92**
	0.30	**0**	1	1	**0**	1	1	**0**	**0**	**0**	**0**	2	7	**0**	**0**	**0**	**0.00**	**0.00**	**0.01**
	0.50	**0**	4	**12**	**0**	**2**	**12**	**0**	**2**	**12**	1	5	18	**0**	**2**	**12**	**0.00**	**2.56**	**12.30**
	0.70	1	8	25	1	7	21	**0**	**6**	21	**0**	9	22	**0**	**6**	21	**0.00**	**6.64**	**20.99**
	0.90	1	7	22	1	**6**	20	**0**	**6**	**19**	**0**	7	21	**0**	**6**	**19**	**0.00**	**6.02**	**19.16**
Avg.		2	5	9	1	3	8	**0**	**3**	**7**	2	6	11	**0**	**3**	**7**	**0.97**	**3.63**	**7.61**

[a] For the gaps of instances with 25 orders reported by Silva et al. (2018), we identify some mistakes because the gaps in their table are smaller than the gaps between their tight upper bounds and the upper bounds by Cesaret et al. (2012). Therefore we decide not to include ILS in this comparison.

Table 4
Gaps (%) of various algorithms on instances with 50 orders (multiple runs for each instance).[a]

n = 50		TS			DRGA			GA			HH			LOS			Sparrow		
τ	R	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
0.10	0.10	1	2	3	1	2	3	1	2	3	3	4	5	1	2	3	**0.51**	**0.79**	**1.27**
	0.30	1	2	4	1	2	3	1	2	4	2	4	5	1	2	3	**0.60**	**1.00**	**1.53**
	0.50	1	2	2	1	1	2	**0**	1	2	1	2	4	**0**	1	2	**0.00**	**0.42**	**0.96**
	0.70	**0**	2	**16**	**0**	2	**16**	**0**	2	**16**	**0**	3	17	**0**	2	**16**	**0.00**	**1.66**	**16.38**
	0.90	**0**	1	2	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	1	**0**	**0**	**0**	**0.00**	**0.00**	**0.00**
0.30	0.10	2	3	3	2	3	4	1	2	4	3	6	8	1	2	3	**0.63**	**1.59**	**2.86**
	0.30	3	4	5	2	3	4	2	3	4	5	6	9	2	3	4	**1.40**	**1.81**	**2.69**
	0.50	1	3	5	1	2	4	1	2	4	1	4	7	1	2	**3**	**0.18**	**1.44**	**3.64**
	0.70	**0**	1	3	**0**	1	2	**0**	**0**	**1**	1	3	5	**0**	1	2	**0.00**	**0.39**	**1.16**
	0.90	1	1	3	**0**	1	2	**0**	**0**	**1**	**0**	2	4	**0**	**0**	**1**	**0.00**	**0.27**	**1.32**
0.50	0.10	3	4	5	3	4	5	**1**	**2**	3	4	6	8	**1**	3	4	**1.11**	**2.01**	**2.64**
	0.30	3	6	8	3	5	7	2	**3**	6	6	8	11	2	4	6	**1.97**	**3.30**	**5.44**
	0.50	2	4	8	2	4	8	**1**	**3**	**6**	3	7	12	**1**	**3**	7	**1.03**	**3.00**	**6.49**
	0.70	2	3	5	1	3	6	**0**	2	**4**	2	5	9	**0**	2	**4**	**0.37**	**1.97**	**4.14**
	0.90	0	3	6	0	2	5	0	**1**	**4**	0	5	7	0	**1**	**4**	**−4.09**	**1.53**	**4.04**
0.70	0.10	4	7	9	4	5	7	**2**	4	5	7	9	11	**2**	4	5	**2.24**	**3.88**	**4.74**
	0.30	4	7	11	4	7	10	3	5	**8**	8	10	14	3	5	9	**2.75**	**4.82**	**8.40**
	0.50	7	9	13	6	8	12	**4**	**6**	11	8	11	15	5	**6**	11	**4.74**	**6.46**	**10.91**
	0.70	2	9	18	3	8	15	2	**7**	15	6	11	20	2	**7**	**14**	**1.73**	**7.13**	15.50
	0.90	4	10	18	4	9	14	**3**	**7**	**13**	5	10	16	**3**	**7**	**13**	**3.26**	**7.63**	**13.02**
0.90	0.10	8	13	18	8	12	19	**6**	11	**16**	10	14	21	7	11	**16**	**6.55**	**10.98**	**16.77**
	0.30	12	16	21	**9**	14	18	**9**	**13**	**17**	13	17	21	**9**	**13**	**17**	**9.28**	**13.43**	**17.80**
	0.50	7	16	20	4	13	19	**3**	**12**	17	5	16	20	**3**	**12**	**16**	**3.05**	**12.31**	**16.78**
	0.70	8	14	21	7	13	18	**5**	**11**	**17**	7	15	22	6	**11**	**17**	**5.62**	**11.50**	18.42
	0.90	11	14	19	10	13	17	**9**	**12**	**16**	12	16	20	10	**12**	**16**	**9.74**	**12.49**	**16.30**
Avg.		3	6	10	3	5	9	**2**	5	8	4	8	12	**2**	5	8	**2.11**	**4.47**	**7.73**

[a] For the gaps of instances with 50 orders reported by Silva et al. (2018), we identify some mistakes because the gaps in their table are smaller than the gaps between their tight upper bounds and the upper bounds by Cesaret et al. (2012). Therefore we decide not to include ILS in this comparison.

Table 5
Gaps (%) of various algorithms on instances with 100 orders (multiple runs for each instance).

n = 100		TS			DRGA			GA			HH			LOS			ILS-Avg			Sparrow		
τ	R	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
0.10	0.10	1	2	3	1	1	2	1	2	3	2	3	5	2	2	3	0.74	0.95	1.35	**0.39**	**0.57**	**0.80**
	0.30	2	2	3	1	1	2	2	2	3	1	3	4	1	2	3	0.44	0.74	1.09	**0.34**	**0.61**	**0.88**
	0.50	1	1	3	1	1	1	1	1	2	1	1	2	1	1	2	0.20	0.37	0.58	**0.00**	**0.13**	**0.28**
	0.70	**0**	1	1	**0**	1	1	**0**	**0**	**0**	**0**	**0**	2	**0**	**0**	1	**0.00**	0.04	0.12	**0.00**	**0.00**	**0.00**
	0.90	**0**	1	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0.00**	0.01	0.10	**0.00**	**0.00**	**0.00**
0.30	0.10	1	3	4	2	3	4	2	3	4	4	6	7	2	2	3	1.00	1.40	1.82	**0.53**	**0.99**	**1.44**
	0.30	2	3	5	1	3	4	1	2	4	2	5	7	1	3	4	0.73	1.38	2.67	**0.57**	**1.24**	**2.49**
	0.50	1	2	4	2	2	3	1	2	3	3	4	6	1	2	3	0.73	1.17	**1.65**	**0.65**	**1.07**	1.75
	0.70	1	2	3	1	1	1	**0**	1	2	1	2	4	**0**	1	2	0.27	0.44	**0.74**	**0.00**	**0.30**	0.79
	0.90	1	1	2	**0**	1	1	**0**	**0**	1	**0**	1	3	**0**	**0**	1	**0.00**	0.25	0.73	**0.00**	**0.11**	**0.42**
0.50	0.10	2	4	5	3	5	7	3	4	6	7	8	11	2	4	5	1.46	2.26	3.11	**1.07**	**1.79**	**2.71**
	0.30	3	4	6	4	4	6	3	4	5	6	7	9	2	3	5	1.72	2.32	3.08	**1.49**	**2.16**	**2.63**
	0.50	3	4	5	3	4	6	2	4	5	5	7	10	2	3	4	1.58	2.40	3.36	**1.31**	**2.33**	**3.35**
	0.70	2	3	4	1	3	4	**0**	2	3	2	5	7	**0**	2	3	0.49	1.61	2.80	**0.35**	**1.49**	**2.38**
	0.90	1	2	5	1	2	4	**0**	1	3	2	3	6	**0**	1	4	0.39	1.16	2.76	**0.19**	**0.92**	**2.40**
0.70	0.10	3	5	6	4	6	8	4	5	6	7	9	11	3	4	7	2.28	3.13	3.77	**1.69**	**2.47**	**3.21**
	0.30	4	7	11	4	6	9	3	5	8	6	9	13	2	5	7	1.89	3.86	5.82	**1.35**	**3.66**	**5.80**
	0.50	4	6	13	4	7	15	3	6	12	6	10	18	**2**	5	12	2.63	**4.25**	**8.69**[a]	**2.23**	4.34	10.65
	0.70	3	7	13	5	8	12	4	6	9	6	9	13	2	5	8	2.66	6.17	9.32	**1.92**	**4.81**	**7.33**
	0.90	5	8	13	4	7	10	4	6	9	5	9	12	3	**5**	8	3.92	6.60	8.27	**2.97**	**5.07**	**6.55**
0.90	0.10	7	9	12	6	9	12	5	7	9	8	11	14	4	6	9	4.40	7.02	9.07	**3.44**	**5.46**	**8.47**
	0.30	7	14	17	8	12	16	6	10	13	11	14	18	6	9	12	8.91	11.83	13.59	**4.90**	**8.65**	**11.20**
	0.50	11	16	18	11	14	19	8	12	17	12	16	20	8	11	16	10.12	14.06	18.40	**7.90**	**10.76**	**15.67**
	0.70	11	15	20	10	14	16	9	12	15	11	16	19	8	11	14	9.67	12.75	15.95	**7.38**	**10.73**	**13.02**
	0.90	11	16	22	8	13	19	6	12	17	8	15	22	4	11	18	7.21	13.23	18.43	**3.32**	**10.67**	**16.53**
Avg.		3	6	8	3	5	7	3	4	6	5	7	10	2	4	6	2.54	3.98	5.49	**1.76**	**3.21**	**4.83**

[a] This value reported by Silva et al. (2018) is smaller than the gaps between their tight upper bounds and the upper bounds by Cesaret et al. (2012).

Note that in Fig. 3 some performance gaps of LOS are smaller than 0, because LOS only reported rounded-down integer values. In general, Fig. 3 shows that gaps between the performance of Sparrow and those of other algorithms tend to grow when there are more orders, and when when τ and R are smaller. These are the cases found to be hard for Sparrow, but these cases are even harder for the other algorithms. Among other algorithms, HSSGA and EA/G-LS have the steadiest performance for instances with different parameters. Like Sparrow, HSSGA and EA/G-LS also work better than others when there are more orders, and when τ and R are smaller. On the contrary, GA and LOS work better when there

Table 6
Gaps (%) of various algorithms on instances with 25 orders (single run for each instance).

n = 25		ABC			HSSGA			EA/G-LS			Sparrow-Average			Sparrow-Best			Sparrow-Worst		
τ	R	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
0.10	0.10	1.45	2.70	3.85	1.09	2.08	3.16	1.09	2.16	3.16	**0.72**	**1.82**	**2.69**	0.72	1.67	2.69	0.72	2.07	3.15
	0.30	0.69	2.18	5.78	0.69	1.66	3.20	0.69	1.58	3.20	**0.68**	**1.53**	**3.19**	0.00	1.32	2.72	0.68	1.61	3.40
	0.50	**0.00**	1.38	2.45	**0.00**	0.90	1.90	**0.00**	0.91	1.77	**0.00**	**0.71**	**1.60**	0.00	0.64	1.41	0.00	0.84	1.84
	0.70	**0.00**	0.63	2.68	**0.00**	0.39	1.67	**0.00**	0.39	1.67	**0.00**	**0.33**	**1.27**	0.00	0.18	1.00	0.00	0.38	1.67
	0.90	**0.00**	0.35	2.09	**0.00**	0.35	2.09	**0.00**	0.35	2.09	**0.00**	**0.34**	**2.08**	0.00	0.26	2.03	0.00	0.37	2.09
0.30	0.10	1.21	3.11	5.24	1.21	2.48	4.55	1.21	2.44	4.55	**0.89**	**2.31**	**4.19**	0.68	2.17	3.84	1.71	2.55	4.54
	0.30	1.12	3.37	6.04	1.12	3.12	**4.62**	2.15	3.46	6.04	**1.11**	**2.94**	5.06	1.11	2.55	4.33	1.11	3.18	5.28
	0.50	1.12	2.40	5.24	1.12	**1.52**	3.50	2.15	1.85	3.50	**0.86**	1.65	**3.49**	0.66	1.48	3.49	0.86	1.87	3.49
	0.70	0.88	1.97	5.32	**0.00**	**1.61**	**4.61**	**0.00**	1.90	5.32	**0.00**	1.69	4.66	0.00	1.32	4.10	0.00	1.94	5.31
	0.90	**0.00**	1.27	3.38	**0.00**	**1.05**	2.82	**0.00**	1.10	2.82	**0.00**	1.06	**2.79**	0.00	0.84	2.73	0.00	1.23	2.82
0.50	0.10	2.80	4.88	6.76	**1.68**	4.17	6.76	**1.68**	4.11	6.76	**1.68**	**3.87**	**6.14**	1.60	3.59	5.61	1.68	4.38	6.75
	0.30	2.85	4.49	7.29	1.82	4.22	7.29	1.82	4.12	7.29	**1.79**	**3.99**	**7.16**	1.68	3.85	7.08	1.81	4.25	7.29
	0.50	1.94	4.48	**6.02**	**0.97**	4.28	6.08	**0.97**	**4.25**	6.08	**0.97**	4.32	6.31	0.97	4.08	6.01	0.97	4.67	7.14
	0.70	0.64	4.20	**7.17**	0.64	**4.16**	**7.17**	0.64	4.19	**7.17**	**0.63**	4.26	7.29	0.63	4.16	7.17	0.63	4.74	8.36
	0.90	1.03	3.37	7.00	1.03	**3.05**	7.00	1.03	3.13	**6.79**	**0.85**	3.10	7.00	0.68	2.97	7.00	1.02	3.42	7.00
0.70	0.10	0.93	7.72	15.88	0.93	**7.43**	**14.86**	0.93	7.53	16.22	**0.92**	7.46	15.81	0.92	7.35	14.86	0.92	7.56	16.55
	0.30	**5.05**	8.64	12.40	5.78	8.49	12.40	5.78	8.53	12.50	5.34	**8.43**	**12.39**	5.05	8.33	12.39	5.77	8.61	13.30
	0.50	**5.37**	10.45	14.03	**5.37**	**10.05**	14.03	**5.37**	10.17	14.03	5.41	10.11	**14.02**	5.37	10.01	14.02	5.78	10.55	14.14
	0.70	1.69	6.90	**12.08**	**1.58**	6.56	**12.08**	**1.58**	**6.47**	**12.08**	1.69	6.80	**12.08**	1.69	6.69	12.08	1.69	7.18	12.08
	0.90	**0.01**	**8.12**	13.64	**0.01**	**8.12**	13.64	**0.01**	8.27	13.64	0.65	8.21	**13.63**	0.00	8.06	13.63	1.19	8.51	13.63
0.90	0.10	**0.00**	0.59	4.93	**0.00**	**0.49**	4.93	**0.00**	**0.49**	4.93	**0.00**	**0.49**	**4.92**	0.00	0.49	4.92	0.00	0.49	4.92
	0.30	**0.00**	0.16	1.48	**0.00**	**0.00**	**0.01**	**0.00**	**0.00**	**0.01**	**0.00**	**0.00**	**0.01**	0.00	0.00	0.00	0.00	0.01	0.10
	0.50	**0.00**	2.58	**12.30**	**0.00**	**2.49**	**12.30**	**0.00**	**2.49**	**12.30**	**0.00**	2.56	**12.30**	0.00	2.49	12.30	0.00	2.72	12.30
	0.70	**0.00**	6.81	**20.99**	**0.00**	6.67	**20.99**	**0.00**	6.67	**20.99**	**0.00**	**6.64**	**20.99**	0.00	6.61	20.99	0.00	6.75	20.99
	0.90	**0.00**	6.09	19.33	**0.00**	**5.99**	**19.10**	**0.00**	**5.99**	**19.10**	**0.00**	6.02	19.16	0.00	5.99	19.09	0.00	6.05	19.33
Avg.		1.15	3.95	8.13	1.00	3.65	7.63	1.08	3.70	7.76	**0.97**	**3.63**	**7.61**	0.87	3.48	7.42	1.06	3.84	7.90

Table 7
Gaps (%) of various algorithms on instances with 50 orders (single run for each instance).

n = 50		ABC			HSSGA			EA/G-LS			Sparrow-Average			Sparrow-Best			Sparrow-Worst		
τ	R	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
0.10	0.10	1.20	1.89	2.81	**0.51**	1.18	1.69	**0.51**	1.08	1.47	**0.51**	**0.79**	**1.27**	0.37	0.64	0.99	0.51	0.94	1.59
	0.30	1.06	1.77	2.33	0.71	1.12	1.57	0.71	1.10	1.57	**0.60**	**1.00**	**1.53**	0.45	0.90	1.53	0.70	1.08	1.53
	0.50	0.34	0.96	2.03	**0.00**	0.43	**0.84**	**0.00**	0.45	0.95	**0.00**	**0.42**	0.96	0.00	0.33	0.83	0.00	0.56	1.25
	0.70	**0.00**	1.90	16.39	**0.00**	1.73	16.39	**0.00**	**1.66**	16.39	**0.00**	**1.66**	**16.38**	0.00	1.63	16.38	0.00	1.78	16.38
	0.90	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	0.00	0.00	0.00	0.00	0.00	0.00
0.30	0.10	1.63	2.59	3.77	1.02	1.80	**2.79**	1.22	1.93	2.86	**0.63**	**1.59**	2.86	0.60	1.30	2.48	0.81	1.85	3.24
	0.30	1.94	2.98	4.49	1.55	2.06	**2.66**	1.54	2.18	3.05	**1.40**	**1.81**	2.69	1.15	1.63	2.51	1.54	2.13	3.23
	0.50	1.94	1.88	3.88	1.55	1.51	3.88	1.54	1.53	3.68	**0.18**	**1.44**	**3.64**	0.00	1.23	3.10	0.37	1.63	4.26
	0.70	**0.00**	0.75	1.56	**0.00**	0.39	**1.03**	**0.00**	**0.35**	1.04	**0.00**	0.39	1.16	0.00	0.24	1.03	0.00	0.66	1.55
	0.90	**0.00**	0.50	1.83	**0.00**	0.29	1.21	**0.00**	**0.23**	**1.02**	**0.00**	0.27	1.32	0.00	0.16	1.01	0.00	0.35	1.62
0.50	0.10	1.71	3.07	4.17	**1.11**	2.20	3.20	**1.11**	2.23	3.38	**1.11**	**2.01**	**2.64**	0.76	1.78	2.43	1.11	2.15	3.05
	0.30	3.08	4.45	6.11	2.26	3.43	**5.15**	2.26	3.53	5.73	**1.97**	**3.30**	5.44	1.64	3.01	5.15	2.25	3.78	5.72
	0.50	2.07	3.78	6.82	**1.03**	3.04	**6.33**	**1.03**	3.17	7.14	**1.03**	**3.00**	6.49	0.83	2.77	6.16	1.03	3.39	7.14
	0.70	0.76	2.28	4.24	**0.36**	2.00	**3.87**	0.38	2.03	4.24	0.37	**1.97**	4.14	0.36	1.67	3.86	0.37	2.22	4.41
	0.90	3.61	1.71	**4.04**	0.00	1.54	4.31	−4.02	1.57	**4.04**	**−4.09**	**1.53**	**4.04**	−4.41	1.17	4.04	−4.01	1.91	4.04
0.70	0.10	3.16	4.57	5.79	2.42	4.17	5.20	2.42	4.18	5.39	**2.24**	**3.88**	**4.74**	1.85	3.58	4.45	2.41	4.13	5.19
	0.30	3.51	5.74	9.36	3.01	5.02	**8.30**	3.17	5.05	**8.30**	**2.75**	**4.82**	8.40	2.47	4.46	8.12	3.00	5.12	9.18
	0.50	5.30	7.09	11.39	5.08	6.71	11.03	5.17	6.70	11.03	**4.74**	**6.46**	**10.91**	4.42	6.10	10.32	5.08	6.88	11.28
	0.70	1.89	7.56	15.26	1.72	7.18	15.38	**1.37**	**7.09**	**14.31**	1.73	7.13	15.50	1.37	6.69	15.38	2.40	7.63	15.81
	0.90	3.69	8.02	13.67	**2.91**	7.69	13.38	3.30	**7.62**	**13.00**	3.26	7.63	13.02	2.91	7.28	13.00	3.68	7.95	13.10
0.90	0.10	7.30	11.33	**16.77**	**6.55**	**10.97**	**16.77**	7.30	11.15	**16.77**	**6.55**	10.98	**16.77**	6.55	10.87	16.77	6.55	11.23	16.77
	0.30	**9.28**	13.77	17.80	**9.28**	13.49	17.60	**9.28**	13.47	**17.40**	**9.28**	**13.43**	17.80	9.28	13.27	17.60	9.28	13.57	18.00
	0.50	3.28	12.77	17.29	3.28	12.33	16.88	3.46	12.45	**16.73**	**3.05**	**12.31**	16.78	2.62	12.16	16.51	3.64	12.67	17.65
	0.70	5.81	11.53	**17.85**	**5.62**	11.48	**17.85**	**5.62**	**11.39**	**17.85**	**5.62**	11.50	18.42	5.62	11.26	17.84	5.62	11.74	19.03
	0.90	10.02	12.38	16.27	**9.57**	12.37	**15.92**	9.58	**12.30**	16.09	9.74	12.49	16.30	9.40	12.22	16.10	10.01	12.81	16.46
Avg.		2.90	5.01	8.24	2.38	4.57	7.73	2.28	4.58	7.74	**2.11**	**4.47**	**7.73**	1.93	4.25	7.50	2.25	4.73	8.06

are fewer orders, and when τ and R are larger.

Last but not least, we discuss the ability of Sparrow to escape from local optima. In the proposed algorithm, the following techniques are used: (1) The combination of BRKGA and ALNS helps ALNS to search more solutions in the space more efficiently. The ALNS search starts from multiple points (i.e., multiple randomly-generated individuals) in the solution space; (2) In the BRKGA part, in each iteration, 10% of total population (i.e., mutant individuals) are generated randomly. This method helps the algorithm to keep the diversity of the population and an escape occurs when the mutant or the offspring is better than the local optimum; (3) In the ALNS part, there is a random removal operator which helps the algorithm search broadly. Besides, a simulated annealing criterion is used to accept a worse solution with some probability, which helps the algorithm to escape from the entrapment of a local optimum.

To test the ability of the algorithm to get out of local optima, there

Table 8
Gaps (%) of various algorithms on instances with 100 orders (single run for each instance).

n = 100		ABC			HSSGA			EA/G-LS			Sparrow-Average			Sparrow-Best			Sparrow-Worst		
τ	R	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
0.10	0.10	1.11	1.75	2.21	0.40	0.77	1.32	0.72	0.94	1.22	**0.39**	**0.57**	**0.80**	0.19	0.46	0.65	0.54	0.69	0.85
	0.30	0.90	1.35	1.71	0.45	0.71	0.98	0.44	0.74	1.06	**0.34**	**0.61**	**0.88**	0.18	0.50	0.81	0.45	0.75	1.01
	0.50	0.36	0.70	1.23	**0.00**	0.19	0.38	**0.00**	0.20	0.37	**0.00**	**0.13**	**0.28**	0.00	0.07	0.18	0.00	0.20	0.37
	0.70	**0.00**	0.09	0.35	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	0.00	0.00	0.00	0.00	0.00	0.00
	0.90	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	0.00	0.00	0.00	0.00	0.00	0.00
0.30	0.10	1.27	2.11	2.71	**0.50**	1.18	1.71	0.69	1.25	1.71	0.53	**0.99**	**1.44**	0.39	0.83	1.35	0.58	1.09	1.53
	0.30	1.40	2.12	3.97	0.66	1.27	2.64	0.66	1.41	2.64	**0.57**	**1.24**	**2.49**	0.56	1.06	2.23	0.74	1.38	2.64
	0.50	1.40	1.67	2.45	0.66	**1.07**	**1.52**	0.66	1.11	1.69	**0.65**	**1.07**	1.75	0.45	0.91	1.52	0.81	1.22	2.03
	0.70	**0.00**	0.63	1.16	**0.00**	0.35	0.85	**0.00**	**0.24**	**0.63**	**0.00**	0.30	0.79	0.00	0.17	0.62	0.00	0.38	0.96
	0.90	**0.00**	0.26	0.82	**0.00**	**0.10**	0.56	**0.00**	0.14	0.47	**0.00**	0.11	**0.42**	0.00	0.05	0.28	0.00	0.20	0.64
0.50	0.10	2.50	3.37	4.33	1.22	2.09	2.99	1.69	2.31	3.27	**1.07**	**1.79**	**2.71**	0.81	1.57	2.47	1.26	2.03	2.91
	0.30	2.46	3.01	4.00	1.54	2.29	2.95	1.67	2.43	2.96	**1.49**	**2.16**	**2.63**	1.14	1.94	2.51	1.53	2.45	3.15
	0.50	2.26	3.13	4.21	**1.22**	2.36	**3.35**	1.48	2.51	3.61	1.31	**2.33**	**3.35**	1.13	2.12	3.17	1.39	2.66	3.69
	0.70	0.77	1.82	2.99	0.51	1.49	2.33	0.51	**1.32**	**2.24**	**0.35**	1.49	2.38	0.34	1.26	2.06	0.51	1.70	2.78
	0.90	0.53	1.42	3.13	**0.18**	1.04	**2.15**	**0.18**	**0.90**	**2.15**	0.19	0.92	2.40	0.00	0.73	2.23	0.35	1.11	2.77
0.70	0.10	3.20	3.99	4.84	2.09	2.84	3.53	2.27	2.98	3.82	**1.69**	**2.47**	**3.21**	1.45	2.25	3.13	1.90	2.73	3.52
	0.30	2.37	4.72	7.16	1.71	3.88	6.07	1.67	4.09	6.26	**1.35**	**3.66**	**5.80**	1.14	3.35	5.43	1.66	3.95	6.07
	0.50	3.27	5.27	11.94	2.25	4.62	10.95	2.62	4.69	10.95	**2.23**	**4.34**	**10.65**	1.97	4.06	10.44	2.44	4.60	11.02
	0.70	2.77	5.37	7.37	2.18	5.16	7.78	2.77	5.39	7.65	**1.92**	**4.81**	**7.33**	1.76	4.46	7.00	2.35	5.19	7.74
	0.90	3.57	5.90	8.33	3.73	5.48	7.49	3.15	5.34	6.88	**2.97**	**5.07**	**6.55**	2.75	4.72	6.16	3.26	5.49	6.95
0.90	0.10	4.70	6.62	9.44	4.57	6.27	9.06	4.75	6.45	9.34	**3.44**	**5.46**	**8.47**	3.26	5.21	8.28	3.63	5.72	8.76
	0.30	5.39	9.40	11.86	5.11	9.03	11.25	5.39	9.09	12.35	**4.90**	**8.65**	**11.20**	4.60	8.40	10.96	5.29	8.98	11.82
	0.50	8.82	11.88	17.25	8.60	11.26	16.29	8.28	11.20	15.84	**7.90**	**10.76**	**15.67**	7.52	10.47	15.52	8.32	11.01	15.86
	0.70	7.96	11.19	**12.85**	8.07	11.23	13.07	8.05	11.07	13.06	**7.38**	**10.73**	13.02	7.10	10.41	12.54	7.71	11.14	13.43
	0.90	4.11	11.31	16.83	3.57	11.07	16.60	4.07	11.02	16.73	**3.32**	**10.67**	**16.53**	3.17	10.41	16.17	3.47	10.92	16.93
Avg.		2.44	3.96	5.73	1.97	3.43	5.03	2.07	3.47	5.08	**1.76**	**3.21**	**4.83**	1.60	3.02	4.63	1.93	3.42	5.10

are usually two methods: (1) for instances for which the global optimum is known, check whether the algorithm finds this optimum; (2) for instances for which the optimum is not known, calculate the gap to the upper bound and see how small it is. For the first method, Table 9 shows the number of instances with known optimum and the number of instances where Sparrow also finds the optimum for each group of ten instances. For 86.4% of all the 353 instances with known optimum, the proposed Sparrow algorithm also finds the optimum. Since we do not have the detailed result for each individual instance of other algorithms, it is impossible to compare this with other algorithms. Regarding the instances for which the optimum is not found, Tables 3–8 show that Sparrow has smaller gaps. From the above analysis, it can be concluded that the proposed Sparrow algorithm has a better ability to get out of local optima compared with state-of-the-art algorithms.

4.3. Problem properties

In this section, we aim to understand the problem further by studying how different properties (other than n, τ and R) of the problem correlate with its difficulty. To achieve this, the performance of Sparrow on instances with different properties is evaluated, because according to the comparison with other algorithms above, Sparrow is the new state of the art for this problem. In this section, the same benchmark as in Sections 4.1 and 4.2 and the tight upper bounds provided by Silva et al. (2018) are used. As in Fig. 2, we select the instances with 25 orders in order to decrease the influence from the upper bounds, because the upper bounds for instances with 25 orders are tighter than those for instances with 50 and 100 orders.

In this section, the following properties of each problem instance are calculated: standard deviation of setup times; standard deviation of window length; standard deviation of revenue of orders; the length of horizon; average window length; congestion ratio (this value is calculated by adding up the processing time and the smallest setup time of each order, divided by the horizon, to evaluate how congested the instance is); standard deviation of order processing time; average conflict ratio (the conflict ratio of an order is calculated by adding up the overlap time between its time window and the time windows of all other orders, divided by the length of its time window); standard deviation of conflict ratio; setup window ratio (this value is the average setup time divided by the average length of window); process window ratio (this value is the average processing time divided by the average length of window); correlation coefficient between processing time and revenue of orders.

Fig. 4 shows how the performance of Sparrow changes with six

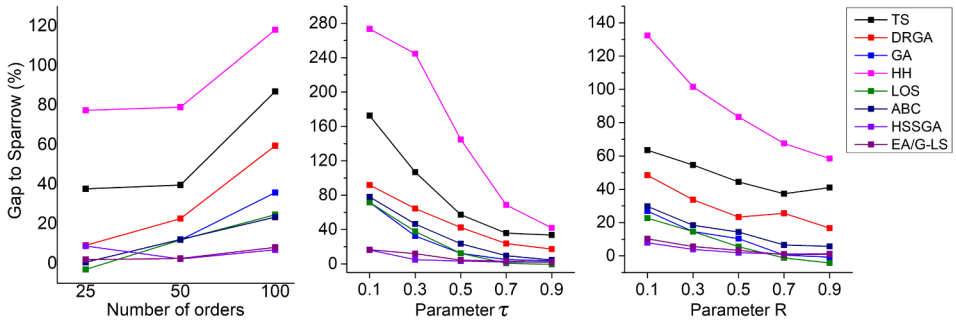

Fig. 3. The gaps of different algorithms to Sparrow. We do not plot the results of ILS in this figure because some gaps reported by Silva et al. (2018) are smaller than the gaps between their tight upper bounds and the upper bounds by Cesaret et al. (2012).

Table 9
The number of instances with known optimum (#Opt) for each group of ten instances and the number of instances where Sparrow finds the optimum (#Sparrow∗).

n, τ, R	#Opt	#Sparrow∗	n, τ, R	#Opt	#Sparrow∗	n, τ, R	#Opt	#Sparrow∗
25, 0.1, 0.1	3	3	50, 0.1, 0.1	0	0	100, 0.1, 0.1	0	0
25, 0.1, 0.3	4	1	50, 0.1, 0.3	0	0	100, 0.1, 0.3	0	0
25, 0.1, 0.5	7	4	50, 0.1, 0.5	3	3	100, 0.1, 0.5	6	6
25, 0.1, 0.7	10	9	50, 0.1, 0.7	10	10	100, 0.1, 0.7	10	10
25, 0.1, 0.9	10	9	50, 0.1, 0.9	10	10	100, 0.1, 0.9	10	10
25, 0.3, 0.1	6	4	50, 0.3, 0.1	0	0	100, 0.3, 0.1	0	0
25, 0.3, 0.3	3	3	50, 0.3, 0.3	0	0	100, 0.3, 0.3	0	0
25, 0.3, 0.5	3	3	50, 0.3, 0.5	1	1	100, 0.3, 0.5	0	0
25, 0.3, 0.7	9	7	50, 0.3, 0.7	6	6	100, 0.3, 0.7	5	5
25, 0.3, 0.9	8	6	50, 0.3, 0.9	8	8	100, 0.3, 0.9	8	8
25, 0.5, 0.1	10	10	50, 0.5, 0.1	0	0	100, 0.5, 0.1	0	0
25, 0.5, 0.3	8	7	50, 0.5, 0.3	1	0	100, 0.5, 0.3	0	0
25, 0.5, 0.5	8	7	50, 0.5, 0.5	0	0	100, 0.5, 0.5	0	0
25, 0.5, 0.7	9	9	50, 0.5, 0.7	0	0	100, 0.5, 0.7	0	0
25, 0.5, 0.9	9	7	50, 0.5, 0.9	1	1	100, 0.5, 0.9	1	1
25, 0.7, 0.1	10	9	50, 0.7, 0.1	0	0	100, 0.7, 0.1	0	0
25, 0.7, 0.3	10	9	50, 0.7, 0.3	1	0	100, 0.7, 0.3	0	0
25, 0.7, 0.5	10	9	50, 0.7, 0.5	1	1	100, 0.7, 0.5	0	0
25, 0.7, 0.7	10	6	50, 0.7, 0.7	7	3	100, 0.7, 0.7	0	0
25, 0.7, 0.9	10	10	50, 0.7, 0.9	7	3	100, 0.7, 0.9	0	0
25, 0.9, 0.1	10	10	50, 0.9, 0.1	10	10	100, 0.9, 0.1	0	0
25, 0.9, 0.3	10	10	50, 0.9, 0.3	10	8	100, 0.9, 0.3	0	0
25, 0.9, 0.5	10	10	50, 0.9, 0.5	10	9	100, 0.9, 0.5	0	0
25, 0.9, 0.7	10	10	50, 0.9, 0.7	10	5	100, 0.9, 0.7	0	0
25, 0.9, 0.9	10	10	50, 0.9, 0.9	10	5	100, 0.9, 0.9	0	0
Total.	207	182		106	83		40	40

properties. The performance with other properties is not shown because the correlations are weak, which shows that Sparrow adapts it well for these properties. According to Fig. 4, the problem becomes harder for Sparrow when: (1) the average conflict ratio is larger; (2) the average window length is larger. This is consistent with the observation in Fig. 2, because when τ is smaller, the average window length is larger; (3) the congestion ratio is larger; (4) the process window ratio is smaller. This case is harder because the space where an order can be shifted within a window is larger. The solution space becomes larger; (5) the setup window ratio is smaller; (6) the correlation coefficient between processing time and revenue is larger. The reason is that when the correlation is larger, the difference among order values is smaller, therefore it is more difficult to select orders.

According to the observations above, it can be concluded that the following properties make the problem harder:

1. Congestion-related properties: larger congestion ratio; larger conflict ratio.
2. Order-related properties: larger correlation between processing time and revenue of orders.
3. Window-related properties: longer time windows; smaller process/setup window ratio.

It is interesting to see that each of the three hard cases corresponds to a real-life situation. The satellite scheduling problem usually has the problem of the scheduling horizon being heavily contested; many real-life situations such as commerce cause longer orders to be more expensive, where the processing time and revenue of an order are more correlated; cases such as the travelling repairman problem involve mostly short processing times, while having relatively longer time windows.

Although the benchmark set by Cesaret et al. (2012) is known to contain difficult instances, the three real-life and difficult scenarios above do not correspond to the instances generated with this methodology. Therefore in the next section, we generate three new sets of instances, each corresponding to one of the scenarios, to compare the performances of different algorithms.

4.4. The performances of different algorithms on new instances

In this section, we evaluate the state-of-the-art algorithms on three new sets: The *satellite scheduling* set is more congested by setting the parameters as follows: $n = \{100, 150, 200, 250, 300\}$, $\tau = 0.1$, and $R = 0.1$; The *commerce* set has a more correlated processing time and revenue by generating a random value γ in [1,20] and calculating the final revenue of order o_i by $((1-q)\gamma + 2q \cdot t_i)$, and setting $q = \{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$, $n = 100$, $\tau = 0.1$, and $R = 0.1$; Finally, the *travelling repairman* set has long time windows whereas the processing times are short, by multiplying t_T with a constant $c = \{1.0, 1.2, 1.4, 1.6, 1.8, 2.0\}$, and $n = 100$, $\tau = 0.1$, and $R = 0.1$. Since multiplying t_T by c makes the instance less congested, $n \cdot c$ orders are generated to the keep the congestion ratio at a similar level. For each set ten instances with the same parameters are generated. Therefore there are $5 \cdot 10 + 6 \cdot 10 + 6 \cdot 10 = 170$ new instances in total.

Besides Sparrow, we include two other algorithms which correspond to the state of the art: ILS by Silva et al. (2018) and HSSGA by Chaurasia and Singh (2017). As obtaining the original implementations of these algorithms proved to be infeasible, we reimplemented the algorithms according to their descriptions with varying degrees of success. For our implementation of ILS the gaps are too dissimilar to the gaps reported for the benchmark instances provided by Cesaret et al. (2012) to argue reproduction of their algorithm. Therefore we refer to our implementation as ILS∗. For HSSGA using proposed parameters our gap was found to be at most 1% larger than the values reported by Chaurasia and Singh (2017), with a similar count of instances solved to optimality for the same set of benchmark instances. We argue that in the latter case the implementation is at least globally representative of the performance of HSSGA as described in the article.

All the three algorithms are run on Intel Core i5-3470 3.20 GHz CPU with 8 GB memory, using a single core. Sparrow uses the same parameters as mentioned in Section 4.2 and the other two algorithms use the same parameters as mentioned in articles (Chaurasia & Singh, 2017; Silva et al., 2018).

We do not have the upper bounds of the new instances. Therefore the gaps between the results of HSSGA and ILS∗ and the results of Sparrow are calculated. The gaps of HSSGA and ILS∗, as well as the CPU

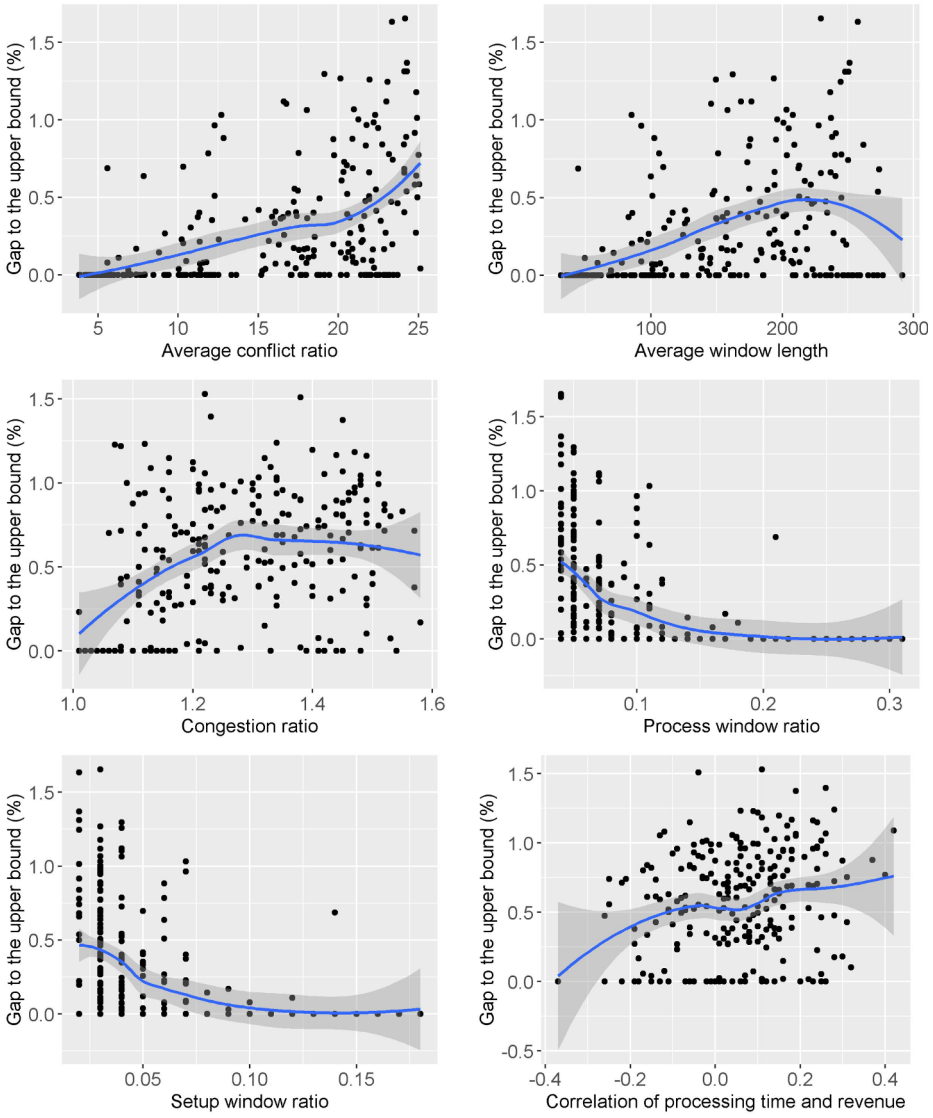

Fig. 4. The correlation of the performance of Sparrow and different properties of the problem. Each point is the average of 10 runs of each instance with 25 orders.

times of the three algorithms on the three new instance sets are shown in Figs. 5–7. Each point is the average result of 10 runs. Since the runtime of ILS∗ turned out to be too long for large instances, a time limit of 3600s is used.

On the satellite scheduling set, the gap between Sparrow and ILS∗ increases when there are more orders, which shows that ILS∗ cannot handle larger instances well. The gap between Sparrow and HSSGA does not change much. This shows that both Sparrow and HSSGA have a stronger ability to solve instances with larger congestion ratio efficiently. For instances with 150 and 300 orders, HSSGA performs better than Sparrow (with the gap smaller than 0). Regarding CPU times, ILS∗ shows a weak scalability. There is decline at order size 300, because many instances in order size 300 were stopped by the time limit, resulting in a lower average CPU time. Sparrow uses the least time. Both HSSGA and Sparrow algorithm are based on a hybridization of a population-based method and a local search method. The two algorithms

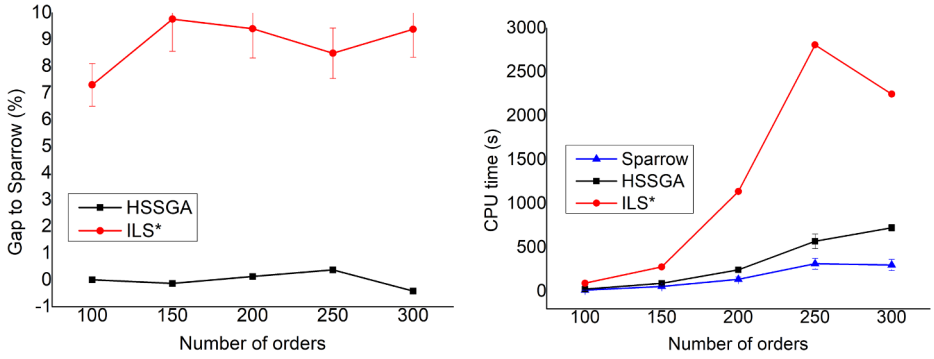

Fig. 5. The gaps of HSSGA and ILS∗ to Sparrow (Left), and the CPU times of three algorithms on the satellite scheduling set (right).

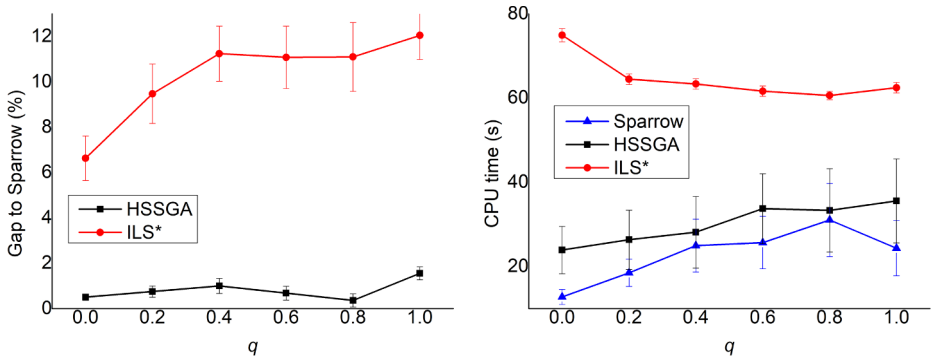

Fig. 6. The gaps of HSSGA and ILS∗ to Sparrow (Left), and the CPU times of three algorithms on the commerce set (right).

integrate the exploration ability of the population-based method and the exploitation ability of the local search method well, making them adapt well for instances with varying sizes.

On the commerce set, the gaps of ILS∗ and HSSGA both increases when q is larger (i.e., when revenue and processing time are more correlated), which shows that Sparrow can solve instances with correlated processing time and revenue well. Sparrow uses the least time and produces the best solutions. Sparrow performs well on this set of instance mainly because of the self-adaption ability of the ALNS component: ALNS uses multiple neighbourhood operators and can choose the most efficient one according to the instances. When the processing time and revenue of orders are more correlated, the operators based on the unit revenue would become less efficient to select orders. In this case Sparrow chooses other operators and the correlation between processing time and revenue does not influence the performance of Sparrow much.

Last, on the travelling repairman set, the changes of the gaps of the two algorithms are not clear. However it is obvious that Sparrow uses the least time and produces the best solutions. We believe this is because of the fast insertion algorithm with the time slack strategy, which provides a good flexibility for the orders to shift in the long time windows.

As a summary, Sparrow has a relatively high ability to handle harder and real-life cases, especially when revenue and processing time are more correlated, and the time window is very long. The HSSGA algorithm performs well on large instances in terms of optimality gap, although taking more time than Sparrow. The ILS∗ algorithm performs worst on these harder instances, and has the worst scalability.

5. Conclusions

This article studied the Order Acceptance and Scheduling (OAS) problem with sequence-dependent setup times and time windows. We proposed a novel memetic algorithm called Sparrow, a hybridization of the biased random key genetic algorithm (BRKGA) and adaptive large neighbourhood search (ALNS). We introduced a new bounded-width gene encoding strategy, a hybrid decoding method, an intelligent crossover operator, several neighbourhood operators, and a fast insertion algorithm, making Sparrow suitable for problems with varying properties.

Sparrow is tested on a set of standard benchmark with 750 instances with 25–100 orders. Compared with state-of-the-art algorithms, Sparrow obtains better-quality solutions with comparable running time. The gaps between Sparrow and other algorithms tend to increase when the problem instances get more difficult. We further study this problem by analyzing the correlation between problem properties and the algorithm performance and find that the congestion ratio of the instance, the length of time windows and the correlation between revenue and processing time are the key properties influencing the difficulty of the problem. Finally, we generate new instances with up to 300 orders, longer time windows and more correlated processing time and revenue, and compare the performances of different algorithms on these new realistic instances to understand their strengths and weaknesses. It is found that Sparrow has a good ability to solve these difficult instances and the HSSGA algorithm from Chaurasia and Singh (2017) performs well on large instances in terms of optimality gap, although taking more time than Sparrow. We believe that the integration of the exploration ability of population-based methods and the exploitation ability of local search methods in Sparrow and HSSGA, and the varying neighbourhood operators and the fast insertion strategy in Sparrow make them successful on the difficult instances.

Our next steps are to further improve the performance of Sparrow on large instances, such as by proposing a better intelligent crossover operator which keeps longer sub-sequences of high quality efficiently. Moreover, we will apply it to other real-world problem domains such as the agile satellite observation scheduling problem which has time-dependent revenue, time-dependent setup times and multiple machines. One challenge of using Sparrow is to determine the population size to balance exploration and exploitation. Therefore, in our future works we are also interested in studying a dynamic Sparrow which can tune this

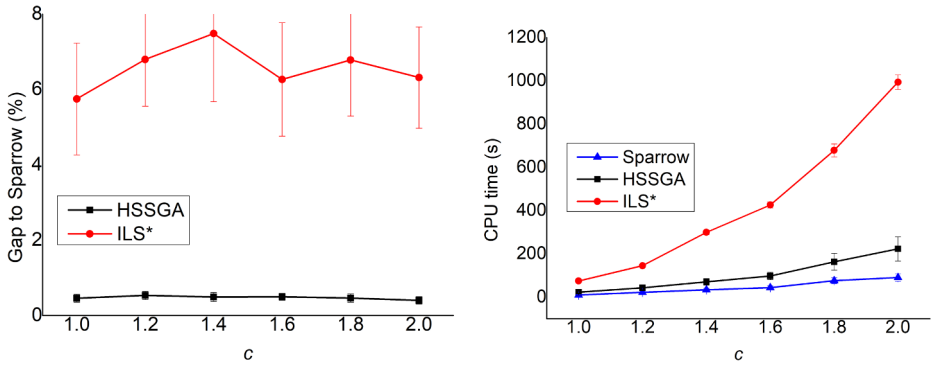

Fig. 7. The gaps of HSSGA and ILS∗ to Sparrow (Left), and the CPU times of three algorithms on the travelling repairman set (right).

automatically, and investigating the performance of other alternatives on our problem, such as the differential evolution algorithm by Wang, Liu, Long, Zhang, and Yang (2017), where a new encoding strategy is used to achieve this balance.

Acknowledgements

We gratefully thank Dr. Yuri Laio T.V. Silva and Dr. Anand Subramanian for providing the tight upper bound of each instance. This work is supported by China Scholarship Council for Lei He's visit at Delft University of Technology, the Netherlands (Grant No. 201703170269), and China Hunan Provincial Innovation Foundation for Postgraduate (Grant No. CX2018B020). We gratefully thank the reviewers for their valuable comments.

References

Asgari, N., Rajabi, M., Jamshidi, M., Khatami, M., & Farahani, R. Z. (2017). A memetic algorithm for a multi-objective obnoxious waste location-routing problem: a case study. *Annals of Operations Research, 250*(2), 279–308.

Bontoux, B., Artigues, C., & Feillet, D. (2010). A memetic algorithm with a large neighborhood crossover operator for the generalized traveling salesman problem. *Computers & Operations Research, 37*(11), 1844–1852.

Cesaret, B., Oğuz, C., & Salman, F. S. (2012). A tabu search algorithm for order acceptance and scheduling. *Computers & Operations Research, 39*(6), 1197–1205.

Chaurasia, S. N., & Kim, J. H. (2019). An artificial bee colony based hyper-heuristic for the single machine order acceptance and scheduling problem. *Decision science in action* (pp. 51–63). Springer.

Chaurasia, S. N., & Singh, A. (2017). Hybrid evolutionary approaches for the single machine order acceptance and scheduling problem. *Applied Soft Computing, 52*, 725–747.

Chaves, A. A., Gonçalves, J. F., & Lorena, L. A. N. (2018). Adaptive biased random-key genetic algorithm with local search for the capacitated centered clustering problem. *Computers & Industrial Engineering, 124*, 331–346.

Chen, C., Yang, Z., Tan, Y., & He, R. (2014). Diversity controlling genetic algorithm for order acceptance and scheduling problem. *Mathematical Problems in Engineering, 2014* Article ID 367152.

Demir, E., Bektaş, T., & Laporte, G. (2012). An adaptive large neighborhood search heuristic for the pollution-routing problem. *European Journal of Operational Research, 223*(2), 346–359.

Ghosh, J. B. (1997). Job selection in heavily loaded shop. *Computers and Operations Research, 24*(2), 141–145.

Gonçalves, J. F., & De Almeida, J. R. (2002). A hybrid genetic algorithm for assembly line balancing. *Journal of Heuristics, 8*(6), 629–642.

Gonçalves, J. F., & Resende, M. G. (2011). Biased random-key genetic algorithms for combinatorial optimization. *Journal of Heuristics, 17*(5), 487–525.

Krasnogor, N., & Smith, J. (2005). A tutorial for competent memetic algorithms: model, taxonomy and design issues. *IEEE Transactions on Evolutionary Computation, 9*(5), 474–488.

Li, Y., Chen, H., & Prins, C. (2016). Adaptive large neighborhood search for the pickup and delivery problem with time windows, profits, and reserved requests. *European Journal of Operational Research, 252*(1), 27–38.

Lin, S.-W., & Ying, K. (2013). Increasing the total net revenue for single machine order acceptance and scheduling problems using an artificial bee colony algorithm. *Journal of the Operational Research Society, 64*(2), 293–311.

Liu, X., Laporte, G., Chen, Y., & He, R. (2017). An adaptive large neighborhood search metaheuristic for agile satellite scheduling with time-dependent transition time. *Computers & Operations Research, 86*, 41–53.

Moscato, P. (1989). On evolution, search, optimization, gas and martial arts: toward memetic algorithms. Tech. rep. California Institute Technology, Pasadena, CA. Caltech Concurrent Comput. Prog. Rep. 826.

Neri, F., & Cotta, C. (2012). Memetic algorithms and memetic computing optimization: A literature review. *Swarm and Evolutionary Computation, 2*, 1–14.

Nguyen, S. (2016). A learning and optimizing system for order acceptance and scheduling. *The International Journal of Advanced Manufacturing Technology, 86*(5–8), 2021–2036.

Nguyen, S., Zhang, M., & Johnston, M. (2014a). A sequential genetic programming method to learn forward construction heuristics for order acceptance and scheduling. *Evolutionary computation (CEC), 2014 IEEE congress on* (pp. 1824–1831). IEEE.

Nguyen, S., Zhang, M., & Johnston, M. (2014b). Enhancing branch-and-bound algorithms for order acceptance and scheduling with genetic programming. *European conference on genetic programming* (pp. 124–136). Springer.

Nguyen, S., Zhang, M., & Tan, K. C. (2015). A dispatching rule based genetic algorithm for order acceptance and scheduling. *Proceedings of the 16th annual conference on genetic and evolutionary computation (GECCO 2015)* (pp. 433–440). ACM.

Oğuz, C., Salman, F. S., & Yalçın, Z. B. (2010). Order acceptance and scheduling decisions in make-to-order systems. *International Journal of Production Economics, 125*(1), 200–211.

Palomo-Martínez, P. J., Salazar-Aguilar, M. A., & Laporte, G. (2017). Planning a selective delivery schedule through adaptive large neighborhood search. *Computers & Industrial Engineering, 112*, 368–378.

Park, J., Nguyen, S., Johnston, M., & Zhang, M. (2013). Evolving stochastic dispatching rules for order acceptance and scheduling via genetic programming. *Australasian joint conference on artificial intelligence* (pp. 478–489). Springer.

Park, J., Nguyen, S., Zhang, M., & Johnston, M. (2014). Enhancing heuristics for order acceptance and scheduling using genetic programming. *Asia-pacific conference on simulated evolution and learning* (pp. 723–734). Springer.

Pisinger, D., & Ropke, S. (2007). A general heuristic for vehicle routing problems. *Computers & Operations Research, 34*(8), 2403–2435.

Prasetyo, H., Fauza, G., Amer, Y., & Lee, S. H. (2015). Survey on applications of biased-random key genetic algorithms for solving optimization problems. *Industrial engineering and engineering management (IEEM), 2015 IEEE international conference on* (pp. 863–870). IEEE.

Silva, Y. L. T., Subramanian, A., & Pessoa, A. A. (2018). Exact and heuristic algorithms for order acceptance and scheduling with sequence-dependent setup times. *Computers & Operations Research, 90*, 142–160.

Slotnick, S. A. (2011). Order acceptance and scheduling: A taxonomy and review. *European Journal of Operational Research, 212*(1), 1–11.

Tangpattanakul, P., Jozefowiez, N., & Lopez, P. (2015). Biased random key genetic algorithm for multi-user earth observation scheduling. *Recent advances in computational optimization* (pp. 143–160). Springer.

Verbeeck, C., Vansteenwegen, P., & Aghezzaf, E.-H. (2017). The time-dependent orienteering problem with time windows: A fast ant colony system. *Annals of Operations Research, 254*(1–2), 481–505.

Wang, H., Fu, Y., Huang, M., Huang, G. Q., & Wang, J. (2017). A nsga-ii based memetic algorithm for multiobjective parallel flowshop scheduling problem. *Computers & Industrial Engineering, 113*, 185–194.

Wang, Y., Liu, H., Long, H., Zhang, Z., & Yang, S. (2017). Differential evolution with a new encoding mechanism for optimizing wind farm layout. *IEEE Transactions on Industrial Informatics, 14*(3), 1040–1054.

Wang, P., Reinelt, G., Gao, P., & Tan, Y. (2011). A model, a heuristic and a decision support system to solve the Earth observing satellites fleet scheduling problem. *Computers & Industrial Engineering, 61*(2), 322–335.