Cilt / Volume 59 | Sayı / Issue 1 | 115-132

VERİMLİLİK DERGİSİ
JOURNAL OF PRODUCTIVITY

Efficient Solution Algorithms for Resource Planning and Scheduling in Seasonal Reservation Systems

Uğur Eliiyi[1]

ABSTRACT

Purpose: In this study, efficient solution algorithms are proposed for a problem which simultaneously optimizes capacity planning and scheduling decisions in reservation systems. The problem is especially important for systems involving appointments/reservations, such as hotel or seat reservations in tourism, operation and treatment reservations in healthcare systems, or port logistics operations.

***Methodology*:** The optimization problem studied involves concurrent decisions of scheduling and dynamic capacity determination, with an objective of maximizing the net return gained from the served appointments. A randomized constructive heuristic exploiting problem's structural characteristics is proposed together with effective improvement procedures. Extensive computational experimentation is conducted in order to test algorithm performance.

Findings: The developed approach performs excellently in both solution quality and time. With up to 200 reservations, the heuristic technique outperforms CPLEX in terms of solution time and quality. The algorithm's performance remains unchanged as the size of the problem increases.

Originality: This study presents the first heuristic approach to solving this significant problem. Through optimization of resource utilization and scheduling, substantial positive social and economic impact on a number of business sectors can be obtained. The efficient problem-solving techniques developed will pave the way for future research.

Keywords*:* Scheduling with Time Windows, Capacity Planning, Reservation Systems, Optimization.

JEL Codes: C44, C54, C61.

Mevsimsel Rezervasyon Sistemlerinde Kaynak Planlama ve Çizelgeleme İçin Etkin Çözüm Algoritmaları

ÖZET

Amaç: Bu çalışmada, rezervasyon sistemlerinde eş zamanlı kapasite planlama ve çizelgeleme kararlarını optimize eden bir problem için etkin çözüm algoritmaları önerilmektedir. Problem özellikle turizmde otel veya koltuk rezervasyonları, sağlık sistemlerinde operasyon ve tedavi rezervasyonları ya da liman lojistik operasyonları gibi randevu/rezervasyon ile çalışan sistemler için önemlidir.

***Yöntem*:** İncelenen optimizasyon problemi, işleme alınan rezervasyonlara ait net getirinin maksimize edilmesi amacıyla, çizelgeleme ve dinamik kapasite belirleme kararlarının eş zamanlı olarak verilmesini içermektedir. Problemin yapısal özelliklerini kullanan rastgele bir inşa edici sezgisel yöntem ile etkili iyileştirme algoritmaları önerilmiştir. Algoritma performansını test etmek amacıyla kapsamlı hesaplamalı deneyler gerçekleştirilmiştir.

Bulgular: Geliştirilen yaklaşım hem çözüm kalitesi hem de zaman açısından mükemmel performans göstermektedir. Sezgisel yöntem 200 rezervasyona kadar CPLEX'e göre çözüm süresi ve kalitesi açısından üstündür. Algoritmanın performansı, problemin büyüklüğü ile değişmemektedir.

Özgünlük: Bu çalışma, bu önemli problem için sezgisel çözümler öneren ilk çalışmadır. Kaynak kullanımı ve çizelgelemenin optimizasyonu yoluyla birçok sektörde önemli sosyal ve ekonomik katkılar elde edilebilir. Geliştirilen hızlı ve etkili problem çözme teknikleri ileri araştırmaların önünü açacaktır.

Anahtar Kelimeler*:* Zaman Pencereli Çizelgeleme, Kapasite Planlama, Rezervasyon Sistemleri, Optimizasyon.

JEL Kodları: C44, C54, C61.

[1] Izmir Bakircay University, Faculty of Economics and Administrative Sciences, Department of Business, İzmir, Türkiye

Corresponding Author: Uğur Eliiyi, ugur.eliiyi@bakircay.edu.tr

DOI: 10.51551/verimlilik.1521438

Research Article | Submitted: 24.07.2024 | Accepted: 01.11.2024

Cite: Eliiyi, U. (2025). “Efficient Solution Algorithms for Resource Planning and Scheduling in Seasonal Reservation Systems”, *Verimlilik Dergisi*, 59(1), 115-132.

1. INTRODUCTION

Scheduling optimization problems, which lie in the field of management science, include several characteristics of tasks to be processed (priority relationships, setup times etc.), and features of resources to be used for processing these tasks (identical or different resources), in addition to a variety of objective functions (minimizing resources' remaining idle times, minimizing waiting times of tasks, maximizing total return etc.). Because of its wide range of applications and practical significance, this area of optimization has received extensive research. In conventional scheduling, the decision maker is free to choose the start times of tasks; however, in interval scheduling (IS), these start times are set as parameters. Reservation systems are used to manage and allocate limited resources by handling and processing booking requests. These systems are employed in various sectors, including airline ticketing and gate assignment, healthcare systems, classroom scheduling, port operations, and hotel accommodations. Such systems require periodic/seasonal reservations. Hence, the IS problem has significant practical applications in these fields, as well as manufacturing operations such as maintenance and shift scheduling.

In this study, an optimization problem for simultaneously optimizing capacity determination and reservation scheduling decisions is considered. The problem is critical and necessary for optimizing the use of resources in order to deliver efficient services to a larger customer base. The problem is applicable to all reservation systems aiming to maximize the net "return" obtained from reservations handled. The concept of "return" can be defined as a material "operating profit" for some systems, whereas it can represent a physical "benefit", "weight" or "priority" for others. Defining the problem in such a flexible manner allows the decision makers to be able to adapt the problem for different contexts. The problem was introduced to the literature under the name of Combined Reservation Scheduling (CRS) problem by Eliiyi (2021). An integer programming model was developed for the optimal solution of the problem. The model was implemented in IBM ILOG CPLEX environment, and its computational performance was assessed under different scenarios through extensive numerical experiments. The results of the experiments were analyzed and the managerial implications were discussed in depth. The complex nature of the and the NP-Hardness of the problem were proven theoretically. It was not possible to solve the problem optimally as problem size increased, and the need for effective and efficient heuristic solutions for large problem instances were stated.

In this respect, this study is unique and practically significant, as it is the first in literature to tackle efficient solutions for this practically important problem. We propose fast, high-quality and easy-to-implement solutions in a problem environment where appointment requests and cancellations can occur. In such a dynamic environment, repetitive quick and effective solutions are mandatory. Therefore, fast and high-quality solutions will benefit the decision makers from a practical point of view. We develop heuristic solution approaches with inherent characteristics specific to the nature of the problem, which can produce effective solutions quickly. The effectiveness and efficiency of the developed algorithms are measured through extensive numerical experiments. For assessing the performance of the developed algorithms, the obtained solutions are compared with the existing optimal solutions in literature, whereas lower and upper bounds are proposed and used for measurement purposes for larger instances of the problem.

The remainder of this paper is structured as follows: Section 2 provides a comprehensive review of existing research on scheduling with time windows. In Section 3, we introduce our proposed solution approach, including three improvement algorithms. Section 4 presents a detailed analysis of the computational experiments conducted to assess algorithm performance. Finally, Section 5 offers concluding remarks and outlines potential opportunities for future research.

2. LITERATURE REVIEW

The IS problem entails two key decisions: Determining whether to accept or reject an incoming request or task/job/reservation, and if accepted, assigning the most suitable resource or machine to process the task. A typical IS setting includes n tasks that are available to be processed on m parallel resources, where the time window of task j is specified by a ready time (r_j) and a deadline (d_j). Tactical and operational IS problems differ in their objectives and scope. Tactical IS problems associate a fixed cost (c_k) with each resource k, and focus on minimizing total costs or resource utilization while processing all tasks. Conversely, operational IS problems operate within a fixed resource constraint, aiming to maximize the total value or quantity of processed jobs. In operational IS, each job (j) is assigned a weight (w_j) representing its priority or value (e.g. profit). The number of resources m is a predetermined parameter in operational IS, while it serves as an upper limit in the tactical problem.

When a job's processing time equals its deadline minus its ready time ($p_j = d_j - r_j$ for every j), resulting in no flexibility in scheduling, the problem is classified as a Fixed Job Scheduling (FJS) problem. In this study, we focus on scheduling n non-preemptive jobs, each with a given processing time (p_j) and an interval for the start time as [r_j, b_j], where $b_j \geq r_j$ for every j. The problem involves time windows that are larger than the

processing times, i.e., each reservation request entering the system at its r_j should start its processing latest on its standby limit b_j, otherwise it will be wasted in terms of return. This variant called as Variable Job Scheduling (VJS), also known as parallel machine scheduling with time windows (Gabrel, 1995), is a generalization of Fixed Job Scheduling (FJS). The start and end of the reservation, the standby limit, and the anticipated profit are all immediately established and specified upon receipt of a request. A request may be considered a fresh one if it is canceled or modified before to processing. Therefore, the problem's parameters can be treated as deterministic and updated as needed. The goal of a Tactical VJS (TVJS) problem is to minimize the total cost of resources required to process all jobs. On the other hand, an Operational VJS (OVJS) problem aims to maximize the total weight of a job subset that can be processed using a fixed number of machines.

Numerous studies have noted the wide range of applications for IS in manufacturing and service operations. Kolen and Kroon (1992) used tactical and operational FJS to study applications related to allocating aircraft to gates and capacity planning for maintenance staff. Wolfe and Sorensen (2000) examined the scheduling of satellites using operational FJS, whereas Fischetti et al. (1987, 1989, 1992) studied bus driver scheduling as an application of the tactical FJS. Additional application areas mentioned are printed circuit board production (Spieksma, 1999), data transmission (Faigle et al., 1999), and class scheduling (Kolen and Kroon, 1991). Some variations include limitations on eligibility (Eliiyi and Azizoglu, 2009), availability (Kolen and Kroon, 1993), machine running time restrictions (Fischetti et al., 1992; Eliiyi and Azizoglu, 2011), or differing machine speeds (Azizoglu and Bekki, 2008). In addition to theory and complexity results, reviews of possible applications of FJS were given by Kovalyov et al. (2007) and Kolen et al. (2007).

Although IS problems have been widely acknowledged as FJS in the literature, VJS has not received the attention it deserves, with just a small number of academics concentrating on it following the landmark study by Gertsbakh and Stern (1978). Using identical machines, these authors stated the fundamental TVJS problem and gave an approximate solution. In the context of data transmission for low-orbit satellites, Gabrel (1995) employed the operational FJS model and claimed that the suggested model and algorithms can be modified to address OVJS. The operational FJS problem with identical weights and eligibility limitations was handled. The problem's lower and upper bounds are established, and the computational results were shown.

There is not much research on OVJS in the literature. A greedy randomized adaptive search strategy (GRASP) and a heuristic based on dynamic programming were presented by Rojanasoonthon et al. (2003) and Rojanasoonthon and Bard (2005). Bard and Rojanasoonthon (2006) created a branch and price algorithm for a similar problem. The computational results showed that the suggested method could solve fairly big instances to optimality. Garcia and Lozano (2005) investigated the two-stage OVJS problem in the context of manufacturing ready-mix concrete. Each task had an ideal start time, and the goal was to maximize the total weight of the processed jobs while minimizing the overall weighted deviation from ideal start times. To solve the problem, they suggested using the tabu search heuristic, and the outcomes showed good performance in terms of both time and solution quality.

Eliiyi et al. (2009) addressed berth allocation in a container port by using a nested eligibility structure to solve the OVJS problem. Two resource types corresponded to small and large berths for allocating vessels, and two job classes represented two different vessel sizes. The authors developed an integer programming model and showed that the problem is NP-hard. To produce near-optimal solutions, a construction heuristic based on constraint graphs was developed. The solutions were boosted using improvement algorithms including the genetic algorithm. They showed that the problem-specific improvement heuristic outperformed the genetic algorithm.

An alternative methodology involves handling each reservation request independently and in real-time. This dynamic approach captures snapshots of the reservation system's current state for each incoming request, allowing for immediate decision-making. The approach is commonly suggested in the literature for managing fluctuating demand and reservations, particularly in the context of communications networks, where changes happen often and planning horizons are generally shorter (Barshan et al., 2016; Steiger et al., 2004). Unlike adversarial online interval scheduling, which makes no assumptions about task parameters, stochastic variations of this problem assume that task parameters follow a specific distribution (Yu and Jacobson, 2020).

The most crucial element in calculating potential profit in a reservation system is the number of resources as it establishes the array of requests that can be served. A reservation system's capacity needs to be carefully considered in this regard. While prior research has employed TVJS for capacity planning, the tactical problem ignores potential cancellations or new requests during the planning period and necessitates a long-term projection of reservations, which may not be accessible. Additionally, capacity

modification requirements are disregarded by TVJS, which is a serious problem for systems exhibiting seasonal demand as in hotel and vehicle rental reservations (Eliiyi, 2021).

The problem considered in this study offers versatility in capacity planning. It can determine optimal capacity expansion for existing systems or establish initial capacity and scheduling for new ones. By addressing the number and assignment of additional resources, it also allows to manage seasonal demand fluctuations. This approach can be applied dynamically across various timeframes, from seasons to days or even hours, assuming reservations within a specific period remain fixed.

3. SOLUTION ALGORITHMS FOR THE CRS PROBLEM

To ensure clarity and completeness, the mathematical model proposed by Eliiyi (2021) is presented before detailing the newly developed construction and improvement algorithms for the CRS problem.

3.1. The CRS Model

The problem defined by Eliiyi (2021) includes n reservations/tasks to be served on at most m available resources. Each resource incurs a fixed usage cost c_k, while each incoming reservation has a specified ready time (arrival time) r_j and a standby limit $b_j(> r_j)$, representing the maximum allowable waiting period before the reservation is considered lost. The fixed usage cost of each resource could be considered as the total cost or rental fee for the whole planning period, which is to be paid in full regardless of the duration of usage. The standby limit enables customers to set a flexible window of days for their booking in $[r_j, b_j)$.

The service time (reservation duration) and the return are denoted in the model by p_j and w_j, respectively. These problem parameters are nonnegative integers and deterministic. If a reservation request is cancelled or changed before it is processed, it is considered a new one, and the parameters are updated as necessary. The planning horizon is divided into T time intervals of equal length to facilitate mathematical modeling; namely $\{t_1, \dots, t_T\}$. P_a is the set of tasks ready to be processed in time interval $[t_a, t_{a+1})$, where $a = 1, \dots, T-1$ ($P_a = \{j | r_j \le t_a, b_j + p_j - 1 \ge t_a\}$).

Set S_j is defined as the set of intervals of task j ($S_j = \{r_j, \dots, b_j + p_j - 1\}$).

The decision variables are:

$$x_{jka}: \begin{cases} 1, \text{ if task } j \text{ is served on resource } k \text{ in time interval } a \\ 0, \text{ otherwise} \end{cases}$$

$$y_{jk}: \begin{cases} 1, \text{ if task } j \text{ is served on resource } k \\ 0, \text{ otherwise} \end{cases}$$

$$z_k: \begin{cases} 1, \text{ if resource } k \text{ is used} \\ 0, \text{ otherwise} \end{cases}$$

As defined by Eliiyi (2021), decision variable x_{jka} equals 1 if task *j* is served by resource *k* during time interval $[t_a, t_{a+1})$. This definition allows for a task to be served by multiple resources across different time intervals. To prevent preemption, the second decision variable, y_{jk}, is introduced. This variable equals 1 if all intervals of task *j* are served by resource *k*. The third decision variable, z_k, is used to calculate resource fixed costs. If a resource is utilized for any task, its corresponding variable becomes 1, activating its fixed cost in the objective function.

The CRS model determines the number and cost of resources will be used in total, in addition to the subset of served reservation requests:

CRS:

$$\text{Maximize} \sum_{j=1}^{n} \sum_{k=1}^{m} w_j y_{jk} - \sum_{k=1}^{m} c_k z_k \quad (1)$$

s.t.

$$\sum_{a \in S_j} x_{jka} = p_j y_{jk}, \quad j = 1, \dots, n; k = 1, \dots, m \quad (2)$$

$$\sum_{k=1}^{m} y_{jk} \le 1, \quad j = 1, \dots, n \quad (3)$$

$$\sum_{j \in P_a} x_{jka} \le 1, \quad a = 1, \dots, T-1; \ k = 1, \dots, m \quad (4)$$

$$p_j x_{jka} - p_j x_{j,k,a+1} + \sum_{t=a+2}^{b_j + p_j - 1} x_{jkt} \le p_j, \quad j = 1, \dots, n; k = 1, \dots, m; a \in S_j \quad (5)$$

$$y_{jk} \le z_k, \quad j = 1, \dots, n; k = 1, \dots, m \quad (6)$$

$$x_{jka}, y_{jk}, z_k \in \{0,1\}, \quad j = 1, \dots, n; a = 1, \dots, T-1; k = 1, \dots, m \quad (7)$$

The objective function in Equation 1 maximizes the net total return from the served reservations. The constraints in Equation 2 and Equation 3 together force all intervals of a task to be assigned to a single resource, while the constraints in Equation 4 dictates each resource to process at most one task in an interval. If a reservation is served, each time interval of it should be served in order by the same resource, as ensured by constraint set Equation 5. Constraint set in Equation 6 binds the two decision variables in the model whereas constraints in Equation 7 defines the sign constraints.

Eliiyi (2021) also proposed an upper bound on the number of resources as $m = \text{Max}_a\{|P_a|\}$, and proved that the problem is NP-hard. Through computational experimentation, the author concluded that the assumption of deterministic parameters provides tractability and ease-of-use. However, repetitive solutions of the problem, which are necessary in a dynamic environment including cancellations and changes, call for fast and high-quality heuristics.

3.2. A Randomized Heuristic for the CRS Problem

Eliiyi et al. (2009) have shown that a problem-specific heuristic designed for the OVJS problem with general weights generates better solutions in less CPU times than a genetic algorithm. Although their study is on the operational variant of VJS, our heuristic approach for the combined problem uses similar ideas for generating near-optimal solutions at its inner loop, as will be explained shortly. First, we provide some definitions that are employed in our heuristic algorithm.

Let s_j ($r_j \leq s_j \leq b_j$, $\forall j$) denote the realized start time at which reservation j is processed, if it is processed at all. We define the concepts of *overlap* and *overlap amount* via the following definitions.

Definition 1: Two reservations i and j such that $s_i \leq s_j$ *overlap* in time if $s_i = s_j$ or $s_i < s_j < s_i+p_i$. In this case, reservations i and j are defined as *overlapping reservations*.

Definition 2: The *overlap amount(i,j)* between two overlapping reservations i and j such that $s_i \leq s_j$ is computed as in Equation 8.

$$overlap\ amount(i,j) = \begin{cases} s_i + p_i - s_j, \text{ if } s_i + p_i \leq s_j + p_j \\ p_j, \text{ otherwise} \end{cases} \quad (8)$$

.

The solution procedure proposed below has a nested structure, which will be explained in detail. The outer (main) loop is used for trying out different capacity levels for determining a near-optimal capacity:

<u>The CRS Algorithm:</u>

(S0) $Z_0 = 0$. The resources are sorted in ascending order of their fixed costs, i.e. $c_1 \leq c_2 \leq \ldots \leq c_m$.

(S1) For all possible capacity levels ($k = 1,\ldots, m$):

Run *AlgorithmCRS(k)* for capacity level k. Let the resulting objective function value be Z_k.

If $Z_k < Z_{k-1}$ then go to (S2),

Else, proceed with the next capacity level.

(S2) Output Z_{k-1} as the best solution obtained.

This primary algorithm progressively introduces resources into the solution, starting with those having the lowest fixed costs. The main loop (S1) repeatedly calls *AlgorithmCRS(k)*, which constitutes the inner loop, to solve the CRS problem with n jobs and k machines. The algorithm terminates adding resources (at a capacity level of k-1) when introducing a new one results in a decreased net marginal return ($Z_k < Z_{k-1}$). To ensure capacity expansion even at the breakeven point ($Z_k = Z_{k-1}$), the stopping condition explicitly excludes equality. In other words, among the solutions having the same objective function, the algorithm favors solutions having a higher capacity and processing a larger number of incoming reservation requests. This characteristic of the algorithm allows for serving more reservations, which is in line with practice and expedites customer satisfaction.

Next, we describe *AlgorithmCRS(k).* This inner loop involves randomized decisions for the start times and the assignment of the reservations to the resources.

<u>AlgorithmCRS(k):</u>

(S0) $Z_k = 0$, $i = 0$.

(S1) $i \leftarrow i+1$.

For each reservation j (j = 1,..., n):

Compute *overlap index(j)* as follows:

1. Assume all n reservations start processing at their ready times ($s_i = r_i$, i = 1,..., n). Compute *overlap amount(i,j)* between reservation j and every other reservation i under this assumption. Let $o_{1j} = \sum_i overlap\ amount(i,j)$.

2. Assume all n reservations start processing at their standby limits ($s_i = b_i$, i = 1,..., n). Compute *overlap amount(i,j)* between reservation j and every other reservation i under this assumption. Let $o_{2j} = \sum_i overlap\ amount(i,j)$.

3. Generate a random start time s_i (such that $r_i \leq s_i \leq b_i$) for each reservation i, i = 1,..., n. Compute *overlap amount(i,j)* between reservation j and every other reservation i under this assumption. Let $o_{3j} = \sum_i overlap\ amount(i,j)$.

4. *overlap index(j)* = $(o_{1j} + o_{2j} + o_{3j})/w_j$.

Re-index the reservations in ascending order of their overlap index values.

For each reservation (with new index) j (j = 1,..., n) do:

Chose one of the below methods *randomly* to assign reservation j to the next resource (among k resources). Continue until all reservations are assigned or no resources are available:

1. Search forward between $[r_j, b_j]$ to assign the reservation at the earliest available time.
2. Search backward between $[r_j, b_j]$ to assign the reservation at the latest available time.
3. Search forward between $[t, b_j]$ to assign the reservation at the earliest available time, where t is a random number between $[r_j, b_j]$.
4. Search backward between $[r_j, t]$ to assign the reservation at the latest available time, where t is a random number between $[r_j, b_j]$.

(S2) Compute the objective function of the solution at iteration i as z_i.

(S3) Sequentially execute *Improvement1*, *Improvement2* and *Improvement3* to the solution in this order. Update z_i and the assignments as necessary.

(S4) If $z_i > Z_k$, then set $Z_k = z_i$. Store the corresponding solution as the best solution.

(S5) If $i \leq NumIter$, go to (S1). Else, output Z_k as the best solution.

In (S1) of *AlgorithmCRS(k)*, three different representative sets of reservation start times are generated to compute possible *overlap amount*s. The *overlap index* of a reservation is then computed as sum of the overlap amounts from these scenarios divided by the original return of the reservation. A low overlap index corresponds to a low average overlap amount, a high return, or both. Hence, by assigning the reservations to the resources in nondecreasing order of their overlap indices, the algorithm favors low-overlap and/or high-return reservations for processing, which is in line with the objective function of the CRS problem.

To introduce randomness, the algorithm selects an assignment method for each reservation randomly and determines the reservation's start time based on this choice. The process is repeated multiple times, and the best solution found across these iterations is retained. The value of the iteration limit *NumIter* is determined via preliminary experimentation.

In (S3) of each iteration, *AlgorithmCRS(k)* calls three basic improvement algorithms in a sequential manner to further enhance the quality of the produced solution. These algorithms are described below. The first one involves an insertion mechanism that tries to assign an unscheduled reservation to one of the used resources without disturbing its current schedule.

Improvement1 (Insert):

(S0) $z_{imp1} = z_i$, A: set of assigned reservations, B: set of unassigned reservations.

Index the reservation in set B in descending order of their w_j values.

(S1) For each reservation j in set B $(j = 1,\ldots,|B|)$ do:

For each resource l in use, do:

Check if reservation j can be inserted into the schedule on resource l without changing the scheduled times of any existing reservations.

If yes, assign reservation *j* to resource *l*, remove *j* from *B* and add it to *A*. Set $z_{imp1} = z_{imp1} + w_j$.

(S2) Output z_{imp1} and the corresponding schedule as the improved solution.

Following the completion of insertion-based improvements by *Improvement1*, the algorithm below attempts to optimize the schedule by swapping reservations between different resources. The swap is executed only if it allows for the scheduling of a previously unallocated reservation, leading to an overall improvement in the objective function value.

Improvement2 (Swap&Insert):

(S0) $z_{imp2} = z_{imp1}$, A_k: set of assigned reservations on resource *k*, *B*: set of unassigned reservations.

Index the reservations in set *B* in descending order of their w_j values.

(S1) For each pair of resources in use (*l* and *p*) do:

For each reservation *j* in set A_l and each reservation *i* in set A_p do:

Check if it is possible to move reservation *j* to resource *p* and reservation *i* to resource *l*, without changing the remaining schedule (assigned reservations and start times) on either of the resources. If yes, make the swap.

For each reservation *s* in set *B* (*s* = 1,...,|*B*|) do:

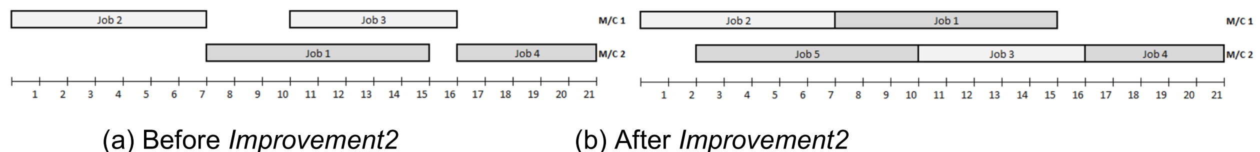
Check if reservation *s* can be inserted into the schedule on resource *l* or resource *p* without changing the scheduled time of any assigned reservations on the resource.

If yes, assign reservation *s* to resource *l* or resource *p*, remove *s* from *B* and add it to A_l or A_p, accordingly. Set $z_{imp2} = z_{imp2} + w_s$.

If no reservation in set *B* could be inserted into the schedule, reverse the swap and continue.

(S2) Output z_{imp2} and the corresponding schedule as the improved solution.

The algorithm prioritizes swap and insert operations based on the decreasing order of reservation weights (w_j) within set *B*. This approach helps to maximize the potential improvement from each swap. It continues until there are no possible moves. The execution of *Improvement2* is illustrated in Figure 1, which shows four reservations assigned on two resources on the timeline where each row represents a resource. Currently, reservation 5 with $r_5 = 2$, $b_5 = 5$ and $p_5 = 8$ cannot be processed on either of the two resources, since it either overlaps with reservation 2 or reservation 3 on the first resource, and reservation 1 on the second resource. While *Improvement2* tries to swap reservations between the resources, reservation 1 and reservation 3 can be swapped without disturbing current schedules. The insertion of reservation 5 to the second resource is then possible, as shown in Figure1(b). The objective function is thereby increased by w_5.



(a) Before *Improvement2* (b) After *Improvement2*

Figure 1. Execution mechanism of *Improvement2*.

Once all possible *swap&insert* combinations are explored by *Improvement2*, the last improvement algorithm tries to insert an unscheduled reservation to a used resource as in *Improvement1*. But unlike *Improvement1*, the following algorithm shifts the scheduled reservations on the resource forward and backward temporally in search of an available time gap for a feasible insertion.

Improvement3 (Shift&Insert):

(S0) $z_{imp3} = z_{imp2}$, *A*: set of assigned reservations, *B*: set of unassigned reservations.

Index the reservations in set *B* in descending order of their w_j values.

(S1) For each reservation *s* in set *B* (*s* = 1,...,|*B*|) do:

For each resource *l* in use do:

Consider interval [r_s, r_s+p_s] on resource l. Let *left_j* be the first reservation on resource l that starts within this interval. Let *right_j* be the first reservation on resource l that starts after *left_j*.

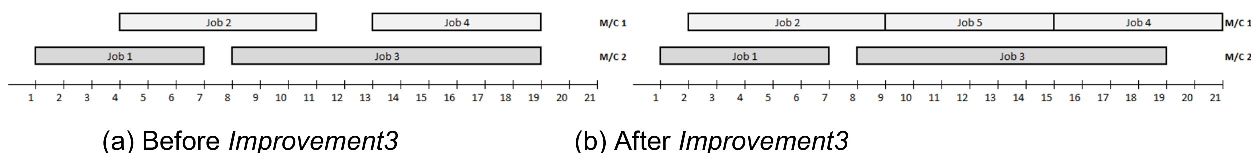
Shift *left_j* and all reservations before it backward in time as much as possible without causing any overlap. Similarly, shift *right_j* and all reservations after it forward in time as much as possible without causing any overlap.

Check if reservation s can be inserted into the schedule on resource l between *left_j* and *right_j* without causing any overlap. If yes, reservation job s to resource l, remove s from B and add it to A. Set $z_{imp3} = z_{imp3} + w_s$.

(S3) Output z_{imp3} and the corresponding schedule as the improved solution.

The algorithm iteratively performs shift and insert operations on reservations in set B, prioritized by their descending weight values (w_j). This process continues until no feasible insertions can be made, indicating a local optimum. Our algorithm is different than the *Shift & Insert* algorithm described in Eliiyi et al. (2009) in the following manner. In their approach, each resource is considered separately, and for each pair (*left_j* and *right_j* in our notation), a gap between them is tried to be created by moving *left_j* and all before that exactly to their ready times while moving *right_j* and all after that exactly to their standby limits. No time intervals in-between the ready times and the standby limits are investigated for possible shifts, and their algorithm does not perform a shift even if only one task could not be moved to its ready time or standby limit. *Improvement3* has a superior search procedure that checks for all possible shift combinations (including all possible time intervals for each reservation) by considering each unscheduled reservation separately.

The execution of *Improvement3* is illustrated through Figure 2. Consider resource 1 in Figure 2(a), processing reservation 2 and 4. Assume that reservation 2 has r_1 = 2 and reservation 4 has b_4 = 15. Reservation 5, having parameters r_5 = 7, b_5 = 10 and p_5 = 6, is currently out of schedule as it overlaps with reservation 3 on resource 2, and reservation 2 or reservation 4 on resource 1. While trying to insert reservation 5 into the schedule of resource 1, *Improvement3* shifts reservation 2 (*left_j*) two units backward, and reservation 4 (*right_j*) two units forward in time. These shifts create an opportunity to insert reservation 5 at time 9, as illustrated in Figure 2(b), improving the objective function value by w_5.



(a) Before *Improvement3* (b) After *Improvement3*

Figure 2. Execution mechanism of *Improvement3*.

The solution approach described in this section is explained under the assumption that all resources are eligible to process all incoming reservation requests, i.e. there is no eligibility constraint in the problem and $w_{jk} = w_j$, $\forall k$. However, the adaptation to the problem with eligibility constraints is rather straightforward. Namely, only the eligible resources should be considered when making the assignment of an incoming reservation or the swaps between any two reservations. In other words, *resource* should be replaced by *eligible resource* wherever necessary in the proposed algorithms.

The CRS problem can be framed as a multi-objective optimization problem, balancing return maximization and cost minimization. Interestingly, *AlgorithmCRS* generates a set of Pareto-optimal solutions across different capacity levels as a byproduct, offering valuable insights into the trade-offs between these competing objectives. The algorithm can be easily modified to store and output optimal solutions for every capacity level without increasing computational complexity. This enhancement transforms the algorithm into a tool for solving the associated multi-objective optimization problem.

4. COMPUTATIONAL EXPERIMENTATION

A computational experiment was conducted to assess the performance of the developed algorithms. The algorithms were implemented using C# and C++ on MS Visual Studio 2008 and executed on a Core 2 Duo 2.8 GHz PC with 4 GB of memory. The experimental setting is identical to that of Eliiyi's (2021) to allow direct comparison with optimal solutions. Therefore, the reader is referred to that study for details of the experiment design, in which 10 test problems were generated for the CRS problem with n = 20, 50, 100, 200 reservations and 36 combinations of parameters, corresponding to 1440 instances. We include all these instances in our study as well as additional 360 new test instances for very large problems with n = 500. Hence, a total of 1800 problem instances are used for performance evaluation purposes.

As stated by Eliiyi (2021), a majority of problem instances with *n* = 50 or more tasks/reservations could not be solved optimally by the commercial solver within the allotted 1200-second time limit, which is a clear display for the difficulty of the problem. Only the best feasible solutions were reported for these instances. In this section, we discuss the performance of *CRS Algorithm* as compared to IBM ILOG CPLEX 12.8 solutions. Our algorithm is executed for 100 iterations for all instances, and a 1200-second time limit is imposed for CPLEX solutions, as in Eliiyi (2021). For instances where an optimal solution could not be determined within the time limit, the algorithm's solution is compared to the best feasible solution found by CPLEX. Tables 1 through 3 present the result of the experimental runs at three different resource cost levels for *n* = 20, 50, 100, and 200.

The column *Avg. UB* gives the upper bound on the number of available resources as $\text{Max}_a\{|P_a|\}$ for each problem instance (Eliiyi, 2021). The results are averaged over 10 problem instances. The other columns compare the average number of the used resources, average percentage of resource utilizations, average percentage of the processed reservations (over all incoming), average percentage of the processed return (over all incoming), the solution times of the *CRS Algorithm* and CPLEX, and the percent gaps between the CPLEX solutions and the lower bound solutions obtained by the algorithm. The last column, *CPLEX ZERO* lists the number of instances (out of 10) for which a feasible solution could not be obtained by CPLEX within the time limit. As no feasible solution could be obtained for any problem instance with *n* = 500 by CPLEX, the solutions could not be compared, therefore not reported.

The average number of used resources over 10 problem instances is given in *Avg. # used* column. For example, in Table 1, for *n* = 100, *b-r* = 1, *w* = 1 and *p* = 1, while all requests could be processed with 12 resources, the *CRS Algorithm* obtained solutions with an average of 4 machines while CPLEX used 1 machines on the average. The *Avg. % util.* (resource load / planning horizon length) illustrates the percentage of resource usage during the planning horizon [0, 200]. For the same example, our algorithm generated solutions with an average resource utilization of 68% for the 4 resources used, whereas CPLEX solutions used only 28% of the 1 resource used. Note that CPLEX could not attain optimality for this set of instances within the time limit. The next two columns list the average percentage of processed reservations (*% processed reservations*) calculated by $\left(\sum_{j=1}^{n} x_{jk}\right)/n$ for the algorithm, and the average percentage of processed return (*% processed return*) computed as $\left(\sum_{j=1}^{n} w_j x_{jk}\right)/\left(\sum_{j=1}^{n} w_j\right)$ for CPLEX. The percentage of processed reservations can be used to assess the solution's effectiveness in capturing potential gains. For instance, the solution by the *CRS Algorithm* that processes 86% of reservations generates 85% of the total potential return for the above example, whereas both ratios are 16% for CPLEX. In scenarios where the return of each reservation varies, the percentage of processed return could exceed the percentage of processed reservations.

The tables reveal that while several problem instances with *n* = 50 or more reservations could not be solved to optimality by CPLEX within the allotted time limit, the *CRS Algorithm* produces very fast solutions; the longest solution time being around 1 minute. The percent gap (*% Diff*) between CPLEX and the lower bound obtained by the algorithm, calculated as $100 * (Z_{CPLEX} - Z_{AlgorithmCRS})/Z_{CPLEX}$, came out to be negative for many instances. This result indicates that the algorithm obtained better solutions than CPLEX in much shorter times. The gap is not computed for instances where CPLEX could not find a feasible solution, as indicated in the last column (*CPLEX ZERO*). CPLEX could not obtain any feasible solution for 910 of the 1800 instances within the time limit. The algorithm's efficiency is influenced by several factors. High resource costs generally lead to faster computation times as fewer reservations are processed. Conversely, longer standby durations, increased processing times, and a larger number of reservations tend to extend solution times. While the number of reservations directly impacts performance, changes in reservation returns appear to have minimal effect.

For *n* = 20, CPLEX found the optimal solutions for all instances. The average CPU time over Tables 1, 2 and 3 is approximately 1 minute. Our algorithm obtained instant solutions for these instances, while the average *% Diff* is around 2.7% for uniformly distributed resource costs, 1.8% for low resource costs and 3.0% for resource costs. The algorithm found optimal solutions in 274 out of 360 instances for *n* = 20. For larger ones, it obtained much better solutions than CPLEX. As an example, examine in Table 1 the row for *n* = 100, *b-r* = 2, *w* = 1 and *p* = 2, and note that the *% Diff* value is 51100.0. The *CPLEX ZERO* indicates that no feasible solutions could be obtained for 9 of the 10 instances of this setting. For the remaining one instance, CPLEX obtained an objective function value of 1, while the algorithm found 511, and hence the corresponding *% Diff* value. On average, the algorithm provides solutions with higher reservation processing, return processing and resource utilization percentages than CPLEX.

Table 1. Results for c_k ~ U{80, 100, 120, 140, 160}, $\forall k$

					Avg. # used		Avg. % util.		% processed reservations		% processed return		Solution time (seconds)			CPLEX ZERO
n	b-r	w	p	Avg. UB	Alg.	CPLEX	Alg.	CPLEX	Alg.	CPLEX	Alg.	CPLEX	Alg.	CPLEX	% Diff	
20	1	1	1	4	1	1	44	45	75	75	76	76	0.1	4.4	2.8	
			2	5	1	1	60	60	60	61	64	65	0.1	159.0	4.2	
		2	1	4	1	1	43	43	74	74	77	77	0.1	1.7	0.0	
			2	5	1	1	25	27	31	31	32	32	0.1	10.1	2.6	
		3	1	4	1	1	43	43	75	75	80	80	0.1	3.3	0.2	
			2	5	1	1	51	52	62	62	68	69	0.1	31.5	3.5	
	2	1	1	5	1	1	52	53	89	89	89	89	0.1	6.9	1.0	
			2	5	1	1	65	65	67	68	69	70	0.1	273.1	3.0	
		2	1	4	1	1	52	52	91	91	92	92	0.0	2.2	0.8	
			2	5	1	1	51	52	56	55	58	59	0.1	144.2	11.0	
		3	1	5	1	1	49	50	86	86	89	90	0.1	3.5	1.3	
			2	5	1	1	63	63	69	69	76	76	0.1	50.3	2.2	
50	1	1	1	8	2	2	70	66	69	75	69	75	0.4	1200.1	-0.1	
			2	9	3	2	69	47	81	41	80	42	0.8	1200.1	-937.2	2
		2	1	8	2	2	62	66	79	69	82	73	0.6	1200.1	-1.4	
			2	9	2	1	71	31	52	19	55	20	0.3	1164.2	-764.0	5
		3	1	7	2	2	59	61	85	83	89	88	0.6	1200.1	0.5	
			2	10	2	2	65	55	69	42	76	46	0.6	1200.1	-404.9	
	2	1	1	10	2	2	64	67	89	67	89	68	0.7	1200.1	-32.9	
			2	10	3	1	71	33	87	18	86	18	1.1	1200.1	-1356.6	3
		2	1	9	2	1	71	66	79	65	82	69	0.6	1200.1	-26.2	
			2	10	2	1	76	31	65	22	69	24	0.5	1200.1	-320.9	5
		3	1	9	2	2	65	62	91	78	93	83	0.8	1200.1	-13.8	
			2	11	2	1	72	34	73	26	77	29	0.8	1200.1	-998.5	2
100	1	1	1	12	4	1	68	28	86	16	85	16	3.2	1200.1	-2015.9	4
			2	15	6	0	71	0	86	0	85	0	5.2	1200.3		10
		2	1	12	3	1	69	29	82	19	86	21	2.9	1200.1	-968.4	4
			2	15	3	1	74	3	52	1	56	1	1.9	1200.3		10
		3	1	13	4	2	66	33	89	29	93	31	4.0	1200.1	-277.7	2
			2	14	5	0	71	10	74	2	80	2	4.0	1200.3	-343.8	9
	2	1	1	15	4	1	71	39	93	21	93	22	4.7	1200.2	-628.9	2
			2	19	6	0	75	4	89	1	88	1	7.1	1200.1	-	9
		2	1	16	3	1	76	50	85	26	87	28	3.4	1200.1	-560.5	
			2	19	4	1	79	5	66	4	70	4	3.7	1201.2		10
		3	1	14	4	2	71	40	93	36	96	39	5.0	1200.1	-538.9	
			2	16	5	1	76	5	82	3	87	3	6.4	1200.1	-487.0	9
200	1	1	1	21	7	0	71	0	90	0	89	0	24.1	1200.2		10
			2	21	11	0	75	0	89	0	87	0	39.9	1200.2		10
		2	1	22	6	0	74	2	82	1	86	1	20.2	1201.8		10
			2	22	6	0	78	0	61	0	65	0	21.2	1202.2		10
		3	1	19	7	3	72	10	90	7	94	7	27.1	1200.3	-3851.2	9
			2	18	9	0	75	0	78	0	84	0	36.0	1200.2		10
	2	1	1	26	7	0	76	16	93	3	92	3	33.1	1202.5	-2977.1	7
			2	28	12	0	75	0	93	0	92	0	60.8	1206.9		10
		2	1	26	6	1	77	4	88	1	91	1	31.5	1205.6	-8837.5	9
			2	28	6	1	82	4	62	1	67	2	29.6	1204.2		10
		3	1	26	7	1	75	11	93	3	96	4	37.3	1200.3	-1599.1	7
			2	26	9	0	79	0	81	0	87	0	51.5	1200.5		10

Table 2. Results for c_k = 80, $\forall k$

					Avg. # used		Avg. % util.		% processed reservations		% processed return		Solution time (s.)			CPLEX
n	*b-r*	*w*	*p*	*Avg. UB*	*Alg.*	*CPLEX*	*Alg.*	*CPLEX*	*Alg.*	*CPLEX*	*Alg.*	*CPLEX*	*Alg.*	*CPLEX*	*% Diff*	*ZERO*
20	1	1	1	4	1	1	49	49	79	79	79	79	0.1	3.1	1.1	
			2	5	1	1	57	57	68	67	68	69	0.1	219.6	1.6	
		2	1	4	1	1	44	44	80	81	83	83	0.1	2.1	1.3	
			2	5	1	1	56	58	65	65	69	70	0.1	11.5	8.0	
		3	1	4	1	1	45	45	80	80	85	85	0.1	3.6	0.1	
			2	4	1	1	53	54	63	63	68	68	0.1	71.3	0.1	
	2	1	1	5	1	1	51	51	90	91	90	91	0.1	3.0	3.7	
			2	5	1	1	69	69	74	75	74	75	0.1	377.0	0.7	
		2	1	5	1	1	50	51	89	90	91	92	0.0	4.3	1.8	
			2	5	1	1	62	63	71	71	75	75	0.1	91.6	1.2	
		3	1	5	1	1	51	52	90	90	92	92	0.0	4.5	0.7	
			2	6	1	1	61	62	70	71	76	76	0.1	335.0	0.7	
50	1	1	1	8	2	2	62	61	85	84	85	84	0.6	1200.1	-3.2	
			2	9	3	3	65	58	83	66	83	67	1.0	1200.1	-58.3	
		2	1	7	2	2	60	61	85	81	87	84	0.6	1200.1	-4.1	
			2	9	2	1	68	60	66	46	69	50	0.5	1200.1	-24.8	1
		3	1	8	2	2	59	58	82	83	88	89	0.7	1200.1	-0.6	
			2	10	3	3	62	54	78	62	83	68	1.0	1200.1	-39.2	
	2	1	1	9	2	2	65	66	91	73	90	74	0.7	1200.1	-29.4	
			2	10	3	1	68	47	91	26	91	27	1.2	1200.1	-181.6	3
		2	1	8	2	2	65	64	91	72	93	76	0.7	1200.1	-20.6	
			2	11	2	1	77	36	62	22	65	24	0.5	1200.1	-166.1	4
		3	1	8	2	2	67	63	91	86	94	90	0.7	1200.1	-8.2	
			2	10	3	1	71	42	85	33	89	37	1.1	1200.2	-223.8	2
100	1	1	1	12	4	3	66	52	90	61	90	61	3.9	1200.2	-73.8	1
			2	15	7	1	68	17	92	7	91	8	6.9	1200.3	-1662.8	6
		2	1	12	4	3	69	59	86	59	89	62	3.3	1200.1	-81.7	
			2	15	4	0	74	17	66	5	70	6	3.0	1200.2	-1745.1	7
		3	1	12	4	4	64	49	91	64	94	68	4.4	1200.1	-291.7	
			2	15	6	3	69	39	84	22	89	26	6.3	1200.3	-587.3	1
	2	1	1	16	4	1	71	37	95	26	94	25	4.9	1199.1	-365.6	3
			2	19	7	1	70	13	94	7	93	8	8.3	1200.3	-1853.9	7
		2	1	15	3	2	75	49	88	43	90	46	3.8	1200.2	-162.2	
			2	18	3	0	79	9	63	4	67	4	3.3	1200.4	-1646.4	8
		3	1	15	4	3	68	42	95	47	97	52	5.3	1200.2	-288.5	1
			2	18	6	2	74	28	87	16	91	19	7.2	1200.2	-1206.2	2
200	1	1	1	21	8	1	67	35	94	11	94	12	29.6	1200.2	-4359.8	2
			2	27	13	0	71	0	95	0	95	0	55.0	1200.3		10
		2	1	19	7	3	73	49	90	32	92	35	26.2	1200.3	-1176.0	
			2	27	7	1	76	2	67	1	71	1	25.6	1200.3		10
		3	1	21	8	2	68	37	94	18	96	21	31.6	1200.2	-896.5	1
			2	28	11	0	72	0	86	0	90	0	50.5	1200.3		10
	2	1	1	27	7	0	75	0	93	0	93	0	34.4	1201.8		10
			2	31	13	0	73	0	96	0	95	0	68.5	1200.5		10
		2	1	28	7	0	76	6	93	1	95	1	38.6	1201.2	-	8
			2	31	7	0	81	2	70	1	74	1	38.5	1200.4		10
		3	1	27	8	0	73	0	96	0	98	0	42.2	1200.3		10
			2	32	10	0	78	0	86	0	91	0	62.8	1200.5		10

Table 3. Results for c_k = 160, $\forall k$

					Avg. # used		*Avg. % util.*		*% processed reservations*		*% processed return*		*Solution time (seconds)*			CPLEX
n	b-r	w	p	Avg. UB	*Alg.*	*CPLEX*	*Alg.*	*CPLEX*	*Alg.*	*CPLEX*	*Alg.*	*CPLEX*	*Alg.*	*CPLEX*	*% Diff*	ZERO
20	1	1	1	4	0	0	0	0	0	0	0	0	0.1	0.1	0.0	
			2	5	0	0	0	7	0	7	0	6	0.2	5.0	10.0	
		2	1	4	0	0	0	0	0	0	0	0	0.1	0.1	0.0	
			2	4	0	0	0	0	0	0	0	0	0.2	0.1	0.0	
		3	1	4	1	1	44	44	78	78	81	81	0.1	2.8	5.8	
			2	5	0	0	15	16	19	19	20	20	0.2	8.0	4.0	
	2	1	1	5	0	0	0	0	0	0	0	0	0.1	0.1	0.0	
			2	6	1	1	58	58	55	55	55	56	0.1	161.2	9.5	
		2	1	4	0	0	0	0	0	0	0	0	0.1	0.1	0.0	
			2	6	0	0	0	0	0	0	0	0	0.2	0.2	0.0	
		3	1	4	1	1	54	54	92	92	95	95	0.0	3.3	0.4	
			2	6	1	1	50	50	58	57	61	62	0.1	71.7	6.0	
50	1	1	1	7	1	1	77	78	53	52	53	54	0.2	1200.1	5.6	
			2	9	1	1	79	81	46	33	43	32	0.3	1200.1	-20.9	
		2	1	7	1	1	70	72	55	55	59	60	0.2	1200.0	0.5	
			2	9	0	0	22	8	13	5	14	5	2.3	1116.3	34.8	9
		3	1	7	2	2	61	64	78	69	84	76	0.6	1200.1	0.1	
			2	9	1	1	72	69	45	42	52	48	0.3	1200.1	-47.6	
	2	1	1	9	1	1	82	41	57	28	56	28	0.3	1200.1	-13.5	5
			2	10	2	0	79	0	65	0	64	0	0.5	1200.1		10
		2	1	10	1	1	79	75	59	56	64	62	0.3	1200.1	-36.8	
			2	11	1	0	39	0	23	0	25	0	1.6	1200.1		10
		3	1	8	2	1	72	68	75	67	82	75	0.5	1200.1	-10.6	
			2	11	2	1	79	46	58	25	65	29	0.4	1200.1	-116.1	3
100	1	1	1	13	2	1	80	49	56	17	55	17	1.2	1200.1	-96.7	4
			2	15	4	0	80	8	59	5	56	4	2.1	1200.2	-50.7	9
		2	1	12	2	1	75	53	59	23	65	26	1.2	1200.1	-377.2	2
			2	15	1	0	79	0	26	0	28	0	0.4	1200.3		10
		3	1	12	3	2	71	57	78	55	84	62	2.3	1200.1	-36.5	1
			2	15	3	1	75	10	56	7	63	8	2.0	1200.3	-396.8	8
	2	1	1	15	3	0	81	0	73	0	71	0	2.2	1200.2		10
			2	18	5	0	81	0	75	0	72	0	3.8	1200.2		10
		2	1	16	2	1	81	40	62	17	67	19	1.6	1200.2	-693.7	4
			2	19	1	0	84	0	28	0	32	0	0.7	1200.3		10
		3	1	14	3	2	77	49	84	36	89	42	3.1	1200.2	-100.4	2
			2	18	3	0	80	4	59	2	66	2	2.5	1200.3	-9850.0	9
200	1	1	1	21	5	0	80	7	69	1	67	1	10.9	1200.2	-1316.7	9
			2	27	8	0	81	0	69	0	65	0	21.7	1200.3		10
		2	1	20	4	1	79	25	59	11	64	13	8.4	1200.3	-374.1	6
			2	26	2	1	81	4	24	2	26	2	2.2	1201.3		10
		3	1	20	6	2	75	36	81	24	87	27	19.0	1200.3	-917.3	3
			2	26	5	0	78	0	55	0	62	0	15.1	1200.3		10
	2	1	1	26	6	0	82	0	79	0	78	0	20.1	1200.2		10
			2	31	9	0	83	0	80	0	76	0	37.7	1200.4		10
		2	1	26	4	0	83	0	66	0	71	0	14.8	1200.3		10
			2	32	2	0	87	0	30	0	33	0	4.2	1200.4		10
		3	1	26	6	0	79	0	86	0	91	0	27.2	1201.2		10
			2	33	6	0	83	0	59	0	67	0	22.7	1200.5		10

Increasing resource costs discourage capacity expansion. This is evident in higher resource utilization rates observed for larger problem instances in Tables 2 and 3. As problem size grows, more overlapping reservations occur, allowing for denser resource scheduling, and consequently, higher reservation and return processing levels. However, this increased efficiency comes at the cost of higher resource utilization, which is directly linked to increased resource costs. In summary, the developed algorithm including all improvement heuristics produces exceptionally high-quality solutions under one second for small problems (n = 20). It significantly outperforms CPLEX in terms of both solution speed and quality when handling larger problem instances. Moreover, the algorithm demonstrates consistent performance across various parameter settings, including return and cost values.

4.1. Effect of Improvement Algorithms and the Number of Iterations

To observe individual performances of the improvement algorithms, the *CRS Algorithm* is executed for 100 iterations including all improvement algorithms, for 600 instances with c = 1 including the largest instances (n = 500). Table 4 presents the results of these experiments. The columns labeled as *Best0* through *Best3* in the table illustrate in how many of the instances the resulting solution is determined by the corresponding improvement algorithm. *Best0* indicates the case of no improvement. As an example, the first row of the table indicates that for 1 of the 10 instances in this setting, the best solution is obtained without applying any improvement. For another problem instance in this setting, the best solution is achieved after *Improvement1*. For the remaining 8 instances, the best solution is output by *Improvement3*. These columns clearly reveal the effectiveness of the improvement heuristics. The best solution is obtained without using any improvement in only 19 of the 600 test instances. For most (546 instances in total), the best solution is found after applying *Improvement3*. While *Improvement3* seems to be the most effective in this regard, it should be noted that improvements are applied sequentially to the solution, i.e. *Improvement3* is applied after *Improvement2*. This result will be investigated further below.

The columns *Imp1%* through *Imp3%* indicate the percentage increase in the objective function by each improvement algorithm. For example, in the last row of the table, we can observe that *Improvement1* was futile, yielding no increase in the objective function value. In contrast, a substantial improvement of 9.1% was achieved after *Improvement2*. After *Improvement3*, an additional 0.4% was obtained. The results in the table reveal that the best improvements can be obtained by *Improvement2*. On average, *Improvement2* (*Swap&Insert*) brings 12.4% enhancement to the objective, followed by *Improvement3* (*Shift&Insert*) with 7.1% and *Improvement1* (*Insert*) with 1.3%.

The columns *CPU1%* through *CPU3%* present average solution times of improvement algorithms as a percentage of the overall solution time of the *CRS Algorithm*, whereas *CPU0%* indicate the time consumed by the remaining parts of the algorithm. As an example, the last row of the table shows that *Improvement2* took 92.6% of the overall solution time on average for n = 500, b-r = 2, w = 3 and p = 2. The whole 100 iterations of the algorithm with no improvement took only 0.7% of the 1302.8 seconds, whereas *Improvement3* used up 6.5% of total solution time. It can be concluded that the best improvements come with a cost of solution time. Averaging over all test instances, *Improvement2* consumes 84.5% of the overall solution time whereas *Improvement3* spends only 10.0%, the iterations take 3.3%, and *Improvement1* only 1.0%. For the largest instances, the percentages of *Improvement2* are amplified, reaching to approximately 93% of the total solution time. The *CPU (sec.)* column presents the average solution times of the algorithms. Only one of the largest set of solution instances have taken more than 1200 seconds on the average, namely the setting with n = 500, b-r = 2, w = 3 and p = 2.

Due to time-consuming nature of *Improvement2*, we further investigate if it is possible to obtain better solutions with different improvement schemes and iteration limits. For this purpose, new runs are carried out for the same instances in Table 4 with 100 and 500 iterations. The results are summarized in Table 5. The *CPU (sec.)* column is identical to the one in Table 4, presenting average solution times with all improvements at each iteration and a total of 100 iterations, which constitute the base setting for the algorithm.

The next two columns exhibit performance when the *CRS Algorithm* is executed for 100 iterations only with *Improvement1* and *Improvement3*. The column *NoImp2%* lists the average percentage decline in the objective function value as compared to the base setting, and *NoImp2 CPU (sec.)* present the average solution times for this case. It is observed that the individual effect of *Improvement2* is up to 5% for large problems, while it brings an average 2% improvement over all instances, which are both quite significant. However, this reward comes at a substantial cost of solution time; there is a dramatic reduction in solution times when *Improvement2* is not applied (from ~150 to ~14 seconds, averaged over all instances).

Table 4. Performance of the improvement algorithms

n	b-r	w	p	Best0	Best1	Best2	Best3	Imp1 %	Imp2 %	Imp3 %	CPU0 %	CPU1 %	CPU2 %	CPU3 %	CPU (sec.)
20	1	1	1	1	1	0	8	6.2	0.0	8.1	12.0	0.0	69.7	18.3	0.1
			2	1	0	0	9	0.5	1.8	74.2	0.8	0.0	82.7	16.5	0.1
		2	1	3	0	0	7	1.5	0.0	21.2	13.2	0.0	64.5	12.3	0.1
			2	7	0	0	3	0.0	0.0	52.9	11.6	5.0	76.4	7.0	0.1
		3	1	1	1	0	8	0.0	0.0	0.5	9.5	2.5	59.3	18.7	0.1
			2	3	1	1	5	0.0	3.1	0.2	7.3	0.0	92.7	0.0	0.1
	2	1	1	0	0	0	10	3.5	0.0	18.0	1.3	3.9	57.6	27.3	0.1
			2	0	0	0	10	0.8	0.3	11.7	0.0	5.3	82.3	12.3	0.1
		2	1	0	0	0	10	8.9	0.0	8.5	8.3	0.0	46.7	25.0	0.0
			2	2	0	0	8	40.0	0.0	81.8	11.5	0.8	71.4	16.3	0.1
		3	1	0	0	0	10	1.4	0.0	3.4	8.3	2.5	52.5	16.7	0.1
			2	0	1	0	9	3.8	0.0	7.0	8.5	0.0	68.5	23.0	0.1
50	1	1	1	0	0	1	9	1.3	10.4	10.9	1.9	0.0	83.3	14.9	0.4
			2	0	0	2	8	0.0	14.4	3.4	5.3	2.0	86.9	5.8	0.8
		2	1	0	0	0	10	1.0	13.8	6.8	1.8	0.6	87.0	10.6	0.6
			2	1	0	1	8	1.9	12.0	13.5	8.3	3.2	71.2	17.3	0.3
		3	1	0	0	3	7	0.0	9.1	3.3	2.5	0.4	89.0	8.1	0.6
			2	0	0	2	8	0.0	8.8	3.2	4.3	1.4	87.9	6.3	0.6
	2	1	1	0	0	0	10	0.8	31.1	5.9	3.3	0.6	84.4	11.7	0.7
			2	0	0	0	10	0.3	19.2	6.8	4.3	2.5	86.9	6.4	1.1
		2	1	0	0	1	9	0.1	13.0	7.2	4.0	2.4	78.8	14.9	0.6
			2	0	0	0	10	2.0	12.6	9.6	4.2	2.5	80.3	12.9	0.5
		3	1	0	0	0	10	0.0	9.2	2.0	3.1	0.4	86.2	10.3	0.8
			2	0	0	1	9	0.4	10.0	3.2	6.6	2.9	81.9	8.6	0.8
100	1	1	1	0	0	0	10	0.1	28.6	3.9	2.7	0.8	87.8	8.6	3.2
			2	0	0	0	10	0.3	18.1	3.7	2.6	1.0	89.9	6.5	5.2
		2	1	0	0	0	10	0.0	14.0	2.9	2.1	0.7	88.9	8.3	2.9
			2	0	0	0	10	0.1	14.9	5.2	4.0	2.3	85.1	8.6	1.9
		3	1	0	0	2	8	0.2	9.5	1.4	2.0	0.5	91.6	5.9	4.0
			2	0	0	0	10	0.0	10.9	1.8	2.8	1.3	88.7	7.2	4.0
	2	1	1	0	0	0	10	0.0	27.3	3.8	1.9	1.2	88.6	8.4	4.7
			2	0	0	0	10	0.1	23.7	3.3	3.4	1.3	87.9	7.4	7.1
		2	1	0	0	0	10	0.5	16.9	3.5	2.4	0.6	86.9	10.0	3.4
			2	0	0	0	10	0.1	26.6	3.4	4.2	2.4	83.6	9.8	3.7
		3	1	0	0	1	9	0.0	9.0	1.1	1.8	0.7	90.7	6.8	5.0
			2	0	0	1	9	0.0	11.9	2.0	2.4	0.7	89.7	7.2	6.4
200	1	1	1	0	0	0	10	0.2	23.5	2.0	1.3	0.4	90.1	8.2	24.1
			2	0	0	0	10	0.2	16.9	1.1	1.7	0.4	90.8	7.1	39.9
		2	1	0	0	0	10	0.0	16.2	1.4	1.2	0.4	90.8	7.7	20.2
			2	0	0	1	9	0.1	19.6	2.8	1.3	0.7	90.5	7.5	21.2
		3	1	0	0	0	10	0.0	8.7	0.7	1.0	0.4	92.5	6.1	27.1
			2	0	0	2	8	0.0	9.7	0.9	1.7	0.5	91.6	6.2	36.0
	2	1	1	0	0	0	10	0.1	26.0	2.5	1.1	0.4	90.5	8.0	33.1
			2	0	0	0	10	0.0	20.7	1.9	1.8	0.5	90.4	7.2	60.8
		2	1	0	0	0	10	0.0	14.1	1.8	1.2	0.4	90.4	8.1	31.5
			2	0	0	2	8	0.1	22.4	1.5	1.7	0.7	89.2	8.4	29.6
		3	1	0	0	0	10	0.0	7.8	0.7	1.1	0.3	91.7	6.9	37.3
			2	0	0	0	10	0.0	9.7	1.1	1.5	0.5	91.7	6.4	51.5
500	1	1	1	0	0	0	10	0.0	21.4	1.1	0.6	0.2	90.1	9.1	408.5
			2	0	0	0	10	0.0	12.1	1.1	1.0	0.3	88.3	10.4	786.2
		2	1	0	0	1	9	0.0	14.4	0.6	0.5	0.2	91.5	7.8	414.8
			2	0	0	0	10	0.0	17.9	1.0	0.7	0.3	90.2	8.8	359.4
		3	1	0	0	0	10	0.0	7.1	0.3	0.6	0.2	91.9	7.4	462.2
			2	0	0	6	4	0.0	9.2	0.1	0.7	0.2	91.2	7.9	775.1
	2	1	1	0	0	0	10	0.0	21.8	1.3	0.6	0.2	91.6	7.7	687.5
			2	0	0	0	10	0.0	16.6	1.2	1.0	0.2	89.7	9.1	1143.8
		2	1	0	0	0	10	0.0	13.8	0.9	0.5	0.1	91.7	7.6	695.7
			2	0	0	2	8	0.0	17.8	0.8	0.7	0.2	92.0	7.1	729.8
		3	1	0	0	0	10	0.0	6.0	0.4	0.5	0.1	92.5	6.9	803.1
			2	0	0	1	9	0.0	9.1	0.4	0.7	0.2	92.6	6.5	1302.8

Table 5. Effect of improvements and number of iterations on algorithm performance

n	b-r	w	p	CPU (sec.)	NoImp2 %	NoImp2 CPU (sec.)	500 NoImp2 %	500 NoImp2 CPU (sec.)	500 NoImp %	500 NoImp CPU (sec.)	500 OneImp %	500 OneImp CPU (sec.)
20	1	1	1	0.1	0.0	0.0	0.0	0.1	1.3	0.1	1.3	0.1
			2	0.1	0.0	0.0	-0.5	0.1	6.0	0.1	5.5	0.1
		2	1	0.1	0.0	0.0	0.0	0.2	2.0	0.1	2.0	0.1
			2	0.1	0.0	0.0	0.0	0.3	0.0	0.1	0.0	0.1
		3	1	0.1	0.0	0.0	-0.2	0.1	0.1	0.1	-0.2	0.1
			2	0.1	0.0	0.0	-2.2	0.2	-1.5	0.1	-2.0	0.1
	2	1	1	0.1	0.0	0.0	-0.9	0.1	4.4	0.1	2.2	0.1
			2	0.1	-0.7	0.0	-0.9	0.3	5.5	0.1	4.8	0.1
		2	1	0.0	0.0	0.0	-0.2	0.2	10.5	0.1	6.0	0.1
			2	0.1	0.0	0.0	-13.1	0.3	10.2	0.1	10.2	0.1
		3	1	0.1	0.0	0.0	-0.4	0.2	1.7	0.1	1.4	0.1
			2	0.1	0.0	0.0	-0.7	0.2	4.9	0.1	3.1	0.1
50	1	1	1	0.4	2.0	0.1	-0.2	0.6	15.2	0.2	8.3	0.2
			2	0.8	1.6	0.2	-0.5	0.9	12.1	0.5	5.3	0.5
		2	1	0.6	0.1	0.1	-1.1	0.5	11.2	0.3	6.4	0.3
			2	0.3	1.8	0.1	-1.5	0.6	12.0	0.2	11.2	0.3
		3	1	0.6	0.3	0.1	-0.1	0.6	8.8	0.3	3.1	0.4
			2	0.6	2.2	0.2	0.5	0.7	8.2	0.3	2.8	0.4
	2	1	1	0.7	1.1	0.2	-2.3	0.8	17.6	0.3	5.5	0.4
			2	1.1	1.6	0.3	-2.5	1.1	17.2	0.6	8.2	0.9
		2	1	0.6	2.0	0.2	0.9	0.8	13.8	0.2	7.9	0.3
			2	0.5	1.6	0.2	-1.1	0.7	12.5	0.3	6.9	0.6
		3	1	0.8	0.8	0.2	0.0	0.7	8.9	0.3	2.8	0.4
			2	0.8	0.2	0.2	-1.8	0.9	9.6	0.4	5.4	0.7
100	1	1	1	3.2	1.3	0.6	-1.5	2.7	20.6	0.7	4.6	0.9
			2	5.2	3.0	0.8	0.9	4.1	14.9	1.3	6.0	1.6
		2	1	2.9	2.4	0.5	1.0	2.3	13.1	0.7	4.0	0.9
			2	1.9	3.0	0.4	-0.2	1.8	13.8	0.7	4.4	0.8
		3	1	4.0	1.5	0.6	0.7	2.5	8.8	0.9	3.1	1.1
			2	4.0	3.0	0.7	2.0	2.9	10.0	1.1	2.5	1.4
	2	1	1	4.7	3.3	0.7	1.9	3.6	21.8	0.9	5.4	1.1
			2	7.1	2.1	1.2	0.7	5.6	18.5	1.7	7.1	2.0
		2	1	3.4	2.0	0.6	1.0	3.0	15.6	0.8	4.1	1.0
			2	3.7	4.0	0.7	0.9	3.3	18.1	0.9	7.0	1.2
		3	1	5.0	1.3	0.7	0.6	3.5	8.1	1.0	3.7	1.2
			2	6.4	2.0	0.9	0.2	4.3	10.7	1.5	5.0	1.8
200	1	1	1	24.1	2.5	3.2	1.9	16.7	18.5	3.1	2.6	3.5
			2	39.9	2.0	5.0	1.8	24.8	13.4	5.6	2.8	6.5
		2	1	20.2	2.0	2.6	1.3	13.2	13.6	2.6	1.9	3.1
			2	21.2	4.7	2.2	3.0	11.8	15.6	2.9	4.9	3.4
		3	1	27.1	1.6	2.9	1.4	14.9	7.6	3.3	1.9	3.9
			2	36.0	2.7	4.3	1.8	19.5	8.8	4.7	2.2	5.6
	2	1	1	33.1	4.1	4.1	2.4	21.4	20.1	4.0	4.2	4.3
			2	60.8	3.1	7.4	1.5	39.2	16.6	8.2	3.2	8.9
		2	1	31.5	2.3	3.8	1.3	19.1	12.5	3.5	2.2	3.9
			2	29.6	2.5	3.5	1.1	18.4	16.2	3.6	3.6	4.4
		3	1	37.3	0.8	4.0	0.4	20.4	6.8	4.0	1.5	4.5
			2	51.5	2.2	5.4	1.3	28.0	9.3	6.0	1.9	7.2
500	1	1	1	408.5	4.8	48.8	4.2	248.3	17.1	24.1	2.9	27.9
			2	786.2	5.0	94.2	3.9	486.5	11.8	52.7	3.9	62.5
		2	1	414.8	3.6	41.7	3.0	203.2	12.1	21.8	2.3	26.2
			2	359.4	5.5	31.9	4.3	157.0	14.6	19.9	3.5	24.0
		3	1	462.2	1.5	44.3	1.5	226.5	6.1	25.4	1.3	30.8
			2	775.1	3.1	72.0	2.6	355.5	8.0	41.7	1.5	49.6
	2	1	1	687.5	4.4	63.9	3.7	328.5	17.8	30.5	3.4	37.0
			2	1143.8	2.7	134.2	1.9	667.0	14.7	68.4	3.5	80.8
		2	1	695.7	2.5	56.4	2.0	284.5	12.0	26.8	1.6	32.6
			2	729.8	4.2	55.5	3.3	292.9	14.7	30.0	4.2	35.9
		3	1	803.1	0.9	60.4	0.8	311.2	5.3	32.6	1.2	39.8
			2	1302.8	1.9	98.9	1.5	491.1	8.4	52.8	2.2	62.9

The columns *500 NoImp2 %* and *500 NoImp2 CPU* present the results when the *CRS Algorithm* is executed for 500 iterations only with *Improvement1* and *Improvement3* at each iteration, in an attempt to improve the quality of solutions without the burden of the extra time by *Improvement2*. It can be seen that the solution times in this case are approximately half of the base setting with *Improvement2* and 100 iterations. Hence, the second improvement algorithm has a higher contribution to the solution time than the number of iterations. The negative percentages for small problems in the *500 NoImp2 %* column imply that better solutions can be obtained with this version as compared to the base setting, meaning that the randomization in the algorithm pays off over a larger number of iterations. On the other hand, although the number of iterations is fewer in the base setting, *Improvement2* still brings up to 4.3% increase in the objective for large problem instances.

The effect of the other two improvement algorithms on solution quality and time are also tested. For this purpose, the next two columns of Table 5 present the results when the *CRS Algorithm* is run for 500 iterations with no improvement. It can be seen that solution times are all less than 1 minute, even for the largest problem instances. However, the decline in the quality of solutions as compared to the base setting is considerable; up to 22% for some problem instances and 11% on the average. The solutions are significantly worse when compared to the previous setting, as well. Although these improvement algorithms take a considerable amount of time, they are quite effective in improving the solutions.

Finally, in an attempt to benefit from the effects of all improvement algorithms while reducing the solution time, the *CRS Algorithm* is run for 500 iterations, and all improvement algorithms are applied only once to the best solution at the end of all iterations. The results are listed in the *500 OneImp %* and *500 OneImp CPU (sec.)* columns. The percentages represent the average decline in the objective value as compared to the base setting. An average 4% decline is observed over all instances while the solution time is drastically reduced. This setting of the algorithm provides the closest solution quality to the base setting with much shorter run times for the largest problems with n = 500, although there is an average decline in solution quality of 2.6% for this instance set.

Due to the dynamic characteristic of the problem environment, short solution times might be preferable. Therefore, if small computation times are desired, the *CRS Algorithm* should be executed with 500 iterations (or more) without *Improvement2*, or with a single post-iterations execution of all improvement algorithms (the last setting). Since the solution times are ignorable for any combination for small problem instances (n = 20 and 50), running all combinations and selecting the best solution may also be an option. For larger problem instances, the solution quality improves with *Improvement2*, and therefore the base setting of the algorithm seems to be most appropriate, as the trade-off between solution quality and computation time is evident especially for these instances. The algorithm's improved solution quality justifies the increased processing time. Additionally, the algorithm demonstrates efficiency in handling large-scale problems, providing quick and effective results. The decision to increase the number of iterations for further refinement remains at the discretion of the decision-maker.

6. CONCLUSION

In this study, we consider the Combined Reservation Scheduling (CRS) problem for deciding the capacity and determining the schedule in systems where the incoming reservation requests have time windows for processing. The resources have varying fixed costs of usage. Many application areas are reviewed for the problem, as well as related literature. Our research contributes to literature and practice by introducing a novel heuristic approach to address this critical problem.

A randomized constructive heuristic is proposed for obtaining near-optimal solutions, employing effective improvement algorithms. We evaluate the performance of the developed algorithms through extensive computational experiments, testing different iteration limits and improvement schemes. The heuristic approach significantly outperforms CPLEX for problem instances up to 200 reservations. While computational time grows with problem size, primarily due to improvement algorithms, the algorithm's overall performance remains robust, meaning that this practical and effective approach can be directly implemented by industry practitioners and decision-makers.

As to the best of our knowledge, ours is the first study to propose heuristic algorithms for simultaneous capacity and scheduling decisions for the CRS problem. Our findings can provide significant positive social and economic benefits across various industries through optimization of resource use and scheduling. It is expected that the fast and effective solution approaches developed for this unique, important and novel problem will shed light on subsequent studies by laying the groundwork for future research. Effective solutions for large instances hold significant value for industries such as tourism, healthcare, transportation, logistics, and manufacturing. For example, our proposed approach can optimize gate assignments at airports by simultaneously determining the optimal number of gates to use and their corresponding schedules, considering varying gate costs.

The modular structure of the randomized algorithm can be easily adjusted by inclusion or exclusion of improvement algorithms and modification of the iteration limit. The proposed heuristic effectively addresses the problem, demonstrating superior performance compared to CPLEX for large problem instances. Its efficiency and adaptability make it a valuable tool for practitioners. While the current approach yields excellent results, future research could explore optimization with machine learning, or the potential of metaheuristics like genetic algorithms, tabu search, or particle swarm optimization. Comparative studies with the proposed heuristic could provide valuable insights. Additionally, incorporating factors such as resource shifts, availability constraints and time-dependent operating costs could enhance the model's applicability to a wider range of real-world scenarios. These additional features may be worth investigating in environments where the resources/servers are outsourced in a daily manner instead of long-term contracts.

Conflict of Interest
No potential conflict of interest was declared by the author.

Funding
Any specific grant has not been received from funding agencies in the public, commercial, or not-for-profit sectors.

Compliance with Ethical Standards
It was declared by the author that the tools and methods used in the study do not require the permission of the Ethics Committee.

Ethical Statement
It was declared by the author that scientific and ethical principles have been followed in this study and all the sources used have been properly cited.

The authors own the copyright of their works published in Journal of Productivity and their works are published under the CC BY-NC 4.0 license.

REFERENCES

Azizoglu, M. and Bekki, B. (2008). "Operational Fixed Interval Scheduling Problem on Uniform Parallel Machines", *International Journal of Production Economics*, 112(2), 756-768. https://doi.org/10.1016/j.ijpe.2007.06.004

Bard, J.F. and Rojanasoonthon, S. (2006). "A Branch-and-Price Algorithm for Parallel Machine Scheduling with Time Windows and Job Priorities", *Naval Research Logistics*, 53(1), 24-44. https://doi.org/10.1002/nav.20118

Barshan, M., Moens, H., Famaey, J. and De Turck, F. (2016). "Deadline-Aware Advance Reservation Scheduling Algorithms for Media Production Networks", *Computer Communications*, 77(1), 26–40.

Eliiyi, D.T. and Azizoglu, M. (2009). "A Fixed Job Scheduling Problem with Machine-Dependent Job Weights", *International Journal of Production Research*, 47(9), 2231-2256. https://doi.org/10.1080/00207540701499499

Eliiyi, D.T. and Azizoglu, M. (2011). "Heuristics for Operational Fixed Job Scheduling Problems with Working and Spread Time Constraints", *International Journal of Production Economics*, 132(1), 107-121. https://doi.org/10.1016/j.ijpe.2011.03.018

Eliiyi, D.T., Korkmaz, A.G. and Çiçek, A.E. (2009). "Operational Variable Job Scheduling with Eligibility Constraints: A Randomized Constraint-Graph-Based Approach", *Technological and Economic Development of Economy*, 15(2), 245-266. https://doi.org/10.3846/1392-8619.2009.15.245-266

Eliiyi, U. (2021). "Seasonal Reservation Scheduling with Resource Costs: A Mathematical Modeling Approach", *İzmir İktisat Dergisi*, 36(2), 409-422. https://doi.org/10.24988/ije.202136211

Faigle, U., Kern, W. and Nawijn, W.M. (1999). "A Greedy Online Algorithm for the k-track Assignment Problem", *Journal of Algorithms*, 31(1), 196-210. https://doi.org/10.1006/jagm.1998.1001

Fischetti, M., Martello, S. and Toth, P. (1987). "The Fixed Job Schedule Problem with Spread-Time Constraints", *Operations Research*, 35(6), 849-858. https://doi.org/10.1287/opre.35.6.849

Fischetti, M., Martello, S. and Toth, P. (1989). "The Fixed Job Schedule Problem with Working-Time Constraints", *Operations Research*, 37(3), 395-403. https://doi.org/10.1287/opre.37.3.395

Fischetti, M., Martello, S. and Toth, P. (1992). "Approximation Algorithms for Fixed Job Schedule Problems", *Operations Research*, 40(S1), 96-108. https://doi.org/10.1287/opre.40.1.S96

Gabrel, V. (1995). "Scheduling Jobs within Time Windows on Identical Parallel Machines", *European Journal of Operational Research*, 83(2), 320-329. https://doi.org/10.1016/0377-2217(95)00010-N

Garcia, J.M. and Lozano, S. (2005). "Production and Delivery Scheduling Problem with Time Windows", *Computers & Industrial Engineering*, 48(4), 733-742. https://doi.org/10.1016/j.cie.2004.12.004

Gertsbakh, I. and Stern, H.I. (1978). "Minimal Resources for Fixed and Variable Job Schedules", *Operations Research*, 26(1), 68-85. https://doi.org/10.1287/opre.26.1.68

Kolen, A.J.W. and Kroon, L.G. (1991). "On the Computational Complexity of (Maximum) Class Scheduling", *European Journal of Operational Research*, 54(1), 23-38. https://doi.org/10.1016/0377-2217(91)90320-U

Kolen, A.J.W. and Kroon, L.G. (1992). "License Class Design: Complexity and Algorithms", *European Journal of Operational Research*, 63(3), 432-444. https://doi.org/10.1016/0377-2217(92)90160-B

Kolen, A.J.W. and Kroon, L.G. (1993). "On the Computational Complexity of (Maximum) Shift Class Scheduling", *European Journal of Operational Research*, 64(1), 138-151. https://doi.org/10.1016/0377-2217(93)90014-E

Kolen, A.J.W., Lenstra, J.K., Papadimitriou, C.H. and Spieksma, F.C.R. (2007). "Interval Scheduling: A Survey", *Naval Research Logistics*, 54(5), 530-543. https://doi.org/10.1002/nav.20231

Kovalyov, M.Y., Ng, C.T. and Cheng, T.C.E. (2007). "Fixed Interval Scheduling: Models, Applications, Computational Complexity and Algorithms", *European Journal of Operational Research*, 178(2), 331-342.

Rojanasoonthon, S., Bard, J.F. and Reddy S.D. (2003). "Algorithms for Parallel Machine Scheduling: A Case Study of the Tracking and Data Relay Satellite System", *Journal of the Operational Research Society*, 54(8), 806-821.

Rojanasoonthon, S. and Bard, J.F. (2005). "A GRASP for Parallel Machine Scheduling with Time Windows", *INFORMS Journal on Computing*, 17(1), 32-51. https://doi.org/10.1287/ijoc.1030.0048

Spieksma, F.C.R. (1999). "On the Approximability of An Interval Scheduling Problem", *Journal of Scheduling*, 2(5), 215-227. https://doi.org/10.1002/(SICI)1099-1425(199909/10)2:5%3C215::AID-JOS27%3E3.0.CO;2-Y

Steiger, C., Walder, H. and Platzner, M. (2004). "Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks", *IEEE Transactions on Computers*, 53(11), 1393–1407. https://doi.org/10.1109/TC.2004.99

Wolfe, W.J. and Sorensen, S.E. (2000). "Three Scheduling Algorithms Applied to the Earth Observing Systems Domain", *Management Science*, 46(1), 148-168. https://doi.org/10.1287/mnsc.46.1.148.15134

Yu, G. and Jacobson, S.H. (2020). "Primal-Dual Analysis for Online Interval Scheduling Problems", *Journal of Global Optimization*, 77, 575–602. https://doi.org/10.1007/s10898-020-00880-5