TECHNOLOGICAL AND ECONOMIC DEVELOPMENT OF ECONOMY
Baltic Journal on Sustainability
2009
15(2): 245–266

OPERATIONAL VARIABLE JOB SCHEDULING WITH ELIGIBILITY CONSTRAINTS: A RANDOMIZED CONSTRAINT-GRAPH-BASED APPROACH

Deniz Türsel Eliiyi[1], Aslıhan Gizem Korkmaz[2], Abdullah Ercüment Çiçek[3]

[1, 2] Izmir University of Economics, 35330, Balcova, Izmir / Turkey
E-mail: [1]deniz.eliiyi@ieu.edu.tr; [2]aslihan.korkmaz@ieu.edu.tr

[3] Sabanci University, 34956, Orhanli, Tuzla, Istanbul / Turkey
E-mail: ercumentc@su.sabanciuniv.edu

Received 12 August 2008; accepted 4 May 2009

Abstract. In this study, we consider the problem of Operational Variable Job Scheduling, also referred to as parallel machine scheduling with time windows. The problem is a more general version of the Fixed Job Scheduling problem, involving a time window for each job larger than its processing time. The objective is to find the optimal subset of the jobs that can be processed. An interesting application area lies in Optimal Berth Allocation, which involves the assignment of vessels arriving at the port to appropriate berths within their time windows, while maximizing the total profit from the served vessels. Eligibility constraints are also taken into consideration. We develop an integer programming model for the problem. We show that the problem is NP-hard, and develop a constraint-graph-based construction algorithm for generating near-optimal solutions. We use genetic algorithm and other improvement algorithms to enhance the solution. Computational experimentation reveals that our algorithm generates very high quality solutions in very small computation times.

Keywords: operational variable job scheduling, eligibility constraints, optimal berth allocation, genetic algorithm, constraint satisfaction, constraint graph.

Reference to this paper should be made as follows: Eliiyi, D. T.; Korkmaz, A. G.; 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.

1. Introduction

In a typical scheduling problem, jobs represent the tasks that have to be processed, and machines correspond to the resources processing these tasks. In conventional scheduling, the decision-maker has the freedom of determining the starting times of the jobs. Making use of this flexibility, a schedule satisfying certain constraints or optimizing a certain criterion can be created. Interval Scheduling (IS) is an area of scheduling, where there is limited or no freedom in determining the starting times of jobs, as they are input to the problem. In IS, the scheduler faces the decisions of whether or not to accept an incoming job, and what resource to allocate to it in a parallel resource environment. Two different objectives may be of concern for an IS problem. A tactical IS problem tries to minimize the total cost of the resources to process all jobs. Alternatively, the operational IS problem assumes a fixed number of resources, and tries to select a subset of jobs for processing in order to maximize the total weight.

Fixed Job Scheduling (FJS), a special problem in IS area, is defined in an environment, where the tasks are to be processed on a number of resources that are working concurrently. In FJS, each job has a predetermined ready time and a deadline, and the processing time of the job is exactly equal to the difference between these times. That is, each task should start processing just as it arrives, or else it will be lost. The FJS problem has many practical applications in manufacturing and service environments, such as maintenance scheduling, transportation systems and shift scheduling.

Variable Job Scheduling (VJS), also referred to as parallel machine scheduling with time windows, is a generalization of the FJS problem that involves time windows larger than the processing times of jobs. In this problem, the decision-maker has some flexibility on determining the start time of an arriving job, since each job may have some slack time before it should start processing. The solution for a tactical VJS problem aims to minimize the total cost or number of the machines to process all jobs, while solution for an operational problem seeks the optimal subset of the jobs that can be processed with a fixed number of machines.

Although there is a considerable amount of literature on FJS problems, research has often been customized to specific applications, and results are rather scattered through literature. A study on Tactical Fixed Job Scheduling (TFJS) problem is the Bus Driver Scheduling Problem (Fischetti *et al.* 1987). Their objective is to find a proper set of driver duties at minimum cost while satisfying the constraints imposed by company regulations and union contracts. TFJS is used as the core model in capacity planning of aircraft maintenance personnel for an airline company (Kroon 1990). Their problem is to decide the number of engineers to carry out inspections on aircrafts in order to avoid aircraft delays. A later study deals with the operational variant of this problem with a given number of maintenance engineers, and priorities defined for maintenance activities (Kroon *et al.* 1995).

Operational Fixed Job Scheduling (OFJS) is studied to model the problem of scheduling earth-observing satellites (Wolfe and Sorensen 2000). Orbiting satellites are used to observe the earth and transmit data for further processing. The authors define priorities for each observation to prevent satellite conflicts, and try to maximize the amount of received data. Another contribution to the OFJS literature is the usage of a branch and bound algorithm that employs size reduction mechanisms and dominance conditions to solve the OFJS problem with spread time constraints (Eliiyi and Azizoglu 2006). In a further study, machine eligibility is taken into consideration (Eliiyi and Azizoglu 2008). In this study, the jobs have machine-dependent weights, i.e. each job may bring different profits depending on the processing machine, and some machines are not allowed to process some jobs. Another study uses a branch and bound algorithm for the OFJS problem with non-identical machine speeds (Azizoglu and Bekki 2008).

Two survey papers have been published recently (Kovalyov *et al.* 2007; Kolen *et al.* 2007). The first one presents a general formulation of the FJS Problem, reviews known models and algorithms. The second one presents the complexity of various FJS problems and suggests appropriate solution algorithms. For more detailed applications and the theory of the FJS problem, the reader is referred to these studies.

While FJS is considered extensively in literature, VJS has not gained its deserved attention. Despite its practical implications in manufacturing and service environments, very few researchers have focused on the problem after the pioneering contribution by Gertsbakh and Stern (1978). Their study proposes an approximate solution to the tactical VJS problem based on the entropy principle of informational smoothing. They use integer programming to formulate the tactical VJS problem. The problem does not include any machine or job classes (eligibility constraints), and all machines are identical in speed.

Another study uses the OFJS model for planning data transmission of low-orbit Earth observing satellites, and claims that the proposed model and algorithms can be adapted to solve the operational VJS (OVJS) Problem (Gabrel 1995). The majority of the study deals with OFJS. Job weights are identical; the objective is to maximize the number of jobs processed. There are machine and job classes, and not every machine is suitable to process every job. An incompatibility graph is defined, where a node represents a pair of a job and a machine such that the job can be processed by that machine. The maximum independent set in the incompatibility graph represents a schedule that maximizes the number of jobs performed. The authors develop lower and upper bounds for the OFJS problem. Computational results of upper and lower bounds are presented. Extension of the upper bounding procedure to OVJS is discussed, but no computational result is presented for the OVJS problem.

As to our best knowledge, there are only 2 studies focusing on OVJS (Rojanasoonthon *et al.* 2003; Rojanasoonthon and Bard 2005). Both consider the problem for a satellite system. The authors provide many interesting application areas for the problem. In both studies there are 2 different job priorities, where high priority jobs are infinitely more important than low priority jobs. Jobs have sequence-dependent setups, different classes and multiple time windows. Sequence-dependent setup structure requires the problem to be formulated like a Vehicle Routing problem. The authors develop a greedy randomized adaptive search procedure (GRASP) for the problem. Two concepts, backward and forward slacks, are used in the execution of GRASP. The definitions are very similar to the concepts of earliest start and latest start in project scheduling. GRASP is made up of 2 phases. In the first phase a restricted candidate list is formed. A solution randomly selected from this list is the starting solution of the second phase, where neighbourhood search is carried out to find a local optimum. The entire procedure is repeated many times. The computational results reveal that GRASP provides the best results when the first phase yields results close to the local optimum.

In a VJS problem, the machines can be identical or different, and jobs may be grouped into different classes in terms of size or priority. Each machine can process one or more classes of jobs. A job cannot be assigned to a machine that is not eligible for its class, and this makes up the eligibility constraint. In this study, the OVJS problem is considered with machine-dependent weight definitions for handling eligibility. An interesting application of the OVJS problem with eligibility constraints is the Berth Allocation Problem (BAP), the most crucial daily activity in port management, which emerged as a critical issue in the last decades (Murty *et al.* 2005). BAP affects the efficiency of all subsequent activities in port management. Each day, a number of ships with certain intervals of standby times arrive at the port and are either unloaded or loaded for commercial purposes. The aim is to maximize the total number or profit of the ships served, so that the port, as a whole, is better-off in economical terms. This requires optimal allocation of appropriate berths to the ships, and the problem can be modeled as an OVJS problem, where ships correspond to jobs to be processed and berths correspond to machines. Eligibility constraints are required since the ships differ in size and draft, and each class of ships has to be served on a suitable berth for its class.

There are many studies in BAP literature (Imai *et al.* 2001; Cordeau *et al.* 2005; Imai *et al.* 2007). In all studies, it is assumed that a ship arriving at the port waits to be berthed until an appropriate berth becomes available. To the best of our knowledge, none of the studies in BAP literature consider the fact that a ship may depart without being processed if the waiting time is too long, especially if it will visit several ports. Long waiting times may also cause the ship owner firms to switch to nearby ports in the long term. Either way, a loss of profit is expected if a ship is waited too long without processing.

Such a case exists in the Port of Izmir, the most important container port in Turkey (800.000 TEU in 2006) located at the far west of Anatolia on the intersection point of heavy traffic. Due to its strategic location in the Aegean Sea, the port is an ideal node for import/export between Europe and Asia. The waiting times of the ships (before they are berthed) are considerably high in this port. In January and February of 2008, waiting times for 41% of the incoming ships are realized in 1–2 days, which is regarded as a very high value (Biricik 2008). Related to this fact, the port has started to lose its importance. The daily cost of waiting ships in Izmir Port is estimated as $300 000. This figure is an estimate of the total short and long term costs realized as a result of long waiting times. Hence, it is obvious that there is a standby time (an upper bound on the waiting time) for each incoming ship, after which the ship owner may switch to other ports either in the short term or in the long one. In this respect, our model also has an economic impact for such ports as the Port of Izmir.

For reasons explained above, our study fills important gaps in both BAP and VJS literature. Many other application areas for the OVJS problem includes similar assignment-type problems from both manufacturing and service environments, such as reservation systems, the scheduling of cars in a car repair service, the assignment of aircrafts to the gates of an airport, or assignment of holiday bungalows to campers. In the next section, the OVJS problem is defined and the mathematical model is formulated. Section 3 presents the development of a randomized constraint-graph-based heuristic. In order to improve the quality of the heuristic solution, a genetic algorithm and other improvement algorithms are developed, and the specifics are presented in the 4th section. Results of the computational experimentation are provided in Section 5. Finally, conclusions and future research issues are noted.

2. The operational variable job scheduling problem

The working time of a machine encloses all the time windows that the machine operates on a single day. We divide a day into time intervals, and the planning horizon extends to the next day, as will be explained later.

There are I jobs to be processed on K unrelated machines working simultaneously. Given job i, the arrival time is denoted by a_i, and the job can wait until the time b_i to start processing. If the job is not assigned to any machine until b_i, it leaves the system without being processed. The processing time and the weight of job i on machine k is p_i and w_{ik}, respectively. We make the following assumptions throughout the study:

- All numerical data, a_i, b_i, p_i, and w_{ik} are non-negative integers and are known with certainty.
- Preemption is not allowed, a job can only be processed in one machine without any interruption.

We assume T time intervals that are equal in length. I_t is the set of jobs available for processing in time interval t, and T_i is the set of time intervals for job i. That is,

$$I_t = \{ i \mid a_i \le t, b_i + p_i - 1 \ge t \},$$

$$T_i = \{ a_i, \ldots, b_i + p_i - 1 \}.$$

We define binary decision variables x_{itk} and y_{ik} as follows:

$$x_{itk} = \begin{cases} 1, & \text{if time interval } t \text{ of job } i \text{ is processed on machine } k \\ 0, & \text{otherwise} \end{cases}$$

$$y_{ik} = \begin{cases} 1, & \text{if job } i \text{ is assigned to machine } k \\ 0, & \text{otherwise} \end{cases}$$

Constraints of the model are listed below:

- If a job is assigned, every time interval of the job should be assigned:

$$\sum_{t \in T_i} x_{itk} = p_i y_{ik}, \qquad i = 1, \ldots, I, \qquad k = 1, \ldots, K. \tag{1}$$

- A job can be assigned to at most one machine:

$$\sum_{k=1}^{K} y_{ik} \le 1, \qquad i = 1, \ldots, I. \tag{2}$$

- For each time interval, any machine available in that interval can process at most one job:

$$\sum_{i \in I_t} x_{itk} \le 1, \qquad t = 1, \ldots, T, \qquad k = 1, \ldots, K. \tag{3}$$

- If a job is assigned, each time interval of the job should be assigned consecutively:

$$p_i x_{itk} - p_i x_{i,t+1,k} + \sum_{j=t+2}^{b_i + p_i - 1} x_{ijk} \le p_i, \qquad i = 1, \ldots, I, \qquad k = 1, \ldots, K, \qquad t \in T_i. \tag{4}$$

– Integrality constraints for job-machine assignments:

$$x_{itk} \in \{0,1\} \quad y_{ik} \in \{0,1\}, \quad i = 1,\dots,I, \quad k = 1,\dots,K, \quad t = 1,\dots,T-1. \tag{5}$$

The objective function requires the maximization of the total weight of the jobs processed:

$$\text{Maximize} \sum_{i=1}^{I} \sum_{k=1}^{K} w_{ik} y_{ik}. \tag{6}$$

We handle eligibility using machine-dependent weight definitions as in Eliiyi and Azizoglu (2008), where a weight of –1 is assigned if a job cannot be processed on a machine, and inappropriate assignments are avoided by the objective function. Namely, we define the weight w_{ik} of the job i, when assigned on machine k as follows:

$$w_{ik} = \begin{cases} w_i, & \text{if machine } k \text{ is eligible to process job } i, \\ -1, & \text{otherwise.} \end{cases}$$

The defined OVJS problem reduces to the OFJS problem with general weights, when $a_i = b_i$, $\forall i$. The OFJS problem with general weights is shown to be NP-hard in the strong sense by Eliiyi and Azizoglu (2008). Hence, so is our problem. As in the case of the berth allocation example, the problem is an operational one, which has to be solved daily or even hourly in some cases, and practical solution procedures are necessary. For this reason, we develop a constraint-graph-based heuristic exploiting the structural characteristics of the problem. In the next section, we define the specifics of our heuristic.

3. RCGA: a randomized constraint-graph-based algorithm

Since the OVJS problem with general weight definitions is NP-hard in the strong sense, a heuristic algorithm called the Randomized Constraint-Graph-Based Algorithm (RCGA) is developed for generating near-optimal solutions. We explain the details of RCGA in this section.

Assuming all jobs are scheduled to a single machine, which will most probably lead to infeasibility, the colliding time windows (overlaps) between any two jobs is defined as a constraint (stress) on those jobs. The structure can be shown in a constraint satisfaction graph, where nodes represent the jobs, an edge represents the constraint existing between any two jobs, and the weights associated with the edges correspond to the number of overlapping time intervals between jobs, i.e. the length of an overlap. The degree of a node (total weight of emanating edges from that node) determines the total constraint on that job.

Unlike the FJS problem where the starting times of the jobs are fixed, the overlaps in a VJS problem may change depending on the starting times of the jobs. That is, there is no single constraint graph representation for the problem as the starting time of a job may vary between its a_i and b_i. For this reason, we generate 3 representative constraint graphs by assigning the jobs on their arrival times (a_i), on their standby limits (b_i), and a random time between a_i and b_i. The total degree of a node is calculated using these graphs to obtain a single value for each job. After that, the degree is divided by the original weight (w_i) of the job, since the objective function tries to maximize the total weight of the assigned jobs. Then, based on a least-constraint-variable approach, the jobs are scheduled to the machines in non-decreasing order of their adjusted degrees. Hence, the algorithm encourages low-overlap and/or high-weight jobs to be assigned first.

We assume 2 job and machine classes: small and large. The eligibility structure is assumed to be nested; small machines can only process small jobs, while large machines can handle both classes of jobs. This assumption is consistent with a BAP example, where large ships (in size or draft) can only be served by large or deeper berths, but small ships can be assigned to any berth. While RCGA is designed for this eligibility structure, it can easily be adapted for different structures, as well.

This structure creates the need to solve the problem in 2 stages: we first develop constraint graphs for small jobs and try to assign them giving priority to small machines. A set of jobs that could not be assigned in this step are combined with the set of large jobs, and further constraint graphs and assignments are made. An unassigned job is assigned on the first available eligible machine at this stage. After the order in which the jobs will be considered is determined, the next question is how and when to assign each job. The following 4 methods are proposed for this purpose:

1. Assign the next job to the earliest time available in the interval $[a_i, b_i]$.
2. Assign the next job to the latest time available in the interval $[a_i, b_i]$.
3. Generate a random integer r in the interval $[a_i, b_i]$. Assign the next job to the earliest time available in the interval $[r, b_i]$.
4. Generate a random integer r in the interval $[a_i, b_i]$. Assign the next job to the latest time available in the interval $[r, b_i]$.

To incorporate randomization into the algorithm, one of the above 4 methods is selected randomly, and the next job is assigned with the selected method. The pseudo code of the RCGA algorithm is presented in Fig. 1. The whole algorithm is executed repeatedly, and the best solution is stored.

We demonstrate the execution of the algorithm with an example. For this purpose, a berth allocation problem with 10 ships and 3 berths (1 small, 2 large) is considered. Problem parameters are given in Table 1. Figures in bold characters belong to large ships. As it was stated before, ships belonging to the small class, which can be processed by both classes of berths, are considered first. Here, the goal is to assign small ships to small berths as much as possible, since large ships cannot be assigned to small berths. If there are more than two job and machine classes in the problem, the adaptation of the algorithm should start with the least capable machine class and its corresponding job class, and then move towards more capable classes.

Table 1. Parameters for the example

Ship i	a_i	b_i	p_i	w_i
1	12	15	8	17
2	15	23	6	33
3	6	17	10	12
4	11	17	6	24
5	6	20	7	26
6	3	8	11	35
7	2	5	12	17
8	4	5	5	9
9	7	20	15	14
10	9	19	5	26

```
CalculateNodeDegrees(joblist)
Input: joblist
Output: degree(joblist)
Begin
  For each job $j \in$ joblist
        Assign $j$ to a dummy berth at $a_j$
        Compute the degree of each job: degree($j$) = total number of overlapping time intervals of $j$ with other jobs
  End For
  For each job $j \in$ joblist
        Assign $j$ to a dummy berth at $b_j$
        degree($j$) = degree($j$) + total number of overlapping time intervals of $j$ with other jobs
  End For
  For each job $j \in$ joblist
        Assign $j$ to a dummy berth at a random time between $[a_j , b_j]$
        degree of each job: degree($j$) = degree($j$) + total number of overlapping time intervals of $j$ with other jobs
  End For
  For each job $j \in$ joblist degree($j$) = degree($j$) / weight($j$)
End


MAIN (RCGA)
Input: joblist, machinelist, NumRep
Output: BestAssignment
Begin
  Do
   I = I + 1;
   CalculateNodeDegrees(joblist);
        Sort joblist in increasing order of degree(joblist);
        For Each job $j \in$ joblist
              Select an assignment method randomly
                  Assign $j$ with the selected method;
        Return assignment;
        Update BestAssignment if assignment is better than the current best;
        Until I = NumRep
        Return BestAssignment
End
```

Fig. 1. The pseudo code of the RCGA algorithm

We demonstrate an iteration of RCGA. Note that each iteration may result in a different feasible solution as the algorithm has random components. After a predetermined number of repetitions, the best solution is output. At the first stage, the time overlaps between the ships are determined. For this purpose, all ships are assigned on their arrival times (a_i), on their standby limits (b_i), and a random time between a_i and b_i. Corresponding overlaps for small ships on a 24-hour timeline are shown in Fig. 2, where each row represents a single ship's assignment on the time axis. Note that all 3 assignments in Fig. 2 are infeasible for a single berth due to overlaps. The lengths of the overlaps between job pairs are used in forming the constraint graphs. The constraint matrices and the corresponding constraint graphs for each assignment in Fig. 2 are presented in Fig. 3.

The degree of a ship is calculated as follows: The degrees on each assignment of that ship are summed, then the sum is divided by the weight of the ship. The resulting adjusted degrees for the small ship class are shown below using numbers in Fig. 3:

degree(Ship 1) = (14+11+14) / 17 = 2.29

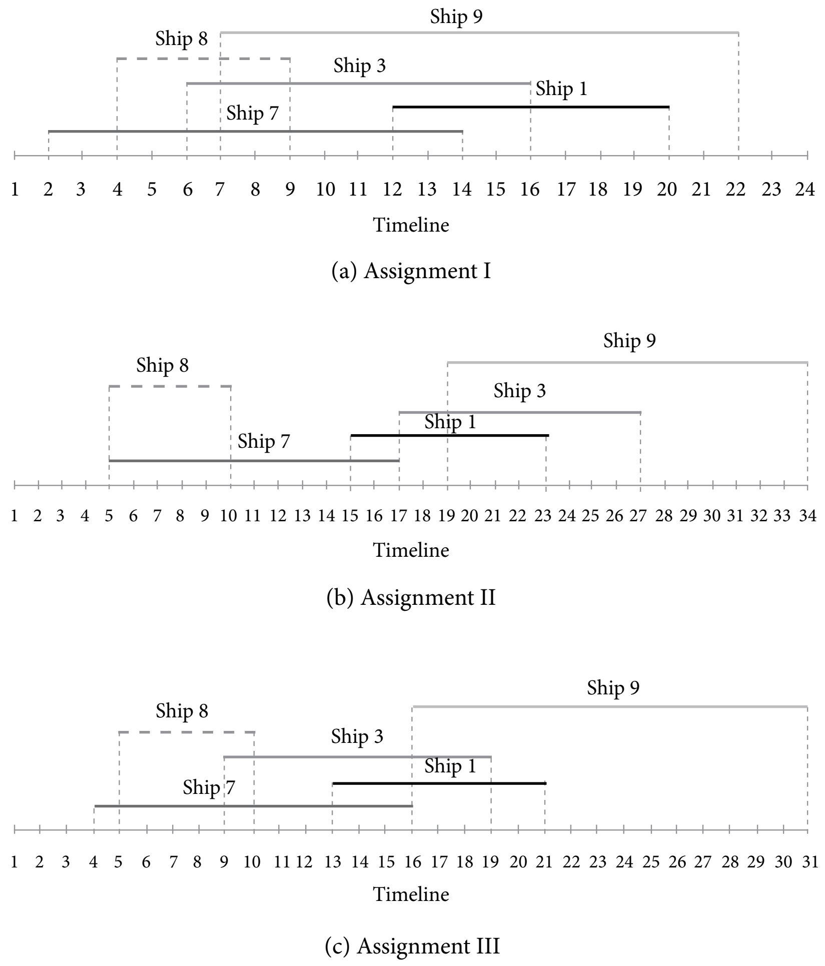


Fig. 2. Overlaps of small ships when ships are assigned at (a) on their arrival times, (b) on their standby limits, and (c) on random times between arrival times and standby limits

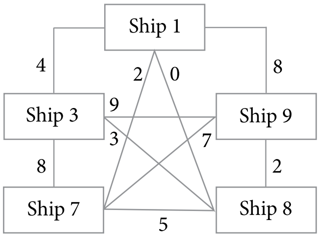
degree(Ship 3) = (24+13+17) / 12 = 4.50
degree(Ship 7) = (22+7+15) / 17 = 2.58
degree(Ship 8) = (10+5+6) / 9 = 2.33
degree(Ship 9) = (26+10+8) / 14 = 3.14

Based on the results, the assignment order of the small ships is the following: Ship 1, Ship 8, Ship 7, Ship 9 and Ship 3. Note that there is only one small berth in the example. The assignment is made by selecting among the 4 methods randomly. For demonstration purposes, assume that method (1) is selected for Ship 1; that is, try to assign the ship to the earliest time available in the interval [a_1, b_1]. As Ship 1 is the first ship to be assigned, the berth is empty and Ship 1 is assigned on $a_1 = 12$. The berth becomes occupied in the interval (12, 20). Next, let's assume method (2) is picked randomly for Ship 8, i.e. the latest assignment possible. After assigning Ship 8 on $b_8 = 5$, the berth is occupied between [5, 10) and [12, 20). For Ship 7, assume that method (3) is selected, i.e. the ship will be assigned to the earliest time after a random point r in $[a_7, b_7]$. Let $r = 3$, which is in [2, 5]. Checking for the availability of the berth during the time intervals [3, 15), [4, 16) and [5, 17), we see overlaps with previously assigned ships. Thus, Ship 7 cannot be assigned and left for the reconsideration with the large ships.

Now assume that method (1) is picked for Ship 9. Considering time windows 7 to 19 yields overlaps, but the berth is idle on time window $b_9 = 20$. The assignment is then made, and the berth becomes occupied at intervals (5, 10), [12, 20) and [20, 35). One can easily examine the example and observe that Ship 3 cannot be assigned. The resulting assignments are illustrated in Fig. 4.

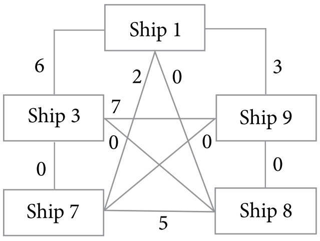
(a) Assignment I

	Ship 1	**Ship 3**	**Ship 7**	**Ship 8**	**Ship 9**
Ship 1	–	4	2	0	8
Ship 3	4	–	8	3	9
Ship 7	2	8	–	5	7
Ship 8	0	3	5	-	2
Ship 9	8	9	7	2	–
Sum	**14**	**24**	**22**	**10**	**26**



(b) Assignment II

	Ship 1	**Ship 3**	**Ship 7**	**Ship 8**	**Ship 9**
Ship 1	-	6	2	0	3
Ship 3	6	–	0	0	7
Ship 7	2	0	–	5	0
Ship 8	0	0	5	–	0
Ship 9	3	7	0	0	–
Sum	**11**	**13**	**7**	**5**	**10**



(c) Assignment III

	Ship 1	**Ship 3**	**Ship 7**	**Ship 8**	**Ship 9**
Ship 1	–	6	3	0	5
Ship 3	6	–	7	1	3
Ship 7	3	7	–	5	0
Ship 8	0	1	5	–	0
Ship 9	5	3	0	0	–
Sum	**14**	**17**	**15**	**6**	**8**

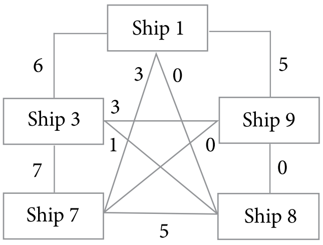


Fig. 3. Constraint matrices and corresponding constraint graphs for the assignments in Fig. 2

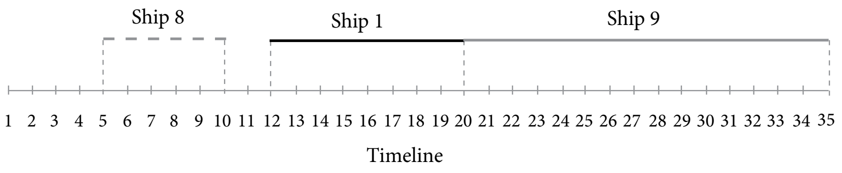


Fig. 4. Assignments of small ships to the small berth, obtained from one execution of RCGA

Next, the algorithm moves on to the larger ship class. The same steps will be followed for the ship set, which is composed of the large ships and the small ships that could not be assigned in the first phase. Therefore, the ship set to be considered in the second phase includes ships 2, 3, 4, 5, 6, 7, and 10. As the execution is identical, the details are skipped. An exemplary final assignment for the large berths is illustrated in Fig. 5.

In the next section, improvement algorithms developed for enhancing the solution generated by RCGA are presented in detail.

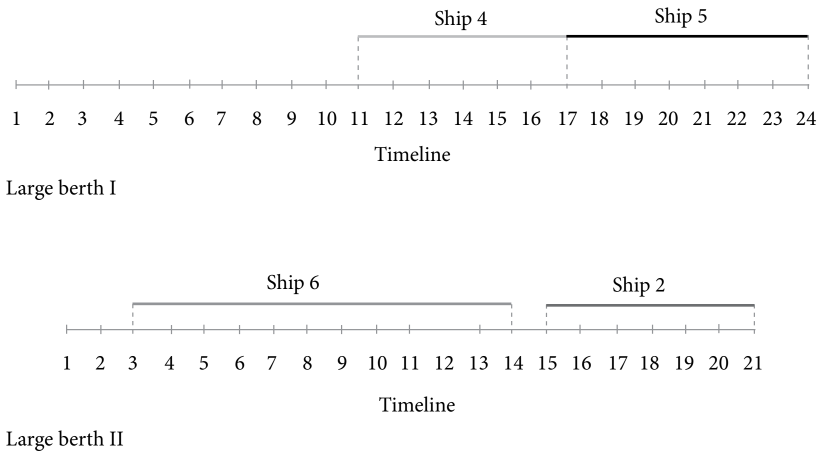


Fig. 5. Assignments for the 2 large berths obtained from one execution of RCGA

4. Improvement algorithms

We propose 3 improvement algorithms to be executed after RCGA for enhancing the solution quality. The first one is a genetic algorithm, details of which are provided in Section 4.1. The second algorithm tries to swap 2 jobs between machines. The algorithm performs a swap only if an unscheduled job can be inserted into the schedule after the swap, hence resulting in a better objective function value. The third improvement algorithm tries to shift the starting times of the jobs, in search of the possible insertion of an unscheduled job.

4.1. Genetic algorithm

Genetic algorithms (Goldberg 1989) are search and optimization algorithms, which are inspired by the rules of natural evolution. A population of candidate solutions is coded by genes, and an optimal solution is searched via creating new chromosomes by crossover and mutation. Genetic algorithm (GA) was used to solve the tactical FJS problem by Santos, Zhong (2001), where each gene contains information about assignments of the jobs to the machines. However, due to the more complex nature of the OVJS problem, starting time of a job is another decision in our study.

An initial population made up of n chromosomes is necessary to start the procedure. Each chromosome represents a feasible solution to the problem, and each gene of the chromosome corresponds to a job. The quality of the solution represented by a chromosome is evaluated by its fitness metric. Fitness of a chromosome is the sum of the weights of assigned jobs in that chromosome. The goal of the algorithm is to maximize the fitness value. After an initial population is formed by selecting pairs of chromosomes (parents) according to a criterion (selection process), crossover operations are performed and 2 new chromosomes (offspring or children) are generated. The offspring are mutated with a certain probability and inserted into the new population. The procedure is repeated for a prespecified number of iterations.

4.1.1. Representation

Each chromosome is a list of size I, where each index contains job assignment information including the assigned machine (k), the starting time interval (t) and the variable weight (w) for a job i on machine k. Below is the representation of a chromosome c. The assignment information is represented as a list with angle brackets. Each index (assignment information for each job) is referred using square bracket notation.

$$c = \{x_1, \dots, x_I\},$$

$$c[x_i] = \begin{cases} \langle w, t, k \rangle & \text{if interval } t \text{ of job } i \text{ is assigned to machine } k \\ 0, & \text{otherwise} \end{cases} \qquad i = 1, \dots, I.$$

The dot (.) notation is used to refer to the attributes <w, t, k> of each index of the chromosome. We store the weight information of a job w_{ik} to handle eligibility. The fitness of a chromosome is calculated as the total weight of jobs assigned at time interval t on machine k:

$$f(c) = \sum_{i=1}^{I} c[x_i].w_{ik}. \tag{7}$$

4.1.2. Selection

In each iteration of the algorithm, parents for the crossover operation are selected in the following manner. The population is sorted in a non-increasing order of the fitness metric. Crossover Probability (CP) is the probability that a chromosome will reproduce. Each individual of the population is directly passed on to the next iteration with (1-CP) probability.

This portion is assumed to include the individuals that are unable to reproduce. The algorithm therefore has a memory; it does not forget all the information stored in a previous iteration. Next, the rest of the population that is subject to crossover is partitioned into 4 equal groups, where the first group represents the fittest chromosomes and the last group represents the worst. The parents are picked with a bias towards better groups, with probabilities 0.4, 0.3, 0.2 and 0.1, respectively. Hence, while fitter chromosomes are favoured with the expectation to produce fitter offspring, the possibility that 2 unfit parents may produce a good offspring is not totally discarded. Value of *CP* is subject to experimentation.

4.1.3. Crossover

Assume that 2 parents (c_1 and c_2) are selected for crossover. For this operation, initially a clone of c_1 is created as offspring c. Then, for each gene on c, the corresponding genes on c_2 are examined. A heuristic function is developed for this purpose, which computes the difference between the weight of the first job (gene) of c and the maximum-weight overlapping job in c_2. So, the gain or loss of assigning the job at that time interval is estimated. With this crossover function, if a job is not assigned in any parent, it won't be assigned in the child. If a job is assigned in one or both of the parents, the assignment maximizing the heuristic function is preferred. The function can be expressed as follows:

$$h\left(c\left[x_i\right], c_2\left[x_a\right]\right) = c\left[x_i\right].w - \max_a\left\{c_2\left[x_a\right].w\right\}, \text{ where } a = 1,\dots,I \wedge c\left[x_i\right].t \text{ overlaps with } c_2\left[x_a\right].t \wedge c\left[x_i\right].k = c_2\left[x_a\right].k. \tag{8}$$

For each parent pair, a second crossover operation is performed in the reverse order of genes. In the end, 2 new chromosomes are obtained.

4.1.4. Mutation

Each newly generated gene is subject to mutation with a predetermined Mutation Probability (*MP*). Mutation aims to prevent being stuck at local maxima by jumping to different points in the solution space. Two types of mutations are equally likely to occur. The first type randomly picks an unassigned job *i* and a machine *k*. Next, the least-cost-interval for the job in [a_i, b_i] is computed based on the sum of the jobs weights that have to be removed from schedule for job *i* to be assigned on machine *k*. The job is then scheduled in its least-cost-interval removing all overlapping jobs. The second type of mutation randomly picks a scheduled job and simply removes it from the schedule. Our mutation scheme tries to favour good mutations by minimizing the cost of mutation, while not totally discarding the option of removing a job from schedule. The value of *MP* is subject to experimentation.

GA can also be used as a construction algorithm for the OVJS problem. For this purpose, an initial population is generated randomly, and GA is run for a prespecified number of iterations. In Section 5.2, we provide the results of the computational experimentation, when our GA is used for both construction and improvement purposes.

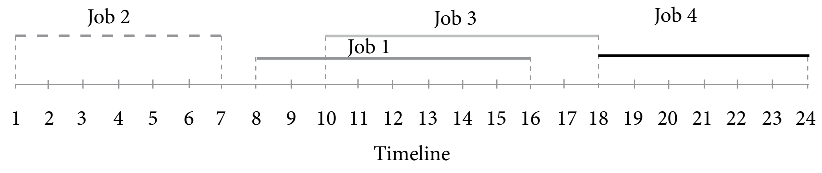
4.2. Swap & Insert

The idea of the second improvement algorithm is the following: two jobs that are assigned on different machines of the same class can swap machines if there is no resulting overlap on any machine. Doing so, it may be possible to open up a space on any of the machines and insert one or more unscheduled jobs into the schedule. The objective function is improved as a result of the insertion. The algorithm performs the best swap & insert move, and continues until there are no possible moves.

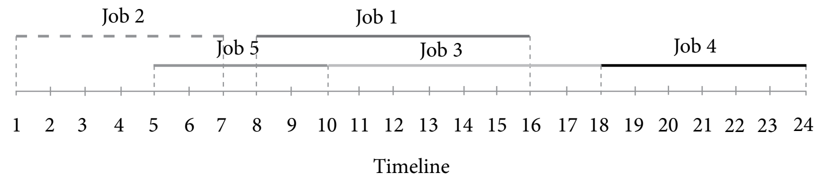
Another example is shown in Fig. 6, where 2 different schedules for 2 machines of the same class are presented. Each row represents the assignment of jobs on a particular machine. Job 5, which can be processed by this class of machines, having parameters $a_5 = 5$, $b_5 = 7$ and $p_5 = 5$, is out of schedule in Fig. 6(a), since it overlaps with Job 2 or 3 on the first machine and Job 1 on the second machine. Note that Job 1 and Job 3 can swap machines without disturbing any other job in schedule. After the swap, the second machine is now available for inserting Job 5 as depicted in Fig. 6 (b). The objective function is thus improved by an amount w_5.

4.3. Shift & Insert

The goal of the third improvement algorithm is to shift the starting time of scheduled jobs for creating a space that may allow the insertion of an unscheduled job. For this purpose, the algorithm considers each machine separately. For each pair of consecutively scheduled jobs on a machine, say job *i* and job *j*, the method tries to shift all jobs assigned before job *i* (including job *i*) backward to their arrival times, and shift all jobs that are assigned after job *j* (including job *j*) forward up to their standby limits. This is done in an attempt to form an available time interval between jobs *i* and *j*. Among the unscheduled jobs, candidates for the



(a) Before Swap & Insert



(b) After Swap & Insert

Fig. 6. Example for demonstrating the execution of the Swap & Insert algorithm

new time space are checked for selecting the job with the best insertion performance. The algorithm continues until no insertion is possible.

An example is shown in Fig. 7. Consider the machine in Fig. 7(a) processing jobs 1 and 4. Assume that Job 1 has $a_1 = 5$, and Job 4 has $b_4 = 18$. Job 6, which can be processed on this class of machines, having parameters $a_6 = 13$, $b_6 = 14$ and $p_6 = 3$, is currently out of schedule, since it overlaps with Job 3 on one machine and Job 1 on the other. Job 1 can be shifted back up to time 5, but Job 4 cannot be shifted forward as it is already scheduled on b_4. As a result of the shift, the machine becomes available for Job 6 to be inserted at time 13 or 14, as depicted in Fig. 7(b), and the objective function is improved by an amount w_6.

5. Computational experimentation

In order to evaluate the performance of our algorithms, an experimental study is carried out. We give the details of the experimental design in section 5.1, and discuss the results in 5.2.

5.1. Experiment design

In order to make our computational study represent a practical situation, the experiment is designed in accordance with the statistics obtained from the Izmir Port authority. The port adopts a 24-hour workday, as three 8-hour shifts. The berths are categorized in 2 classes, as are the incoming ships. The weight of a ship is correlated with the number of containers loaded/unloaded.

Random test problems are generated accordingly for 10, 20, 30 and 40 jobs and 2, 3 and 4 machines. The port has a fixed number of berths. However, the effect of the number of

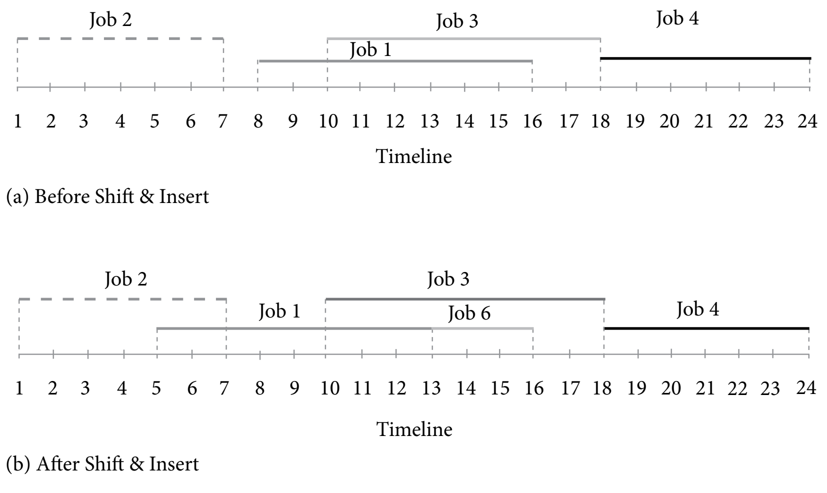


Fig. 7. Example for demonstrating the execution of the Shift & Insert algorithm

machines on the performance of the algorithms also needs to be investigated. Hence, the data is generated for different problem sizes. Arrival times and standby limits are distributed uniformly between [0, 24], processing times are distributed uniformly between [5, 15], and the weights are distributed uniformly between [5, 10]. There are 2 job and machine classes, small and large. We assume that the machines are divided equally between the two classes. But in the 3-machine instances, one machine is assumed to be small and the other two larger. Small machines cannot process jobs with weights greater than 7.5, the midpoint of the weight range [5, 10].

All algorithms are coded in C# programming language using MS Visual Studio 2005, built on .NET 2.0 Framework. Solution performances are measured on a PC with 1.70 GHz Intel Pentium M processor, 592 MHz FSB and 512 MB RAM. The optimum solutions of the problem instances are found using LINGO 8.0 Unlimited Edition with MIP Solver. The same PC hardware configuration is used for LINGO runs.

5.2. Computational results

A pilot experimentation is performed initially in order to observe the performance of the GA, when it is used as a construction algorithm for our problem. This initial experiment also reveals the best setting for algorithmic parameters for the GA.

For this purpose, 72 different levels are set for the algorithmic parameters, with population sizes 160, 320 and 600, iteration numbers 150, 300, 500 and 1000, mutation probabilities (*MP*) 0.5 and 1.0, and crossover probabilities (*CP*) 0.9, 0.8, and 0.5. The initial population is generated randomly. The computation times and the quality of the resulting lower bounds are compared. Table 2 presents the best 4 parameter settings that yield minimum gaps and shortest average times after the pilot runs.

Table 2. Best levels of parameters observed in the pilot experimentation

GA #	Population Size	# of Iterations	*MP*	*CP*
1	600	1000	1	0.9
2	160	1000	1	0.9
3	600	1000	0.5	0.8
4	600	150	0.5	0.8

The pilot experiments also reveal that the procedures Swap & Insert and Shift & Insert take practically no time but they are very useful in terms of improving solution quality. Therefore both algorithms are executed after each GA to come up with better solutions. Table 3 presents the performance of the GA with the above parameter settings. Problem instances are identified by their number of jobs and machines, i.e. 10/2 represents a problem with 10 jobs and 2 machines. The column heading "GA1+ Imp." indicates the execution of GA1, Swap & Insert and Shift & Insert in a sequential manner. Average gap columns represent the percentage gaps between the lower bounds found by GA and the optimum objective function value, computed as:

$$\frac{Optimum - OBJ(GA)}{Optimum}.$$

In cases where optimal solutions cannot be obtained, the best feasible solutions up to that moment, found by LINGO, are used in average gap computations.

Table 3. Performance of GA as a construction algorithm

Problem	LINGO (Optimum)	GA1+ Imp.		GA2+ Imp.		GA3+ Imp.		GA4+ Imp.	
	CPU Time	Avg. Gap	CPU Time	Avg. Gap	CPU Time	Avg. Gap	CPU Time	Avg. Gap	CPU Time
10 / 2	26.50	0.00	238.46	0.00	27.29	0.00	224.85	0.01	34.03
10 / 3	293.00	0.00	287.02	0.03	27.74	0.00	227.85	0.05	34.37
10 / 4	204.10	0.01	287.37	0.02	27.86	0.01	227.33	0.03	34.47
20 / 2	268.20	0.02	322.95	0.05	36.49	0.02	261.54	0.06	39.62
20 / 3	3687.50 ^{5}	0.01	326.98	0.09	37.65	0.01	261.41	0.10	39.89
20 / 4	4140.80 ^{1}	−0.02	331.20	0.07	38.08	−0.02	263.52	0.06	40.50
30 / 2	1449.80 ^{7}	0.07	358.40	0.08	47.42	0.07	294.12	0.07	44.49
30 / 3	4486.60 ^{2}	0.04	361.82	0.08	48.60	0.05	291.09	0.06	44.55
30 / 4	7200 ^{10}	0.05	369.29	0.07	50.90	0.05	296.15	0.05	45.75
40 / 2	3730.00 ^{5}	0.07	400.28	0.07	50.02	0.07	333.47	0.10	50.44
40 / 3	4261.90 ^{3}	0.09	404.28	0.11	61.19	0.10	327.85	0.09	50.83
40 / 4	7200 10	0.08	401.23	0.09	60.82	0.07	325.98	0.08	50.58

- All CPU times are in seconds.
- The superscripts denote the number of the instances (out of 10) that could not be solved within 2 hours of computation time.

As expected, Table 3 reveals that the average computation time decreases as population size and number of iterations decrease. It can be seen that the population size and the number of iterations affect the result more than the mutation probability and the crossover probability, as GA1 cannot compete with GA2, which only differ in population size. This is also true between GA3 and GA4 due to the difference in number of iterations. There is a trade-off between the quality of the result and time needed. It seems that for small problem sizes, LINGO outperforms GA, as it terminates upon finding the optimal solution but GA runs for a predetermined number of iterations. As the problem size increases, LINGO fails to terminate even within 2 hours, while GA finds good solutions in much smaller times. Note that for the 20/4 setting, GA1 and GA3 found better solutions than the best lower bound found by LINGO, resulting in negative gap values. One might question why GA3 performs worse than GA4 on the 40/3 setting, since they have identical parameters, except the number of iterations. This is because in one or more instances of this set, although an assignment with a worse lower bound is obtained with GA4, the improvement algorithms reach an objective function value that is even higher than the one obtained by GA3 in 1000 iterations.

Solution times do not vary among different instances when GA is used as a construction algorithm, since a prespecified number of iterations are run with a given population size. Note that a 20/3 instance is a moderate size instance for the problem, as it can be seen from the optimal solution time. The CPU time of any GA for this instance is much less than that of LINGO. As a trial, a 30/3 instance is allowed to run on LINGO until the optimal solution is found. The solution is found in approximately 13 hours, while the slowest GA provides a solution in only 6 min. The maximum gap is only about 0.13 for this trial instance.

Next, the performance of RCGA is evaluated. Among the four GAs presented in Table 2 and 3, GA3 with $MP = 0.5$ and $CP = 0.8$ is selected as the improvement algorithm since it dominates GA2 and GA4 in terms of solution quality, and GA1 in terms of computation time. RCGA is executed as explained in Section 3. Improvement algorithms Swap & Insert and Shift & Insert are executed as well, as they improve the solution with no time cost at all. Table 4 presents the average gaps and CPU times when RCGA is run for 600 repetitions, when GA3 is executed as an improvement algorithm after RCGA600, and when RCGA is run for 10 000 repetitions.

Table 4. Performance of RCGA

Problem	LINGO (Optimum)	RCGA600 + Imp.		RCGA600 + GA3 + Imp.		RCGA10000 + Imp.	
	CPU Time	Avg. Gap	CPU Time	Avg. Gap	CPU Time	Avg. Gap	CPU Time
10 / 2	26.50	0.01	0.34	0.00	225.87	0.00	5.69
10 / 3	293.00	0.01	0.33	0.00	227.88	0.00	5.66
10 / 4	204.10	0.01	0.29	0.01	228.20	0.01	5.52
20 / 2	268.20	0.03	0.69	0.02	262.48	0.03	12.04
20 / 3	3687.50 ^{5}	0.01	0.74	0.01	262.38	0.01	13.84
20 / 4	4140.80 ^{1}	–0.02	0.78	–0.02	264.63	–0.04	16.10
30 / 2	1449.80 ^{7}	0.02	1.08	0.01	295.57	0.01	18.36
30 / 3	4486.60 ^{2}	–0.03	1.24	–0.03	292.39	–0.04	23.96
30 / 4	7200 ^{10}	–0.06	1.43	–0.06	296.99	–0.07	29.85
40 / 2	3730.00 ^{5}	0.00	1.64	0.00	334.12	0.00	27.69
40 / 3	4261.90 ^{3}	–0.01	1.95	–0.02	329.53	–0.03	38.08
40 / 4	7200 ^{10}	–0.03	2.05	–0.03	324.12	–0.05	44.05

– All CPU times are in seconds.
– The superscripts denote the number of the instances (out of 10) that could not be solved within 2 h of computation time.

As summarized in Table 4, RCGA performs superbly even with a small number of repetitions. The computation times for "RCGA600 + Imp." are simply negligible, and the solution qualities are outstanding. For problem sizes larger than 20/4, the algorithm finds better solutions than LINGO in almost all cases. When the performance of "RCGA10000 + Imp." is examined, it is obvious that the solution quality improves as the number of repetitions is increased, and the solution times are still less than 1 minute. This means that the randomization in the algorithm pays off. An increase in the number of repetitions leads to better solutions as they help avoiding local optima. For 4-machine instances, the solutions found by RCGA10000, in very small computation times are on the average 5.4% better than those found by LINGO. Hence, the algorithm has an excellent performance for the OVJS problem with eligibility constraints.

When GA3 is used as an improvement algorithm after RCGA600, the solution times are affected, however the average gaps do not seem to improve significantly. Therefore, one can say that the increase in the computation time is not justified by the increase in solution quality. One reason for this is clearly the excellent performance of RCGA; it may very well be the case that most of the instances are solved to optimality by the algorithm, hence GA3 has very little left to improve.

For observing the effects of the number of machines on the performance of the algorithms better, additional runs are made with 10 and 20 jobs and 6 and 8 machines. The results of these runs are presented in Table 5. The instances for these problems are also consistent with the small set of real data obtained from the Port of Izmir. The port has 10 berths, serving 10 to 20 ships arriving daily. Berth allocation is currently made manually by the port authority, and no explicit mathematical method is used.

Table 5. Effect of the number of machines on algorithm performance

Problem	LINGO (Optimum)	RCGA600 + Imp.		RCGA600 + GA3 + Imp.		RCGA10000 + Imp.	
	CPU Time	Avg. Gap	CPU Time	Avg. Gap	CPU Time	Avg. Gap	CPU Time
10 / 2	26.5	0.01	0.33	0.00	224.85	0.00	5.69
10 / 4	204.1	0.01	0.29	0.01	226.85	0.01	5.52
10 / 6	44.9	0.00	0.29	0.00	227.33	0.00	5.09
10 / 8	375.9	0.00	0.32	0.00	236.48	0.00	5.50
20 / 2	268.2	0.03	0.70	0.02	261.54	0.03	12.04
20 / 4	4140.80 ^{1}	–0.02	0.78	–0.02	263.52	–0.04	16.10
20 / 6	4113.20 ^{2}	–0.60	0.83	–0.60	271.85	–0.60	14.27
20 / 8	7200 ^{10}	–15.71	0.87	–15.71	276.60	–16.21	14.46

– All CPU times are in seconds.
– The superscripts denote the number of the instances (out of 10) that could not be solved within 2 h of computation time.

The table reveals that, as the number of machines increases, the negative gap increases as well. This means that the number of machines greatly affects the quality of the lower bound found by the optimization software within the 2-hour limit. On the other hand, such an effect is not observed on the performance of RCGA, hence the gaps increase in favour of our algorithm as the problem size increases. Note that most of the instances with 20 jobs and

4, 6 and 8 machines cannot be solved optimally by LINGO. The average gap between the RCGA with 10,000 iterations and the solution found by the software is about – 16.21% for the largest instances, which is quite high, especially when computation times are considered. It seems that the outstanding performance of the RCGA also holds for instances with more number of machines.

Consistent with the results observed in Table 4, although the GA makes small improvements on some of the solutions, it does not prove to be effective as an improvement algorithm considering the extra computation times.

6. Conclusion

In this study, the Operational Variable Job Scheduling Problem with eligibility constraints is considered through. Among many other practical applications, the Berth Allocation Problem is an interesting application area of the problem. Our study therefore contributes to both areas. We show that the problem is NP-hard, and propose several construction and improvement algorithms.

A computational experiment is designed for observing the performance of the developed algorithms. The proposed genetic algorithm performs very well as a construction algorithm for the problem. The two improvement algorithms prove also very useful in improving the solution quality in practically no time, hence both are utilized after the initial feasible solutions are found.

The results also reveal that our Randomized Constraint-Graph-Based Algorithm (RCGA), which exploits special structural characteristics of the problem, outperforms GA as a construction algorithm in both solution time and quality. Together with the improvement algorithms, RCGA performs extremely well even for large instances of the problem, producing very good solutions in negligible computation times. GA does not seem effective when used as an improvement algorithm after RCGA. However, it may prove useful in larger instances of the problem.

As the computational experiment reveals, our RCGA algorithm can be used for finding near-optimal solutions to the berth allocation problem. The performance of the algorithm does not deteriorate as the number of berths or ships increase. Since it takes very little time, the algorithm can be run daily or even more frequently. It can also serve many other application areas of the problem, such as the assignment of gates to incoming aircrafts in an airport. For different application areas, some extra considerations may also prove useful, such as shifts or availability constraints for the machines.

References

Azizoglu, M.; Bekki, B. 2008. Operational fixed interval scheduling problem on uniform parallel machines, *International Journal of Production Economics* 112(2): 756–768.

Biricik, T. 2008. *Izmir'de gemiler baska liman ariyor* [Vessels in Izmir are considering other Ports]. Deniz Haber Ajansi. Available from Internet: <http://www.denizhaber.com.tr/LIMANLAR/13139/Izmirde-gemiler-baska-liman-ariyor.html>. [Accessed 10 Jun 2008].

Cordeau, J. F.; Laporte, G.; Legato, P.; Moccia, L. 2005. Models and tabu search heuristics for the berth-allocation problem, *Transportation Science* 39(4): 526–538.

Eliiyi, D. T.; Azizoglu, M. 2006. Spread time constraints in operational fixed job scheduling, *International Journal of Production Research* 44(20): 4343–4365.

Eliiyi, D. T.; Azizoglu, M. 2009. A fixed job scheduling problem with machine-dependent job weights, *International Journal of Production Research* 47(9): 2231–2256.

Fischetti, M.; Martello, S.; Toth, P. 1987. The fixed job schedule problem with spread-time constraints, *Operations Research* 35(6): 849–858.

Gabrel, V. 1995. Scheduling jobs within time windows on identical parallel machines, *European Journal of Operational Research* 83(2): 320–329.

Gertsbakh, I.; Stern, H. I. 1978. Minimal resources for fixed and variable job schedules, *Operations Research* 26(1): 68–85.

Goldberg, D. E. 1989. *Genetic algorithms in search, optimization, and machine learning*. New York: Addison-Wesley.

Imai, A.; Nishimura, E.; Papadimitriou, S. 2001. The dynamic berth allocation problem for a container port, *Transportation Research Part B* 35(4): 401–417.

Imai, A.; Sun, X.; Nishimura, E.; Papadimitriou, S.; Hattori, M. 2007. Berth allocation at indented berths for mega-containerships, *European Journal of Operational Research* 179(2): 579–593.

Kolen, A. W. J.; Lenstra, J. K.; Papadimitriou, C. H.; Spieksma, F. C. R. 2007. Interval scheduling. *A survey, Naval Research Logistics* 54(5): 530–543.

Kovalyov, M. Y.; Ng, C. T.; Cheng, T. C. E. 2007. Fixed interval scheduling: Models, applications, computational complexity and algorithms, *European Journal of Operational Research* 178(2): 331–342.

Kroon, L. G. 1990. *Job scheduling and capacity planning in aircraft maintenance*. Ph.D. Thesis, Rotterdam School of Management, Erasmus University, The Netherlands.

Kroon, L. G.; Salomon, M.; Van Wassenhove, L. N. 1995. Exact and approximation algorithms for the operational fixed interval scheduling problem, *European Journal of Operational Research* 82(1): 190–205.

Murty, K. G.; Liu, J.; Wan, Y. W.; Linn, R. 2005. A decision support system for operations in a container terminal, *Decision Support Systems* 39(3): 309–332.

Rojanasoonthon, S.; Bard, J. F. 2005. A GRASP for parallel machine scheduling with time windows, *INFORMS Journal on Computing* 17(1): 32–51.

Rojanasoonthon, S.; Bard, J. F.; 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.

Santos, Jr. E.; Zhong, X. 2001. Genetic algorithms and reinforcement learning for the tactical fixed interval scheduling problem, *International Journal on Artificial Intelligence Tools* 10(1–2): 23–38.

Wolfe, W. J.; Sorensen, S. E. 2000. Three scheduling algorithms applied to the earth observing systems domain, *Management Science* 46(1): 148–168.

KINTAMOS TRUKMĖS DARBŲ PLANAVIMAS ĮVERTINANT TINKAMUMO APRIBOJIMUS: ATSITIKTINIŲ APRIBOJIMŲ GRAFINIS METODAS

D. T. Eliiyi, A. G. Korkmaz, A. E. Çiçek

Santrauka

Nagrinėjama nepastovių darbų planavimo problema, susijusi su mašinų darbo planavimu laiko tarpais. Ši problema yra bendresnė fiksuotų darbų planavimo problemos versija, kai laiko tarpai ilgesni už darbų trukmę. Siekiama rasti optimalų atliekamų darbų poaibį. Geras pavyzdys yra optimali laivų priežiūra prieplaukoje, kai laivai prisišvartuoja laiko tarpais, taip maksimizuodami iš laivų gaunamą pelną. Taip pat įvertinami tinkamumo ribojimai. Problemai spręsti sukurtas sveikųjų skaičių programavimo modelis. Sukurtas apribojimų ir kreivių pavidalo algoritmas, gebantis generuoti apytikslius sprendinius. Jiems patikslinti naudojamas genetinis algoritmas ir kiti korekciniai algoritmai. Kompiuteriniai eksperimentai atskleidė, kad sukurtieji algoritmai generuoja labai tikslius sprendinius per labai trumpą laiko tarpą.

Reikšminiai žodžiai: kintamos trukmės darbų planavimas, tinkamumo ribojimai, optimali laivų priežiūra, genetinis algoritmas, apribojimų išpildymas, apribojimų kreivė.

Deniz Türsel ELİİYİ. Assistant Professor in the Department of Business Administration at Izmir University of Economics, Turkey. Received BS, MS and PhD degrees in Industrial Engineering from the Middle East Technical University, Ankara, Turkey. Research interests: production and operations management, scheduling, combinatorial optimization.

Aslıhan Gizem KORKMAZ. MBA student in Izmir University of Economics, Faculty of Economics and Administrative Sciences Department of Business Administration, Izmir, Turkey. Holds BA degree from Dokuz Eylul University, Izmir, Turkey. Research interests: Operations Research (Scheduling, Optimization), Finance (security markets, stock exchanges), Accounting (international accounting).

Abdullah Ercüment ÇİÇEK. M.Sc. student in Sabanci University, Faculty of Engineering and Natural Sciences, Computer Science and Engineering Program, Istanbul, Turkey. Research ınterests: data mining, knowledge discovery in distributed environments, optimization.