Metaheuristic Algorithms for Related Parallel Machines Scheduling Problem with Availability and Periodical Unavailability Constraints

Mihály Gencsi

Department of Computational Optimization, University of Szeged, Árpád tér 2, 6720 Szeged, Hungary, gencsi@inf.u-szeged.hu

Abstract: The Related Parallel Machine Scheduling Problem (R-PMSP) is a type of optimal job scheduling problem. The problem is to assign different types of jobs to different parallel machines. Every machine has a speed rate that can execute a job faster or slower than other machines. This paper focuses on an R-PMSP, with availability and periodical unavailability constraints. Some jobs can also have machine preferences. The problem with these constraints is NP-hard. This study describes three metaheuristic algorithms for solving the problem. Namely, the algorithms are Genetic Algorithm (GA), Simulated Annealing (SA), and Discrete Grey Wolf Optimizer (DGWO). This article focuses on examining the performance of the algorithms, determined by the required time, to find a suboptimal threshold. Simulated Annealing proved to be the best in terms of efficiency and time required to find the suboptimal threshold. In addition, the study describes a benchmark generator method for this problem, which guarantees to create a problem with given properties and with a given optimum.

Keywords: Parallel Machines Scheduling; Availability and Periodical Unavailability Constraint; Genetic Algorithm; Simulated Annealing; Grey Wolf Optimizer

1 Introduction

Task scheduling problems can be seen in any part of our lives, for example, scheduling production lines, scheduling task execution, and assigning clients to service queues. That is why scheduling is one of the most dominant problems to be solved today. There are many approaches to solving these problems in the literature, but the diversity of the problem means many open areas. Time is one of the essential resources. Valuable time can be saved by properly allocating tasks to the machines. The motivation for this research comes from two problems that need to be solved. The first is the scheduling of patients to testing laboratories for individual tests. This can include CT, MRI, blood tests, cancer prevention tests, etc. The duration of tests varies and patients have the option of choosing laboratories. For instance, the patient has an agreement with the laboratory. The testing laboratories have different characteristics: maximum availability (opening times) and periodical unavailability time (break times, maintenance time). Some laboratories perform their tasks faster than others (diversity of MRI machines, etc.). The second example is the scheduling of tasks to various computing devices. With so many different types available, such as PCs, supercomputers, and microcontrollers, the processing speed varies, and tasks must be assigned accordingly to optimize their running time. Utilizing a fast machine can reduce the running time of all tasks. Additionally, certain tasks may be delegated to predetermined machines for reasons such as data protection or contractual obligations. These machines possess varying characteristics, including reboot time, maintenance schedules, and cleaning intervals. The machine scheduling problem can be precisely matched with the patient scheduling problem. The characteristics of two examples are perfectly reconcilable. Patients are the tasks, and laboratories are the machines. In both cases, the goal is to minimize the makespan, i.e. minimizing the time difference between the start and the end of the task sequence. This problem is called the Related Parallel Machine Scheduling Problem (R-PMSP) with constraints.

In the next section, in chronological order, the state of the art on the problem in more detail is presented. Then, the types of scheduling problem and present current methods are described. In Section 3, a mathematical model of our problem is defined. Section 4 presents a benchmark generation method that yields the optimum. The used metaheuristic and their problem-specific modifications are described in Section 5. The computational results are discussed in Section 6. Finally, in the last section, Section 7, the results are concluded, summarized and the possibilities for further improvements are discussed.

2 Related Work

Researchers have recently defined categories and types of optimal scheduling problems. Two broad categories can be distinguished:

- **Single-stage job scheduling** Where each job consists of only one execution phase
- **Multi-stage job scheduling** Where each job consists of several execution phases that must be executed in parallel or a predefined order according to different rules

There are several types of single-stage job scheduling: single-machine scheduling, identical-machine scheduling, related-machine scheduling, and unrelated machine scheduling. Three types of multi-stage scheduling problems are known: open-shop scheduling, flow-shop scheduling, and the job-shop scheduling problem. These fundamental problems can be extended with different machine or job constraints. Machine constraints can be as follows: a machine can work for a particular time, each machine has its own time to process the information needed to complete the task, and machines can stop periodically. Task constraints can also be of many kinds: tasks must run within a given time, a task can be moved to another machine, tasks can only be available after a particular time, and tasks can be split up or not. There can be different objective functions to minimize [12]: makespan, maximum lateness, the total completion time, number of late jobs, or the total tardiness. The interested reader is referred to [5] [16] and the references therein.

2.1 The Single-Machine Scheduling Problem with Constraints

The simplest version of the problem is the single-machine scheduling problem, to which periodic or random breaks can be assigned. In [7], the single-machine scheduling problem with availability constraints is discussed. Two types of availability constraints are introduced. A machine must stop maintenance after a specific time, or a tool must be replaced after a particular processed job. In this case, the goal is to minimize the makespan. It has been shown that a single-machine problem with two maintenance constraints is NP-hard. Six types of heuristic algorithms are proposed to solve the problem, and it is shown that the best performing among them is the decreasing order with first fit algorithm (DFF). In [11], the single-machine problem is addressed under tasks due dates and machine unavailability constraints. The goal is to minimize the sum of maximum earliness and tardiness. A mathematical formulation was developed to exactly solve small problems. The Variable Neighborhood Search (VNS) was used to solve real-life problems. The VNS was extended with two knowledge module-based local searches, which improves the weaknesses of the random search of VNS. Experimental results have shown that the modified VNS can achieve optimal or near-optimal solutions in a reasonable time. In addition to these works, numerous other studies in the literature formulate the problem as a MILP model and solve the problem using various proven methods [2] [18].

2.2 The Two-Machine Scheduling Problem with Constraints

Many papers in the literature focus on the two-machine R-PMSP with various constraints. In [9], the problem was studied under the periodic availability constraint of a machine. Their goal is to minimize the makespan. They showed that the Longest Processing Time first algorithm (LPT) has a worst-case ratio of 3/2 if the problem is offline. In [8], the two-machine probability was graded under the constraint that one machine is unavailable at a given time. Their goal is to minimize the total weighted completion time. They developed a fully polynomial-time approximation scheme (FPTAS) for this problem. They also generalized this scheme to m parallel machines. In [1], the two-machine scheduling problem with unavailability of a single machine for a specific time was addressed. Their goal was to minimize the makespan. They separately chose five cases for this problem and developed a separate solution method for each case. It was shown that these methods are efficient even for a large number of items.

2.3 Multiple Machine Scheduling Problem with Constraints

There are also papers in the literature related to multi-machine task scheduling. For multi-machine task scheduling, several types of constraints can be found in the literature, for which different algorithms have been developed. The work [3] deals with a pseudo-analysis of the classical scheduling problem, for which unavailability times for machines and release dates and delivery deadlines for tasks are introduced. A branching strategy and a new lower and upper bound for the tasks are developed based on a representation taking all the permutations of tasks. It is shown that embedding a semi-preemptive lower bound based on max-flow computations in a branch-and-bound algorithm yields very promising performance. Using this method, they were able to solve 700 tasks with 20 machines within a reasonable CPU time. The authors of [10] focus on the multi-machine task scheduling problem without extra constraints. Their goal is to minimize the maximum delay. To solve this problem, the Largest the sum of Processing time and Delivery Time first Simulated Annealing algorithm called LPDT-SA is developed. The initial solution was generated using a heuristic LPDT method. In addition to these, they used an effective solution for the representation that efficiently implements swapping and insertion into the neighborhood and avoids worse solutions. The resulting algorithm is able to solve problems with 350 tasks in 90 seconds, and the average error between the lower bounds for all 2400 random instances is 0.339%. In the study [13], the parallel machine scheduling problem is studied with multiple scheduled unavailability periods. In their presented case, they allow the tasks to be restarted. Their goal is to minimize the makespan. They first formulate the problem as a MILP for small, medium, and moderately large instances. They proposed an enumeration algorithm using lexicographic sequencing. They compared this method with the MILP model. It is shown that the proposed algorithm obtains the optimal solution and is faster. In [4], the scheduling problem of static m identical parallel machines with shared server and sequence-dependent setup times is addressed. Their goal is to minimize the makespan. They describe a MILP model for the problem and, in addition, implement a Simulated Annealing and Genetic Algorithm for large-scale problems. After comparing the efficiency of the three methods, they concluded that the GA algorithm provides better quality solutions in a reasonable computation time.

3 Model Description

In this section, we describe our problem. The problem studied is single-stage offline job scheduling. In other words, we want to assign a predefined set of jobs to machines (offline). All tasks are unrelated; this means that they do not form jobs, and we cannot break the jobs into smaller tasks or pause them (single-stage). Thus, we will use task and job as synonyms. Data are fixed and deterministic. We assign machine-specific constraints to the problem: maximum availability constraint and periodical unavailability constraint. Machines can have different speeds, which means that a machine can execute all tasks faster (or slower). Thus, the lengths of tasks are given in number of machine instructions (NMI), where one machine instruction is done in one-time unit on a unit-speed machine. We can consider one unit of time as one minute for an easier discussion. Suppose that we have an examination with ten machine instructions, a laboratory with 0.9 speed, and a faster laboratory with 0.5 speed. For example, the second laboratory has faster CT equipment and better-qualified staff than the first one. The first laboratory can complete this task in nine minutes, while the second laboratory in five minutes. The machine availability constraint means that a machine is unavailable after a particular time. For example, if we have a laboratory with a four-hour daily work schedule, we cannot assign more than four hours of examination. The periodic unavailability constraint consists of two components: maintenance time and uptime. After the end of the uptime, the machines must be periodically suspended for maintenance time. Suppose that we have a laboratory whose internal rules require that workers need to take a ten-minute break every 120 minutes. In this case, the uptime is 120 minutes, and the maintenance time is 10 minutes. The machine parameters are different for each machine. If the same constraint were imposed on all machines, we would obtain a particular case of the problem. Similarly, as we can introduce features for a machine, we can also introduce features for tasks. The machine preference for a task is set to a predefined machine and specifies that the task must be executed on that machine. It is important to point out that splitting a task into smaller sub-tasks in our problem is impossible. The tasks have no setup time, no appearance date, and no execution date. All tasks are known and available at the starting time. Our goal is to minimize the makespan.

Table 1
Notations

$J = \{j_i\}$	Jobs, $i = 1, \ldots, n$
$P = \{p_i\}$	Number of machine-instructions of each job
$JP = \{jp_i\}$	Machine preference of each job
$M = \{m_j\}$	Machines, $j = 1, \ldots, m$
$S = \{s_j\}$	Speed of each machine
$UT = \{ut_j\}$	Periodic uptime of each machine
$MT = \{mt_j\}$	Maintenance time of each machine
$A = \{a_j\}$	Available time of each machine

To formalize the problem, we need to introduce notations, summarized in Table 1. J represents the index array of the tasks. Each task is given by its NMI, p_i, from the array P. In addition, each task is assigned a jp_i value, which indicates the machine on which a task needs to be executed. If we do not have a machine preference, this value is null. M represents the index array of the machines. Each machine is assigned its speed s_j from the interval (0, 1.0], where a speed 0.5 means a two-times faster machine than speed 1 in relation to processing time. For each machine, ut_j and mt_j specifies the intervals at which the machine is active or paused. In addition, the maximum availability time of each machine a_j is also given. From the last parameters, one can compute the maximum number of segments for each machine, namely, $ns_j = \left\lfloor \frac{a\_j+mt\_j}{ut_j+ mt_j} \right\rfloor$.

Based on the above, we can formulate the following decision variables.

$$x_{kj}^{i} = \begin{cases} 1, & \text{if the } i^{th} \text{ job is run on the } k^{th} \text{ segment of the } j^{th} \text{ machine;} \\ 0, & \text{otherwise.} \end{cases}$$

With these parameters and variables, one can formulate the mathematical model of the problem. Then, such a model can be used to solve the problem by any method for ILP. Given the complexity of our problem, aiming for a concrete mathematical model is not worthy because ILP solvers cannot solve reasonable problems efficiently. Therefore, the goal was to describe the problem at hand, so that the reader could use and recreate the problem as presented.

4 Benchmark Generating Method

There is no benchmark for this problem in the literature to provide an optimal solution. In the papers mentioned in Section 2, randomly generated test instances were used. The benchmarks in the literature are not designed for such a problem: they do not contain breaks, and the constraints defined are not included. Adapting these existing benchmarks for the task is very time-consuming, and one would be unable to provide the optimum. Therefore, a benchmark generating method is presented, to ensure the optimal solution or to define a value close to the optimal one, generate gaps while keeping the optimal solution, and handle all constraints. Several parameters of our generating method are controlled, which allows for the specification of the generated problem. For example, one can set limits on the machine instruction, the maximum availability constraint, the periodical uptime, the speed of the machines, and the probability of machine preference. In addition, the number of machines and the number of gaps for each machine can be specified.

4.1 Description of the Generating Method

In the first step, the aim is to create a problem for which we know the optimal solution. It can be achieved by setting the total time of each machine to the given optimum. That is, take random uniform integer values for each machine, from a given interval, which will give the total NMI it can process (these have to be larger than the optimum and do not take maintenance times into account at this point). Set the speed of each machine, such that the execution time of their generated NMIs takes exactly the optimal time. After that, assign maintenance times and uptimes to each machine between the limits, such that the last segment is always shorter than the others, but strictly larger than 0. This will ensure that the optimum will be the given value. When generating tasks, make sure that the limit of the number of instructions is respected and that there are no gaps (idle times) on the machines. Each part of a segment is linked to a job. This means that tasks on the machine must follow each other, and only maintenance time can be added between them. Assign the maximum availability constraints to the machines randomly between the maximum availability limits. Here, we make sure that the generated value is not less than the optimum.

By careful generation, we know that all limits have been met and the optimal solution is known, since there are no idle-time slices except for pause times.

4.2 The Rules for Generating the Gaps for m Related Machines

A gap in a solution is the idle time when the machine is active but not working. In the previous step, we generated test cases with no gaps in the optimal solution. However, in most practical cases, this does not occur. Therefore, our goal is to artificially introduce gaps into a test case such that its optimal solution, Opt, does not change.

One can observe that for m related-machines, one can reduce the NMI of $m-1$ tasks by one, without affecting the optimal solution.

To see the above statement, suppose that we have $m > 1$ machines and $n \geq m$ tasks, where the NMI of all tasks are integer, that is, the smallest NMI is one. Let us randomly reduce the NMI of $m-1$ tasks by one. Consider the case where we reduce the NMI of the last tasks on the last segment of the first $m-1$ machines. Still, the optimum cannot change because on the last machine there were no reductions, and the last task finishes at the same time as before. As there are no tasks with smaller units, there is no better distribution of the tasks.

We also observe that when we reduce more than $m-1$ machine instructions in total, the optimum might decrease. Still, we can obtain a lower bound on the optimum by calculating the maximum possible decrease of the optimum. Let $r \geq m$ be smaller than the smallest NMI of the last segments times m. If we reduce the total NMI of all tasks by r, then the optimum can improve at most by $\frac{r}{\sum_{i=1,..,m} s_i}$ unit time.

To understand this formula, suppose that we have $m > 1$ machines and $n \geq m$ tasks. We randomly reduce the NMI of some tasks, in total, by r. To analyze the worst case, we reduce the last tasks of the jth machine by $\frac{r}{\sum_{i=1,..,m} s_i} \cdot s_j$. Thus, the length of each machine is reduced by the same amount, $\frac{r}{\sum_{i=1,..,m} s_i}$. It modifies the optimal solution to $Opt - \frac{r}{\sum_{i=1,..,m} s_i}$.

In the deletion mechanism, we can choose whether maintaining the optimum is necessary or whether a good lower and upper bound on the optimum is sufficient. We aim to reduce the NMI of any task in the non-last segment such that it cannot be changed by any task in the last segment. Note that by generation, the last segment is always shorter than the other segments on all machines. As a consequence, the gaps cannot be moved to the last segment. This also implies that there is a maximum reduction which can be made.

Namely, we can reduce the time of tasks from a non-last segment until the combination of all possible tasks does not fill the segment better. Generating all possible solutions (combinations) is too expensive because the segments can be placed in any order within a machine having the same optimal value. Therefore, the algorithm controls this with a parameter to display all solutions or just the first possible one. The second step results in a test case that adheres to the task specification and contains gaps. If only $m - 1$ NMI is reduced in total, optimality is guaranteed, while if reduction is higher, we can bound the optimum.

The result of the generating method is shown in Figure 1, visualized in Figure 2. The figure shows a test case with three machines. It is observed that the optimal solution is 15 units, the number of tasks is 13, and there are gaps (marked with a red striped box) in the solution. The generated case obeys all constraints of the control parameters.

5 Algorithms

The R-PMSP problem with the introduced constraints is NP-hard, so we cannot give an exact algorithm that can solve our problem for real-life test cases in a reasonable time. There are exact algorithms for problems with two machines or without constraints, but if we increase the number of machines, tasks, and/or add constraints, we can only solve the problem using heuristics. Simulated Annealing (SA) and Genetic Algorithm (GA) are commonly used methods to solve these problems [4] [10]. We have used these algorithms and introduced our modifications to obtain results close to the optimal solution. In addition, we applied a discrete version of the Grey Wolf Optimizer (DGWO) to solve the problem.

5.1 Common Representation and Operators of the Algorithms

The three algorithms use the same representation for the solutions. In addition, we use the same fitness function, crossover, mutation, and initial solution generation procedure.

5.1.1 Representation of Individuals or Solutions

We can achieve faster convergence with a well-chosen representation of an individual or a solution. A good representation allows us to better explore our search space. A one-dimensional array represents individuals or solutions. The i^{th} element of the array correspond to the task j_i, and encode which segment of which machine it is assigned (see the Solution line in Figure 1). To do this, we first enumerate each segment of each machine (within the available time of the machine) sequentially from the first machine to the last. We store which segments belong to which machines in an index array. In addition, to speed up the calculations, we store the gaps (idle times) of each solution in an array, that is, how much empty space is available on each machine segment. In this way, during crossover and mutation, we do not need to calculate which segment has enough space for a task. The required storage space increases by storing the gaps in the segments but reduces the computational time considerably.

Two solutions are equivalent, where only the permutation of non-last segments of any machine is different. These individuals differ because the segments are in a different order, but the tasks assigned to a given segment are mutually equivalent. The above representation considers these solutions different. Thus, we order the segments, except the last, within the machine in descending order based on their idle time. With this sorting, the loaded segments are moved to the front (see the segments gaps in Figure 1). In the case where two segments have the same idle time, the choice is based on the number of tasks in the segments.

For example, we can see the test case from Figure 1 on the machines in Figure 2. We have three machines, 13 tasks (jobs), and the optimum is 15. In the example, we see one gap (marked with a red striped box) on the first machine and one on the first segment of the second machine. The characteristics of the machines are shown in Table 2. We see the solution representation of the same in Figure 1. The first array contains the NMI of the tasks, the second describes the segment index of the machines, the next encodes the solution, and the last array presents the gaps on the segments. We can see in the solution array the segment indexes. For example, the first task, named Job 1 in Figure 2, will be performed on the segment with index one, which is the segment of the first machine. The third task, called Job 3 in Figure 2, will be executed on the segment with index five, which is the first segment of the third machine.

Table 2
Machine characteristics

Constraints	m_1	m_2	m_3
Maximum availability	22	22	23
Speed	0.75	1.0	0.5
Uptime	30	7	16
Maintenance time	0	2	4

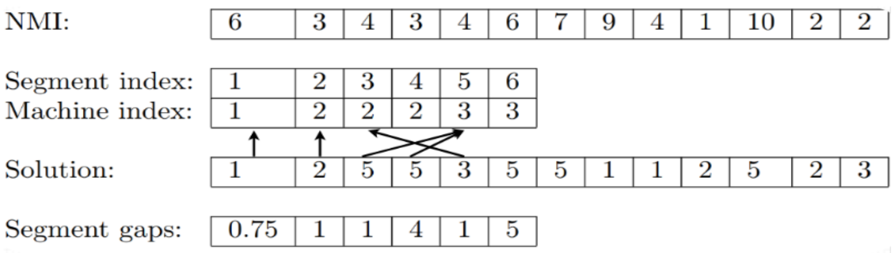

NMI: 6 3 4 3 4 6 7 9 4 1 10 2 2
Segment index: 1 2 3 4 5 6
Machine index: 1 2 2 2 3 3
Solution: 1 2 5 5 3 5 5 1 1 2 5 2 3
Segment gaps: 0.75 1 1 4 1 5

Figure 1
Solution representation

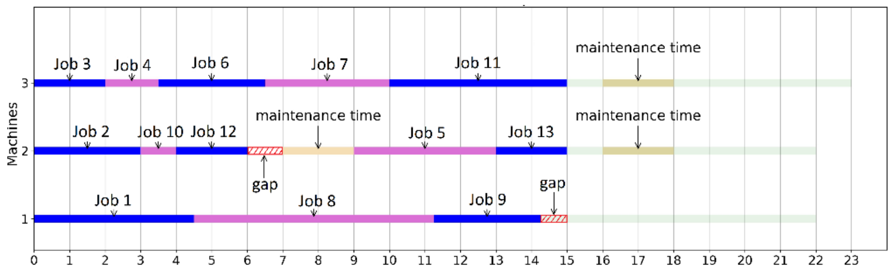


Figure 2
Generated test case with 3 machines, 13 jobs with the optimum 15

5.1.2 Generation of Initial Solutions or Individuals

The initial population or solution is generated at random. We first assign tasks with machine preference to their corresponding machines, but the segment within a machine is chosen randomly. Second, the remaining tasks are assigned to randomly selected machine segments from those where they fit in. Finally, we regenerate the individual or solution if the task cannot be assigned to a machine even after multiple attempts.

5.1.3 Fitness Function

The fitness function measures how close a given solution is to the optimal solution of the problem. It is a numeric value assigned to an individual or a solution. We could use only the makespan as a fitness function, but instead we use a more complex fitness function: we add the number of occupied segments and the total idle time on the occupied segments to the makespan. We can formalize the fitness function as $Fitness(x) = occupied\_seg(x) + total\_idle(x) + makespan(x)$.

5.1.4 Crossover

A crossover is a genetic operator in which genetic information from two selected individuals (parents) is combined to produce new individuals (children). We have implemented several types of crossover operators that can be used with the one-dimensional array representation. Different crossovers per algorithm were shown to be better. We have implemented k-point, uniform, and three-parents crossover [14]. We swapped randomly selected segments of the individuals. For each substitution, we checked the feasibility of the solution, and dropped the children which were not feasible (the new segment does not fit into its new place). We observed that the crossovers used in this way were almost useless, and the methods converge to a wrong solution quite quickly.

Consequently, we created modified crossovers that perform segment replacement by randomly selecting the segment from those that fit on the replacement machine. In cases where it was not possible to swap due to machine preference or none of the segments fit, we left the element unchanged. In this way, the crossover operators generate more correct solutions and better traverse the search space.

5.1.5 Mutation

A mutation is a genetic operator responsible for diversifying a population. It is a slight modification of an individual or a solution. The discrete version of the Grey Wolf Optimizer requires several mutations to maintain population, namely, pack of wolves, diversification. We have implemented bit flip, swap, inverse, and reverse mutation [15]. In these implementations, segments are modified to other segments. We consider a mutation to have been used if the resulting individual satisfies the constraints of the problems. Otherwise, a new individual is generated from scratch. In addition, we implemented a modified version of each mutation named above. In these implementations, we swapped the machines, but we chose the segments randomly.

5.1.6 Selection

Selection is a rule that determines the individuals in the following population or the individuals participating in the crossover and mutation. For crossover and mutation, we randomly select individuals from the current population. We use the tournament selection rule to create a new population for our problem [6].

5.2 The Algorithms and their Adaptations

We modify the SA, the GA, and the DGWO with changes that achieve faster convergence and improve the quality of the solutions.

5.1.7 Simulated Annealing

The Simulated Annealing algorithm originates from metallurgy, which aims to change the physical properties of a material by heating and controlled cooling. The method employs an iterative motion according to the varying temperature parameters based on the annealing operation of metals.

Algorithm 1 describes the Simulated Annealing algorithm we use, where p_m denotes the mutation, p_{co} denotes the cooling scheme parameters, and $data$ encodes the problem to be solved.

The algorithm requires an initial temperature, a reduction scheme, a number of iterations, and an initial solution, which will be called the current solution. There are several temperature reductions schemes: linear, logarithmic, exponential, and quadratic. The linear scheme is the best for the problem. The temperature is reduced from the initial temperature according to the reduction mechanism. In each iteration, we generate a neighboring solution using the current solution. The current solution is compared with the neighboring solution. It is swapped if its fitness value is better than the current fitness value. Otherwise, it is swapped with a certain probability. The role of the acceptance probability is to be able to move out of the local minimum points and move towards better solutions. As the temperature decreases, the value of this acceptance probability decreases. Several stopping conditions can be introduced: after a given number of steps r, if the best solution has not improved, or the maximum number of iterations has been reached. Due to the heuristic nature of the algorithm, the optimal solution can be reached after multiple runs.

In our version, we use the mutation operators to generate a neighboring solution. In contrast to the basic algorithm, we do not generate an adjacent solution per iteration. Instead, we generate n and compare the best one with the current solution.

Algorithm 1 SimulatedAnnealing($data$, T, $iter_{max}$, p_m, p_{co}, $neighboor_{size}$)

```
1:  $x_{cur} \leftarrow$ GenenerateP opulation($data$, 1);
2:  $i \leftarrow 0$; $x_{best} \leftarrow x_{cur}$; $run \leftarrow true$;
3:  **while** ($i < iter_{max}$ && $run$) **do**
4:    $temp_{cur} \leftarrow$ CalcTemp($i$, $T$, $p_{co}$);
5:    $X \leftarrow$ GenerateNeighboors($x_{cur}$ , $p_m$, $neighboor_{size}$);
6:    $x_{tmp} \leftarrow$ SelectBestNeighboor($X$);
7:    **if** Fitness($x_{tmp}$) $\leq$ Fitness($x_{cur}$) **then**
8:      $x_{cur} \leftarrow x_{tmp}$;
9:      **if** Fitness($x_{tmp}$) $\leq$ Fitness($x_{best}$) **then**
10:       $x_{best} \leftarrow x_{tmp}$;
11:     **end if**
12:   **else**
13:     **if** Random(0, 1) < exp((Fitness($x_{cur}$)–Fitness($x_{tmp}$))/ $temp_{cur}$) **then**
14:       $x_{cur} \leftarrow x_{tmp}$;
15:     **end if**
```

```
16:     end if
17:     $run$ ← ExamineStopCriterion();
18: end while
19: return $x_{best}$, Fitness($x_{best}$)
```

Algorithm 1
Simulated Annealing

5.1.8 Genetic Algorithm

The Genetic Algorithm is a population-based metaheuristic algorithm inspired by natural selection. Instead of one solution, we work with a set of solutions, namely, population. The elements of the population are called individuals, and the number of iterations is called generations.

Algorithm 2 presents the pseudocode of the Genetic Algorithm, where p_c is the crossover, p_m is the mutation, and p_s is the selection parameter.

```
Algorithm 2 GeneticAlgorithm($data$, $pop_{size}$, $gen_{size}$, $p_c$, $p_m$, $p_s$)
1:  $pop_{cur}$ ← GeneratePopulation($data$, $pop_{size}$);
2:  $x_{best}$ ← SelectBestIndividual($pop_{cur}$);
3:  for $i = 1, . . . , gen_{size}$ do
4:    $crossover$ ← {}; $mutation$ ← {};
5:    for $j = 1 , . . . , \frac{pop_{size}}{2}$ do
6:      $r_1, r_2, r_3$ ← RandInt(0, $pop_{size}$);
7:      $crossover$ ← $crossover$ ∪ Crossover($p_c$, $pop_{cur}$, $r_1$, $r_2$);
8:      $mutation$ ← $mutation$ ∪ Mutation($p_m$, $pop_{cur}$, $r_3$);
9:    end for
10:   $pop_{cur}$ ← $pop_{cur}$ ∪ $mutation$ ∪ $crossover$;
11:   $x_{cur}$ ← SelectBestIndividual($pop_{cur}$);
12:   if Fitness($x_{best}$) > Fitness($x_{cur}$) then
13:     $x_{best}$ ← $x_{cur}$;
14:   end if
15:   $pop_{cur}$ ← Selection($p_s$, $pop_{cur}$);
16: end for
17: return $x_{best}$, Fitness($x_{best}$)
```

Algorithm 2
Genetic Algorithm

The algorithm requires a population size, a generation size, and an initial population. Individuals from the current population are selected according to some selection rule. We use our modified crossover and mutation operators. Then, the population of the new generation is selected from the current population, including mutation and crossover results. This selection can be based on age or fitness. The algorithm can be stopped when a generation number is reached or if no improvement is observed after r steps.

In our implementation, we work with a fixed population. The initial population is generated randomly according to the method described in Subsection 5.1.2. In each iteration or generation, we generate new individuals equal to half the population size by crossover and mutation operators. The individuals involved here are selected randomly. We use tournament selection for the selection of new populations.

5.1.9 Discrete Version of the Grey Wolf Optimizer

The Gray Wolf Optimizer is a population-based metaheuristic algorithm inspired by nature. The method attempts to mimic the hierarchy and hunting mechanism of grey wolves. Grey wolves are organized into a hierarchy: alpha, beta, delta, and omega wolves. Each level of the hierarchy has its role, and no one can be absent from the hierarchy. The alpha wolf is at the top of the hierarchy. His role is to make decisions about the pack and to lead the pack. The beta wolf is the second level of the hierarchy. His role is to assist the alpha wolf in decision-making, to relay the decisions of the alpha wolf to the pack, but he also has a leadership role. The lowest level of the hierarchy is the omega wolf, which represents the weakest wolf in the pack. He does not have an important role in the pack, but his absence can create internal conflict. All wolves that do not belong to the previous three are delta wolves. Their role is to scout, protect, and hunt. The hunting mechanism of grey wolves also plays an important role in the algorithm. The algorithm mimics hunting, prey search, prey enclosure, and prey attack in its methods.

Algorithm 3 presents the pseudocode for the used Grey Wolf Optimizer, where p_c is the crossover, p_m is the mutation, pb_{max} and pb_{min} are the local and global stage measures, respectively.

Algorithm 3 DiscreteGWO($data$, $pack_{size}$, gen_{size}, p_c, p_m, pb_{max}, pb_{min})

```
1:  $pack \leftarrow$ GeneratePopulation($data$, $pack_{size}$);
2:  $x_\alpha, x_\beta, x_\delta \leftarrow$ SelectBestWolfs($pack$);
3:  $x_{best} \leftarrow x_\alpha$;
4:  for $i = 1, \ldots, gen_{size}$ do
5:    $x_\alpha, x_\beta, x_\delta \leftarrow$ LocalSearchUpdate($x_\alpha$, $x_\beta$, $x_\delta$, $p_m$);
6:    for $j = 1, \ldots, pack_{size}$ do
7:      if Random(0, 1) $\leq pb_{max} - (pb_{max} - pb_{min}) \cdot i/gen_{size}$ then
8:        $pack_i \leftarrow$ SeekingMode($pack_i$, $p_c$);
9:      else
10:       $pack_i \leftarrow$ TracingMode($pack_i$, $p_c$);
11:     end if
12:   end for
13:   $x_\alpha, x_\beta, x_\delta \leftarrow$ SelectBestWolfs($pack$);
14:   if Fitness($x_{best}$) > Fitness($x_\alpha$) then
15:     $x_{best} \leftarrow x_\alpha$;
16:   end if
17: end for
```

```
18:    return $x_{best}$, Fitness($x_{best}$)
```

Algorithm 3
Discrete version of the Grey Wolf Optimizer

The algorithm requires a population or pack size and a number of iterations. In the algorithm, the tails α, β and δ indicate the first three solutions with the lowest fitness values. The ω denotes all other solutions. The first step of the algorithm is to generate an initial pack, from which we select the first three wolves. Then, in each iteration, a local search method is applied to the wolves α, β and δ and their values are updated. Then, for each wolf in the pack, we use the seeking mode or the tracing mode in addition to the selection probability. The stopping conditions can be the maximum number of iterations or no change in our best solution after r steps.

In detail, the local search and update algorithm (line 5 in Algorithm 3) calls the local search for the wolves α, β and δ and sorts these three wolves by fitness value. In the local search, we improve the fitness value using three types of mutation. We use two search modes in the algorithm, corresponding to the local and global search stages. We apply the search selection mentioned in line 7 in Algorithm 3, which explores the global space in the first stages of the search, clustering around local optima. In later stages, it converges to the global optimum. In the seeking mode, we aim to preserve population diversity and avoid premature convergence. This can be achieved by using the crossover between a randomly selected individual and the current individual. The tracking mode is used for local searches. The method selects the α, β and δ with fitness-based selection, and execute a crossover on the selected with the current solution [17].

6 Experimental Results

In this section, we present our computational results. First, we show the test cases, which were generated using the method described in Section 4. Then, we discuss how we determined the parameters of the algorithms. Finally, we also discuss the methods for comparing the algorithms.

6.1 Generating Test Cases

Test cases were generated using the method described in Section 4. We wanted to see the results compared to a known optimal solution for each test case, so we generated only as many gaps that did not change the optimal solution. It is impossible to directly control the number of segments and the number of tasks in the generating method. Therefore, some parameters of the generation method are controlled and others are left with a higher degree of freedom. We strictly set the parameter controlling the machine preference to 0.1 (so 10% of the tasks have preference), the number of machines to $m = 5, 10, 20$, and the machine speed to the interval [0.8, 1.0]. We set maximum availability at [30, 120], periodical uptime at [10, 40] and the maintenance time at [0, 10]. The number of segments per machines are either 0, or chosen from [1,10] by the generator, denoted by $\overline{ns} = 0, 10$. We set the number of tasks to be chosen from three intervals: $n \in$ [25-50], [51-100] and [101-200]. For the easier, we will denote these by $\bar{n} = 50$, 100, 200. For each task, NMI is set to [1,18]. With these parameters, we generated 25 test cases. From each set of test cases, we randomly selected 5 test cases.

In total, we have the following classes for both $\overline{ns} = 0$, 10 (See in Table 3).

Table 3

Benchmark generating test case classes

m	5			10			20		
$\bar{n}$	50	100	200	50	100	200	50	100	200

6.2 Parameter Tuning of the Algorithms

We fine-tune all parameters of the three metaheuristic algorithms by controlled grid search. The test cases for fine-tuning are randomly generated. We considered each combination mentioned in Section 6.1. Each test case was run ten times. We try to define the parameters and operators based on improving the mean, minimum, and variance per iteration. We focus on keeping the variance large initially and decreasing slightly per iteration so that our search space is well-traversed. We also ensured that the average converges to the minimum by the end of the iterations. This method allowed us to filter out most of the parameters. To determine the additional parameters, we considered how many times out of ten runs reached 98% of the optimal value. We consider the winning parameter to be the one that reaches the limit more times and is closer to the optimum on average.

6.3 Experiences and Results

All runs were performed on a 2.00 GHz Intel Xeon Processor (E5-2660 v4 35M cache) with 64 GB RAM. We examine the average performance of the algorithms from 12 runs on the test cases presented in the Subsection above. Additionally, we compare the time required for the algorithms to reach a suboptimal threshold.

6.1.1 The Performance of the Algorithms

We study the performance of the algorithms on the generated test cases. First, we divide the test cases into two groups, one that does not contain periodical availability constraints, and the other that contains periodical availability constraints. Then, we run the algorithms on each test case 12 times and examine the optimum. If the algorithm does not stop after 3600 seconds, it is stopped, and the best solution is assigned to the run. We normalize the result of each run by dividing the absolute error of each result of the run with the optimal value. This value represents the relative deviation error from the optimum. Next, we calculate the maximum, minimum, and mean of this relative for each test class. These values are grouped in increasing order by algorithm, number of machines, and tasks. For example, the group $m = 5$, $n = [51, 100]$, SA means $m = 5$, $n = [51, 100]$ using the SA. Finally, we calculate the mean, the mean minimum, and maximum error percentage of the group. In the figures, the colored column gives the relative average error, while the thin black line shows the range by the minimum and maximum of the relative error, all in percentages.

Figure 3 shows the results for the test cases without periodical availability constraints. The three algorithms obtain the optimum with a small error for the test cases with $m = 5$. As the number of machines increases, the percentage error increases linearly. The algorithms obtained the minimum out of 12 runs for all test cases. For the 20 machine runs, the maximum average error increased, but the average error did not exceed 3%. As the number of tasks increased, the performance of the algorithms increased. There may be several possible solutions for many tasks close to the optimum. In conclusion, as the number of tasks increases, the average error decreases. As the number of machines increases, the error increases.

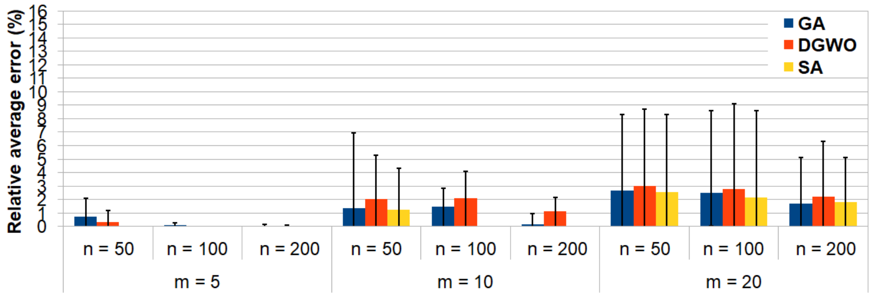


Figure 3

Test cases without periodical unavailability constraints. Relative average error from the optimum, with the minimum and maximum errors grouped by class

We can see in Figure 4 the relative average error grouped by machine, task, and algorithm for test cases with periodical availability constraints. The relative average error increases linearly as a function of the number of machines. The algorithms obtained the minimum out of 12 runs for all test cases. The relative average error of the algorithms does not exceed 5%, even for problems with 20 machines. In a few cases, it is observed that the maximum error decreases when increasing the number of tasks. The best performing algorithm here is also the SA. In this case, the maximum average does not exceed 15.5%.

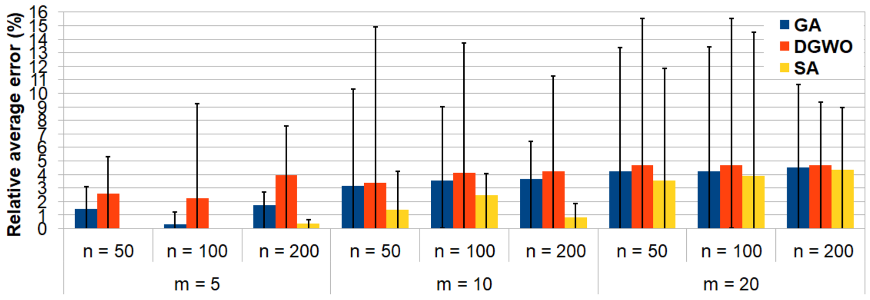
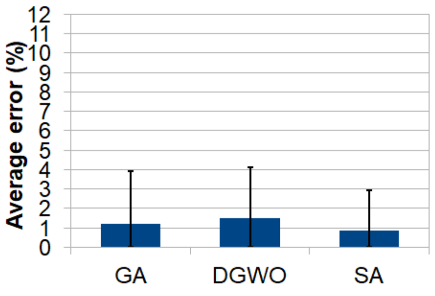


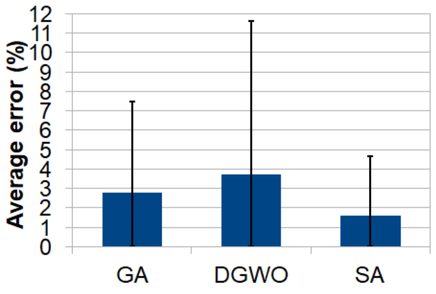
Figure 4
Test cases with periodical unavailability constraints. Relative average error from the optimum, with the minimum and error grouped by class

We can see in Figure 5, the overall average error calculated for each algorithm taking into account all test cases without periodical availability constraints. In general, SA has the lowest average failure rate of all the test cases. The relative average error of the algorithms does not exceed 1.5%.

Figure 6 shows the overall average error calculated for each algorithm for the all test cases without periodical availability constraints. The average error percentage for all algorithms for all test cases is below 5%, and the average maximum error percentage is below 12%. Furthermore, we can see that the average minimum error percentage is equal to 0%, which means that all algorithms find the optimal solution at least once in all test cases.



Test cases without periodical unavailability constraints. Average error from the optimum, with the minimum and maximum errors grouped by algorithm.



Test cases with periodical unavailability constraints. Average error from the optimum, with the minimum and maximum errors grouped by algorithm.

6.1.2 The Time Required to Find a Suboptimal Threshold

We study the average time required to reach some suboptimal threshold. We introduce four types of the suboptimal threshold: 30%, 20%, 10% and 5% the optimal solution plus the optimal solution. We divide the test cases into two groups, one that does not contain periodical availability constraints and the other that contains periodical availability constraints. In the first case, we set the maximum runtime to 600 seconds because it does not require more time to find the suboptimal threshold and in the second case, it to 3600 seconds. We run the algorithms on each test case 12 times and examine the required time. If the algorithm did not reach the suboptimal threshold, we set the time to maximum runtime. We calculated the average of 12 runs. Then, we calculate the average of the average using some grouping rule. For example, we can group by the number of machines, the range of tasks, and the algorithms. In the figures, the points represent the average time required to reach the suboptimal threshold.

Figure 7 shows the average time required to reach some suboptimal threshold for test cases with periodical availability constraints. The test cases are grouped by the number of machines, tasks, and algorithms. If we increase the thresholds, the time required increases linearly. We observe that DGWO reaches the first two thresholds faster than GA. After that, DGWO slows down. SA is the fastest in all three categories.

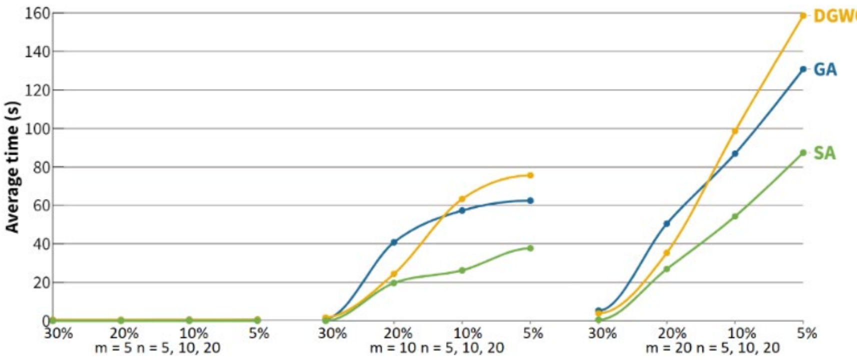


Figure 7

Test cases without periodical unavailability constraints. Relative average error from the optimum, with the minimum and maximum errors grouped by class.

Figure 8 shows the average time to reach some suboptimal threshold for test cases with periodic availability constraints. The test cases are grouped as above. If we increase the number of machines, the average running times increase exponentially.

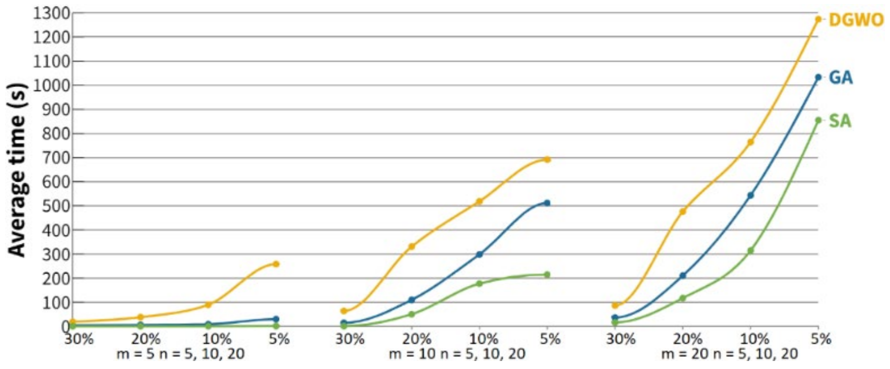


Figure 8
Test cases without periodical unavailability constraints. Relative average error from the optimum, with the minimum and maximum errors grouped by class.

Summary and Conclusions

This work focused on the related parallel machine problem (R-PMSP) with availability and periodical unavailability constraints. A robust benchmark generating method was proposed for the problem, which can generate a large variety of test cases, by controlling the parameters of the methods and knowing the optimum with certainty. In addition, a deletion mechanism that generates gaps in the solution, was offered. Gaps can be created, keeping the optimum or giving a lower and upper bound on the optimum. An implementation of three metaheuristic algorithms to solve the problem are given: Genetic Algorithm, Simulated Annealing and the Discrete version of Grey Wolf Optimizer. We introduced an efficient representation of the solution and methods to improve the algorithms in terms of solving the problem. Simulated annealing was also complemented with multiple neighborhood methods. Common operators and functions were used in the algorithms.

The performance of the algorithms was examined and compared to the average time required to find a suboptimal threshold. In the case without periodical unavailability constraints, the average relative error and the average of maximum relative errors of the algorithms are below 1.5% and 4.5%, respectively. Furthermore, with unavailability constraints, the average errors of the algorithms were below 4% and 11.5%, respectively. The running times of the algorithms increase exponentially by decreasing the threshold for many machines. However, for relatively few machines, this indicator is linear. The Simulated Annealing algorithm performs the best on average. The DGWO algorithm is not efficient for this problem, as it tends to get stuck in local optima and is time-consuming. GA takes longer on average than SA. However, it generates several near-optimal solutions, which can be beneficial in various cases where multiple solutions are needed, or we want to examine critical points. Therefore, we recommend utilizing this approach when sufficient computing time is available and several near-optimal or optimal solutions are required.

Future goals are to improve the SA and GA methods, with local search and procedures to reduce the searching space. The goal would be to generalize the SA method for semi-online R-PMSP, with constraints using local methods, but also to develop a Matheuristic, that handles the semi-online case.

References

[1] A. A. Masmoudi, M. Benbrahim. New heuristics to minimize makespan for two identical parallel machines with one constraint of unavailability on each machine. In 2015 International Conference on Industrial Engineering and Systems Management (IESM), pp. 476-480, 2015

[2] A. B. Keha, K. Khowala, J. W. Fowler. Mixed integer programming formulations for single machine scheduling problems. Computers & Industrial Engineering, 56(1):357-367, 2009

[3] A. Gharbi, M. Haouari. Optimal parallel machines scheduling with availability constraints. Discrete Applied Mathematics, 148(1):63-87, 2005

[4] A. Hamzadayi, G. Yildiz. Modeling and solving static m identical parallel machines scheduling problem with a common server and sequence dependent setup times. Computers & Industrial Engineering, 106:287-298, 2017

[5] A. Jain, S. Meeran. Deterministic job-shop scheduling: Past, present and future. European Journal of Operational Research, 113(2):390-434, 1999

[6] A. Shukla, H. M. Pandey, D. Mehrotra. Comparative review of selection techniques in genetic algorithm. In 2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE), pp. 515-519, 2015

[7] C. Low, M. Ji, C.-J. Hsu, C.-T. Su. Minimizing the makespan in a single machine scheduling problems with flexible and periodic maintenance. Applied Mathematical Modelling, 34(2):334-342, 2010

[8] C. Zhao, M. Ji, H. Tang. Parallel-machine scheduling with an availability constraint. Computers & Industrial Engineering, 61(3):778-781, 2011

[9] D. Xu, Z. Cheng, Y. Yin, H. Li. Makespan minimization for two parallel machines scheduling with a periodic availability constraint. Computers & Operations Research, 36(6):1809-1812, 2009

[10] K. Li, S.-L. Yang, H.-W. Ma. A simulated annealing approach to minimize the maximum lateness on uniform parallel machines. Mathematical and Computer Modelling, 53(5):854-860, 2011

[11] M. Yazdani, S. M. Khalili, M. Babagolzadeh, F. Jolai. A single-machine scheduling problem with multiple unavailability constraints: A mathematical model and an enhanced variable neighborhood search approach. Journal of Computational Design and Engineering, 4(1):46-59, 10 2016

[12] N. G. Hall, C. N. Potts, C. Sriskandarajah. Parallel machine scheduling with a common server. Discrete Applied Mathematics, 102(3):223-243, 2000

[13] N. Hashemian, C. Diallo, B. Vizvári. Makespan minimization for parallel machines scheduling with multiple availability constraints. Annals of Operations Research, 213(1):173-186, Feb 2014

[14] P. Kora, P. Yadlapalli: Crossover operators in genetic algorithms. A review. International Journal of Computer Applications, 162:34-36, 03 2017

[15] S. M. Lim, A. B. M. Sultan, M. N. Sulaiman, A. Mustapha, K. Y. Leong. Crossover and mutation operators of genetic algorithms. International journal of machine learning and computing, 7(1):9-12, 2017

[16] T. Cheng, C. Sin. A state-of-the-art review of parallel-machine scheduling research. European Journal of Operational Research, 47(3):271-292,1990

[17] T. Jiang, C. Zhang, H. Zhu, G. Deng. Energy-efficient scheduling for a job shop using grey wolf optimization algorithm with double-searching mode. Mathematical Problems in Engineering, 2018:8574892, Oct 2018

[18] V. Nguyen, N. Huynh Tuong, H. Tran, N. Thoai. An MILP-based makespan minimization model for single-machine scheduling problem with splitable jobs and availability constraints. pp. 397-400, 01 2013