J Math Model Algor (2009) 8:271–292
DOI 10.1007/s10852-008-9101-1

Hybrid Flow-Shop: a Memetic Algorithm Using Constraint-Based Scheduling for Efficient Search

Antoine Jouglet · Ceyda Oğuz · Marc Sevaux

Received: 3 March 2006 / Accepted: 11 December 2008 / Published online: 21 January 2009

Abstract The paper considers the hybrid flow-shop scheduling problem with multiprocessor tasks. Motivated by the computational complexity of the problem, we propose a memetic algorithm for this problem in the paper. We first describe the implementation details of a genetic algorithm, which is used in the memetic algorithm. We then propose a constraint programming based branch-and-bound algorithm to be employed as the local search engine of the memetic algorithm. Next, we present the new memetic algorithm. We lastly explain the computational experiments carried out to evaluate the performance of three algorithms (genetic algorithm, constraint programming based branch-and-bound algorithm, and memetic algorithm) in terms of both the quality of the solutions produced and the efficiency. These results demonstrate that the memetic algorithm produces better quality solutions and that it is very efficient.

Keywords Multiprocessor task scheduling · Hybrid flow-shop · Genetic algorithm · Constraint programming · Memetic algorithm

A. Jouglet (✉)
HEUDIASYC, UMR CNRS 6599, Centre de Recherche de Royallieu,
Université de Technologie de Compiègne, BP 20529,
60205 Compiègne cedex, France
e-mail: antoine.jouglet@hds.utc.fr

C. Oğuz
Department of Industrial Engineering, Koç University,
Rumeli Feneri Yolu, Sarıyer, 34450 İstanbul, Turkey
e-mail: coguz@ku.edu.tr

M. Sevaux
UEB—Lab-STICC, UMR CNRS 3192, Centre de Recherche,
Université de Bretagne Sud, BP 92116, 56321 Lorient cedex, France
e-mail: marc.sevaux@univ-ubs.fr

1 Introduction

In this paper, we consider the following scheduling problem: a set $J = \{1, 2, \ldots, n\}$ of n independent jobs has be processed in a k-stage flow-shop. We assume that all processors and all jobs are available from time $t = 0$, and each processor can process at most one job at a time. Each job j is composed of multiprocessor tasks, and each stage i has m_i identical parallel processors (machines), $i = 1, 2, \ldots, k$; $j \in J$. It is convenient to view a job as a sequence of k tasks, where each task of a job j corresponds to a stage i (T_{ij}), and the processing of any task can commence only after the completion of the preceding task. Each task T_{ij} is characterized by its processing time p_{ij} and its processor requirement $size_{ij}$, and we say that task T_{ij} has to be processed simultaneously on $size_{ij}$ of m_i identical parallel processors of stage i for an uninterrupted period of p_{ij} units of time, $i = 1, 2, \ldots, k$; $j \in J$. Let $\pi_i(\sigma)$ denote the sequence of jobs at stage i in schedule σ and $C_{ij}(\sigma)$ be the completion time of the i-th task of job j in schedule σ, $i = 1, 2, \ldots, k$; $j \in J$. Our objective is to minimize the maximum completion time of all jobs in schedule σ, that is the makespan, which is denoted by $C_{\max}(\sigma) = \max_{j \in J}\{C_{kj}(\sigma)\}$. Our problem can be denoted by $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ by using the well-known three-field notation (see, for example, Błażewicz et al. [3]).

This scheduling problem arises in several settings such as computer systems [13], container terminals for berth allocation [15], manufacturing environments for maintenance works [6], and finally machine vision systems [24]. Furthermore it is a generalization of the $F2(1, 2)||C_{\max}$ problem, that is makespan minimization in a two-stage hybrid flow-shop with one processor at one stage and with classical job definition. The $F2(1, 2)||C_{\max}$ problem is an $\mathcal{NP}$-hard problem [10] and one can see that researchers resort to either heuristics or branch-and-bound algorithms to solve its different versions [21, 30]. Furthermore Portman et al. [27] showed that using a genetic algorithm within a branch-and-bound algorithm improves the performance of the branch-and-bound algorithm considerably for a hybrid flow-shop scheduling problem. Due to the computational complexity result of the $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem, heuristic and metaheuristic algorithms have been proposed for this problem as well [24–26, 28]. These studies showed that genetic algorithm performs very well in terms of both the solution quality and the efficiency.

In view of the success of combining an exact algorithm with the genetic algorithm for a hybrid flow-shop scheduling problem, in this paper, we incorporated a constraint programming based branch-and-bound algorithm into a genetic algorithm and propose a memetic algorithm to provide a better quality solution to the $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem in a shorter time especially for the large-size instances of the problem.

The rest of the paper is organized as follows. Section 2 first describes the special features of the genetic algorithm and then presents its specific implementation details. Section 3 gives the details of the constraint programming based branch-and-bound algorithm. Section 4 presents the memetic algorithm which combines the genetic algorithm with the constraint programming based scheduling method. Section 5 illustrates the computational experiments employed to test the performance of the memetic algorithm. After describing the data generation, we subsequently report the computational results. We first compare the performance of our genetic algorithm with that of the genetic algorithm of Oğuz and Ercan [24]. We then compare the

performance of our genetic algorithm with that of the constraint programming based branch-and-bound algorithm alone and with that of the memetic algorithm. We further compare the performance of the memetic algorithm with that of the algorithms provided in the literature. Section 7 concludes the paper.

2 The Genetic Algorithm

Genetic algorithms, motivated by the natural evolution, was first introduced by Holland [11] and widely adapted to solve optimization problems (see, for example, Goldberg [8]). Starting from a set of solutions (*population*), genetic algorithms iteratively work on certain solutions (*parents*) by means of genetic operators (*selection, crossover, mutation* and *replacement*) to obtain better solutions (*offspring*). Each solution is referred to as a *chromosome* and it is composed of *genes* that characterize the solution. The search in genetic algorithms, which is stochastic in nature, is guided by the *fitness* of the chromosomes, which is determined by the associated value of the objective function for each solution.

In the following, we will explain our implementation of the genetic algorithm for the $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem by giving details for each component of the genetic algorithm. The genetic algorithm developed in this paper is in some parts similar to one of the variants presented by Oğuz and Ercan [24]. For completeness, we will describe how a solution is *coded* to a chromosome in the population and how a feasible schedule can be constructed from this coded solution. Note that the *crossover* and *mutation* operators are the same as in Oğuz and Ercan [24]. For further details of these components and their implementation, we refer the reader to that paper. On the other hand, the generation of *initial solutions*, the *stopping condition*, and the *selection* are different in our implementation and will be explained in detail.

Coding a solution A solution to single machine scheduling problems is usually coded as a permutation of jobs, which is the natural representation of a solution in this case. For the hybrid flow-shop scheduling problems, there exist several representations with varying degrees of complexity that are used for coding solutions. In Portmann (unpublished manuscript), a list of permutations at each stage is used in a general genetic algorithm approach. A more complex representation is used in Portmann et al. [27]. In that paper, two types of variables are necessary, the assignment variables for each stage and each operation (the indices of the machines which perform the operation) and the sequencing variables for each stage, in form of a matrix that contains the precedence constraints between the operations. For this latter representation, the crossover and mutation operators are designed very carefully to generate the offspring that represent solutions themselves.

In our case, as in Oğuz and Ercan [24], we prefer to keep an easier representation, one that can easily be coded for implementation. The permutation of the jobs at the first stage will be the coding of a solution. It is clear that using this coding, a permutation could generate several different schedules. This will be explained next.

Decoding of a solution A list scheduling algorithm is used to obtain a feasible schedule for the permutation that codes a solution. This algorithm chronologically constructs the schedule by iteratively assigning the jobs to the processors according to the ordered list of completion times at the previous stage. This list scheduling

Table 1 Data for the 2-stage 5-job example

Jobs j	1	2	3	4	5	6	7	8	9
p_{1j}	4	5	5	4	3	2	1	1	2
$size_{1j}$	1	3	3	3	3	1	2	2	2
p_{2j}	2	6	2	1	1	4	1	2	1
$size_{2j}$	4	5	2	5	3	2	1	2	3

algorithm preserves the general order induced by the representation. In that way, semi-active schedules are built but non-permutation sequences can be constructed. A simple example from Oğuz and Ercan [24] is given below.

Example Consider a 2-stage 5-job example for which the data are given in Table 1 (p_{ij} and $size_{ij}$ represent respectively the processing time and the processor requirement at stage i of job j). Each stage can run up to five processors in parallel. The ordered sequence of jobs for the example considered will be $\pi_1 = \{2, 3, 1, 4, 7, 6, 5, 8, 9\}$.

First, according to the sequence $\pi_1 = \{2, 3, 1, 4, 7, 6, 5, 8, 9\}$, the jobs are scheduled at the first stage by the decoding algorithm. Note that even if job 1 might have been scheduled at time 0 (enough time and processors are available), the algorithm did not choose this option to follow the general order induced by the representation. The alternative way of scheduling, which schedules job 1 at time 0, has the advantage of more densely packing the jobs but at the same time, it may lead to an identical solution for many different permutations, so many sequences would be handled by the genetic algorithm "for nothing". Figure 1 presents the resulting schedule after

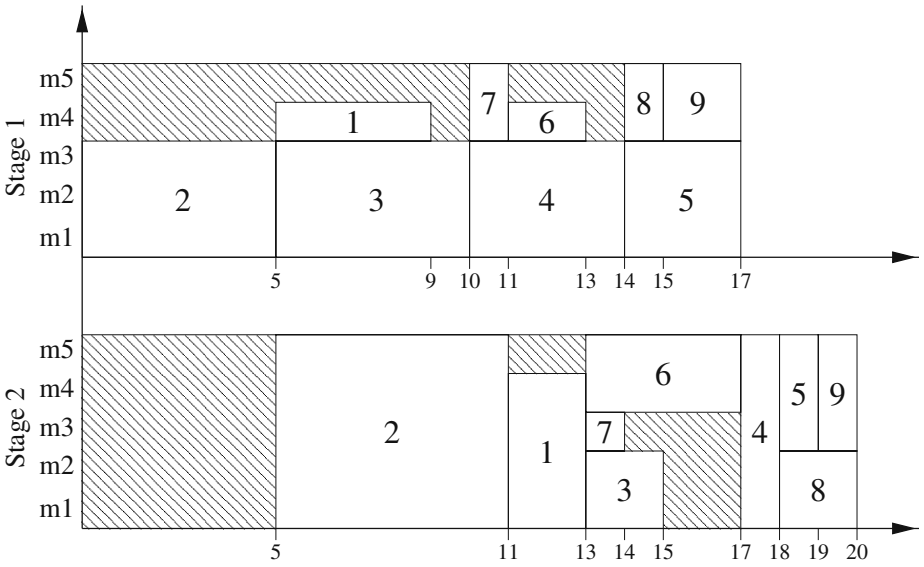

Fig. 1 The schedule generated by the decoding algorithm for the example

applying the decoding algorithm. After sequencing the jobs at the first stage, the new order is $\pi_2 = \{2, 1, 3, 7, 6, 4, 8, 5, 9\}$. Jobs at stage 2 are scheduled in that order.

Initial solutions An initial population is randomly generated. As usually recognized by the researchers, good solutions that are introduced in the initial population speed up the convergence of the genetic algorithm. In Oğuz et al. [25], different priority lists have been used for a two-stage hybrid flow-shop problem. We will consider four best performing priority lists from that paper after extending them to the multi-stage problem. These priority list rules are described below:

1. Find a sequence S by sorting the jobs in non-increasing order of last stage's processing times multiplied by last stage's processor requirements.
2. Find a sequence S by sorting the jobs in non-decreasing order of stage 1 processing times multiplied by stage 1 processor requirements.
3. Find a sequence S by sorting the jobs in non-increasing order of stage 1 processor requirements.
4. Find a sequence S by sorting the jobs in non-increasing order of last stage's processor requirements.

Selection The binary tournament technique is one of the simplest and most reliable techniques to implement. When selecting a chromosome for crossover operation, we want to make a biased random selection (giving more chance to the best chromosomes to be selected). To obtain such a chromosome, we select at random (with uniform repartition) two chromosomes in the population and keep the best as the first parent. We perform this operation again for obtaining the second parent. To avoid useless crossover operations, we force the second parent to be different from the first one.

Crossover and mutation operators The NXO crossover operator described in Oğuz and Ercan [24] is used in this genetic algorithm. This operator has been specially designed for hybrid flow-shop problems with multiprocessor tasks and performs well. It preserves the interesting characteristics of the parents, such as collections of consecutive jobs, in the resulting offspring. In Oğuz and Ercan [24], the PMX crossover operator has also been tested but the results indicate that the NXO operator gives better results when combined with the insertion mutation operator.

The insertion mutation operator is one of the most commonly used operator for scheduling problems. One of the jobs of a sequence is picked at random and inserted at a random place elsewhere in the chromosome structure. This mutation operator preserves the structure of several consecutive jobs but inserts one job in a new position. So depending on the place where the job is picked up from and inserted into, the effect of the mutation operator will be important or not. We note here one difference with the genetic algorithm of Oğuz and Ercan [24] regarding the application of mutation operator. We apply the mutation operator with probability p_m to an offspring only if its fitness is worse than the fitness of the best chromosome in the population. However, the mutation operator was performed on an offspring with probability p_m without checking its fitness in Oğuz and Ercan [24].

Stopping conditions One of the aims of this study is to perform a comparison between different solution procedures and for a fair comparison, the same duration to

run each solution procedure is used. After few experiments on the largest instances, the total duration given to each solution method was set to 15 min (900 s) of CPU time. In addition, we note that it is not necessary to spend more time trying to solve an instance if we reach a lower bound, an optimal solution, or a local minimal solution from which we cannot escape. For this reason, an additional stopping condition of a maximum number of 10,000 iterations without improvement of the best solution has been added with also a check on the lower bound value.

3 The Constraint Programming Based Branch-and-Bound Algorithm

Constraint programming is a programming paradigm aimed at solving combinatorial optimization problems. Indeed, these problems can be solved by defining them as a Constraint Satisfaction Problem: the problem is described by a set of variables, a set of possible values for each variable, and a set of constraints between the variables. The set of possible values of a variable is called the variable domain. A constraint between variables expresses which combination of values for the variables are allowed. The question to be answered is whether there exists an assignment of values to variables, such that all constraints are satisfied. If such an assignment exists, it is a solution to the Constraint Satisfaction Problem. The power of constraint programming method is largely due to the fact that constraints can be used in an active process called "constraint propagation" in which certain deductions are made to reduce computational effort needed to solve problems, by removing values from the domains, deducing new constraints, and detecting inconsistencies. In recent years, constraint programming techniques have been successfully used to model and solve scheduling problems [2]. Thus "constraint-based scheduling" is defined as the discipline that studies how to solve scheduling problems using constraint programming methods. Combining ideas from Artificial Intelligence and Operations Research, constraint-based scheduling is able to maintain the best of both approaches, i.e., the flexibility of Artificial Intelligence scheduling systems and the efficiency of Operation Research algorithms.

The aim of this section is to propose a branch-and-bound algorithm with constraint propagation techniques to solve the $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem under the constraint that a permutation π_1 is given. We employed the constraint-based scheduling tool ILOG SCHEDULER [12] to develop this branch-and-bound algorithm. In Section 3.1, we describe the problem as an instance of the Constraint Satisfaction Problem. We then describe in Section 3.2 the proposed branch-and-bound algorithm. Finally, we describe in Section 3.3 some techniques which are used to improve the performance of the branch-and-bound procedure.

3.1 Modelling the Problem in Constraint Programming

Recall that a solution is encoded as a permutation of n jobs at stage one in the genetic algorithm and then decoded into a schedule by a list scheduling algorithm. The list scheduling algorithm used in the genetic algorithm is very efficient to build a feasible solution from a given permutation and to compute the fitness of the solutions quickly, but has some drawbacks.

Let $C^l_{\max}(\sigma(\pi_1))$ be the makespan of the schedule σ built by the list scheduling algorithm from the permutation π_1. Let $C^*_{\max}(\sigma(\pi_1))$ be the optimal makespan which is found under the constraint that the jobs at the first stage are sequenced in the order defined by π_1. Note that due to the restriction in the list scheduling algorithm that the order imposed from the previous stage is preserved at each stage, $C^l_{\max}(\sigma(\pi_1))$ is not necessarily equal to $C^*_{\max}(\sigma(\pi_1))$. Of course, $C^*_{\max}(\sigma(\pi_1))$ is also not necessarily the optimal makespan of the $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem, that is, $C^*_{\max}(\sigma)$. By these remarks, we note that the genetic algorithm may not be able to produce the optimal solution in certain cases because of the decoding algorithm used.

Nevertheless, the solutions produced by the genetic algorithm were observed to be of high quality most of the time, i.e., very near to the optimum. If, in addition, we make the hypothesis that the sequence π_1 obtained by the genetic algorithm is very close to the sequence which can produce the optimal makespan, then π_1 seems to be a precious information on the structure of the optimal solution.

The main idea of our algorithm is to propose a kind of refining method which allows us to exploit the results of the genetic algorithm and to find a better solution, which can be proven in certain cases to be the optimal solution. For that purpose, we propose to use a branch-and-bound procedure using constraint programming.

In constraint programming, the $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem can be efficiently encoded in terms of variables and constraints in the following way [2]. Let us remind that T_{ij} is the task i of job j; $i = 1, 2, \ldots, k$, $j \in J$. For each task T_{ij} two variables are introduced, $start(T_{ij})$ and $end(T_{ij})$, representing the start time and the end time of the task T_{ij}, respectively; $i = 1, 2, \ldots, k$, $j \in J$. Note that the domains of these variables are tightened during the branch-and-bound procedure as a result of a combination of decisions and constraints propagation.

Temporal relations between the tasks are expressed by linear constraints between the start and the end variables of the tasks. Then, the precedence between two successive tasks T_{ij} and $T_{i+1,j}$ of the same job j is modelled by the linear constraint $end(T_{ij}) \leq start(T_{i+1,j})$; $i = 1, 2, \ldots, k-1$, $j \in J$. Such constraints are easily propagated using a standard arc-B-consistency algorithm [16], which ensures that, when a schedule has been found, all precedence constraints are respected.

Cumulative resource constraints represent the fact that the tasks require some amount of a resource throughout their execution. For our problem, the propagation of the resource constraints mainly consists of maintaining arc-B-consistency on the formula

$$\forall i \in \{1, 2, \ldots, k\},\ \forall t, \sum_{\substack{j \in \{1,2,\ldots,n\} \text{ such that} \\ start(T_{ij}) \leq t < end(T_{ij})}} size_{ij} \leq m_i.$$

In other words, the sum of processor requirement of those tasks, for which $start(T_{ij}) \leq t < end(T_{ij})$, at stage i and at time t, has to be lower than the number of available processors m_i at stage i; $i = 1, 2, \ldots, k$, $\forall t$ (see for instance Le Pape [14]).

The makespan criterion is represented by an additional variable $\overline{C}_{\max}$. Its value is determined by the constraint

$$\overline{C}_{\max} = \max_{i,j} end(T_{ij}),$$

and arc-B-consistency is used to propagate this constraint as well. It ensures that when a schedule has been found, the value of $\overline{C}_{\max}$ is actually the makespan of the schedule.

3.2 Solving the Problem

To find an optimal solution, we solve successive variants of the decision problem, i.e., a problem in which a constraint on the makespan value is added to the problem ($\overline{C}_{\max} \leq \mathrm{UB}$). For the decision problem, the constraint programming approach returns a feasible schedule for this maximum makespan value (UB) or fails. Each time a feasible solution is found (with possibly a new upper bound UB′), the maximum makespan value UB (or UB′) is decreased by one unit and the branch-and-bound algorithm is restarted. Each time, the search resumes at the last visited node of the previous iteration. When no other solution can be found, the last one found is the optimal solution of the problem.

We use a schedule-or-postpone method to solve the decision problem which works as follows: at each step of the branch-and-bound method, we choose an unscheduled task T_{ij} of minimal earliest start time and we schedule it as early as possible, as allowed by the previously scheduled tasks (which have smaller start times than T_{ij}) and by the preceding tasks of the same job (on other resources). Note that each time such a job is scheduled, the constraint propagation techniques are triggered to reduce the domains of variables and then to reduce the search space. If that decision leads to a failure, i.e., no solution can be found according to this start time, choosing the start time of this activity is postponed.

This activity might be considered again when its earliest start time is removed. Such an adjustment can occur as a result of a combination of decisions and propagation. The schedule is built by using a depth first strategy.

At each step of the schedule, the job is selected as follows. A priority sequence of the jobs is provided to the branch-and-bound procedure. When a job has to be scheduled, we then choose the task which is not postponed and which can be scheduled at the stage with the smaller index. Ties are broken choosing the job with the smaller index in the priority sequence among those which can be scheduled the earliest.

If the branch-and-bound procedure is used in the memetic algorithm (which will be explained in Section 4) to perform a hybrid search in cooperation with the genetic algorithm, the priority sequence is provided by the genetic algorithm. If the branch-and-bound algorithm is used alone to solve the problem, the priority sequence is obtained by sorting the tasks in non-decreasing order of stage 1 processing times multiplied by stage 1 processor requirements. Indeed, this heuristic has been shown to give the best results among several other ones in Oğuz et al. [25].

If the branch-and-bound is stopped before reaching the optimal solution, the best schedule found so far (if any) is returned.

3.3 Improving the Performance of the Branch-and-Bound Procedure

Several ways are used to improve the performance of the branch-and-bound procedure. Particularly, several methods relying on jobs' time-windows to propagate the resource constraints are used to update and adjust these sets. These rules are

often pre-implemented in constraint-based scheduling systems such as ILOG SCHEDULER [12].

- At each node of the search tree, we compute a lower bound of the cost of scheduling remaining unscheduled jobs. If this lower bound is greater than the upper bound $\overline{C}_{\max}$ of the decision problem, then a backtrack occurs. For that, we use the lower bound described by Oğuz and Ercan [24] for this problem.
- We use the propagation of the disjunctive constraint which compares the temporal characteristics of pairs of tasks: two tasks T_{ij} and T_{ik} such that $size_{ij} + size_{ik} > m_i$ cannot overlap in time since they require the same resource i and since scheduling both tasks at the same time requires more processors than the maximum number of processors at stage i. Hence, either T_{ij} precedes T_{ik} or T_{ik} precedes T_{ij}, i.e., the disjunctive constraint holds between these tasks. Arc-B-consistency can be then achieved by the formula

$$\left[size_{ij} + size_{ik} \leq m_i\right] \vee \left[end\left(T_{ij}\right) \leq start\left(T_{ik}\right)\right] \vee \left[end\left(T_{ik}\right) \leq start\left(T_{ij}\right)\right].$$

  If n tasks require the same resource, the constraint can be implemented as $n(n-1)/2$ disjunctive constraints.
- We use the edge-finding propagation techniques [2, 4] which are also able to adjust the time-windows according to the resource constraint. Rather than considering only pair of tasks (T_{ij}, T_{ik}) to prove that T_{ij} must precede T_{ik} or vice-versa, the edge-finding constraint propagation techniques make a decision considering the order in which tasks are processed on the resource. The goal is to determine whether a task T_{ij} must be sequenced before (or after) a given set of tasks [22, 23]. Two types of conclusions can then be drawn: new ordering relations ("edges" in the graph representing the possible orderings of jobs) and new time-bounds (earliest and latest start and completion times).
- We use energetic reasoning techniques [7, 17]. Considering the quantities of energy supplied by resources and consumed by tasks within given intervals, the energetic approach aims at developing satisfiability tests and time-bound adjustments to ensure that either a given schedule is not feasible or to derive some necessary conditions that any feasible schedule must satisfy.

4 The Memetic Algorithm

In this section, we describe the memetic algorithm proposed for the $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem. A memetic algorithm [18, 19] is a metaheuristic based on a genetic algorithm in which a local search is performed on new solutions generated during the search. Recent advances on memetic algorithm are presented in Moscato and Cotta [20]. In the proposed memetic algorithm we combine the genetic algorithm described in Section 2 with the branch-and-bound algorithm described in Section 3 to benefit from the advantages of these approaches.

The main idea is, with probability p_{hs}, to improve an offspring with the branch-and-bound algorithm using constraint programming if the offspring is better than all of the chromosomes in the population before considering its inclusion in the population. Such a local search performed on the offspring will definitely increase the quality of the solutions obtained.

In our approach, we have chosen an *incremental* version of the genetic algorithm to replace the existing chromosomes with the selected better ones. In our opinion, this version, compared to a *population replacement* version, is able to take benefits from a new chromosome at the immediate next iteration by being able to potentially use it for a new crossover operation. Moreover, since some solutions will be generated by the branch-and-bound method, and these solutions will probably be improved a lot, we would like to be able to gain benefits from them as soon as possible.

Description of the memetic algorithm Algorithm 1 describes the memetic algorithm, which is a combination of the genetic algorithm with the constraint programming based branch-and-bound algorithm.

Algorithm 1: Memetic algorithm

```
1  input: Population size n_pop, mutation rate p_m, hybrid search rate p_hs
2  Generate an initial population P of n_pop chromosomes
3  Identify best (b) and worst (w) chromosomes
4  while Stopping conditions are not met do
5  |  Selection: p_1 and p_2 from P by binary tournament
6  |  Crossover: p_1 ⊗ p_2 → c // apply NXO crossover operator
7  |  if f(c) ≥ f(b) then
8  |  |  Mutation: mutate c under probability p_m
9  |  endif
10 |  CP Search: apply the CP Search to c with probability p_hs
11 |  if f(c) ≥ f(w) then
12 |  |  Discard c
13 |  else
14 |  |  Select r by reverse binary tournament
15 |  |  c replaces r in P
16 |  endif
17 endw
```

First, an initial population of n_{pop} chromosomes is generated as described in Section 2 and two chromosomes, namely best and worst, are identified. The main loop is then repeated until some stopping conditions are met. Within the loop, the first operation performed is the selection of two chromosomes as parents for the crossover operator. The selection is made by the binary tournament technique. Next, the crossover operation is performed using the NXO operator and one offspring, c, is generated. The quality of the offspring, $f(c)$, is immediately evaluated. Here $f(c)$ refers to the value of $C^{l}_{\max}(\sigma(c))$ If the quality of the offspring is better than that of the best solution found so far, the mutation is not performed at this iteration (test line 7 of Algorithm 1), otherwise the mutation is performed with probability p_m.

With a probability p_{hs}, we apply the local search using the constraint programming based branch-and-bound algorithm (CP search). One common rule applied for accepting the new chromosome to the population is that they at least improve the worst chromosome (line 11 of Algorithm 1). If the new chromosome is accepted to the population, to keep a population of a fixed size, we have to remove one chromosome. To be sure to remove one of the worst, we can use the binary tournament as well, but

this time in a reverse way (select two different chromosomes and remove the worst one). Another strategy would be to remove the worst individual at each iteration, but as shown by Goldberg and Deb [9] the systematic deletion of this worst individual from the population induces a high selective pressure on the population (in other words, it results in a too rapid convergence).

We note that the CP search is first applied for 1 s of CPU time and each time the current solution is improved an additional second of CPU time is allowed. These stopping conditions have been defined experimentally. It appears, in most of the cases, that the constraint programming algorithm either finds very quickly an improved solution or does not find one at all. On the average, when the search is successful, the constraint programming algorithm never runs for more than 5 s. It is important to note that each time a CP search is stopped, the explored search space is kept to avoid redundant and inefficient searches among all calls of the CP search.

When CP search finds a better solution than the incumbent, the fitness value and the permutation are updated accordingly in the genetic algorithm. If CP search finds more than one better solution, only the best one is kept for the mutation.

5 Computational Experiments for $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ Problem

All of the algorithms were implemented in C++ and run on a PC Pentium 4, 1.8 GHz processor with 512 Mb memory. In the sequel, GA will denote the genetic algorithm described in Section 2, CP will denote the constraint programming based branch-and-bound algorithm of Section 3 when it is used alone and MA will denote the memetic algorithm that combines the GA and the CP as described in Section 4. Table 2 summarizes the parameters used for different algorithms and their values.

The CPU time limit of 900 s for a branch-and-bound algorithm may appear as a very strict limit. We note that, after some experiments, we concluded that the branch-and-bound algorithm cannot prove the optimality of the solution even after 3600 s which confirms the difficulty of the problem and the inadequacy of the exact solution procedures alone. In more detailed tests, we observed also that only very few solutions have been improved (and not significantly) when the time limit was extended to 1800 s of CPU time.

We will first show in the rest of this section that the genetic algorithm described in Section 2, that is GA, is of similar efficiency with the genetic algorithm described in Oğuz and Ercan [24] (GA_OE). We will then compare the three algorithms presented in this paper, namely, GA, CP and MA. The data used for the evaluation

Table 2 Parameters used in the computational experiments

Parameter	Value	Algorithm	Comments
p_m	0.1	GA, MA	Mutation rate
p_{hs}	0.0001	MA	Hybrid search rate
CPU_Max	900 s	GA, CP, MA	Maximum CPU time allowed
MaxIterImp	10 000	GA, MA	Maximum number of iterations without improvement of the best solution
CPU_HS	1s+1s+...	MA	Maximum CPU time allowed for the CP search in MA (1 s + an additional second each time the solution is improved)

of the proposed method for the $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem in these experiments are the ones generated in Oğuz and Ercan [24]. Two types of instances are provided depending on the setting of the processor configurations. For each type, the number of stages are 2, 5 or 8 and the number of jobs are 5, 10, 20, 50 or 100. This leads to a set of 300 instances. We will finally compare the performance of the proposed algorithms with that of the genetic algorithm proposed in Serifoğlu and Ulusoy [28] with their data set.

5.1 Comparison of the Two Genetic Algorithms

We first note that the stopping conditions for GA and GA_OE are not the same. In GA_OE, 30 min were allowed to produce results, i.e., 1800 s of CPU time, whereas GA uses 900 s of CPU time. We note that, increasing the CPU time for GA did not significantly improve the results. Table 3 provides results for GA_OE and GA. The columns under "Av. $C_{\max}$" give the average $C_{\max}$ value over 10 instances. The other columns under "# best" denote the number of time each method finds the best solution.

From this table, it seems that the two methods are almost equivalent. For type-1 instances, our genetic algorithm provides better results (in terms of average $C_{\max}$ values and number of best solution found) as the number of jobs increases. For type-2 instances, the genetic algorithm provided by Orğuz and Ercan [24] give better results in terms of the number of best solutions found but GA_OE and GA are equivalent when comparing the average $C_{\max}$ values. Based on these observations, in the sequel, we will focus only on GA.

5.2 Studying Algorithm MA

In Table 4, we study the stability and the convergence of the method MA. For each instance type (1 or 2), for each number of stages, we have launch the method

Table 3 Comparing the two genetic algorithms

k	n	Type 1 instances				Type 2 instances			
		Av. $C_{\max}$		# best		Av. $C_{\max}$		# best	
		GA_OE	GA	GA_OE	GA	GA_OE	GA	GA_OE	GA
2	5	267.60	267.60	10	10	256.40	256.40	10	10
	10	451.10	451.10	10	10	409.50	426.30	9	4
	20	876.70	876.50	9	9	757.40	809.50	10	5
	50	2049.40	2048.50	6	8	1671.10	1731.80	9	4
	100	4355.00	4351.50	4	10	3205.30	3242.50	9	1
5	5	472.10	472.10	10	10	423.80	423.80	10	10
	10	639.10	648.40	8	4	616.30	606.80	8	3
	20	1072.10	1077.70	9	4	948.50	950.90	8	2
	50	2604.00	2574.80	4	8	2074.60	1971.70	2	8
	100	4755.00	4755.90	4	6	4145.30	3822.30	0	10
8	5	641.60	641.60	10	10	614.20	614.20	10	10
	10	836.90	850.50	9	2	813.10	840.40	6	3
	20	1319.30	1319.90	6	4	1148.00	1144.70	6	4
	50	2669.60	2634.30	2	8	2321.80	2292.20	5	5
	100	5327.70	5260.20	1	9	4216.60	4412.10	9	1

Table 4 Studying MA

Type	k	$C_{\max}$ values			χ			Number of CP calls		
		min	av.	max	min	av.	max	min	av.	max
1	2	666	670.4	676	0.809	0.903	0.993	117	129	149
	5	1167	1167.0	1167	0.529	0.924	0.994	63	73	81
	8	1387	1398.6	1426	0.529	0.780	0.927	47	56	64
2	2	760	760.0	760	0.938	0.961	0.983	113	122	126
	5	1042	1042.2	1044	0.332	0.821	0.967	61	70	76
	8	1071	1093.3	1106	0.089	0.424	0.821	44	52	62

MA, 10 times on an instance of 20 jobs which is not proved to be optimal. The search is stopped after 900 s. We report the minimum, the average and the maximum value on the 10 launches of the method. To study the convergence, for each launch is computed the value $\chi = (nb\_total\_iter - nb\_best\_iter)/nb\_total\_iter$ where nb_total_iter and nb_best_iter are respectively the total number of iteration and the number of iterations when the best value has been found. We report then report the minimum, the maximum and the average of χ on the 10 launches. Moreover we report the minimum, the maximum and average number of times the CP search is run all along the method MA. The method seems to be rather stable for a $k = 2$ or $k = 5$. Nevertheless, for $k = 8$, it seems that the method need more CPU times. Analyzing the convergence seems confirm this situation since, contrary to instances with $k = 2$ and $k = 5$, the best value is found later in the search.

5.3 Comparison of GA, CP and MA

In Table 5, we present the average $C_{\max}$ values found by each method over the 10 instances of each class of data. From this table, we can observe that when the number of jobs is small (5 or 10) CP is able to give optimal results in most of the cases,

Table 5 Comparing $C_{\max}$ average values

k	n	Type 1 instances			Type 2 instances		
		GA	CP	MA	GA	CP	MA
2	5	267.6	267.0	267.0	256.4	253.5	253.5
	10	451.1	451.1	451.1	426.3	422.0	422.1
	20	876.5	889.8	877.7	809.5	832.5	807.6
	50	2048.5	2087.0	2046.1	1731.8	1794.1	1722.2
	100	4351.5	4417.7	4348.6	3242.5	3325.9	3215.0
5	5	472.1	466.6	466.6	423.8	418.4	418.4
	10	648.4	637.8	637.8	606.8	596.8	594.9
	20	1077.7	1151.0	1070.3	950.9	1066.5	945.3
	50	2574.8	2680.7	2571.7	1971.7	2134.1	1960.7
	100	4755.9	4868.5	4746.9	3822.3	3965.2	3802.0
8	5	641.6	616.7	616.7	614.2	599.9	599.9
	10	850.5	854.0	840.3	840.4	820.9	824.0
	20	1319.9	1468.2	1315.2	1144.7	1256.2	1133.4
	50	2634.3	2950.2	2623.1	2292.2	2569.2	2315.7
	100	5260.2	5643.9	5267.2	4412.1	4525.8	4330.7

Table 6 Comparing the number of the best solution found

k	n	Type 1 instances			Type 2 instances		
		GA	CP	MA	GA	CP	MA
2	5	8	10	10	8	10	10
	10	10	10	10	5	10	9
	20	9	6	8	7	3	9
	50	5	4	10	4	2	9
	100	7	5	10	0	0	10
5	5	5	10	10	6	10	10
	10	3	10	10	1	8	7
	20	6	1	10	3	0	9
	50	7	2	9	3	0	8
	100	5	1	7	3	2	5
8	5	2	10	10	4	10	10
	10	2	8	7	1	8	5
	20	5	2	8	5	0	7
	50	3	0	7	7	0	3
	100	7	0	4	1	2	7

however, the performance of CP gets worse as the number of jobs increases. We further see that MA performs better than GA as the number of jobs increases. We also note that for the small size instances ($n = 5$ or 10), optimal solutions are found as soon as the CP Search is started in MA, and the memetic algorithm is stopped immediately. This is the reason why either the first line or the first two lines of each instance class present similar results for CP and MA. Overall, it is easy to see that the best results are produced by MA.

In Table 6, we present the number of time the best $C_{\max}$ values (among the three methods) are retrieved by each method. In this case, we can again make the similar observations as for Table 5. Among the three methods, MA provides the best solutions in 90% of all cases. Again, for small size instances (up to 10 jobs) CP solves the problem efficiently. Most of the time, CP is able to find the optimal solution very

Table 7 Comparing the number of the optimal value found

k	n	Type 1 instances			Type 2 instances		
		GA	CP	MA	GA	CP	MA
2	5	8	10	10	8	10	10
	10	10	10	10	4	9	8
	20	5	5	5	4	3	4
	50	4	4	5	3	2	4
	100	5	5	5	0	0	0
5	5	5	10	10	6	10	10
	10	3	10	10	0	7	5
	20	3	0	3	0	0	0
	50	4	2	5	0	0	0
	100	2	1	6	0	0	0
8	5	2	10	10	4	10	10
	10	2	6	5	0	5	4
	20	2	2	2	0	0	0
	50	0	0	1	0	0	0
	100	1	0	1	0	0	0

Table 8 Deviation of C_{max} value from the optimal value or from the lower bound (in %)

k	n	Type 1 instances			Type 2 instances		
		GA	CP	MA	GA	CP	MA
2	5	0.29	0.00	0.00	1.23	0.00	0.00
	10	0.00	0.00	0.00	10.45	9.33	9.36
	20	0.44	2.59	0.66	9.63	12.90	9.34
	50	0.63	2.79	0.49	5.33	9.29	4.70
	100	0.15	1.96	0.07	2.67	5.30	1.77
5	5	1.35	0.00	0.00	1.44	0.00	0.00
	10	1.64	0.00	0.00	3.71	1.92	1.60
	20	3.49	10.85	2.78	7.83	20.78	7.15
	50	0.59	5.30	0.51	4.90	13.61	4.35
	100	2.50	5.19	2.33	10.67	14.89	10.12
8	5	4.15	0.00	0.00	2.38	0.00	0.00
	10	9.38	10.32	8.02	9.32	6.80	7.24
	20	5.69	17.98	5.32	17.26	28.52	16.02
	50	2.17	14.42	1.71	15.62	29.62	16.87
	100	2.02	9.49	2.18	18.85	21.97	16.64

quickly, hence, the first time the exact method is called inside MA, it also finds the optimal solution. On the other hand, for bigger instances MA proves its superiority over CP and most of the time over GA.

In Table 7, we present the number of the optimal value found by each method. In this case, optimal values can be obtained either from the equality with the lower bound or from the CP. We observe that similar conclusions as above can be drawn from this table as well.

In Table 8, the average deviation of C_{max} values from the optimal value (if known, or from the lower bound) in percentage is presented with 1% significance. In Table 9, we present the CPU times for each algorithm. Finally, in Table 10, we provide the average total number of iterations ("total") performed by each method and the average number of iterations performed by each method (for GA and MA) until

Table 9 Comparing the CPU times (in seconds)

k	n	Type 1 instances			Type 2 instances		
		GA	CP	MA	GA	CP	MA
2	5	598.8	0.03	0.8	586.3	0.02	0.7
	10	900	0.05	0.9	900	91.19	763.1
	20	900	450.54	450.4	900	648.35	541.2
	50	900	540.69	454.9	900	725.09	629.8
	100	900	451.62	458.8	900	900	900
5	5	900	0.04	1.1	900	0.04	1.7
	10	900	60.55	541.6	900	411.13	843.6
	20	900	900	634.6	900	900	900
	50	900	721.31	467.8	900	900	900
	100	900	812.03	551.7	900	900	900
8	5	900	0.11	2.2	900	0.06	1.2
	10	900	386.8	720.4	900	560.7	900
	20	900	720.88	721	900	900	900
	50	900	900	816.1	900	900	900
	100	900	900	865.1	900	900	900

Table 10 Comparing the number of iterations

k	n	Type 1 instances					Type 2 instances				
		GA		CP	GA		GA		CP	GA	
		Best	Total	Total	Best	Total	Best	Total	Total	Best	Total
2	5	524	10000000	42	4897	13624	72	10000000	111	2315	9757
	10	8961	9183121	559	1417	7618	18324	8734659	2145364	14781	1646121
	20	135891	4832355	7169095	21172	1441496	383025	4574501	12216085	132555	1708922
	50	545632	1775613	4948851	411678	712311	640107	1658976	4642825	645033	978821
	100	290061	782857	1397422	171486	296016	611295	692942	1399499	500985	603711
5	5	587	5846651	261	4210	8077	198	6696837	339	5169	12663
	10	65641	3534139	717247	451196	1565928	9669	3922615	6053157	214045	2353470
	20	104085	1807291	6724966	300039	1147494	201167	1997714	9812707	587018	1573041
	50	144148	689480	1875092	119454	354410	618607	700449	5416996	538762	635221
	100	192770	275685	1273432	142998	153517	253181	273937	1608930	163469	237803
8	5	555	4724608	935	7972	11562	449	4806862	452	3352	6612
	10	19611	2721753	2956198	249414	1593900	275096	2682765	3856187	794809	1771754
	20	62287	1388172	5025718	231034	930972	279865	1324964	6328804	476577	997060
	50	238199	494018	2853294	287951	412197	399889	450401	4237140	238851	350263
	100	116626	194004	795773	117577	150815	131009	171275	1046851	109575	132472

the best value is found. These tables complements the previous three tables. One can easily note that these results confirm the above observations and that MA performs very good, especially for Type 1 instances. It is apparent that the combination of the constraint programming based branch-and-bound algorithm with the genetic algorithm, in the form of a memetic algorithm, improves the search in terms of number of best solutions found, number of optimal solutions retrieved and deviation from the optimal value or the lower bound. We can conclude that even though CP can produce optimal or near-optimal solutions for small-size problem instances efficiently, it is necessary to have a metaheuristic method to solve larger instances.

Finally we have remarked that if we use a more sophisticated approach in the CP search, such as energetic reasoning techniques, then CP alone performs better but, in MA, this approach disturbs the search. Indeed, time consuming techniques ran at each node of the search tree prevents MA converging to solutions when the number of jobs is large. From our computational experiments, we can conclude that each technique used in CP of MA has to be very efficient from the point of view of CPU time if we hope to explore an interesting search space during the short time allowed to CP each time MA calls it.

5.4 Comparison with Serifoğlu and Ulusoy [29]

To prove the efficiency of our proposed approach, we further test our three methods (GA, CP and MA) on the instances from Serifoğlu and Ulusoy [28] and compare the results with those provided directly by the authors themselves (GA_SU). In Table 11, like for previous tests, we report the average $C_{\max}$ values. From the observation,

Table 11 Comparing $C_{\max}$ average values for [28]

k	n	Type 1 instances				Type 2 instances			
		GA_SU	GA	CP	MA	GA_SU	GA	CP	MA
2	5	275.3	275.3	275.3	275.3	252.2	252.2	251.7	251.7
	10	506.9	506.9	506.9	506.9	390.6	392.9	388.6	388.6
	20	820.2	821.5	838.4	820.6	691.3	690.6	723.5	688.3
	50	2024.3	2007.1	2069.2	2004.2	1737	1704.1	1779	1703.2
	100	4183.4	4127.4	4230	4135.4	3395.4	3305.8	3398.3	3303
5	5	452.9	452.9	446	446	438.6	438.6	429.2	429.2
	10	668.2	669.2	660.1	661.6	627.4	635.5	611.9	614.5
	20	1109.6	1110.4	1179	1109.2	924	931.6	1006.1	917.1
	50	2486	2472.6	2540.8	2468.3	1971.5	1931.1	2148.7	1935.1
	100	4665.2	4636.7	4790.4	4622.2	3910.9	3954.7	4056.3	3849.1
8	5	632.4	631	621.4	621.4	614.1	613.4	592.7	592.7
	10	854.2	860.1	847.4	840.5	829.1	841	826.8	819.4
	20	1337.3	1345	1464.8	1336.8	1181.4	1195.4	1320.3	1179.6
	50	2777.5	2741.2	3023.2	2742.6	2275.8	2290.2	2518.4	2261.5
	100	5142.3	5155.7	5542.4	5157.7	4290.3	4382	4498.4	4320.6
10	5	809.9	810	794.3	794.3	748.5	748.5	724.8	724.8
	10	999.9	1010.8	997.2	988.1	899.9	905.7	887	874.3
	20	1459.3	1455.9	1645.3	1462.7	1303.9	1336.7	1463	1324.7
	50	2922	2899.4	3351.4	2893.3	2541	2552.4	2850.9	2544.7
	100	5358.8	5329.1	5722.6	5323.7	4490.4	4622.1	4727.9	4505.9

Table 12 Comparing the number of the best solution found for [28]

k	n	Type 1 instances				Type 2 instances			
		GA_SU	GA	CP	MA	GA_SU	GA	CP	MA
2	5	10	10	10	10	9	9	10	10
	10	10	10	10	10	6	6	10	7
	20	10	8	5	9	5	5	3	8
	50	4	8	4	9	0	7	2	3
	100	3	9	3	6	0	7	1	10
5	5	3	3	10	10	6	6	10	10
	10	3	3	9	7	0	0	10	5
	20	9	6	3	9	2	0	0	4
	50	4	5	2	9	1	6	0	6
	100	3	2	0	9	0	0	0	4
8	5	3	3	10	10	3	3	10	10
	10	2	1	6	8	3	0	5	5
	20	6	3	0	5	4	2	0	4
	50	0	6	0	8	4	0	0	6
	100	7	4	0	3	6	0	0	4
10	5	4	4	10	10	2	2	10	10
	10	4	2	7	5	1	1	4	8
	20	3	4	0	6	6	2	0	2
	50	3	4	0	7	4	2	0	4
	100	4	5	0	6	7	0	0	3

we can say that better solutions are found by MA comparing to the ones obtained by GA and CP, but also by GA_SU. MA performs better than GA_SU in 31 cases out of 40 and performs the same in two cases.

Table 13 Comparing the number of proven optima for [28]

k	n	Type 1 instances				Type 2 instances			
		GA_SU	GA	CP	MA	GA_SU	GA	CP	MA
2	5	1	0	10	10	0	0	10	10
	10	4	0	9	7	0	0	10	5
	20	1	0	4	5	0	0	3	1
	50	4	0	6	7	0	0	2	0
	100	3	0	3	1	0	0	1	0
5	5	0	0	10	10	0	0	10	7
	10	2	0	9	2	0	0	8	0
	20	3	0	3	8	0	0	0	0
	50	4	0	2	6	0	0	0	0
	100	3	0	0	2	0	0	0	0
8	5	0	0	10	10	0	0	10	6
	10	0	0	3	0	0	0	2	0
	20	0	0	0	0	0	0	0	0
	50	0	0	0	0	0	0	0	0
	100	4	0	0	0	0	0	0	0
10	5	0	0	10	4	0	0	10	7
	10	0	0	3	1	0	0	3	0
	20	1	0	0	0	0	0	0	0
	50	0	0	0	0	0	0	0	0
	100	1	0	0	0	0	0	0	0

The same conclusion can be drawn from Table 12 on the number of time the best solution is found by each method. We can also observe that MA performs very well for all type 1 instances and for type 2 instances with 5 stages or less.

In Table 13, the number of proven optima is reported for each method. Overall MA is able to find the proven optima more often than GA_SU. We note that while GA_SU cannot find any proven optima for type 2 instances, we can still prove some optimal solutions when the number of jobs is 5 or 10.

From the above results, we can say that the MA outperforms the competitive method from Serifoğlu and Ulusoy [28] for the $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem.

6 Computational Experiments for $Fk(Pm_1, \ldots, Pm_k)||C_{\max}$ Problem

In this section, we report the performance of the three approaches proposed, namely, GA, CP and MA, on the hybrid flow-shop problems with classical job definition. In classical hybrid flow-shop problems, the resource requirements of each task is equal to 1, i.e.the machines considered at each stage are non-cumulative ones. For the computational experiments, we considered the benchmark data from Vignier [29], Carlier and Néron [5] and Néron et al. [21].

Néron et al. [21] reported solving all instances of Vignier [29] very quickly and our computational experiments resulted the same. Hence we do not give the results as a table but they are available upon request.

We have next run our algorithms on the instances of Carlier and Néron [5]. As Néron et al. [21] reported improved results on these instances, we compare the performance of GA, CP and MA with that of the algorithm given in Néron et al. [21] ("NBG"). In their paper, the authors use energetic reasonning, satisfiability tests and time adjustments of the release dates and deadlines within two branch-and-bound approaches that differ by their branching scheme. The first one is based on the selection scheme of Carlier and Néron [5] whereas the second one is based on the chronological branching scheme from Baptiste et al. [1]. The latter approach,

Table 14 Comparing the average $C_{\max}$ values for Carlier and Néron [5]

k	n	Type	NBG	GA	CP	MA
5	10	a	111.60	111.60	111.60	111.60
		b	122.67	122.67	122.67	122.67
		c	71.00	71.50	71.00	71.00
		d	66.83	67.50	66.83	66.83
	15	a	161.83	161.83	161.83	161.83
		b	161.17	161.17	161.17	161.17
		c	87.83	87.00	86.50	86.00
		d	105.50	96.67	97.83	96.33
10	10	a	148.00	148.33	148.00	148.00
		b	163.00	163.00	163.00	163.00
		c	128.00	117.00	116.67	115.67
		d	_	112.67	112.33	111.83
	15	a	207.00	206.83	206.83	206.83
		b	211.83	211.83	211.83	211.83
		c	_	136.67	143.17	135.67

Table 15 Comparing the number of the best solution found for Carlier and Néron [5]

k	n	Type	NBG	GA	CP	MA
5	10	a	5	5	5	5
		b	6	6	6	6
		c	6	3	6	6
		d	6	3	6	6
	15	a	6	6	6	6
		b	6	6	6	6
		c	4	2	4	6
		d	1	4	5	6
10	10	a	6	5	6	6
		b	6	6	6	6
		c	1	1	4	6
		d	_	2	5	6
	15	a	5	6	6	6
		b	6	6	6	6
		c	_	1	0	6

which is more efficient, is also a constraint-based branch-and-bound algorithm. The results are presented in Tables 14 and 15, by comparing the average values of $C_{\max}$ and the number of the best solution found, respectively.

Similar conclusions can be drawn when we compare GA, CP and MA; memetic algorithm performs the best on the average compared to GA and CP. From Tables 14 and 15, we can draw two important conclusions:

- Even though CP and NBG are very similar methods, we observe that CP provides better results on some instances, while providing the same results for the rest. For two cases ($k = 10$, $n = 10$, type d, and $k = 10$, $n = 15$, type c), CP succeeds to find solutions, while NBG fails to do so.
- MA either produces the same results as NBG or it improves the results (25 instances over 89). We also observe that MA improves the results of CP in twelve of the instances given.

7 Conclusion

In this paper we proposed a memetic algorithm for the $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem and described its implementation. We had combined a genetic algorithm with a constraint programming based branch-and-bound algorithm to perform the local search on the population of solutions. Even though the proposed genetic algorithm draws heavily from the genetic algorithm of Oğuz and Ercan [24], it includes some differences to make the implementation of the algorithm more efficient. Our computational experiments showed that the quality of the results provided by these genetic algorithms are comparable.

We employed an extensive computational experiment (with several sets of instances from the literature) to test the performance of the three algorithms developed: the genetic algorithm alone (GA), the constraint programming based branch-and-bound algorithm alone (CP), and the memetic algorithm which is a combination of GA and CP (MA). The results showed that MA performs better than both GA and CP in terms of both the quality of the solutions produced and the efficiency.

Our results revealed that even though the performance of GA is quite close to that of MA in terms of the quality of the solutions produced, it is evident that MA is a better approach for the $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem when we consider the efficiency of the algorithms. From these, we can conclude that the memetic algorithm, which combines a genetic algorithm with a constraint programming based branch-and-bound algorithm, is an effective approach for the $Fk(Pm_1, \ldots, Pm_k)|size_{ij}|C_{\max}$ problem.

We then tested the performance of the proposed algorithms for the $Fk(Pm_1, \ldots, Pm_k)||C_{\max}$ problem to see whether they can be employed for other scheduling problems. In these experiments, we used different benchmark data sets from the literature. The results showed that the proposed algorithms are successful for the $Fk(Pm_1, \ldots, Pm_k)||C_{\max}$ problem as well. We particularly note that the memetic algorithm was able to improve some of the results from the literature and to provide solutions for those where no results were provided. This supports the robustness of the proposed memetic algorithm and we can conclude that it can be used for different flow-shop scheduling problems.

Acknowledgements The work described in this paper was partially supported by a grant from The Hong Kong Polytechnic University (Project No. G-T247) and by EGIDE, 28 rue de la Grange aux Belles, 75010 Paris, under the French-Hong-Kong joint programme "PAI Procore" n°05655UK-2004. The major part of this research was conducted when the second author was at the Hong Kong Polytechnic University, Department of Logistics, Hong Kong SAR.

References

1. Baptiste, Ph., Le Pape, C., Nuijten, W.: Satisfiability tests and time bound adjustments for cumulative scheduling problems. Ann. Oper. Res. **92**, 3305–3333 (1999)
2. Baptiste, Ph., Le Pape, C., Nuijten, W.: Constraint-Based Scheduling, Applying Constraint Programming to Scheduling Problems, vol. 39. International Series in Operations Research and Management Science. Kluwer, Deventer (2001)
3. Błażewicz, J., Ecker, K.H., Pesch, E., Schmidt, G., Węglarz, J.: Scheduling Computer and Manufacturing Processes, 2nd edn. Springer, Berlin (2001)
4. Carlier, C., Pinson, E.: A practical use of jackson's preemptive schedule for solving the job-shop problem. Ann. Oper. Res. **26**, 269–287 (1990)
5. Carlier, J., Néron, E.: An exact method for solving the multiprocessor flowshop. RAIRO-RO **34**, 1–25 (2000)
6. Chen, J., Lee, C.-Y.: General multiprocessor task scheduling. Nav. Res. Logist. **46**, 57–74 (1999)
7. Erschler, J., Lopez, P., Thuriot, C.: Raisonnement temporel sous contraintes de ressource et problèmes d'ordonnancement. Rev. Intell. Artif. **5**(3), 7–32 (1991)
8. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison Wesley, Redwood City (1989)
9. Goldberg, D.E., Deb, K.: A comparative analysis of selection schemes used in genetic algorithms. In: Rawlins, G.J.E. (ed.) Foundations of Genetic Algorithms, pp. 69–93. Morgan Kaufman, San Mateo (1991)
10. Gupta, J.N.D.: Two stage hybrid flowshop scheduling problem. J. Oper. Res. Soc. **39**(4), 359–364 (1988)
11. Holland, J.H.: Adaption in Natural and Artificial Systems. University of Michigan Press, Ann Arbor (1975)
12. Ilog: Ilog Scheduler Reference Manual. Ilog, Gentilly (2004)
13. Krawczyk, H., Kubale, M.: An approximation algorithm for diagnostic test scheduling in multicomputer systems. IEEE Trans. Comput. **34**, 869–872 (1985)
14. Le Pape, C.: Implementation of resource constraints in ILOG SCHEDULE: a library for the development of constraint-based scheduling systems. Intell. Syst. Eng. **3**(2), 55–66 (1994)

15. Lee, C.-Y., Cai, X.: Scheduling one and two-processor tasks on two parallel processors. IIE Trans. **31**, 445–455 (1999)
16. Lhomme, O.: Consistency techniques for numeric CSPs. In: Thirteenth International Joint Conference on Artificial Intelligence, Chambéry, August 1993
17. Lopez, P., Erschler, J., Esquirol, P.: Ordonnancement de tâches sous contraintes: une approche énergétique. RAIRO Autom. Prod. Inform. Ind. **26**(6), 453–481 (1992)
18. Moscato, P.: On evolution, search, optimization, genetic algorithms and martial arts: towards memetic algorithms. Technical Report C3P 826, Caltech Concurrent Computation Program, (1989)
19. Moscato, P.: Memetic algorithms: a short introduction. In: Corne, D., Dorigo, M., Glover, F. (eds.) New Ideas in Optimization, pp. 219–234. McGraw-Hill, New York (1999)
20. Moscato, P., Cotta, C.: A gentle introduction to memetic algorithms. In: Glover, F., Kochenberger, G.A. (eds.) Handbook of Metaheuristics, pp. 105–144. Kluwer, Deventer (2003)
21. Néron, E., Baptiste, Ph., Gupta, J.N.D.: Solving hybrid flow shop problem using energetic reasoning and global operations. Omega **29**, 501–511 (2001)
22. Nuijten, W.: Time and resource constrained scheduling: a constraint satisfaction approach. Ph.D. thesis, Eindhoven University of Technology (1994)
23. Nuijten, W., Aarts, E.H.L.: A computational study of constraint satisfaction for multiple capacitated job-shop scheduling. Eur. J. Oper. Res. **90**(2), 269–284 (1996)
24. Oğuz, C., Ercan, M.F.: A genetic algorithm for hybrid flow-shop scheduling with multiprocessor tasks. J. Sched. **8**(4), 323–351 (2005)
25. Oğuz, C., Ercan, M.F., Cheng, T.C.E., Fung, Y.-F.: Heuristic algorithms for multiprocessor task scheduling in a two-stage hybrid flow-shop. Eur. J. Oper. Res. **149**, 390–403 (2003)
26. Oğuz, C., Zinder, Y., Do, V.H., Janiak, A., Lichtenstein, M.: Hybrid flow-shop scheduling problems with multiprocessor task systems. Eur. J. Oper. Res. **152**, 115–131 (2004)
27. Portmann, M.-C., Vignier, A., Dardilhac, D., Dezalay, D.: Branch and bound crossed with GA to solve hybrid flow shops. Eur. J. Oper. Res. **107**, 389–400 (1998)
28. Serifoğlu, F.S., Ulusoy, G.: Multiprocessor task scheduling in multistage hybrid flow-shops: a genetic algorithm approach. J. Oper. Res. Soc. **55**(5), 504–512 (2004)
29. Vignier, A.: Contribution à la Résolution des Problèmes d'Ordonnancement de type Monogamme, Multimachines Flow-shop hybride. Ph.D. thesis, University of Tours (1997)
30. Vignier, A., Billaut, J.-C., Proust, C.: Hybrid flowshop scheduling problems: state of the art. Rairo-Rech. Oper.-Oper. Res. **33**, 117–183 (1999)