Hybrid Evolutionary Methods for the Solution of Complex Scheduling Problems

José Antonio Vázquez Rodríguez* and Abdellah Salhi

Mathematical Sciences Department, University of Essex
Wivenhoe Park, Colchester, U. K.
javazq@gmail.com, as@essex.ac.uk

Abstract. This paper is concerned with the minimisation of makespan and maximum lateness when scheduling Flexible Flow Shops (FFS). Even though, hybrid evolutionary methods have provided with competent solution tools for several hard combinatorial problems, their efficiency for FFS is not known. In this paper, the idea of hybridising Genetic Algorithm (GA) with the Shifting Bottleneck Procedure (SBP) and GA with a Local Search (LS) procedure, is explored. The proposed algorithms, named the Shifting Bottleneck Genetic Algorithm (SBGA) and Memetic Algorithm (MA); and the simple algorithms, GA and SBP, were compared on solving well known benchmarks and two large sets of randomly generated instances. The results revel that both hybrid methods are very successful.

1 Introduction

This paper is concerned with the minimisation of makespan and maximum lateness when scheduling shops with multiple stages and multiple machines per stage. We focus on the particular case in which all the jobs follow the same processing direction. This manufacturing environment is known as Flexible Flow Shop (FFS), [3]. Following the notation in [13], the problems of interest are denoted as: $FFc||C_{\max}$ and $FFc|r_j|L_{\max}$ for makespan and maximum lateness minimisation, respectively.

The FFS scheduling problem is intractable, even the two stages shop version is NP-Hard, [6]. It is still NP-hard if preemptions are allowed, [7]. We refer the reader to [16], [8], [10] and [15] for surveys on approaches to it, including exact methods, deterministic algorithms, heuristics and metaheuristics. The focus here is on metaheuristics and bottleneck exploiting heuristics.

The processing stage with the lowest capacity/work-load ratio, named bottleneck or critical stage, constraints the global performance of the system. Efficient heuristics such as the Shifting Bottleneck Procedure (SBP) exploit this knowledge by maximising the bottleneck utilisation. Examples of successful SBP adaptations to the FFS are described in [19] and [4]. Metaheuristics, on the other

* Corresponding author

© A. Gelbukh, S. Torres, I. López (Eds.)
Advances in Artificial Intelligence
Research in Computing Science 20, 2006, pp. 17-28

hand, are the basis of several state of the art methods for a wide range of combinatorial problems, these include the FFS. For instance in [9], [2] and [12], Genetic Algorithms (GA) were designed to efficiently solve different variants of the problem. In [17] and [11], Tabu Search (TS) was utilised. The combination of heuristics, specially a global search method, such as an evolutionary algorithm, with a problem tailored heuristic, is a good strategy to generate even stronger solution approaches. Memetic Algorithms (MA), which combine GA with an improving procedure, are a good example of them. However, our literature review revels that this idea has not been studied in the case of the FFS problem.

This paper introduces two new algorithms based in the idea of hybridising GA with two specialised methods. The first algorithm uses the SBP to support the GA and was named the Shifting Bottleneck Genetic Algorithm (SBGA). The second algorithm uses a local search method to improve the solution provided by GA. We refer to this method as Memetic Algorithm (MA).

In order to test the "synergy" produced by the combinations of methods in both hybrids, SBGA, MA, GA and SBP were used to solve well known instances of the problem, as well as a large set of randomly generated problems. The reported results show that both of the new methods are effective solution tools for the problem.

The rest of the paper is organised as follows. Section 2 is the formal description of the problem; assumptions notation and a model are presented. Section 3 describes the SBGA. Section 4 presents GA and the MA. Section 5 describes the numerical experiments, including the problems and their origins, the results and a discussion. Section 6 concludes the paper.

2 Problem Description

It is assumed that n jobs are to be processed through m different stages. Each stage has at least one machine, with one or more having at least two parallel identical machines. Any machine can process at most one job at a time and any job is processed on at most one machine at a time. Furthermore, every job is processed on at most one machine in any stage. Preemptions are not allowed, i.e. once the processing of a job has started on a given machine, it can not be stopped until it is finished.

2.1 Notation

- j = job index; k = stage index; l = machine index;
- n = number of jobs; m = number of stages; m_k = number of machines at stage k;
- o_{jk} = operation of job j to be processed on k; $O_k = \cup_j o_{jk}$;
- p_{jk} = processing time of o_{jk}; r_{jk} = release time of o_{jk}; d_{jk} = due date of o_{jk};
- C_j = completion time of job j; $L_j = \max\{C_j - d_{jm}, 0\}$, lateness of job j.

2.2 Problem Formulation

Let A^{kl} be a set of operations $o_{jk} \in O_k$ assigned for processing to machine l at stage k. Let S^{kl} be a permutation of the elements in A^{kl} representing the order in which operations must be processed. Let $S^k = \cup_{l=1}^{m_k} S^{kl}$ and $S = \cup_{k=1}^{m_k} S^k$. Given that the functions approached are regular, our interest is limited to non delay schedules. Because of this, S, being the set of sequences of jobs in all machines, represent a unique schedule. For S, to be feasible, the following must hold: (1) $\cup_{l=1}^{m_k} A^{kl} = O_k \ \forall k$ and (2) $\cap_{l=1}^{m_k} A^{kl} = \emptyset \ \forall k$. These constraints guarantee that all operations to be processed in k are assigned for processing strictly once. Let ψ be a FFS problem instance and Ω^ψ the set of all feasible schedules for ψ. The problem is to find a $S \in \Omega^\psi$ such that its makespan $C_{\max}$,

$$\min_{S \in \Omega^\psi} \max_j C_j(S) \tag{1}$$

or its maximum lateness $L_{\max}$,

$$\min_{S \in \Omega^\psi} \max_j L_j(S) \tag{2}$$

is minimum.

3 The Shifting Bottleneck Genetic Algorithm

The SBP was introduced in [1], and so far, is one of the best established algorithms to approach multiple-stage shops. SBP breaks the overall problem into sub-problems and solves one at a time. In [4], an efficient adaptation of the original SBP is presented (for the rest of the paper SBP refers to this version). SBP uses the reversibility principle of the parallel machines problem (see Section 3.2) to obtain good solutions for it efficiently. The proposed approach, uses GA to prioritise the jobs in the critical stage of the shop. The rest of the stages are scheduled in the order dictated by their criticality with the heuristic employed by SBP.

3.1 Critical Stage

As mentioned, SBGA schedules the stages according to their criticality. The criticality of stage k is the minimum time required so that all the jobs with an operation in k pass through the shop. The one that requires the highest time, restricts the total capacity of the shop and is the critical stage or bottleneck. The criticality of stage k, CS_k, is calculated as follows,

$$CS_k = \frac{1}{m_k}\left(\sum_{y=1}^{m_k} LSA_y^{k-1} + \sum_{j=1}^{n} p_{jk} + \sum_{y=1}^{m_k} RSA_y^k\right) \tag{3}$$

where $LSA_y^k = \{LSA_1^k, LSA_2^k, ..., LSA_n^k\}$ are the sums $\sum_{b=1}^{k} p_{jb}$, $\forall j$ in the ascending order and $RSA_y^k = \{RSA_1^k, RSA_2^k, ..., RSA_n^k\}$ the sums $\sum_{b=k+1}^{m} p_{jb}$, $\forall j$

in the ascending order [4]. In the brackets, the first term is the minimum time elapsed before k becomes fully active, the second is the work load of k (which divided by m_k provides the minimum processing time in k) finally, the third term is the minimum time required by the last jobs processed in k to exit the shop. The critical stage k is the one with the largest CS_k value.

3.2 Heuristic Approach for the Parallel Machines Scheduling Problem

SBP and SBGA decompose the original problem into stages and schedule one at a time fixing the rest. A sub-problem for stage k, can be handled as a parallel machines problem with release dates r_{jk} and due dates d_{jk} ($Pm|r_{jk}, d_{jk}|L_{\max}$, see notation in [13]) and a solution to it can be obtained as follows.

Algorithm *procedureA*()

1. **while** $O_k \neq \emptyset$ **do**
   a. set t as the maximum between the time that the first machine in k becomes idle and the minimum release time of the operations in O_k;
   b. select the operation $o'_{jk} \in O_k : r_{jk} \leq t$ with the smallest due date (d_{jk}), break ties by preferring longer p_{jk};
   c. assign o'_{jk} to the first idle machine;
   d. $O_k = O_k \backslash o'_{jk}$.

Given a $Pm|r_{jk}, d_{jk}|L_{\max}$ problem, its inverse $P'm|r'_{jk}, d'_{jk}|L_{\max}$, is obtained using the negatives of the due dates and release dates of Pm as release and due dates, respectively, in $P'm$, i.e. $r'_{jk} = -d_{jk}$ and $d'_{jk} = -r_{jk}$. Let $\pi = (\pi(1), \pi(2), ..., \pi(|O_k|))$ be a sequence of tasks in a given machine, then $\pi' = (\pi(|O_k|), \pi(|O_k| - 1), ..., \pi(1))$ is the reverse of π. The algorithm presented in [4], for the $Pm|r_j, d_j|L_{\max}$ problem is as follows.

Algorithm $Pm()$

1. obtain $P'm$;
2. solve Pm and $P'm$ using *procedureA*(), reverse the solution obtained for $P'm$, calculate $L_{\max}$ for both solutions.
3. Return the schedule with the minimum maximum lateness.

Note that *procedureA* is not an exact algorithm, therefore, the solutions provided by it, when applied to the original problem and its inverse, may be different. We are interested in the best of them.

3.3 Representation and Evaluation of Solutions

A classical permutation representation, as explained in [5], was adopted. In this, every individual is a permutation $\pi = (\pi(1), \pi(2), ..., \pi(|O_k|))$ where $\pi(i)$ is a job index. π represents the sequence to prioritise the operations at the critical stage k'. In k', operations are assigned in the order dictated by π to the machine

that allows them the fastest completion time. Let K be the set of stage indexes (k) sorted in decreasing order of their CS_k values (see Section 3.1), in such a way that K_1 points to the bottleneck of the shop. The procedure to evaluate an individual is as follows.

Algorithm $CP(\pi, K)$

1. set $S = \emptyset$ (an initial empty schedule).
2. generate S^{K_1} by assigning each o_{jK_1} in the order dictated by π to the machine l in stage K_1 that allows it the fastest completion time, $S = S \cup S^{K_1}$;
3. update release times and due dates as described in Section 3.4;
4. **for** $i = 2 : m$ **do**
   a. generate S^{K_i} using $Pm()$, $S = S \cup S^{K_i}$;
   b. update release times and tails as described in Section 3.4;
5. **return** $\max_j C_j(S)$ and S.

Note that at steps 2 and 4a, S is updated with the information of sub-schedule S^{K_i} obtained for stage K_i.

3.4 Updating Release Times and Due Dates

At the initialisation, the release times of operations to be processed at stages posterior to the first one are calculated as follows $r_{jk} = r_{j1} + \sum_{b=1}^{k-1} p_{jb}$, $k > 1$; and the due dates of previous stages to the last one, as follows $d_{jk} = -d_{j1} + \sum_{b=m}^{k+1} p_{jb}$, $k < m$. The release times at the first stage and due dates at the last one are given as part of the problem, otherwise they are set to 0.

Let us suppose that stage $\hat{k}$ is the one just scheduled at step 2 or 4a of the $CP()$ procedure. The release times must be updated for the operations in every stage $k \geq \hat{k}$ and the due dates for the operations in every stage $k \leq \hat{k}$. Remember that S^{kl} is the sequence of jobs assigned to machine l in k (see Section 2.2), let $q = |S^{kl}|$. The release times are updated as follows.

Algorithm $URT()$
for $k = \hat{k} : m$ **do**
   a. **if** $S^k = \emptyset$ (if stage k has not been scheduled) **do**
      i. $r_{jk} = r_{jk-1} + p_{jk-1}, \forall j$;
      ii. **end**
   b. **for** $l = 1 : m_k$ **do**
      i. $r_{S_1^{kl}k-1} = r_{S_1^{kl}k-1} + p_{S_1^{kl}k-1}$;
      ii. **for** $h = 2 : q$ **do**
         A. $r_{S_h^{kl}k} = \max\left\{r_{S_h^{kl}k-1} + p_{S_h^{kl}k-1}, r_{S_{h-1}^{kl}k} + p_{S_{h-1}^{kl}k}\right\}$.

The due dates are updated with the following procedure.

Algorithm $UDD()$
for $k = \hat{k} : 1$ **do**

a. **if** $S^k = \emptyset$ (if stage k has not been scheduled) **do**
  i. $d_{jk} = d_{jk+1} + p_{jk+1}, \forall j$;
  ii. **end**
b. **for** $l = 1 : m_k$ **do**
  i. $d_{S_q^{kl}k} = d_{S_q^{kl}k+1} + p_{S_q^{kl}k+1}$;
  ii. **for** $h = q - 1 : 1 : -1$ **do**
    A. $d_{S_h^{kl}k} = \max\left\{d_{S_h^{kl}k+1} + p_{S_h^{kl}k+1}, d_{S_{h+1}^{kl}} + p_{S_{h+1}^{kl}k}\right\}$.

3.5 Genetic Operators

SBGA maintains a population *Pop* of N individuals. At every generation, a new population *Pop'* is generated sampling in the search space by means of the genetic operators and using the information of the best individuals in *Pop*. Each new individual is created using one of the GA operators: crossover, mutation or direct reproduction. The order crossover (OX) was chosen as recombination method. It showed a better performance in a pre-experimental stage when compared with other crossover methods such as edge, cycle and partially mapped crossover (see [5]). In OX, two individuals (parents) and two crossing points are randomly chosen. The elements of the first parent that are in between the two crossing points are copied in the same positions to the new one. The rest are copied in the same order than in the second parent following a toroidal path (see [5] for more details). The mutation employed is the random generation of a new individual. Let *rep* be N minus the number of individuals not created through crossover or mutation. In order to provide elitism, the *rep* best fitted individuals in *Pop* are copied to complete *Pop'*.

The selection method of individuals, to participate for crossover or mutation, is binary tournament selection. In this, 2 individuals are selected randomly from the population, they compete among them, and the fittest participates in the crossover process, [5].

3.6 SBGA General Framework

At the initialisation N individuals are generated randomly and evaluated using the $CP()$ procedure. The criticality CS_k of every stage k is calculated using formula 3 and the ordered set K' is generated (see Section 3.3). The release times r_{jk} and due dates d_{jk} are calculated as described in Section 3.4.

The general framework of SBGA is as follows.

Algorithm *SBGA*()

1. calculate the ordered set K' (see Section3.3), initialise release times and due dates (see Section 3.4).
2. generate a set *Pop* of N random permutations, $CP_{i=1}^{N}(Pop_i, K')$, i.e. evaluate each solution using the procedure described in Section 3.3, let π^* be the best solution in *Pop*;
3. generate a set *Pop'* of N new solutions by applying the GA operators (Section 3.5), $CP_{i=1}^{N}(Pop'_i, K')$;

4. let Pop be the best N solutions in $Pop' \bigcup \pi^*$, let π^* be the best solution found so far;
5. **if** stopping condition not met **go to** 3;
6. **return** π^*.

4 Alternative Methods

Given that SBGA is a hybrid algorithm, it is important to know if its SBP component provides any added value to the final algorithm. In order to explore this, a GA named Multiple Stages Representation GA (MSRGA), whose individuals represent full schedules, was encoded. MSRGA uses the same genetic operators as SBGA, but it does not use any specialised information. Since SBP has been proved to be an effective scheduling tool, it is expected that it will provide good information to SBGA. However, to evaluate how good this information is, it will be compared with the one provided by a Local Search (LS) method based on an efficient, already tested, neighbourhood function, [11]. A third algorithm, which uses LS to improve the performance of MSRGA, was designed. Given that these sort of hybrid methods are usually referred to as Memetic Algorithms, this third algorithm was named MA. Finally, in order to know to what extent the addition of a stochastic method to SBP, is beneficial, SBP as described in [4], was implemented and tested.

The rest of this section describes the implementation details of MSRGA and MA. We refer the reader to [4], for details on SBP.

4.1 Multiple Stages Representation GA

Representation and Evaluation of Individuals Every individual in MSRGA is a set $\Pi = \{\pi_1, \pi_2, ..., \pi_m\}$ of permutations, one for every stage of the shop, representing the assigning order of jobs at every stage. To evaluate an individual, operations at stage k are scheduled in the order dictated by π_k in the machine l in k that allows them the fastest completion time. This procedure is as follows.

Algorithm $CP_{MSRGA}(\Pi)$

1. set $S = \emptyset$ (an initial empty schedule);
2. **for** $k = 1 : m$ **do**
   a. generate S^k by assigning the operations in the order dictated by π_i to the machine that allows them the fastest completion time;
   b. $S = S \cup S^k$, update release times (see Section 3.4);
3. **return** $\max_j C_j(S)$.

Note that stages are scheduled in the order $1, 2, ..., m$, because of this, the release times need to be updated just for the stage being scheduled (k) and, if $k < m$, for stage $k + 1$ too. The due dates are not updated.

GA General Framework At initialisation, N individuals are generated randomly and evaluated using the $CP_{MSRGA}(\Pi)$ procedure. At every iteration, a new population Pop' is generated using the genetic operators as described in Section 3.5. But in this case, let A and B be the set of m permutations of parents A and B chosen to be recombined through crossover. Every permutation c_k of the new individual C, is the result of applying OX to the permutation $a_k \in A$ and $b_k \in B$.

The general framework of MSRGA is as described in Section 3.6 for the SBGA, but at step 1, there is no need to calculate K' neither the due dates. In steps 2 and 3 $CP(\pi, K')$ is substituted by $CP_{MSRGA}(\Pi)$.

4.2 Memetic Algorithm

In MA, the representation of individuals and the genetic operators are as described for MSRGA. However, in MA the procedure to evaluate individuals requires the use of a Local Search (LS) procedure. We refer the reader to [11] for a description and theoretical basis of the neighbourhood function on which LS is based.

To evaluate an individual, this is decoded by means of $CP_{MSRGA}()$, LS is then used to improve it. In algorithmic form, the evaluation of individuals in MA is as follows.

Algorithm $CP_{MA}(\Pi)$
1. $S = CP_{MSRGA}(\Pi)$, i.e. generate initial schedule;
2. $S' = LS(S)$, i.e. improve S using LS;
3. return $\max_j C_j(S')$.

The general framework of MA is as the one described for MSRGA, but substituting $CP_{MSRGA}(\Pi)$ with $CP_{MA}(\Pi)$.

5 Computational Experience

In order to evaluate the described methods, they were run on three different problem sets. The first one is, perhaps, the best known test-set for the FFS problem, [18]. The second and third are random instances generated in a similar fashion as in [19], [4] and [9].

Being SBP a deterministic heuristic, it does not require input parameters and it was run once on every instance. For the rest of the algorithms, the following parameter settings showed being appropriate.

- population size: SBGA 100, MSRGA 100, MA 30
- crossover rate: 95% for the three algorithms
- mutation rate: 1% for the three algorithms
- stopping condition: MSRGA was run on every problem until it did not show improvement for 100 consecutive generations or it reached a maximum processing time of 60 seconds. This time was recorded and set as the stopping condition for MA and SBGA.

Table 1. performance on IBM Wittrock's instances

Instance	SBP	MSRGA	MA	SBGA
1	761	820	**760**	**760**
2	769	793	763	**755**
3	761	784	767	**759**
4	781	787	772	**761**
5	**961**	**961**	**961**	**961**
6	667	674	665	**659**

SBGA, MSRGA and MA were run 5 times on every instance, the best found solution was reported. The four algorithms were encoded in Java S.E. 5.0. All the experiments were executed on identical PC's (Pentium IV, 3.0GH, 1Gb RAM) running Windows XP.

5.1 Results on IBM Wittrock's Instances

Here we report the performance of the 4 algorithms on Wittrock's test-bed, [18], from an IBM production line. This line, inserts components into printed circuit cards. Every card is transported in a magazine that holds 100 identical cards. The magazines have to go through three different types of machines: two "DIP inserters", three "SIP inserters" and three robots. The required time for each magazine in every stage depends on the type of card that it holds. The problem instances are the production needs for 6 days, for which, a daily schedule with minimum makespan, is required. Any transportation times between stages were neglected. Table 1 presents the results.

In 5 out of 6 instances SBGA outperformed SBP and GA and it was better than MA in 4 out of 6. MA, on the other hand, was superior to MSRGA in 5 out of 6 instances and to SBP in 4 out of 6. These results suggest that SBP and LS provide useful information to GA, being the one by SBP superior. However, conclusions can not be obtained from such a small sample. Next, the results in two large sets of randomly generated instances are reported.

5.2 Randomly Generated Instances

A set of 1080 instances for makespan and 1296 for maximum lateness were generated in a similar fashion as in [9] and [4]. The makespan and maximum lateness obtained by the algorithms, on each of these, were compared with the lower bound (LB) described in [14]. Since our problems are minimisation ones, when a solution reaches a makespan or lateness value equal to the one provided by LB, an optimal solution has been found. Table 2 displays the success rate of each algorithm on both objectives.

The most successful method is SBGA, followed by MA, SBP and finally MSRGA. Enough evidence has been collected to conclude that a generic method such as GA is not competitive. Moreover it is evident that SBP and GA work quite well together, the success rate of SBGA is around twice the one of SBP

Table 2. success rate on randomly generated instances

Opt. criterion	MSRGA	MA	SBP	SBGA
makespan	11.42%	32.72%	24.26%	**54.91%**
lateness	9.26%	46.38%	38.04%	**60.57%**

and around 5 times higher than that of MSRGA. MA, on the other hand, is also more competitive than MSRGA and SBP, but not as SBGA. Next, the mean deviations from the lower bounds obtained by the algorithms are presented. The results are reported for subsets of instances dictated by their characteristics, this will provide an idea of the effects of the instance characteristics on the algorithms performance.

The deviation on $C_{\max}$, DC_{ai} max, of algorithm $a \in \{MSRGA, SBGA, ...\}$ on instance i with respect to the lower bound value LB_i, is calculated as:

$$DC_{ai}\,\max = \frac{C\max_{ai} - LB_i}{LB_i} * 100. \tag{4}$$

Given a set I of problem instances of interest, the deviation with respect to the lower bound for $L_{\max}$ is calculated as:

$$DL_{aI}\,\max = \frac{\sum_{i \in I}(L\max_{ai} - LB_i)}{\sum_{i \in I} LB_i} * 100 \tag{5}$$

where DL_{aI} max is the maximum lateness deviations from LB, obtained by algorithm a on a set of instances I.

Table 3 presents the mean DC_{ai} max and DL_{aI} max values of each algorithm on the sets of instances described on the first two columns.

The best performing algorithm, on the whole and on the different instance subsets presented in Table 3, is SBGA. It can be concluded from this, that GA and SBP collaborate positively. On the other hand, MA also reported better results than SBP and GA which shows the benefits of adding LS to MSRGA. The results on Table 3, let no place for discussion the need of using specialised information to approach FFS problems. However, it was necessary its testing to safely conclude that SBP and LS enhance the performance of GA. Moreover, the quality of information provided by SBP is comparable and superior, at least on the testbed presented, to that of a competitive local search method.

Regarding the computational times, SBP is far faster than any of the three other methods. Its required time never exceeded 5 seconds even for the largest instances. On the other hand the rest of the methods run for 1 minute in most of the instances with 6 and 7 stages and 100 jobs. For the rest of the problems this time limit was rarely reached. On the practice 1 minute of computational cost, to schedule a 700 operations shop, is reasonable. For an acceptable extra cost, GA can enhance the performance of SBP from an average deviation of 5.86% to 1.38% and from 3.15% to 0.83% for makespan and maximum lateness, respectively. This could represent important savings on real world shops.

Table 3. deviation from lower bound on randomly generated instances

parameter	value	$\overline{DC_{ai}}$ max				$\overline{DL_{aI}}$ max			
		MSRGA	MA	SBP	SBGA	MSRGA	MA	SBP	SBGA
n									
	10	14.36	4.97	8.99	2.68	112.16	9.56	18.32	5.72
	20	22.38	3.26	7.15	1.58	98.56	4.23	6.05	1.10
	50	88.46	1.65	4.06	0.73	79.87	1.32	1.99	0.39
	100	127.88	1.84	3.21	0.54	65.23	0.79	1.29	0.35
m									
	[2, 3]	36.45	2.35	4.79	1.29	39.65	0.79	1.35	0.50
	[4, 5]	61.35	2.59	5.75	1.23	48.11	1.12	1.72	0.37
	[6, 7]	92.01	3.85	7.03	1.63	59.36	1.37	2.11	0.40
m_k									
	[1, 2]	70.36	2.87	6.83	0.84	38.37	0.68	2.75	0.31
	[1, 4]	54.68	2.66	5.42	1.14	59.84	1.09	1.99	0.39
	[1, 8]	64.77	3.26	5.31	2.17	77.59	0.98	1.90	1.18
p_{jk}									
	[10, 20]	40.02	2.37	3.87	1.13	65.23	0.69	1.13	0.40
	[10, 100]	86.52	3.49	7.83	1.64	89.26	0.78	2.60	0.59
total									
		63.27	2.93	5.86	1.38	78.34	2.67	3.15	0.83

6 Conclusion

A new hybrid GA and SBP for the solution of FFS scheduling problems has been presented. This algorithm was compared versus a Multiple Stage Representation GA, a Memetic Algorithm and the Shifting Bottleneck Procedure on the solution of well known benchmarks and two large randomly generated test sets.

The proposed algorithm outperformed its competitors in all tests. It obtained deviations of 1.38% and 0.83% from lower bounds for makespan and lateness minimisation respectively and success rates of 54.9% and 60.57% for the same problems. Given that SBGA outperformed both, SBP and GA, it is reasonable to conclude that there is an important collaboration between the GA component and the SBP one. In other words, the relation GA and SBP, as in SBGA, seems to be highly "synergic". On the other hand, it is also clear that the information provided by SBP to GA is of a higher quality than that of a well established local search method.

To implement SBGA does not represent more challenge than SBP, moreover, the former improved the results of the latter by a 4% and 2%, on average, for makespan and maximum lateness minimisation, respectively. Because of this, and the fact that the CPU cost required by SBGA is reasonable, SBGA is a sensible choice in the practice.

Despite the extent of this investigation, it remains to test SBGA on problems with other optimisation criteria; and investigate its performance on other problems such as job, open and assembly shops.

7 Acknowledgements

It is a pleasure to acknowledge the support from **CONACyT** through grant 178473.

References

1. J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34:391–401, 1988.
2. S. Bertel and J. C. Billaut. A genetic algorithm for an industrial multiprocessor flow shop scheduling problem with recirculation. *European Journal of Operational Research*, 159:651–662, 2004.
3. S. A. Brah and J. L. Hunsucker. Branch and bound algorithm for the flow shop with multiple processors. *European Journal or Operational Research*, 51:88–99, 1991.
4. J. Cheng, Y. Karuno, and H. Kise. A shiting bottleneck approach for a parallel-machine flow shop scheduling problem. *Journal of the Operations Research Society of Japan*, 44:140–156, 2001.
5. A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer, 2003.
6. J. N. D. Gupta. Two-stage hybrid flow shop scheduling problem. *Operational Research Society*, 39:359–364, 1988.
7. J. A. Hoogeveen, J. K. Lenstra, and B. Veltman. Preemptive scheduling in a two-stage multiprocessor flow shop is NP-hard. *European Journal of Operational Research*, 89:172–175, 1996.
8. M. E. Kurz and R. G. Askin. Comparing scheduling rules for flexible flow lines. *International Journal of Production Economics*, 85:371–388, 2003.
9. M. E. Kurz and R. G. Askin. Scheduling flexible flow lines with sequence dependent set-up times. *European Journal of Operational Research*, 159:66–82, 2003.
10. R. Linn and W. Zhang. Hybrid flow shop scheduling: A survey. *Computers & Industrial Engineering*, 37:57–61, 1999.
11. E. Nowicki and C. Smutnicki. The flow shop with parallel machines: A tabu search approach. *European Journal of Operational Research*, 106:226–253, 1998.
12. C. Oguz and M. F. Ercan. A genetic algorithm for hybrid flow shop scheduling with multiprocessor tasks. *Journal of Scheduling*, 8:323–351, 2005.
13. M. Pinedo. *Scheduling Theory, Algorithms and Systems*. Prentice Hall, 2002.
14. D. L. Santos, J. L. Hunsucker, and D. E. Deal. Global lower bounds for flow shops with multiple processors. *European Journal of Operational Research*, 80:112–120, 1995.
15. A. Vignier, J. C. Billaut, and C. Proust. Les problèmes d'ordonnancement de type flow-shop hybride: état de l'art. *Operations Research*, 33:117–183, 1999.
16. H. Wang. Flexible flow shop scheduling: Optimum, heuristics and artifical intelligence solutions. *Expert Systems*, 22:78–85, 2005.
17. B. Wardono and Y. Fathi. A tabu search algorithm for the multi-stage parallel machines problem with limited buffer capacities. *European Journal of Operational Research*, 155:380–401, 2004.
18. R. J. Wittrock. An adaptable scheduling algorithm for flexible flow lines. *Operations Research*, 36:445–453, 1988.
19. Y. Yang. *Optimization and Heuristic Algorithms for Flexible Flow Shop Scheduling*. PhD thesis, Columbia University, 1998.