Hybrid Particle Swarm Optimizers in the Single Machine Scheduling Problem: An Experimental Study

Leticia Cagnina[1], Susana Esquivel[1] and Carlos A. Coello Coello[2]

[1] Lab. de Investigación y Desarrollo en Inteligencia Computacional (LIDIC)**
Universidad Nacional de San Luis
Ejército de los Andes 950 - 5700 - San Luis - Argentina
[2] CINVESTAV-IPN (Evolutionary Computation Group)
Departamento de Ingeniería Eléctrica, Sección Computación
Av. IPN No. 2508, Col. San Pedro Zacatenco, México D.F. 07360, México

Summary. Although Particle Swarm Optimizers (PSO) have been successfully used in a wide variety of continuous optimization problems, their use has not been as widespread in discrete optimization problems, particularly when adopting non-binary encodings. In this chapter, we discuss three PSO variants (which are applied on a specific scheduling problem: the Single Machine Total Weighted Tardiness): a Hybrid PSO ($HPSO$), a Hybrid PSO with a simple neighborhood topology ($HPSO_{neigh}$) and a new version that adds problem-specific knowledge to $HPSO_{neigh}$ ($HPSO_{kn}$). The last approach is used to guide the blind search that PSO usually does and reduces its computational cost (measured in terms of the objective function evaluations performed). It is also shown that $HPSO_{kn}$ obtains good results with a lower computational cost, when comparing it against the other PSO versions analyzed, and with respect to a classical PSO approach and to a multirecombined evolutionary algorithm (*MCMP-SRI-IN*), which contains specialized operators to tackle single machine total weighted tardiness problems.

1 Introduction

Particle Swarm Optimization (PSO) is a bio-inspired heuristic that was proposed by James Kennedy and Russell Eberhart [16]. PSO is a population-based stochastic heuristic that simulates the flight of a flock of birds. In PSO, each particle in the swarm (i.e., the population) is a possible solution within the multidimensional search space. Such a particle has some properties such as a position (within the search space), a velocity of exploration which is constantly updated, and a record of its past behavior. Each particle evaluates

** The LIDIC is supported by the Universidad Nacional de San Luis and the ANPCyT (National Agency to Promote Science and Technology).

its relative position with respect to a goal (fitness) at every iteration and it adjusts its own velocity using the best position that it has found so far and the best position reached by any particle in its neighborhood (or in the swarm, if no neighborhood topology is adopted). Then, the velocity is used to update the position of each particle. The update is done using the following equations:

$$vel_{ij} = w * vel_{ij} + c_1 * r_1 * (p_{ij} - part_{ij}) + c_2 * r_2 * (p_{gj} - part_{ij}) \quad (1)$$

$$part_{ij} = part_{ij} + vel_{ij} \quad (2)$$

where vel_{ij} is the velocity of the particle i in the dimension j, w is the inertia factor [15] whose goal is to balance global exploration and local exploitation, c_1 and c_2 are the personal and social learning factors, r_1 and r_2 are two random numbers in the range (0,1), p_{ij} is the best position reached by the particle i and p_{gj} is the best position reached by any particle in the neighborhood (or swarm).

PSO was originally designed to work in continuos search spaces, and the specialized literature reports a significant amount of research that makes evident the great search capabilities of PSO in such type of search spaces. However, the use of *PSO* in discrete search spaces is relatively scarce, particularly when non-binary encodings (e.g., permutations) are adopted (see for example [24, 31, 12, 22]).

The authors recently proposed a hybrid PSO approach, which was called *HPSO* [8]. *HPSO* incorporates a *random keys* representation [4] for the particles and a dynamic mutation operator similar to the one used in evolutionary algorithms. The use of the *random keys* encoding allows to represent permutations using real numbers. This, in turn, allows us to use *PSO* with real numbers instead of having to rely on more complex encodings to represent a permutation of integers. In further work by the authors, $HPSO_{neigh}$ was introduced [7]. This approach adds to *HPSO* a local neighborhood (known as *circle topology* [25]) to each particle.

In this chapter, we propose a new *PSO* variant, which we call $HPSO_{kn}$. This algorithm is an extended version of $HPSO_{neigh}$, which incorporates problem-specific knowledge to guide the search.

The three previously indicated *PSO* approaches are used to solve a hard combinatorial optimization problem called *Total Weighted Tardiness Scheduling* (TWT). To the authors' best knowledge, this chapter constitutes only the third reported attempt to use PSO in scheduling (the two other attempts are reported in [38] and [8]).

The main goal of this chapter is to show the performance of our proposed $HPSO_{kn}$ using some instances of the TWT problem in single machine environments. We also aim to compare the results produced by the new algorithm against those obtained with the classical *PSO*, the *HPSO*, the $HPSO_{neigh}$, and a multirecombined evolutionary algorithm (*MCMP-SRI-IN*) [14] that was

specially designed for dealing with the problem of our interest. Such an approach also adopts the knowlegde insertion concept (adopted in this chapter) that consists of incorporating in the population three seeds generated with other traditional heuristics.

The remainder of the chapter is organized as follows. In Section 2, the scheduling problem of our interest is properly defined. In Section 3, we briefly review the previous related work. Section 4 describes the PSO algorithms adopted for our experimental study, including our new proposed approach. Section 5 contains a description of our experimental design, including the parameters settings adopted. Our results are shown and discussed in Section 6. Finally, our conclusions and some possible paths for future research are provided in Section 7.

2 Single Machine Total Weighted Tardiness Problem

The single machine scheduling model is the simplest of all possible machine environments and it is a special case of more complicated machine environments. This model was selected because the results obtained for it provide the basis to develop heuristics for more complex machine environments. In this work only the deterministic model is analyzed.

The term *machine* is used to specify any resource that will process an assignment. In the single machine system just one resource is available; thus, only one job can be processed by the machine at any time. Each job or task consists of one or more operations (sub-tasks).

The objective function or criterion selected to evaluate the quality of the schedule was the Total Weighted Tardiness (TWT) because it is important in a wide range of production activities. In this problem, the jobs or assignments that have to be processed are characterized by several elements:

- *Processing time (**p**)*, the amount of time the job needs the resource to complete its task. It includes a setup and a knock-down time;
- *Weight (**w**)*, a value indicating the importance of the job with respect to the other jobs in the system. It represents a factor of priority, that is, what job should be chosen (among all the available jobs) to be processed next;
- *Due date (**d**)*, in which the job should finish and free the resource. It denotes the date the job is promised to be delivered to the customer.

Assuming the deterministic model and that the system consists of a set of n jobs ($j = 1, \ldots, n$) to be processed without preemption in a single machine, each job j has its own p_j (processing time), w_j (weight) and d_j (due date). For a given processing order of all jobs, the earliest *completion time* C_j can be defined like the time the job j uses from the moment in which it enters the system and until it leaves the system. Also, for each job j the *tardiness* T_j is defined like the maximum value among zero and the completion time minus the due date: $T_j = max\{0, C_j - d_j\}$. Then, the TWT problem consists

of finding an appropriate processing order of the jobs with the purpose of minimizing the number of weighted tardy jobs, that is, to minimize the *Total Weighted Tardiness*:

$$\sum_{j=1}^{n} w_j T_j$$

over the n jobs in the system.

3 Previous Related Work

The single machine total weighted tardiness problem is an NP-hard [27] scheduling problem. The TWT problem has been tackled by a number of exact methods such as Branch and Bound [37, 21, 33], where some schedules are discarded because they exceed the objective function value set as a bound. A competitive technique in this context is dynamic programming [37, 23], which constructs all possible sets of jobs and recursively obtains a solution. The problem with these two approaches (branch & bound and dynamic programming) is the exponential growth and the considerable computer resources (computational time and memory requirements) that they require as the size of the problem grows.

Several enumerative methods have also been proposed, such as those that use dominance rules to restrict the search for the optimal solution [23] and those that characterize adjacent jobs in the optimal sequence [36]. An experimental study of these methods might be found in [37].

Some schedule construction heuristics have also been proposed to tackle this problem. These heuristics generate good, but not necessarily optimal solutions. For example, some authors have proposed dispatching rules to build a solution by fixing a job in a position in the sequence at each step of the process. There are a lot of rules widely used for the TWT problem. Comparisons between *weighted shorted processing time* (WSPT), *earliest due date* (EDD), *modified cost over time* (MCOVERT) and *apparent urgency* (AU) might be found in [34]. Additionally, an experimental study of this sort of heuristic may be found in [2]. The apparent tardiness cost (ATC) was proposed and tested in [40]. Then, in [10], the same rule was tested with other dispatching rules in job and flow shops, showing its effectiveness in minimizing the average tardiness.

A dominance rule for the most general case of total weighted tardiness problem is presented in [1], showing the sufficient condition for local optimality and how it generates schedules that cannot be improved by adjacent job interchanges.

There are other useful methods, such as the method of interchanges. Such interchanges require an initial sequence over which the change will take place. If the changed solution is better than the non-changed one, the method keeps it; otherwise, the changed solution is discarded. When the solution cannot be

improved, the interchanges stop and the process returns the sequence solution. Comparisons among several heuristics (including interchanges) might be found in [34]. Some results indicate that the pairwise interchange methods are very good for this problem.

There exist several local search algorithms that propose to solve the TWT problem using insertion and swap movements to find a good schedule. These heuristics compute the neighborhood of a solution through movements of jobs in the sequence. For example, an exponentially sized "dynasearch" that swaps positions within the neighborhood in polynomial time is described in [35], where every swap is a single movement. The authors of that paper showed that their results were the best known so far in terms of both solution quality and computational time. In [13], the common swap neighborhood is extended with generalized pairwise interchanges, showing how effective are the neighborhoods for some scheduling problems. An enhanced dynasearch swap neighborhood was developed in [20], precisely by adding generalized pairwise interchanges. A fast and efficient algorithm is presented in [17], which combines the insertion, swap and twist neighborhoods; its searching process takes $O(n^2)$ time.

Metaheuristics offer a good compromise between computational effort and solution quality. In the case of TWT, a number of metaheuristics have been applied to its solution, including simulated annealing, tabu search, genetic algorithms, ant colony optimization and, more recently, particle swarm optimization.

SA and TS are advanced local search techniques. SA uses a parameter named *temperature* for changing the probability of moving from one point within search space to another one [26]. This technique is based on a thermodynamic analogy: "start heating a row of materials to a fusing state for growing a crystal. Then reduce the temperature T until the crystal structure is frozen. But if the cooling is done quickly, bad things might occur (irregularities in the crystal structure, for instance, and the level of energy trapped is higher than a perfect crystal structured)". The state can be looked as a feasible solution, ground state as an optimal solution, temperature control parameter T, and the energy as the evaluation function. In the process, the T parameter used to influence the search of a better value, is updated periodically. Usually T starts with a high value (doing the procedure similar to a purely random search) and gradually decreases its value. In each iteration the best value is updated. The process is executed until some external condition is reached. SA approaches for the TWT problem are stated in [29, 34, 21].

TS, instead has a memory, which forces the algorithm to explore new areas without visiting previous ones [19]. The solutions examined recently become "tabu" (forbidden) points to select as a new solution and are stored in a list H. The process is structurally similar to that of SA. It returns an accepted solution which needs not be better. The acceptance is based on the previous history of the search H. The process makes a new movement in the search

space only when the search is stuck in a local optimum, although SA does not have this condition. In [21], TS was applied to solve the TWT problem.

Genetic Algorithms (GAs) are a particular type of Evolutionary Algorithm (EA) which normally adopt a binary encoding for their individuals. GAs are based in the "survival of the fittest" principle from Darwin's evolutionary theory. GAs choose the fittest individuals to recombine, aiming to increase the fitness of all the population over time. GAs use operators such as selection, mutation and crossover to create a new population. Comparisons of methods that include GAs might be found in [14]. In that work, the authors presented a competitive GA to solve the TWT. This GA uses problem-specific knowledge which is inserted with the aim of removing some of the "blindness" at the search traditionally performed by a GA. This GA outperformed other evolutionary algorithms in the TWT, which showed the efficacy of using problem-specific knowledge.

The Ant Colony Optimization (ACO) is a paradigm inspired by the trail following behavior observed in colonies of real ants. ACO was applied to TWT in [30], in which a pheromone summation evaluation was adopted for the probability of transition, and a specific heuristic was tailored for the TWT. Better results were presented in [28] and [6]. The latter introduced local search which is combined with the constructive phase obtaining an algorithm that uses heterogeneous colonies of ants.

Particle Swarm Optimizer (PSO) is a population-based stochastic heuristic which is inspired in the flight patterns of a flock of birds, in which a group (called the "swarm") follows a leader. As indicated before, PSO has been scarcely applied to scheduling problems. In [38], there is a comparative study between PSO, ACO and Iterative Local Search algorithm in the TWT problem. In [8], the authors proposed to adopt the random keys encoding for the individuals combined with a dynamic mutation operator. In [8], results are compared with respect to conventional heuristics and with respect to an evolutionary algorithm [14] that was fairly competitive at that time. In both cases, results indicated that PSO is a promising heuristic to tackle the TWT problem.

4 Improved Hybrid PSO Algorithms for the TWT Problem

In this section, the three PSO variants adopted in this chapter (i.e., $HPSO$, $HPSO_{neigh}$, and $HPSO_{kn}$) are described. However, we first present the pseudocode of the classical PSO algorithm (see Figure 1), because it will serve as the basis for all the other algorithms.

As can be seen in Figure 1, once the swarm, the velocities of each particle and the particle best memory are initialized (lines 2 to 4), the swarm is evaluated and the *leader* (the best particle of the swarm or the best in the neighboorhod, if appropriate) is selected (line 5). Then, at each iteration,

```
1. InitializeSwarm(Part)
2. InitializeVelocities(v)
3. Copy(Part, PartBests)
4. EvaluateParticles(Part, ObjectiveFunction)
5. Remember Leader of Swarm
6. do
7.     UpdateVelocities(v)
8.     UpdatePositionParticles(Part)
9.     EvaluateParticles(Part, ObjectiveFunction)
10.    UpdateParticleMemory(PartBests) if appropriate
11.    SelectNewLeader
12. while (¬termination)
```

Fig. 1. General outline of the classical *PSO* Algorithm

the velocities and positions of the particles are updated using equations (1) and (2) defined in Section 1 (lines 7 and 8). After the update process takes place, each particle is evaluated at its new position (line 9). If the new particle is better than its personal best position (line 10), then this last one is accordingly updated, i.e. $PartBest_i$ is set to $Part_i$.

4.1 *HPSO* Algorithm Description

As indicated before, there is sufficient evidence of the good performance of the *PSO* algorithm in continuous search spaces. The main motivation for the development of the *HPSO* algorithm was to preserve such efficiency when dealing with discrete optimization problems. Thus, we decided to adopt the random keys encoding proposed in [4] so that we could preserve a real-numbers encoding when dealing with permutations. The main idea of the random keys encoding is to adopt a set of randomly generated real numbers, which are then sorted and decoded in such a way that their position in the sequence is interpreted as a permutation position. In the scheduling problem studied, each particle is an n-dimensional vector and each dimension (a real number with two digits of precision) corresponds to a job. The components are randomly generated when the algorithm starts within the range $(0, 1)$. Then, the particle is transformed into a schedule by sorting those values in ascending order. Let's illustrate this with an example: for a nine-job problem, let's assume that we have the particle vector <0.23, 0.08, 0.97, 0.96, 0.32, 0.55, 0.18, 0.87, 0.99>. If we sort this list of real numbers in ascending order, we have the following sequence: <0.08, 0.18, 0.23, 0.32, 0.55, 0.87, 0.96, 0.97, 0.99>. Now, from this sorted list, we extract the mapping that we need: the first value (0.08) corresponds the the integer 1, the second value (0.18), corresponds to the integer 2, and so on. Going back to the original (unsorted) list of real numbers, the permutation that it encodes can be obtained by replacing the integers that

1. InitializeSwarm($Part$)
2. InitializeVelocities(v)
3. Copy($Part$, $PartBests$)
4. EvaluateParticles($Part$, $ObjectiveFunction$)
5. Remeber Best Leader of Swarm
6. **do**
7. CalculateProbability Mutation(p_{mut})
8. UpdateVelocities(v)
9. UpdatePositionParticle($Part$)
10. EvaluateParticles($Part$, $ObjectiveFunction$)
11. UpdateParticleMemory($PartBests$) if appropriate
12. MutateSwarm($Part$)
13. EvaluateParticles($Part$, $ObjectiveFunction$)
14. UpdateParticleMemory($PartBests$) if appropriate
15. SelectNewLeader
16. **while** ($\neg termination$)

Fig. 2. General outline of the $HPSO$ Algorithm

we produced from the sorted list. So, we have the following schedule: <3 1 8 7 4 5 2 6 9>. This is thus the permutation evaluated to determine the objective function value of this particle. It is worth noting, however, that due to the redundancy of the representation, many random key vectors may result in the same schedule. So, with the aim of maintaining diversity in the population, we adopted a dynamic mutation operator.

The mutation operator is applied to change the value of a component of a particle, with a probability pm varying between max_pm and min_pm, which depends on the total number of cycles max_cycles and the $current\_cycle$.

$$pm = max\_pm - \frac{max\_pm - min\_pm}{max\_cycle} \times current\_cycle \tag{3}$$

where max_pm and min_pm are the maximum and minimum values that pm can take, max_cycle is the total number of cycles that the algorithm will iterate, and $current\_cycle$ is the current cycle in the iterative process. In this way, mutation is more frequently applied at the beginning of the search process and its application decreases as the number of iterations increases. The particle is updated only if the objective function value of the new particle is better than the objective function value prior to applying mutation. Figure 2 displays the pseudocode for the $HPSO$ approach.

The differences between $HPSO$ and the PSO algorithm are described in Figures 1 and 2, and are expressed in lines 7, 12, 13, and 14, where the $HPSO$ algorithm includes the mutation operator and the re-evaluation of the swarm to see if each mutated particle is better than its ancestor; if this is the case, then the best position memory is updated.

4.2 $HPSO_{neigh}$ Algorithm Description

As was observed in [8], $HPSO$ converges to a local optimum in some difficult instances of the TWT, which causes stagnation in the search. In order to avoid this problem, $HPSO$ was improved through the use of a neighborhood circle topology (see Figure 3). In this topology, each particle is influenced both by the best value found by the particle itself and by the best value found in the neighborhood so far (neighborhood leader).

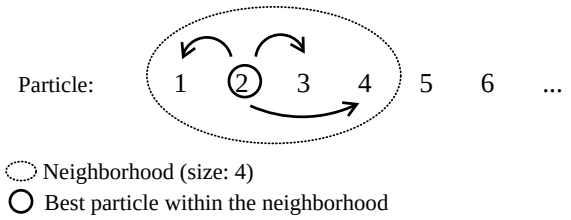

Fig. 3. Graphical illustration of the circle topology adopted by the $HPSO_{neigh}$ algorithm.

For example, if we have a swarm with 6 particles and the neighborhood size is 4, then the following neighborhoods are considered: 0 1 2 3, 1 2 3 4, 2 3 4 5, 3 4 5 0, 4 5 0 1, and 5 0 1 2 (the numbers indicating the particle index). Then, each particle is influenced by the performance of the leader of a smaller group instead of being influenced by the performance of the best global leader (i.e., of the complete swarm). Figure 4 presents the pseudocode of the $HPSO_{neigh}$ algorithm. In line 7 the neighborhood of any $part_i$ is composed by the particles whose index are in the interval $[i, i + neighborhood\_size - 1]$ if $i + neighborhood\_size - 1 < number\_particles$. Otherwise, the neighborhood of any $part_i$ consisting of particles whose index are in the interval $[i, neighborhood\_size - 2]$ $(\forall i = 1, \ldots, number\_particles)$.

Besides the inclusion of the neighborhood handler it is important to note that $HPSO_{neigh}$ differs from $HPSO$ in that the former does the particle processing asynchronously, whereas the last one does such processing synchronously. In the asynchronous update, the neighbors on one side of the particle to be adjusted have been updated, while the neighbors on the other side have not. In the synchronous update, the leader is the same for all the particles; therefore, they can be updated in parallel [9].

The algorithms presented in this work were implemented following these criteria since there is prior empirical evidence of the efficiency of these types of processing [18, 9].

```
1. InitializeSwarm(Part)
2. InitializeVelocities(v)
3. Copy(Part, PartBests)
4. do
5.     for i = 1 to number_particles do
6.         CalculateProbabilityMutation(p_mut)
7.         Search the leader in the neighborhood of part_i
8.         UpdateVelocity(v_i)
9.         UpdateParticle(part_i)
10.        EvaluateParticle(part_i, ObjectiveFunction)
11.        UpdateParticleMemory(part_i, PartBest_i) if appropriate
12.        MutateParticle(part_i)
13.        EvaluateParticle(part_i)
14.        UpdateParticleMemory(part_i, PartBests_i) if appropriate
15.    end
16. while (¬termination)
```

Fig. 4. General outline of the $HPSO_{neigh}$ Algorithm

4.3 $HPSO_{kn}$ Algorithm Description

To improve the previous approach ($HPSO_{neigh}$), we inserted problem-specific knowledge through three seeds generated by three good heuristics: *Rachamadagu and Morton Heuristic* (R&M), *Covert* and *Montagne Heuristic* [32] whose principal property is not only the quality of the results, but also to give an ordering of the jobs (schedule) close to the optimal sequence.

The Rachamadagu and Morton Heuristic, provides a schedule according to the following expression:

$$\pi_j = (w_j/p_j)[exp\{-(S_j)^+/kp_{av}\}] \tag{4}$$

where $S_j = [d_j - (p_j + Ch)]$ is the slack of job j at time Ch and Ch is the total processing time of the jobs already scheduled, k is a parameter of the method (usually $k = 2.0$) and p_{av} is the average processing time of the jobs competing for top priority. In this heuristic, jobs are scheduled one at a time and every time a machine becomes free, a ranking index is computed for the remaining ones. The job with the hightest ranking index is selected to be processed.

The Covert Heuristic, works in a similar way to R&M in cases of a single resource (our case), but applies instead the expression:

$$\pi_j = (w_j/p_j)\{1 - (S_j)^+/kp_j\} \tag{5}$$

The Montagne Heuristic, for its part, uses the following equation:

$$\pi_j = (w_j/p_j)[1 - (d_j)\sum_{i=1}^{n} p_i] \quad (6)$$

This equation does not consider the slack factor, but the due date of every job (d_j) and the sum of all the processing time (p_i).

Several other heuristics previously proposed for the TWT problem in the specialized literature were also tested (using the PARSIFAL package [32]), but we found the three above heuristics to be the most effective and therefore our choice. Needless to say, all of these heuristics are representative of the state-of-the-art in this problem.

As the seed values for each of these three heuristics are very close from each other (in most cases, the Euclidean distance among them is less than one unit in objective function value), we hypothesized that if we put them together, they would influence each other and, slowly, they would also influence the other solutions. That was the reason why we decided to introduce the three seeds within the initial population of particles. Note however, that different positions of the population were adopted for the insertion in each case (see Figure 5). The R&M seed is inserted randomly within the first third of the population, the Montagne seed in the second third, and Covert in the last third of the population (this was done considering the positions of the particles within the storage structure). In that way, each particle is forced to be influenced by some of these good permutations. In some cases, the particles located in the limit of each range might be influenced by two seeds. However, the final value will be the result of the influence of the best of them. Figure 6 shows the pseudocode for our *HPSO_kn* algorithm.

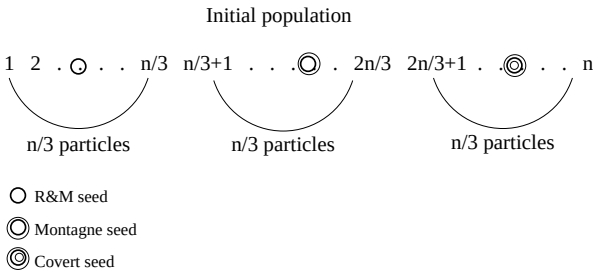

Fig. 5. Graphical illustration of the way in which the three types of seeds (produced by the three heuristics adopted) are inserted in the population.

Finally, we will proceed to briefly describe the evolutionary algorithm used to compare our results. The MCMP-SRI-IN [14] approach considers the mating of an evolved individual (the *stud*) with both random and seed immigrants. The process for creating offspring is the following. From the old population, the stud is selected by means of proportional selection and inserted into the mating pool. A number of n_1 parents in the mating pool is completed

```
1. InitializeSwarm(Part)
2. InitializeVelocities(v)
3. Copy(Part, PartBests)
4. // Seeds Insertion
5. s = rnd(0, number_particles/3)
6. CopySeed(seed_{R&M}, part_s)
7. s = rnd(number_particles/3 + 1, 2 * number_particles/3)
8. CopySeed(seed_{Covert}, part_s)
9. s = rnd(2 * number_particles/3 + 1, number_particles)
10. CopySeed(seed_{Montagne}, part_s)
11. do
12.     for i = 1 to number_particles do
13.         CalculateProbabilityMutation(p_{mut})
14.         Search the leader in the neighborhood of part_i
15.         UpdateVelocity(v_i)
16.         UpdateParticle(part_i)
17.         EvaluateParticle(part_i, ObjectiveFunction)
18.         UpdateParticleMemory(part_i, PartBest_i) if appropriate
18.         MutateParticle(part_i)
19.         EvaluateParticle(part_i)
20.         UpdateParticleMemory(part_i, PartBests_i) if appropriate
21.     end
22. while (¬termination)
```

Fig. 6. General outline of the $HPSO_{kn}$ Algorithm

both with randomly created individuals (the "random immigrants") and with "seed immigrants". The stud mates every other parent. The couples undergo crossover (partial mapped crossover) and $2 \times (n_2 - 1)$ offspring are created. The best of these offspring is stored in a temporary children pool. The crossover operation is repeated n_1 times, for different cut points each time, until the children pool is full. Finally, the best offspring created from n_2 parents and n_1 crossover operations is inserted into the new population. Figure 7 displays this process.

5 Experimental Design

As indicated before, the goal of the work reported here was to determine the performance of different PSO optimizers when used to solve the total weighted tardiness problem in single machine environments. As indicated before, even with this relatively simple formulation, this model leads to an optimizacion problem that is NP-hard.

The algorithms were tested on twenty instances of 40 and 50 jobs, which were extracted from the OR-Library [5]. The numbering of the problems are

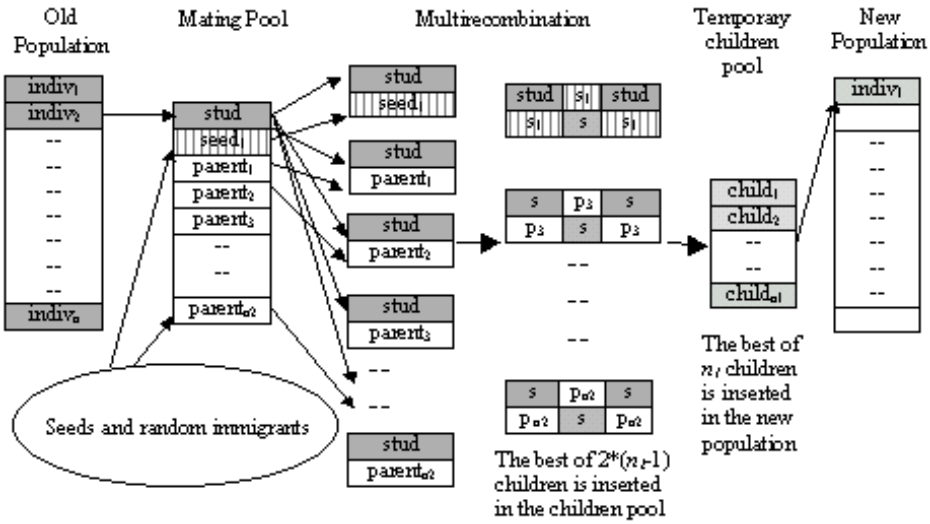

Fig. 7. General outline of $MCMP - SRI - IN$ approach

not consecutive because each one was randomly selected from different groups. The tardiness factor, which is an instance parameter that controls the number of tardy jobs, is harder for those with a higher identifier number. That means that a higher identifier number of instances involves a greater number of tardy jobs.

As it is well-known for researchers working with metaheuristics, the parameters setting of the technique is a very important issue that deserves special attention. Thus, we conducted some preliminary experiments in order to determine the most suitable values for the PSO approaches considered in our study. The values of w (inertia factor), c_1 and c_2 (personal and social learning factors, respectively) were defined following the suggestions from van den Bergh [39]. Analogously, the neighborhood size was fixed between the 8% and 10% of the total swarm size. The values adopted for these parameters in all the experiments conducted are shown in Table 1. The swarm size was set proportional to the permutation length, as suggested by Clerc [11]. 30 independent runs were performed in each experiment. The maximum number of iterations was fixed as follows: $HPSO$ 6000 (40 jobs) and 9000 (50 jobs); $HPSO_{neigh}$ and $HPSO_{kn}$ 50000 (40 jobs) and 65000 (50 jobs). These values were empirically derived after an exhaustive series of experiments. Initially, $HPSO$ ran for the same number of cycles as the other approaches, but its performance did not improve. Thus, as a consequence, we decided to reduce its total number of iterations.

For $HPSO_{neigh}$ and $HPSO_{kn}$, it was neccesary to determinate the values for the mutation probability (pm). This parameter depends of two values:

Table 1. Parameter settings for the PSO algorithms considered.

Parameters	$HPSO$	$HPSO_{neigh}$	$HPSO_{kn}$
Inertia factor	0.3	0.5	0.5
Learning factors	1.3	1.5	1.5
Neighborhood size	-	4	4

min_pm and max_pm which, in our case, were fixed to 0.1 and 0.4, respectively.

Additionally, the parameter settings for $MCMP-SRI-IN$ were taken from [14] and are the following: the evolutionary algorithm ran for 200 generations with a population size of 100 individuals. The crossover probability was 0.65 and the mutation probability was 0.05. The algorithm performed 14 crossover operations on each pair of parents and it used 16 parents to recombine. The number of seed was 3 (generated with R&M, Covert, and Modified R&M heuristics).

5.1 Performance Metrics

To compare the algorithms, the following performance metrics were chosen:

- **Best**: It indicates the best value found by an algorithm.
- μ**Best**: It is the mean objective value obtained from the best found particles throughout all runs.
- σ**Best**: It is the standard deviation of the objective values corresponding to the best found particles throughout all runs with respect to μ**Best**.
- (σ/μ)**Best**: This coefficient of variation is calculated as the $\sigma Best$ and $\mu Best$ ratio. It represents the desviation as a percentage of the $\mu Best$ value. The closer this value is to zero, the higher the robustness of the results obtained by an algorithm.
- **Mean Evaluations (ME)**: It is the mean number of evaluations necessary to obtain the best value of the objective function found throughout the runs performed.
- **Hit Ratio (HR)**: It is the percentage of runs where the algorithm reaches the best known values for each test function.

6 Analysis of Results

In this section, we present the results obtained for the algorithms compared as well as a brief discussion of them. First, we present the results obtained by the classical PSO, which are displayed in Tables 2 and 3 for instances of

40 and 50 jobs, respectively, where **IN** denotes the problem instance number and the **Best Known Values**, were taken from the OR-Library [5].

Table 2. *PSO* performance for problem instances of 40 jobs

IN	Best Known Value	Best	σ/μ	ME	HR
1	913	913	0.4631	320	0.03
6	6955	8708	0.1363	3220	0.00
11	17465	20652	0.1324	120002	0.00
19	77122	81184	0.0501	85233	0.00
21	77774	81057	0.0583	125512	0.00
26	108	108	0.8715	240	0.03
31	6575	9832	0.1789	135522	0.00
41	57640	63311	0.0643	2445	0.00
46	64451	67088	0.0570	289874	0.00
51	0	661	0.5225	47877	0.00
56	2099	2779	0.2827	586588	0.00
66	65386	75419	0.0617	298854	0.00
71	90486	93072	0.0510	147455	0.00
76	0	0	1.8088	200954	0.70
91	47683	57484	0.0706	568852	0.00
96	126048	130657	0.0333	75665	0.00
101	0	0	0.0000	1552	1.00
106	0	0	0.0000	2544	1.00
116	46770	56139	0.0872	185587	0.00
121	122266	128107	0.0581	299847	0.00

From the results shown in Tables 2 and 3, it can be seen that the classical *PSO* is unable to reach the best known values in almost all the instances. This is indicated by the zero values for the HR metric, except for instances 101 and 106 in the case of 40 jobs. This is the reason by which in the remainder of this section, only the results for $HPSO$, $HPSO_{neigh}$, $HPSO_{kn}$ and $MCMP-SRI-IN$ are discussed.

Tables 4 and 5 summarize the best objective function values found by the *PSO* variants and by the evolutionary algorithm for problem instances with 40 jobs and 50 jobs, repectively. Observing these values in both tables, we can see that the $HPSO$ algorithm has, for some instances, the worst performance (marked with boldface). This is due to the fact that each particle in the swarm is attracted towards the position of the global best particle, which leads to a stagnation of the algorithm in a local optimum. In the case of 40 jobs, $HPSO_{neigh}$ and $HPSO_{kn}$ converge to the same best values, and both algorithms outperform to $MCMP-SRI-IN$ in instance 21. The results for the 50 jobs problems (Table 5) show that in instance 6, $HPSO_{kn}$ obtains the worst best value (but yet it is closer to the best known value). For instances

Table 3. *PSO* performance for problem instances of 50 jobs

IM	Best Known Value	Best	σ/μ	ME	HR
1	2134	2259	0.22623	98558	0.00
6	26276	29241	0.08538	568856	0.00
11	51785	53844	0.08870	866585	0.00
19	89299	99698	0.06118	248898	0.00
21	214546	221119	0.03595	25442	0.00
26	2	10	1.11608	17452	0.00
31	9934	14426	0.16843	34252	0.00
41	123893	124855	0.02765	27784	0.00
46	157505	167009	0.46806	89552	0.00
51	0	0	0.7619	131905	0.03
56	1258	1258	0.19776	57884	0.10
66	76878	76991	0.01623	25995	0.00
71	150580	151322	0.02495	69899	0.00
76	0	0	1.0000	27741	0.20
91	9298	39787	0.02778	98778	0.00
96	77909	187222	0.00991	33541	0.00
101	0	0	0.9935	37787	0.50
106	0	0	0.9000	47785	0.35
116	35727	38544	0.03077	78448	0.00
121	8315	79884	0.02304	35884	0.00

Table 4. *Best* metric values for TWT 40 jobs problem size

IN	Best Known Value	$HPSO$	$HPSO_{neigh}$	$MCMP-SRI-In$	$HPSO_{kn}$
1	913	913	913	913	913
6	6955	6955	6955	6955	6955
11	17465	17465	17465	17465	17465
19	77122	77122	77122	77122	77122
21	77774	77774	77774	77774	77774
26	108	108	108	108	108
31	6575	6575	6575	6575	6575
41	57640	57640	57640	**57876**	57640
46	64451	**64459**	64451	64451	64451
51	0	0	0	0	0
56	2099	2099	2099	2099	2099
66	65386	**65402**	65386	65386	65386
71	90486	**90523**	90486	90486	90486
76	0	0	0	0	0
91	47683	47683	47683	47683	47683
96	126048	126048	126048	126048	126048
101	0	0	0	0	0
106	0	0	0	0	0
116	46770	**46771**	46770	46770	46770
121	122266	**122304**	122266	122266	122266

Table 5. *Best* metric values for TWT 50 jobs problem size

IN	Best Known Value	$HPSO$	$HPSO_{neigh}$	$MCMP-SRI-In$	$HPSO_{kn}$
1	2134	2134	2134	2134	2134
6	26276	26276	26276	26276	**26281**
11	51785	51785	51785	51785	51785
19	89299	**89308**	**89308**	89299	89299
21	214546	**214585**	**214744**	**214555**	**214555**
26	2	2	2	2	2
31	9934	9934	9934	9934	9934
44	123893	**124261**	123893	123893	123893
46	157505	**157536**	157505	157505	157505
51	0	0	0	0	0
56	1258	1258	1258	1258	1258
66	76878	**76948**	76878	76878	76878
71	150580	**150667**	150580	150580	150580
76	0	0	0	0	0
91	89298	**89543**	**89323**	**89448**	**89474**
96	177909	**178007**	177909	177909	177909
101	0	0	0	0	0
106	0	0	0	0	0
116	35727	**35830**	**35728**	35727	35727
121	78315	**78396**	78315	78315	78315

19, 21, 91, and 116, $HPSO_{kn}$ is the algorithm with the best performance, and specially in instance 21 where none of the algorithms reaches the best know values, $HPSO_{kn}$ obtains the same value than $MCMP-SRI-IN$. As a conclusion, we can say that except for some instances (marked with boldface), all the algorithms find the best known values. In fact, even when these values are not reached, $HPSO_{kn}$ and $MCMP-SRI-IN$ converge to very similar values.

Nevertheless, it is important to analyze these results in more details, by using other performance metrics such as Hit Ratio and the mean number of evaluations that each algorithm has to perform to find the best value.

Figure 8 shows the analysis of the Hit Ratio metric. In this case, we can see that $HPSO_{neigh}$ finds the best known values approximately 70% of the time for the case of 40 jobs and around 50% of the time for the case of 50 jobs. In contrast, $HPSO_{kn}$ reaches the best known values in approximately the 80% and 70% of the runs for the 40 and 50 jobs instances, respectively. Also, we can observe that the results obtained with $MCMP-SRI-IN$ are slighly better than those found by $HPSO_{kn}$, although none of the algorithms finds the best known values for all the instances in all the runs. With the previous observations in mind, we can conclude that $HPSO_{kn}$ is superior to $HPSO_{neigh}$ and its results are comparable to those obtained by $MCMP-SRI-IN$ (which can be seen as an evolutionary algorithm that has been carefully tailored for the problem being solved in this study).

Figure 9 shows the cost measured as the mean number of evaluations that an algorithm performs to reach the best known values. In this case, $HPSO_{kn}$ performs, on average, a lower number of evaluations when compared with $HPSO_{neigh}$ and $MCMP-SRI-IN$, a difference that becomes even higher

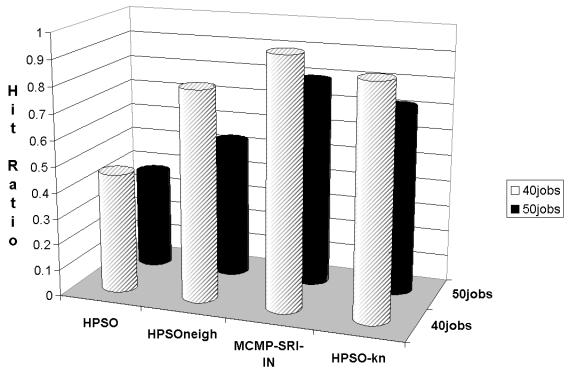

Fig. 8. Performance evaluation with respect to the Hit Ratio metric

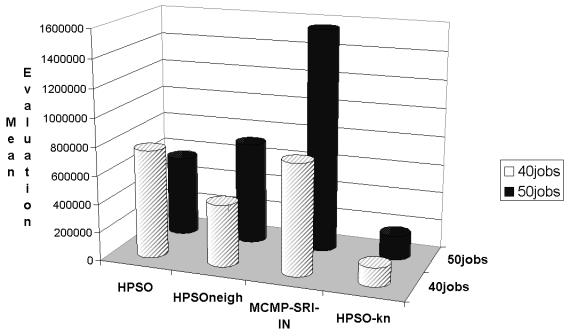

Fig. 9. Performance evaluation with respect to the Mean Evaluation metric

when the problem size is increased. This difference was somehow expected (with respect to $HPSO_{neigh}$) due to the guided search that the $HPSO_{kn}$ performs. The idea of including knowledge about the problem in the algorithm is not new, since it has been successfully applied in the past in several evolutionary algorithms [3].

Table 6. $(\sigma/\mu)Best$ mean values obtained by the PSO variants compared

Problem Size	HPSO	HPSO_neigh	HPSO_kn
40	0.003825	0.002880	0.001950
50	0.003320	0.001565	0.000100

In Table 6, we show the mean values over all the coefficients of variation of the best values calculated for all the instances, for the different PSO variants for the two instances studied (40 jobs and 50 jobs). These values are grouped around the mean. Although not all the coefficient values were equal to zero, they are very close, which suggests robustness of the algorithms with respect to the results that they found.

7 Conclusions and Future Work

In this chapter, three improved PSO variants were presented to deal with permutation problems. To determine the performance of the algorithms studied, the weighted tardiness scheduling on the single machine environments problem was selected as a case of study. $HPSO$ is a hybridized PSO in the sense that a suitable representation and a dynamic mutation operator were adopted to make it more competitive in sequencing problems. However, we saw that this approach in which the global leader is always followed, is prone to converge to a local optimum, causing a premature convergence of the algorithm.

As a way of dealing with this drawback, we proposed an approach called $HPSO_{neigh}$, which incorporates a simple neighborhood topology, so that each particle is only influenced by the best local particle in its neighborhood. This modification allowed that the algorithm could find all the best known values for the 40 jobs problem size and increased the number of instances in which the algorithm found the best known values for the instances of 50 jobs (instances 19, 21, 44, 46, 66 and 71). A further modification was introduced, which consisted of the incorporation of specific domain knowledge by means of the inclusion of seeds (generated with another heuristic) in the swarm. This new version was named $HPSO_{kn}$. All these algorithms were compared among themselves and with respect to $MCMP-SRI-IN$, which is an evolutionary algorithm specially tailored for the problem of interest and which also uses the inclusion of knowledge through seeds. Although $HPSO_{neigh}$, $HPSO_{kn}$ and $MCMP-SRI-IN$ found objective values which are similar, $HPSO_{kn}$ and $MCMP-SRI-IN$ exceeded widely to $HPSO_{neigh}$ in the number of runs in which they reached the best known values as was shown with the Hit Ratio values. In spite of that, the cost (measured in the number of evaluations performed to reach the best known values) of $HPSO_{kn}$ is fairly smaller than the one required by $MCMP-SRI-IN$ and also (as expected) is about a 50% lower than the cost of $HPSO_{neigh}$. We believe that these preliminary results are good enough to consider $HPSO$ variants as a promising approach for scheduling problems. Thus, we are convinced that this topic deserves further study.

As part of our future work, we are considering different possibilities. The first one is to minimize the redundancy of the encoding currently adopted by exploring alternative encodings. Second, we aim to study the effect of incorporating and adapting other operators which have been typically used

with evolutionary algorithms to solve permutations problems. Finally, it is of great relevance for us the study of the behavior of our proposed approach in much larger instances of this problem (between 100 and 200 jobs).

Acknowledgments

The third author gratefully acknowledges support from CONACyT project number 45683-Y.

References

1. M. S. Akturk and M. B. Yildirim. A New Dominance Rule for the Total Weighted Tardiness Problem. *Production Planning and Control*, 10(2):138–149, 1999.
2. B. Alidaee and K. R. Ramakrishnan. A computational experiment of covert_au class of rules for single machine tardiness scheduling problem. *Computers and Industrial Engineering*, 30(2):201–209, 1996.
3. Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz. *Handbook of Evolutionary Computation*. IOP Publishing LTD and Oxford University Press, 1997.
4. James C. Bean. Genetics and random keys for sequencing and optimization. *ORSA Journal on Computing*, 6(2):154–160, 1994.
5. J. E. Beasley. OR Library, Scheduling: Weighted Tardiness. http://people.brunel.ac.uk/~mastjjb/jeb/info.html.
6. M. Den Besten, T. Stutzle, and M. Dorigo. Ant colony optimization for the total weighted tardiness problem. In *Proc. of PPSN-VI: Sixth International Conference on Parallel Problem Solving from Nature*, volume LNCS 1917, pages 611–620, 2000.
7. L. Cagnina and S. Esquivel. Particle swarm optimization para un problema de optimización combinatoria. In *Memorias del X Congreso Argentino de Ciencias de la Computación*, pages 1847–1855, La Matanza, Buenos Aires, Argentina (in Spanish), 2004. http://www.lidic.unsl.edu.ar/publicaciones/info_publicacion.php?id_publicacion=199.
8. L. Cagnina, S. Esquivel, and R. Gallard. Particle swarm optimization for sequencing problems: a case study. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation (CEC'2004)*, pages 536–541, Portland, Oregon, USA, 2004.
9. A. Carlisle. *Applying The Particle Swarm Optimization to Non-Stationary Environments*. PhD thesis, Auburn University, USA, December 2002.
10. K. Caskey and R. L. Storch. Heterogeneous dispatching rules in job and flow shops. *Production Planning and Control*, 7:351–361, 1996.
11. M. Clerc. Discrete particle swarm optimization illustred by the traveling salesman problem, 2000. http://www.mauriceclerc.net.
12. Carlos A. Coello Coello, Erika Hernández Luna, and Arturo Hernández Aguirre. Use of particle swarm optimization to design combinational logic circuits. In

Andy M. Tyrell, Pauline C. Haddow, and Jim Torresen, editors, *Evolvable Systems: From Biology to Hardware. Proceedings of the 5th International Conference, ICES 2003*, pages 398–409, Trondheim, Norway, 2003. Springer, Lecture Notes in Computer Science Vol. 2606.

13. F. Della Croce. Generalized pairwise interchanges and machine scheduling. *European Journal Operations Research*, 83:310–319, 1995.
14. M. De San Pedro, D. Pandolfi, A. Villagra, M. Lasso, G. Vilanova, and R. Gallard. Adding problem-specific knowledge in evolutionary algorithms to solve wt scheduling problems. In *Memorias del VIII Congreso Argentino de Ciencias de la Computación*, pages 343–353, Buenos Aires, Argentina (in Spanish), 2002. http://www.lidic.unsl.edu.ar/publicaciones/info_publicacion.php?id_publicacion=198.
15. R. Eberhart and Y. Shi. A modified particle swarm optimizer. In *International Conference on Evolutionary Computation, IEEE Service Center*, Anchorage, AK, Piscataway, NJ, 1998.
16. R. C. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *6th International Symposium on Micro Machine and Human Science (Nagoya, Japan)*, pages 39–43, Piscataway, NJ., 1995. IEEE Service Center.
17. O. Ergun and J. B. Orlin. A fast algorithm for searching insertion, swap, and twist neighborhoods for the single machine total weighted tardiness problem. In *Working Paper, Operations Research Center, MIT*, 2004.
18. Susana C. Esquivel and Carlos A. Coello Coello. On the Use of Particle Swarm Optimization with Multimodal Functions. In *Proceedings of 2003 Congress on Evolutionary Computation (CEC'2003)*, volume 2, pages 1130–1136, Piscataway, NJ., December 2003. IEEE Press.
19. Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, Boston, Massachusetts, 1997.
20. A. Grosso, F. Della Croce, and R.Tadei. An enhanced dynasearch neighborhood for the single-machine total weighted tardiness scheduling problem. *Operations Research Letters*, 32:68–72, 2004.
21. C. N. Potts H. A. J. Crauwels and L. N. Van Wassenhove. Local search heuristics for the single machine total weighted tardiness scheduling problem. In *INFORMS Journal on Computing*, volume 10(3), pages 341–350, 1998.
22. X. Hu and R. Eberhart. Swarm intelligence for permutation optimization: a case study on n-queens problem. In *Proceeding of the IEEE Swarm Intelligence Symposium*, pages 243–246, Indianapolis, Indiana, USA, 2003.
23. A. H. G. Rinnooy Kan, B. J. Lageweg, and J. K. Lenstra. Minimizing total costs in one-machine scheduling. *Operations Research*, 23(3):908–927, 1975.
24. J. Kennedy and R. Eberhart. A discrete binary version of particle swarm algorithm. In *Proceedings of the World Multiconference on Systemics, Cybernetics and Informatics*, pages 4104–4109, Piscataway, NJ, 1997.
25. James Kennedy and Russell Eberhart. *Swarm Intelligence*. Morgan Kaufmann Publishers, 2001.
26. S. Kirkpatrick, C.D. Gellatt, and M.P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.
27. J. K. Lenstra, A. H. G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problem. In P. L. Hammer, E. L. Johnson, B. H. Korte, and G. L. Nemhauser, editors, *Studies in Integer Programming, volume I of Annals of Discrete Mathematics*, pages 343–362. North-Holland, The Netherlands, 1977.

28. Y. C. Liang. *Ant colony optimization approach to combinatorial problems*. PhD thesis, Department of Industrial and Systems Engineering, Auburn University, 2001.
29. T. E. Matsuo, C. J. Suh, and R. S. Sullivan. A controlled search simulated annealing method for the single machine weighted tardiness problem. *Annals of Operations Research*, 21:95–108, 1989.
30. D. Merkle and M. Middendorf. An ant algorithm with a new pheronome evaluation rule for total tardiness problem. In *Proc. of EvoWorkshops 2000: Real-World Applications of Evolutionary Computing*, volume LNCS 1803, pages 287–296, 2000.
31. C. Mohan and B. Al-Kazemi. Discrete particle swarm optimization. In *Proceeding of the Workshop on Particle Swarm Optimization*, Indianapolis, IN, 2001.
32. T. Morton and D. Pentico. *Heuristic Scheduling Systems*. Wiley series in Engineering and Technology management, John Wiley and Sons, 1993.
33. C. N. Potts and L. N. Van Wassenhove. A branch and bound algorithm for the total weighted tardiness scheduling problem. In *Operations Research*, volume 33 number 2, pages 363–377, 1985.
34. C. N. Potts and L. N. Van Wassenhove. Single machine tardiness sequencing heuristics. *IIE Transactions*, 23(4):346–354, 1991.
35. C. N. Potts R. K. Congram and S. Van de Velde. An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem. *INFORMS Journal on Computing*, 14:52–67, 2002.
36. R. M. V. Rachamadugu. A note on weighted tardiness problem. *Operations Research*, 35:450–452, 1987.
37. C. N. Potts T. S. Abdul-Razaq and L. N. Van Wassenhove. A survey of algorithms for the single machine total weighted tardiness scheduling problem. In *Discrete Applied Mathematics*, volume 26, pages 235–253, 1990.
38. M. Fatih Tasgetiren, Mehmet Sevkli, Yun-Chia Liang, and Gunes Gencyilmaz. Particle Swarm Optimization Algorithm For Single Machine Total Weighted Tardiness Problem. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation (CEC'2004)*, pages 1412–1419, Portland, Oregon, USA, 2004.
39. Frans van den Bergh. *An Analysis of Particle Swarm Optimization*. PhD thesis, Faculty of Natural and Agricultural Science, University of Petroria, Pretoria, South Africa, November 2002.
40. A. P. J. Vepsalainem and T. E. Morton. Priority rules for job shops with weighted tardiness costs. *Management Science*, 33:1035–1047, 1987.