A Hybridization of Harmony Search and Simulated Annealing to Minimize Mean Flow Time for the Two-Machine Scheduling Problem with a Single Server

Keramat Hasani[1], Svetlana A. Kravchenko[2], Frank Werner[3*]
[1]Islamic Azad University, Malayer Branch, Malayer, Iran
hasani@iau-malayer.ac.ir
[2]United Institute of Informatics Problems, Surganova St. 6, 220012 Minsk, Belarus
kravch@newman.bas-net.by
[3]Fakultät für Mathematik, Otto-von-Guericke-Universität Magdeburg,
Postfach 4120, 39016 Magdeburg, Germany
frank.werner@ovgu.de

November 15, 2013

Abstract

In this paper, we consider the problem of scheduling a set of jobs on two parallel machines to minimize the sum of completion times. Each job requires a setup which must be done by a single server. It is known that this problem is strongly NP-hard. We propose an improved harmony search algorithm. The performance of this algorithm is evaluated and compared with a pure simulated annealing algorithm for instances with up to 250 jobs.

Keywords: Scheduling, Parallel machines, Single server, Mean flow time, Harmony search algorithm, Simulated annealing algorithm

1. Introduction

The problem considered in this paper can be described as follows. There are n independent jobs and two identical parallel machines. For each job j, $j = 1, \ldots, n$, its processing time p_j is known. Before processing, a job must be loaded on the same machine M_q, $q = 1,2$, where it is processed, which requires a known setup time s_j. During such a setup, the machine M_q is also involved into this process for s_j time units, i.e., no other job can be processed on this machine during this setup. All setups have to be done by a single server which can handle at most one job at a time. The objective is to determine a feasible schedule which minimizes the sum of completion times. So, using the common scheduling notation, we consider the problem $P2,S1||\sum C_j$. This problem is strongly NP-hard, since it is known that problem $P2,S1|s_j = s|\sum C_j$ is unary NP-hard, see Hall et al. (2000). Some special cases of this problem were already

* corresponding author.

considered. The problem $P2,S1|p_j = p|\sum C_j$ is binary NP-hard, see Brucker et al. (2002). There exists a polynomial algorithm for the problem $P2,S1|s_j = 1|\sum C_j$, see Hall et al. (2000). For the problem $P3,S1|s_j = 1|\sum C_j$, Brucker et al. (2002) developed a polynomial algorithm with the complexity $O(n^7)$. For the problem $P,S1|s_j = 1|\sum C_j$, Kravchenko and Werner (2001) proposed an algorithm which creates a schedule with the following estimation:

$$f_h - f_{opt} = \sum_{i=1}^{n} \widetilde{C}_i - \sum_{i=1}^{n} C_i^* \le n'(m-2),$$

where $n' = |\{i \mid p_i < m-1\}|$, f_h denotes the heuristic function value and f_{opt} denotes the optimal function value. In Wang and Cheng (2001), a $(5-\frac{1}{m})$ - approximation algorithm was proposed for the problem $P,S1||\sum w_j C_j$, and it was shown that the SPT schedule is a $\frac{3}{2}$ - approximation for the problem $P,S1|s_j = s|\sum C_j$.

For the problem $P2,S1||\sum C_j$, nothing is known about heuristic algorithms. However, some results for close models are known. In Weng et al. (2001), the problem of scheduling a set of independent jobs on unrelated parallel machines with job sequence dependent setup times so as to minimize the weighted mean completion time was considered. Seven heuristic algorithms were presented and compared to each other and the best algorithm was selected. The heuristics were tested on instances with up to 120 jobs and 12 machines. In Dunstall and Wirth (2005), the problem with identical parallel machines, jobs divided into families and sequence-independent setup times was considered. The objective was to minimize the weighted sum of completion times. Several heuristics were proposed and tested on instances with up to 8 families, 25 jobs, and 5 machines. In Azizoglu and Webster (2003), several branch-and-bound algorithms for the identical parallel machine scheduling problem with family setup times and the objective of minimizing total weighted flow time was considered. They applied their methods to instances with up to 25 jobs, 8 families, and 5 machines. In Guirchoun et al. (2005), a parallel machine scheduling problem with a server was considered, however, the general requirement that loading requires the server and the machine was omitted. Thus, the considered problem was modelled as a two-stage hybrid flow shop with no-wait constraint between the two stages. A mathematical formulation for the problem was proposed and some polynomially solvable special cases were considered.

In this paper, we consider the problem $P2,S1||\sum C_j$ and present a simulated annealing algorithm. Then we develop a hybridization of harmony search and simulated annealing. We apply both algorithms to instances with up to 250 jobs.

The remainder of the paper is organized as follows. Section 2 presents some basic concepts for the problem under consideration. Section 3 describes the simulated annealing algorithm. Section 4 introduces the standard harmony search algorithm. In Section5, a combination of harmony search and simulated annealing is presented. In Section 6, computational results for both algorithms are given and their performance is compared. Section 7 gives some concluding remarks.

2. Basic Concepts

One can see that an optimal schedule for the problem $P2,S1||\sum C_j$ can be found in the class of list schedules, i.e., one can consider a sequence of the jobs, where all jobs are placed into a list and the algorithm schedules the first unscheduled job from the list whenever a machine becomes idle.

Example 1. We consider an instance with five jobs and the setup and processing times given in Table 1. Let the current job sequence be π_0= (3, 1, 4, 2, 5).

j	**1**	**2**	**3**	**4**	**5**
s_j	2	2	1	2	1
p_j	4	3	5	4	2

Table 1. Setup and processing times of the jobs in Example 1.

According to the given sequence, we assign the jobs in a greedy manner to the machines. In this way, we get the job sequence (3, 4, 5) on machine M_1 and the job sequence (1, 2) on machine M_2. Thus, we get the schedule described in Figure 1, where the grey parts represent the setups. The total completion time value is 53.

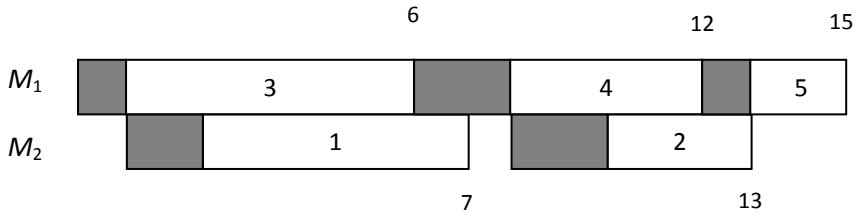

Figure 1: A schedule with five jobs resulting from π_0.

For evaluating the quality of the heuristic solutions, two lower bounds are used. In Hasani et al. (2013a), the following composite lower bound *LB* has been used:
Denote by L_j the sum $s_j + p_j$ for $j = 1,\ldots,n$. Suppose that all jobs are ordered in such a way that $L_1 \leq \ldots \leq L_n$ holds. For the first lower bound LB_1 we have $\sum_j C_j^1 = LB_1$, where

$C_j^1 = L_j + L_{j-2} + \ldots$.

For the second lower bound LB_2 we have $\sum_j C_j^2 = LB_2$, where

$C_j^2 = L_j + ss_{j-1} + ss_{j-2} + \ldots$,

and $ss_1, \ldots, ss_n$ are the setup times in a non-decreasing order of their values, i.e.,

$ss_1 \leq \ldots \leq ss_n$.

3. Simulated Annealing algorithm

This section presents a simulated annealing algorithm to minimize the total completion time for the two-machine scheduling problem with a single server. Simulated annealing tries to avoid cycling by randomization and simulates an annealing process in physics, see e.g. Kirkpatrick et al. (1983). In any iteration, a neighbor is determined by means of random decisions. In the case when the generated neighbor has a better objective function value than the starting solution, the neighbor is always accepted as the new starting solution while in the case of a worse neighbor, this solution is only with a certain probability accepted. For the quality of the results by a simulated annealing algorithm, the chosen neighborhood and the cooling scheme applied are important. Below we give more details.

Neighborhood

The definition of an appropriate neighborhood for a current scheduling solution has usually a large influence on the quality of the final solution. The algorithm presented later is based on the generation of a neighbor in a specific neighborhood. For permutation problems, one can use e.g. the following operators for generating a neighbor:

- **Swap operator**: Here two randomly selected jobs are swapped. Given π_0 in Example 1 and assume that the two randomly selected positions $a = 3$ and $b = 5$, we obtain the sequence

  $Swap(\pi_0, a, b) = Swap(\pi_0, 3, 5) = (3, 1, 5, 2, 4)$.

- **Adjacent Swap operator:** Here two adjacent and randomly selected jobs are swapped. Given $\pi_0$in Example 1 and assume that the two randomly selected positions $a = 3$ and $b = 4$ have been selected, we obtain the sequence

  $SwapAdj(\pi_0, a, b) = SwapAdj(\pi_0, 3, 4) = (3, 1, 2, 4, 5)$.

- **Swap Block operator**: Here a randomly selected block of ℓ jobs is swapped with another randomly selected block of ℓ jobs as well. The block length ℓ is a

randomly selected integer from the set $\{2, 3, \ldots, \left\lfloor \frac{n}{2} \right\rfloor\}$. If the block length ℓ has been chosen, we randomly determine a position a of the first job of the first block and a position b of the first job of the second block. Here we have $a + \ell \leq b \leq n - \ell + 1$. Given the sequence π_0 in Example 1 and assume that first the block length $\ell = 2$ and then the positions $a = 2$ and $b = 4$ have been selected. Then we obtain the sequence

$SwapBlock(\pi_0, a, b, \ell) = SwapBlock(\pi_0, 2, 4, 2) = (3, 2, 5, 1, 4)$.

- **Insert operator**: Here one randomly selected job is removed from its position a and is put on a randomly determined new position b. Given π_0 in Example 1 and assume that the two randomly selected positions are $a = 2$ and $b = 4$, we obtain the sequence

  $Insert(\pi_0, a, b) = Insert(\pi_0, 2, 4) = (3, 4, 2, 1, 5)$.

- **Insert Block operator**: Here one randomly selected block is removed from its position a and it is put on a randomly determined new position b. The block length ℓ is a randomly selected integer from the set $\{2, 3, \ldots n - b\}$. If the block length ℓ has been chosen, we randomly determine two positions a and b. Here we have $b + \ell - 1 \leq n$. Given the sequence π_0 in Example 1 and assume that first the block length $\ell = 2$ and then the positions $a = 2$ and $b = 4$ have been selected. Then we obtain the sequence

  $InsertBlock(\pi_0, a, b, \ell) = InsertBlock\ (\pi_0, 2, 4, 2) = (3, 2, 5, 1, 4)$.

- **Reverse Block operator**: Here a part of the sequence with length ℓ is reversed. Given the sequence π_0 in Example 1 and assume that the block length $\ell = 3$ and then the position $a = 2$ has been selected. Then we obtain the sequence

  $ReverseBlock(\pi_0, a, \ell) = ReverseBlock\ (\pi_0, 2, 3) = (3, 2, 4, 1, 5)$.

- **Insert Block and reverse operator:** Here after applying the insert block operator, the inserted block has been reversed. Given the sequence π_0 in Example 1 and assume that first the block length $\ell = 2$ and then the positions $a = 2$ and $b = 4$ have been selected. Then we obtain the sequence

  $InsertBlockRvs(\pi_0, a, b, \ell) = InsertBlockRvs(\pi_0, 2, 4, 2) = (3, 2, 5, 4, 1)$.

In any iteration of our simulated annealing algorithm, each of these seven operators is used and seven neighbors are randomly generated. Then the neighbor with the best objective function value among them is taken as the generated neighbor and compared with the current starting solution by the simulated annealing acceptance criterion. We

have found that the composite neighborhood worked better than each of the single neighborhoods.

Cooling scheme

Typical cooling schemes used in a simulated annealing algorithm are a geometric, an exponential, a Lundy-Mees and a linear reduction scheme. We tested some different cooling schemes for the problem under consideration and found that often the geometric scheme is slightly superior. In many other applications to scheduling problems, a geometric cooling scheme is also preferred. Therefore, in the following we test exclusively geometric schemes. The geometric cooling scheme reduces the current temperature to the new temperature in the next epoch according to

$$T_k = \alpha\, T_{k-1},\ k = 1,2,\ldots$$

where $0 < \alpha < 1$. We have found that the initial temperature should be chosen such that about 25 percent of worse solutions should be accepted at the beginning. For the problem under consideration, we have chosen the initial temperature $T = 15$. On the other hand, the final temperature should be low enough so that worse solutions do not longer replace solutions with better objective function values. Moreover, in our experiments it turned out that the value $\alpha = 0.999$ worked good. We have chosen an epoch length of 1, i.e., after each iteration the temperature is reduced. Alternatively, only after a period with constant temperature, one may reduce the temperature. In fact, updating the temperature must be done in a way that when the defined run time limit denoted by *TL* is going to be finished, the temperature becomes very close to zero. According to the geometric cooling scheme, at the final stage of the algorithm, $T_N = \alpha^N T_0$. Therefore, we have

$$N = \log_\alpha \frac{T_N}{T_0}$$

With the given value $T_N = 0.0005$, the algorithm will be finished after $N = 10304$ iterations.

Algorithm

In our simulated annealing algorithm, we used a randomly generated starting solution. As a stopping criterion, we used the first of the following events:

- a maximal run time limit or
- within the last 2000 iterations, no improvement of the best objective function value was obtained, or
- for the currently best value *BestSumC* of the sum of completion time, the inequality $BestSumC - LB < 1$ holds.

The complete simulated annealing algorithm is given below as Algorithm 1, where *Rand*(0,1) denotes a uniformly distributed random number from the interval (0,1) and *SumC* denotes the objective function value.

Algorithm 1. Simulated Annealing (SA)

```
BEGIN
   Generate an initial feasible solution π and determine SumC(π);
   BestSol = π; BestSumC = SumC (π)
   T = initial temperature;
   WHILE (stopping criterion is not met) DO
      π′ = best neighbor among the generated neighbors of π;
      ΔC = SumC (π′) – SumC (π);
      prob = Rand (0,1);
      IF ((ΔC ≤ 0) or (prob < e^(-ΔC/T))) THEN
         π = π′; SumC (π) := SumC (π′);
         IF (SumC (π) < BestSumC) THEN
            BestSumC = SumC (π); BestSol = π;
         END IF
      END IF
     T = Update (T);
   END WHILE
   Output BestSol together with its SumC value;
END.
```

4. Introduction to a Standard Harmony Search Algorithm

This section presents the standard harmony search algorithm (HSA). HSA is one of the population-based meta-heuristic algorithms, inspired by musical improvisation, see Lee and Geem (2004). In music performance, each musician plays one musical note at a time. Those musical notes are combined together to form a harmony, measured by aesthetic standards. In optimization, each variable during the optimization process is assigned a value at a time; those values all together form a solution for the considered problem, evaluated by the objective function, see Ingram and Zhang (2009).

Similar to a group of musicians developing their harmonies iteratively, HSA improves the solutions iteratively based on good candidate solutions from the initial population, i.e., the *harmony memory*, see Geem et al. (2001). It carries out a stochastic random search on a solution, i.e., a vector of decision variables, via a number of improvisations. The harmony memory is updated between the improvisations. At each improvisation, the stored values of the decision variables in the harmony memory are adapted according to a *considering rate*. The variable values in the solution are adjusted according to a *pitch adjusting rate*. HSA does not require any starting values of the decision variables nor does it require complex derivatives to adjust the variable values for the new generated

solutions, see Yang (2008). In the standard harmony search algorithm, the optimization procedure consists of steps 1–5, see Geem et al. (2001) and Geem (2008), as follows:

Step 1. Initialize the parameters of the algorithm. The parameters of HSA are the harmony memory size (HMS), the harmony memory considering rate (HMCR), the pitch adjusting rate (PAR), the distance bandwidth (BW), and the number of improvisations (NI). HMS is similar to the population size in genetic algorithms. During the improvisation process, HMCR is used to decide whether the variables of the solution should take the value of any one in the harmony memory (HM). The probability of choosing a solution in HM is determined by HMCR. It takes a value in the range [0,1]. PAR is also used during the improvisation process to decide whether a chosen solution should be changed to a neighbor. PAR takes a value in the range [0,1]. NI specifies the number of iterations in HSA.

Step 2. Build HM. HM is used to store all the solution vectors. In this step, the HM matrix is filled with randomly generated solution vectors. In this paper, a harmony (solution vector) $X^i = (x_1^i, x_2^i, ..., x_n^i)$ is represented as an *n*-dimensional vector of real numbers (jobs) which corresponds to a sequence of jobs.

$$\mathrm{HM} = \begin{bmatrix} X^1 \\ X^2 \\ \vdots \\ X^{HMS} \end{bmatrix} = \begin{bmatrix} x_1^1 & x_2^1 & \cdots & x_n^1 \\ x_1^2 & x_2^2 & \cdots & x_n^2 \\ \vdots & \vdots & \vdots & \vdots \\ x_1^{HMS} & x_2^{HMS} & \cdots & x_n^{HMS} \end{bmatrix}$$

Step 3. Improvise a new harmony. A new harmony solution vector, $X^i = (x_1^i, x_2^i, ..., x_n^i)$, can be constructed based on a memory consideration, a pitch adjustment and a random selection. The diversification and intensification of HSA are maintained in this step by the parameters PAR and HMCR, see Mahdavi et al. (2007). The procedure works as follows:

Procedure Improvising a new harmony

BEGIN
FOR *i* = (1 to NI) DO */* NI is the maximum number of improvisations*/*
  IF (rand(0,1) ≤ HMCR) THEN
    choose a vector randomly from HM
    IF (rand(0,1) ≤ PAR) THEN
      Change the chosen *solution* vector to a random neighbour and put the new solution
      into X^{new};
    ELSE

put the chosen *solution* into X^{new} without changing it

END IF

ELSE generate a *solution* randomly and put it into X^{new}

END IF

END FOR

END

Step 4. Update HM. Evaluate the objective function value of the new harmony vector. The new harmony vector will replace the worst one in HM, if the new harmony vector is better.

Step 5. Repeat Steps 3 and 4 until the termination criterion is met.

5. Hybrid Harmony Search algorithm (HHS)

In this section, we present a hybridization of harmony search and simulated annealing (HHS) to minimize total completion time for the problem of scheduling two parallel machines with a single server. The behaviour of the proposed algorithm is a combination of harmony search and simulated annealing. In our preliminary tests, a standard harmony search algorithm did not meet our expectations to achieve results which are better than simulated annealing proposed in Section 2. This fact motivated us to develop an algorithm which combines the advantages of the harmony search and the simulated annealing algorithms. Therefore, we made some improvements in the harmony improvisation procedure of the standard harmony search. In the following, we describe the new harmony improvisation procedure.

New harmony improvisation

As we discussed in step 3 of the standard harmony search algorithm, improvising a new harmony is generated based on the following rules: memory consideration, pitch adjustment and random selection. The improvisation by memory considerations depends on the HMCR parameter. HMCR is the probability which decides whether to choose a solution vector X^i randomly from HM, or to generate it randomly. In our algorithm, if the HMCR condition is not met, generating a random solution (not by means of HM solution vectors) is performed by using a step of the simulated annealing algorithm. By this way, the chance of finding a random solution which is eligible to be stored in HM may increase. While in a standard HSA, a solution which is generated randomly, may not be acceptable to be stored in HM, and this may influence the diversification and intensification of HSA.

If the HMCR condition is met, we will choose two solution vectors randomly from the HM, then the solutions may be adjusted based on PAR. To increase the chance of finding **'**deeper' solutions, in our algorithm the pitch adjustment is done in two stages. The first

stage is applying a two cut-point crossover called *PMX crossover*. It should be noted that there exist different crossover operators for permutation problems. For an overview on crossover operators applied to such permutation problems, the reader is referred to the survey by Werner (2011). All these crossovers have been tested, the influence on the solution quality was marginal, and it has been decided to use the *PMX crossover*.

The second stage consists in the application of a neighborhood operator as we did in SA described in Section 3. In the following, we give a pseudo-code of the hybrid algorithm.

Algorithm

In the HHS algorithm, we used a randomly generated starting solution. As a stopping criterion, we used the first of the following events:

- a maximal run time limit or
- within the last NI = 2000 iterations, no improvement of the best objective function value was obtained, or
- for the currently best value of the sum of completion times *SumC*(*BestSol*), the inequality *SumC*(*BestSol*) – *LB*< 1 holds.

The complete hybrid algorithm is given below as Algorithm 2, where *Rand*(0,1) denotes a uniformly distributed random number form the interval (0,1).

Algorithm 2. Hybridization of Harmony Search and Simulated Annealing

BEGIN
Build the HM
Initialize the parameters
Initialize *T*
count = 0; ∂ = 500;
Put X^{Best} (the best solution in HM) into π;
BestSol = *BestSASol* = π;
FOR *i* = (1 to *count*) do
  IF (*rand*(0,1) ≤ HMCR) THEN
    Choose two solution vectors denoted by υ_1 and υ_2 randomly from HM;
    IF (*rand*(0,1) ≤ PAR) THEN
      υ = apply a PMX crossover to υ_1 and υ_2;
      X^{new} = the best neighbor among the generated neighbors of υ;
      IF (*SumC*(X^{new}) < *SumC*(X^{worst})) THEN

```
                Replace X^worst by X^new //Updating the HM
            END IF
        END IF
    END IF
    ELSE
        π′ = the best neighbor among the generated neighbors of π;
        ΔC = SumC (π′) – SumC (π);
        prob = Rand (0,1);
        IF ((ΔC ≤ 0) or (prob < e^(-ΔC/T))) THEN
            π = π′; X^new = π′;
            IF (SumC( X^new ) < SumC( X^worst )) THEN
                Replace X^worst by X^new ; //Updating the HM
            END IF
            IF (SumC(π) < SumC (BestSASol)) THEN
                BestSASol = π;
            END IF
        END IF
        T = Update (T);
    END ELSE
    IF (SumC( X^Best ) < SumC(BestSol) )
        BestSol = X^Best ; Count = 0;
    END IF
    IF (SumC (BestSASol) - SumC(BestSol) ≥ ∂)
        π = BestSol;
    END IF
END FOR
Output BestSol together with its objective function value;
END.
```

6. Computational Results

Both heuristics SA and HHS have been implemented using the Java programming language. They have been run using JDK 1.3.0, with 2GB of memory available for working storage on a personal computer Intel(R) Core(TM) i5-2430M CPU @2.4GHz.

Generation of test instances

The performance of the proposed heuristics SA and HHS has been tested on the data generated in the same way as it was described in Hasani et al. (2013b). For the comparison of the obtained results, we used the same run time limit of (300/8)n seconds for the instances with $n \in \{8, 20, 50\}$ and 3600 seconds for the other instances.

For $n \in \{8, 20\}$, the data sets were generated for server load values ranging between 0.1 and 2 with 0.1 increments, i.e., for each $L \in \{0.1, 0.2, \ldots, 2\}$, the value s_j is distributed uniformly in (0, 100L). For each value of L, 10 instances were randomly generated with $p_j \overset{d}{=} \bigcup(0, 100)$, i.e., p_j is uniformly distributed in the interval (0, 100).

For $n \in \{50, 100, 200, 250\}$, 5 instances were generated for each $L \in \{0.1, 0.5, 0.8, 1, 1.5, 1.8, 2\}$ with $p_j \overset{d}{=} \bigcup(0, 100)$ and $s_j \overset{d}{=} \bigcup(0, 100L)$.

Comparative study

Before comparing both heuristics SA and HHS, some results from the initial tests are presented which motivated our decisions in HHS. We first examined the impact of HMS by a set of experiments. Five instances with 100 jobs and L = 1.0 have been chosen for these experiments. In these experiments, we used the following parameter values: HMCR = 0.89, PAR = 0.2. Table 2 presents the experimental results (average out of 5 runs) of HHS using different HMS parameter values, where ***ave*** $(\frac{\sum C}{LB})$ gives the average value of the ratios of the heuristic function value and the lower bound.

HMS	***ave*** $(\frac{\sum C}{LB})$
10	1.0494
20	**1.0443**
30	1.0447
40	1.4690
50	1.0474
60	1.0476
70	1.0467
80	1.0482
90	1.0485
100	1.0498

Table 2. Results of HSA with different parameter values of HMS.

Based on the experimental results in Table 1, HMS = 20 appears to be the most suitable parameter value compared to the other HMS values.

To find suitable values for HMCR and PAR, we did some experiments with several HMCR and PAR values. Five instances with 50 jobs and L = 1, as shown in Table 3, and five instances with 100 jobs and L = 1, as shown in Table 4, have been chosen for these experiments. In the literature, the recommended values range from 0.79 to 0.99 for HMCR and 0.1 to 0.3 for PAR, see Lee and Geem (2004).

HMCR	PAR	***ave*** $(\frac{\sum C}{LB})$
0.79	0.1	1.0623
	0.2	1.0621
	0.3	1.0622
0.89	0.1	1.0620
	0.2	1.0623
	0.3	**1.0620**
0.95	0.1	1.6249
	0.2	1.0627
	0.3	1.0620
0.99	0.1	1.0689
	0.2	1.0686
	0.3	1.0691

Table 3. Results with different parameter values of HMCR and PAR for 50 jobs and $L = 1$.

HMCR	PAR	***ave*** $(\frac{\sum C}{LB})$
0.79	0.1	1.0492
	0.2	1.0470
	0.3	1.0471
0.89	0.1	1.0495
	0.2	1.0477
	0.3	**1.0450**
0.95	0.1	1.0455
	0.2	1.0481
	0.3	1.0485
0.99	0.1	1.0468
	0.2	1.0470
	0.3	1.0473

Table 4. Results with different parameter values of HMCR and PAR for 100 jobs and $L = 1$.

From Table 3 and Table 4, we found that the best results were obtained when HMCR = 0.89 and PAR = 0.3. For the instances with a larger number of jobs different suitable parameter values may be found. However, their influence on the results is not substantial. Therefore, according to these results, we set HMCR as 0.89 and PAR as 0.3 for all tested instances.

Moreover, for all instances, it was counted in SA how often the particular neighborhoods have led to accepted neighbors with a better objective function value. For all instances considered in Figure 2, the "adjacent swap" operator has the largest contribution to the generation of better solutions. For the small instances with 8 and 20 jobs, on the next rank, the neighborhood operators such as "insert", "insert block" and "swap" have a rather impressive contribution. However, for the large instances, the contribution of the "adjacent swap" operator is more impressive (about 60 %) while the other operators have contributed far less. In particular, for the problems with a larger number of jobs, the "swap block" and "insert block + reverse" neighbors have only a small contribution. This shows that for the total completion time problem, the block neighbors cannot be as efficient as for the makespan problem, see Hasani et al. (2013b).

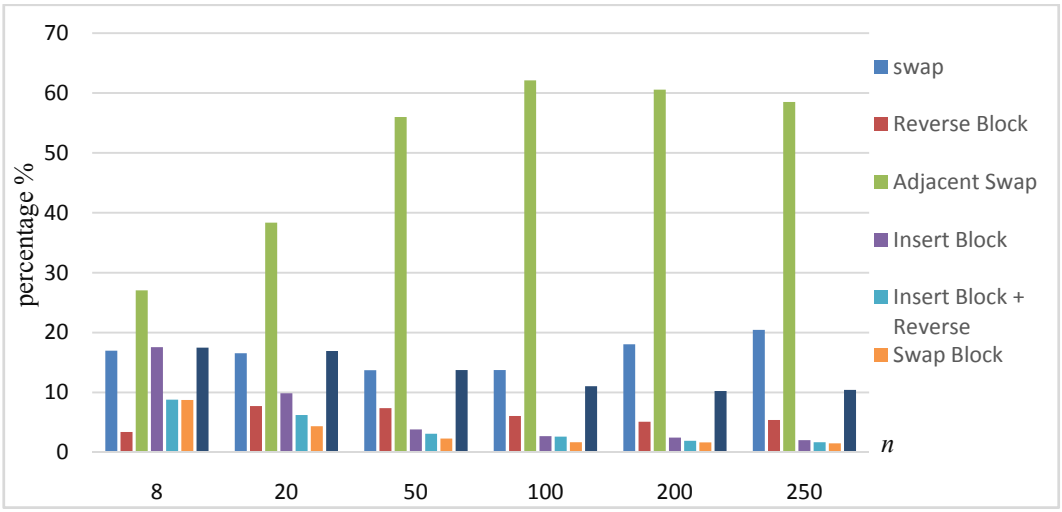

Figure 2 – Contribution of each of the neighborhood operators in producing solutions which have been accepted in SA.

Next, the results for both algorithms SA and HHS obtained for the instances with $n \leq 250$ and evaluated and compared with each other. In the following tables, we give the number of jobs n in the first column, the server load value L in the second column the heuristic H in the third column, in column 4 - 6, we give the minimum, average and maximum computational time per instance and in columns 7 – 9, we give the minimum, average and maximum values of the ratios of the heuristic function value and the lower bound.

n	*L*	*H*	*min time*	*ave time*	*max time*	*min* $(\frac{\sum C}{LB})$	*ave* $(\frac{\sum C}{LB})$	*max* $(\frac{\sum C}{LB})$
8	0.1	SA	9.49	10.47	13.39	1.00	1.00	1.00
		HHS	12.07	12.14	12.17	1.00	1.00	1.00
	0.5	SA	10.61	13.20	15.44	1.00	1.00	1.00
		HHS	12.04	12.11	12.15	1.00	1.00	1.00
	0.8	SA	10.97	13.78	15.54	1.00	1.00	1.00
		HHS	0.00	9.68	12.16	1.00	1.00	1.00
	1.0	SA	13.15	14.30	15.71	1.00	1.00	1.00
		HHS	11.87	12.01	12.17	1.00	1.00	1.00
	1.5	SA	12.98	14.97	15.92	1.00	1.00	1.00
		HHS	11.96	12.02	12.06	1.00	1.00	1.00
	1.8	SA	11.72	14.98	16.04	1.00	1.00	1.00
		HHS	0.02	9.48	11.96	1.00	1.00	1.00
	2.0	SA	12.51	14.88	16.54	1.00	1.00	1.00
		HHS	0.00	7.14	11.97	1.00	1.00	1.00

Table 5. Results for instances with 8 jobs.

In Table 5, the results are shown for n=8 jobs. Both algorithms could find optimal solutions for all instances. However, the HHS algorithm was faster. HHS could obtain an optimal solution within no more than 12.17 seconds, while SA obtained it only within no more than 16.54 seconds.

n	***L***	***H***	***min time***	***ave time***	***max time***	***min*** $(\frac{\sum C}{LB})$	***ave*** $(\frac{\sum C}{LB})$	***max*** $(\frac{\sum C}{LB})$
20	0.1	SA	41.4	44.1	48.2	1.0000	1.0033	1.0052
		HHS	0.2	36.5	62.8	1.0000	1.0033	1.0052
	0.5	SA	55.5	60.4	65.1	1.0099	1.0253	1.0659
		HHS	30.7	37.1	44.6	1.0099	1.0253	1.0659
	0.8	SA	56.3	62.7	71.6	1.0129	1.0278	1.0601
		HHS	31.1	38.8	47.2	1.0129	1.0270	1.0601
	1.0	SA	57.6	64.9	68.6	1.0468	1.0647	1.0800
		HHS	30.4	40.7	46.8	1.0453	1.0644	1.0800
	1.5	SA	61.4	66.0	67.6	1.0265	1.0629	1.0986
		HHS	29.7	32.0	36.5	1.0265	1.0629	1.0986
	1.8	SA	52.7	58.4	64.2	1.0032	1.0326	1.0532
		HHS	29.2	29.9	30.3	1.0032	1.0326	1.0532
	2.0	SA	62.3	70.9	79.4	1.0095	1.0286	1.0710
		HHS	29.3	29.6	30.0	1.0095	1.0286	1.0710

Table 6. Results for instances with 20 jobs.

In Table 6, the results are presented for n=20 jobs. SA and HHS gave almost exactly the same results except for L = 1.0 and L = 0.8, where HHS worked better. This means that for the harder instances, the HHS algorithm has obtained smaller deviations from the lower bound. Also, the HHS algorithm worked faster than SA.

n	***L***	***H***	***min time***	***ave time***	***max time***	***min*** $(\frac{\sum C}{LB})$	***ave*** $(\frac{\sum C}{LB})$	***max*** $(\frac{\sum C}{LB})$
50	0.1	SA	369	459	586	1.0000	1.0015	1.0033
		HHS	373	1267	1586	1.0000	1.0015	1.0033
	0.5	SA	372	547	734	1.0031	1.0133	1.0205
		HHS	1341	1645	2113	1.0032	1.0131	1.0189
	0.8	SA	463	510	619	1.0171	1.0283	1.0406
		HHS	1094	1181	1274	1.0090	1.0234	1.0359
	1.0	SA	444	529	759	1.0500	1.0653	1.0893
		HHS	949	991	1042	1.0444	1.0615	1.0868
	1.5	SA	331	436	509	1.0261	1.0465	1.0842
		HHS	956	1002	1068	1.0266	1.0433	1.0792
	1.8	SA	336	410	540	1.0206	1.0456	1.0652
		HHS	901	1023	1104	1.0206	1.0444	1.0652
	2.0	SA	338	420	519	1.0211	1.0518	1.0985
		HHS	845	881	934	1.0206	1.0501	1.0946

Table 7. Results for instances with 50 jobs.

For n = 50, as it can be seen from Table 7, the HHS algorithm outperformed SA substantially.

n	***L***	***H***	***min time***	***ave time***	***max time***	***min*** $(\frac{\sum C}{LB})$	***ave*** $(\frac{\sum C}{LB})$	***max*** $(\frac{\sum C}{LB})$
100	0.1	SA	1792	1865	1960	1.0001	1.0002	1.0004
		HHS	3600	3600	3600	1.0002	1.0003	1.0005
	0.5	SA	1843	2065	2212	1.0027	1.0069	1.0110
		HHS	3600	3600	3600	1.0015	1.0052	1.0087
	0.8	SA	1592	1764	2086	1.0133	1.0200	1.0390
		HHS	1655	2831	3600	1.0090	1.0160	1.0357
	1.0	SA	1925	2041	2332	1.0229	1.0412	1.0665
		HHS	3600	3600	3600	1.0204	1.0386	1.0634
	1.5	SA	1525	1778	1886	1.0241	1.0451	1.0690
		HHS	3600	3600	3600	1.0197	1.0416	1.0626
	1.8	SA	1561	1779	2432	1.0324	1.0439	1.0540
		HHS	3600	3600	3600	1.0296	1.0406	1.0517
	2.0	SA	1561	1833	2349	1.0067	1.0276	1.0459
		HHS	1769	2575	3600	1.0066	1.0255	1.0412

Table 8. Results for instances with 100 jobs.

Table 8 gives the results for n = 100 jobs. The HHS algorithm was always superior to SA with the exception for L = 0.1, where SA was slightly superior.

n	***L***	***H***	***min*** $(\frac{\sum C}{LB})$	***ave*** $(\frac{\sum C}{LB})$	***max*** $(\frac{\sum C}{LB})$
200	0.1	SA	1.0002	1.0005	1.0006
		HHS	1.0004	1.0010	1.0016
	0.5	SA	1.0077	1.0101	1.0130
		HHS	1.0075	1.0084	1.0097
	0.8	SA	1.0134	1.0150	1.0173
		HHS	1.0105	1.0134	1.0157
	1.0	SA	1.0156	1.0246	1.0361
		HHS	1.0115	1.0196	1.0303
	1.5	SA	1.0488	1.0667	1.0919
		HHS	1.0484	1.0655	1.0958
	1.8	SA	1.0186	1.0322	1.0506
		HHS	1.0166	1.0304	1.0466
	2.0	SA	1.0180	1.0425	1.0613
		HHS	1.0164	1.0410	1.0613

Table 9. Results for instances with 200 jobs.

In Table 9, we give the results for n = 200 with a time limit of 3600 seconds. For most instances, a smaller deviation from the lower bound was obtained by using HHS. The only exception are instances with L = 0.1 and L = 0.5, where SA worked better.

n	***L***	***H***	***min*** $(\frac{\sum C}{LB})$	***ave*** $(\frac{\sum C}{LB})$	***max*** $(\frac{\sum C}{LB})$
250	0.1	SA	1.0006	1.0008	1.0012
		HHS	1.0015	1.0019	1.0023
	0.5	SA	1.0051	1.0127	1.0198
		HHS	1.0068	1.0133	1.0197
	0.8	SA	1.0121	1.0178	1.0232
		HHS	1.0147	1.0190	1.0313
	1.0	SA	1.0178	1.0237	1.0285
		HHS	1.0196	1.0234	1.0274
	1.5	SA	1.0260	1.0513	1.0910
		HHS	1.0246	1.0495	1.0921
	1.8	SA	1.0244	1.0343	1.0433
		HHS	1.0229	1.0318	1.0390
	2.0	SA	1.0196	1.0366	1.0511
		HHS	1.0205	1.0358	1.0496

Table 10. Results for instances with 250 jobs.

For n = 250, a time limit of 3600 seconds was used. As illustrated in Table 10, for the instances with L = {1.0, 1.5, 1.8, 2.0}, HHS works better.

7. Concluding Remarks

In this paper, we considered the minimization of mean flow time for the problem of scheduling a set of jobs on two parallel machines with a single server. We developed a simulating annealing algorithm and a hybridization of harmony search and simulated annealing. The two algorithms have been calibrated for the same way of generating the instances as in previous works and compared with each other on a set of instances up to 250 jobs. Both algorithms performed very well in terms of the deviation from the known lower bounds. However, the HHS algorithm outperformed SA in most cases.

References

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

[2] Brucker P., Dhaenens-Flipo C., Knust S., Kravchenko S.A., Werner F. (2002) Complexity results for parallel machine problems with a single server. Journal of Scheduling 5, 429-457.

[3] Kravchenko S.A., Werner F. (2001) A heuristic algorithm for minimizing mean flow time with unit setups, Information Processing Letters 79, 291-296.

[4] Wang G., Cheng T.C.E. (2001) An approximation algorithm for parallel machine scheduling with a common server, Journal of the Operational Research Society 52, 234-237.

[5] Weng M.X., Lu J., Ren H. (2001) Unrelated parallel machine scheduling with setup consideration and a total weighted completion time objective, Int. J. Production Economics 70, 215-226.

[6] Dunstall S., Wirth A. (2005) Heuristic methods for the identical parallel machine flowtime problem with set-up times. Computers & Operations Research 32, 2479–2491.

[7] Azizoglu M., Webster S. (2003) Scheduling parallel machines to minimize weighted flowtime with family set-up times. Int. J. Production Research 41, 1199–1215.

[8] Guirchoun S., Martineau P., Billaut J.-C. (2005) Total completion time minimization in a computer system with a server and two parallel processors. Computers & Operations Research 32, 599–611.

[9] Hasani K., Kravchenko S.A., Werner F. (2013a) Minimizing Mean Flow Time for the Two-Machine Scheduling Problem with a Single Server. Preprint 12/13, Faculty of Mathematics, Otto-von-Guericke-University Magdeburg, 14 pages.

[10] Kirkpatrick S., Gelatt Jr. C.D., and Vecchi M.P. (1983) Optimization by simulated annealing. Science, 200, No. 4598, 671 - 680.

[11] Lee K.S., Geem Z.W. (2004) A new structural optimization method based on the harmony search algorithm, Journal of Computers and Structures 82, 781-798.

[12] Ingram G., Zhang T. (2009) Overview of applications and developments in the harmony search algorithm, in: Z.W. Geem (Ed.) Music-inspired harmony search algorithm, Springer Berlin / Heidelberg, pp. 15-37.

[13] Geem Z.W., Kim J.H., Loganathan G.V. (2001) A new heuristic optimization algorithm: Harmony search, Journal of Simulation 76, 60-68.

[14] Yang X.S. (2008) Nature-inspired metaheuristic algorithms, Luniver Press.

[15] Geem Z.W. (2008) Novel derivative of harmony search algorithm for discrete design variables. Applied Mathematics and Computation 199, 223–230.

[16] Mahdavi M., Fesanghary M., Damangir E. (2007) An improved harmony search algorithm for solving optimization problems, Applied Mathematics and Computation 188, 1567-1579.

[17] Werner F. (2011) Genetic algorithms for shop scheduling problems: A survey, Preprint 11/31, Faculty of Mathematics, Otto-von-Guericke-University Magdeburg, 65 pages.

[18] Hasani K., Kravchenko S.A., Werner F. (2013b) Two heuristics for minimizing the makespan for the two-machine scheduling problem with a single server. Preprint 08/13, Faculty of Mathematics, Otto-von-Guericke-University Magdeburg, 20 pages.