Computers & Operations Research 39 (2012) 1213–1217

Contents lists available at ScienceDirect

Computers & Operations Research

journal homepage: www.elsevier.com/locate/caor

Iterated local search and very large neighborhoods for the parallel-machines total tardiness problem

F. Della Croce [a], T. Garaix [a], A. Grosso [b,*]

[a] DAI, Politecnico di Torino, Italy
[b] Dip. di Informatica, Università di Torino, Italy

ARTICLE INFO

Available online 23 October 2010

Keywords:
Scheduling
Tardiness
Parallel machines
Very large neighborhood

ABSTRACT

We present computational results with a heuristic algorithm for the parallel machines total weighted tardiness problem. The algorithm combines generalized pairwise interchange neighborhoods, dynasearch optimization and a new machine-based neighborhood whose size is non-polynomial in the number of machines. The computational results significantly improve over the current state of the art for this problem.

© 2010 Elsevier Ltd. All rights reserved.

1. Introduction

In the $Pm|\,|\sum_j w_jT_j$ problem a set of jobs $N=\{1,2,\ldots,n\}$ is given, with processing times p_j, weights w_j and due dates d_j specified for each $j\in N$. The jobs are to be processed in a schedule S on a set $M=\{1,2,\ldots,m\}$ of identical parallel machines so that their completion times C_j minimize the objective function

$$T(S)=\sum_{j=1}^{n} w_jT_j=\sum_{j=1}^{n} w_j\max\{C_j-d_j,0\}.$$

For $m=1$ the problem reduces to the single-machine total tardiness problem, that is well-studied and solved in both the exact and heuristic frameworks—we refer to [4,11,13,14,5,7,8] for recent developments.

The literature seems to be fairly limited for the problem with parallel machines; the most recent references are [2,3,9,13,14] to the authors' knowledge.

Iterated Local Search (ILS, see [10] for a survey) is a local search framework that can be seen as a tradeoff between the naive multistart and complex metaheuristics. In multistart a local search driven optimization starts several times (often a huge number of times) from randomly generated initial solutions, in order to achieve a wide exploration of the solutions set. In metaheuristics a number of sophisticated devices (genetic crossover, short or long-term memory, etc) are employed in order to escape poor local optima. In ILS the search is simply restarted from a slightly perturbed version of the best-known solution. With this type of restart, the starting point of each local search is not completely random, and the perturbation – called "kick" – aims at projecting the search "not too far" from previously explored local optima, without completely loosing their partially optimized structure.

Very Large Neighborhood Search (VLNS) denotes local search methods that define and explore complex neighborhoods for combinatorial optimization problems; such neighborhoods are characterized by having an exponential number of neighbor solutions – with respect to the problem size – but can be explored in polynomial time by means of exact or heuristic procedures (see [1]).

ILS is often successfully coupled with VLNS, hence moving the complexity of the search from the overall algorithm to the neighborhood exploration. We refer to [5,7] for a successful application of such a VLNS technique (called *dynasearch*) to the $1|\,|\sum w_jT_j$ problem.

Rodrigues et al. [13] proposed a simple and quite effective ILS algorithm for the $Pm|\,|\sum_j w_jT_j$ problem, using a local search based on pairwise interchange operators. That algorithm was tested on a batch of 100 instances with $n=40{,}50$ and $m=2{,}4$ derived from a subset of the $1|\,|\sum w_jT_j$ problem instances available in the OR-library.[1] Notice that, on that batch, the algorithm was able to detect all but one optimal solutions.

This paper aims at defining an improved ILS algorithm for $Pm|\,|\sum_j w_jT_j$ by incorporating VLNS techniques. Particularly, we introduce a dynasearch optimization on each machine in the shop and a new "Very Large" neighborhood whose size is non-polynomial *in the number of machines.*

We illustrate the basic building blocks of the algorithm and present computational experiments for assessing their effectiveness in Section 2. The complete algorithm is described in Section 3, where also the computational results are discussed. The proposed ILS algorithm outperforms the ILS of [13] on instances with a number of jobs n ranging from 40 to 300, and a number of machines

* Corresponding author.
E-mail address: grosso@di.unito.it (A. Grosso).

[1] http://people.brunel.ac.uk/~mastjjb/jeb/info.html

0305-0548/$ - see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cor.2010.10.017

m ranging from 2 to 20. The advantage of the new algorithm grows on instances with large m thanks to the new neighborhood.

2. The basic neighborhoods

2.1. Generalized pairwise interchanges

The well-known GPI operators work on a sequence of jobs σ producing a new sequence σ'. Let $\sigma = \alpha i \pi j \omega$, with jobs i and j in position k and l, respectively. The most common GPI operators are

(1) *Swap* $\alpha i \pi j \omega \rightarrow \alpha j \pi i \omega$ (π may be empty);
(2) *Forward insertion* $\alpha i \pi j \omega \rightarrow \alpha \pi j i \omega$;
(3) *Backward insertion* $\alpha i \pi j \omega \rightarrow \alpha j i \pi \omega$;
(4) *Twist* $\alpha i \pi j \omega \rightarrow \alpha j \overline{\pi} i \omega$ with $\overline{\pi} = \pi$ reversed.

The implementation of such operators is straightforward in single-machine sequencing problems with regular cost functions, since the machine is never idle and the sequence σ *is* the schedule. The so-called GPI dynasearch neighborhood for single-machine sequencing problems combines possibly many *independent* moves of types (1)–(4); two moves are said to be *independent* if the pairs of positions (k,l) and (p,q) on which they act are non-overlapping, i.e $\max\{k,l\} < \min\{p,q\}$. In a single-machine environment with an additive objective function the contributions of independent moves combine additively, and the best set of independent moves can be worked out by dynamic programming (see [5] for details). A GPI dynasearch neighborhood exploration for an n jobs sequence requires $\mathcal{O}(n^2)$ time with its best implementation (see [7]).

In parallel-machines environments, GPI operators can be applied provided that a sequence σ can be converted to a schedule. Rodrigues et al. [13] proposed a simple yet quite effective ILS algorithm for the $Pm| |\sum_j w_j T_j$ problem. The algorithm applies GPI operators – limited to (1)–(3) in their implementation – on a sequence of jobs; the schedule on parallel machines associated with this sequence $\sigma = (j_1, j_2, \ldots, j_n)$ is computed from scratch by means of the most natural *dispatching rule*: assign the next job in the sequence to the earliest available machine. The neighborhood exploration is performed with a first-improve strategy, and frequent restarts are applied (one kick every five complete descents).

Whereas the basic GPI neighborhood can be easily adapted to the parallel machines environment, this is not the case for the GPI dynasearch neighborhood: since the job starting times are determined by applying the dispatching rule, the contribution of independent moves is no longer purely additive. Rodrigues et al. [13] do not provide a different notion of independent moves, neither it is easy to see an obvious one.

2.2. Integrating GPIs on parallel machines and dynasearch

A possible drawback of the basic GPI neighborhood is that, in a parallel environment, the working sequence on each single machine is poorly optimized, since the machine-sequencing criterion is extremely crude. We then investigated the opportunity of adding a single-machine optimization phase through the use of a dynasearch neighborhood. We tested the following algorithms, called A1 and A2, respectively, on a set of random instances.

Algorithm A1. The GPI iterated local search of [13] (kindly provided by the authors).

Algorithm A2. The same algorithm, where after building a schedule by the dispatching rule, each single-machine is optimized by a full descent using the GPI dynasearch neighborhood where moves (1)–(4) are used.

Table 1
Basic GPI local search (Algorithm 1) and GPI+dynasearch (Algorithm 2). Comparison for $n=100$, $m=4$.

R	T	Algorithm A2			Algorithm A1		
		CPU_{avg}	$Ndesc_{avg}$	#Bests	CPU_{avg}	$Ndesc_{avg}$	#Bests
0.2	0.2	21.17	3.60	0	1.36	4.40	0
0.2	0.4	27.65	3.20	0	7.10	27.00	0
0.2	0.6	1164.99	147.80	1	301.41	1266.40	0
0.2	0.8	2246.95	252.00	1	540.10	3793.40	0
0.2	1.0	1690.30	205.20	2	723.50	6689.40	0
0.4	0.2	121.81	21.00	0	5.43	22.60	0
0.4	0.4	46.43	4.80	0	25.07	99.20	0
0.4	0.6	752.06	90.80	2	1167.06	5924.60	1
0.4	0.8	1662.31	178.60	1	1057.19	8114.60	4
0.4	1.0	1370.86	165.20	2	976.78	8721.60	3
0.6	0.2	4.00	0.00	0	0.07	0.00	0
0.6	0.4	143.21	19.00	0	163.62	614.40	0
0.6	0.6	414.91	45.00	2	1472.98	8029.20	0
0.6	0.8	2075.77	223.00	3	2085.35	14456.20	2
0.6	1.0	901.48	91.40	3	2178.29	18166.40	2
0.8	0.2	4.20	0.00	0	0.07	0.00	0
0.8	0.4	546.14	84.40	0	210.41	890.80	1
0.8	0.6	1808.82	196.40	2	1781.18	10169.60	2
0.8	0.8	2378.09	244.40	4	1562.70	10504.00	1
0.8	1.0	3004.82	339.80	3	1149.06	9383.20	1
1.0	0.2	4.05	0.00	0	0.07	0.00	0
1.0	0.4	591.58	84.00	0	420.00	2359.80	0
1.0	0.6	1459.21	147.00	3	1855.63	10910.00	2
1.0	0.8	1015.41	107.20	5	2073.00	13795.40	0
1.0	1.0	1879.91	210.40	3	2338.92	17907.40	2

All random instances considered in this work are adapted from the well-established literature on tardiness problems in single-machine environments. The single-machine instances are characterized by uniformly distributed random data with processing times p_i and weights w_i from [1,100], and due dates from the uniform distribution $[(1-T-R/2)\sum_{i=1}^{n} p_i, (1-T+R/2)\sum_{i=1}^{n} p_i]$. The two real-valued parameters R, T are called *due date range* and *tardiness factor*—they determine the practical difficulty of the instances, accordingly with (among others) Refs. [6,12]. The due dates are adapted to the parallel machines case by scaling the due dates by $\frac{1}{m}$ (rounding down the obtained values).

In order to test A1 and A2 we considered a batch of 125 random instances with $n=100$, $m=4$. We recorded the performances of the algorithms in terms of time spent for reaching the best solution and number of local search descents performed. For both algorithms we allowed 1 h of CPU time. Table 1 points to a comparison of the computational costs of the two algorithms in terms of CPU time and number of descents, detailing them by (R,T) pairs – each (R,T) "class" is made of five instances. Out of the 125 instances in the batch, Algorithm A2 delivered better solutions in 37 cases, and worse solutions in 27 (columns labeled "#Bests"). The higher number of better solutions comes at the cost of higher CPU times to be spent in the search. The number of descents required to reach the best solution is always consistently less for Algorithm A2 than for Algorithm A1, but Algorithm A2 – quite expectedly – exhibits in most cases higher CPU times, since every solution undergoes a full dynasearch descent on each machine. Anyway, in the details of the tests we were able to observe that on 18 instances of the batch, Algorithm A2 finds a better solution *and* requires less CPU time to reach the optimum; this behaviour comes out with dramatic evidence on some classes of instances like $R=0.6$, $T=0.6$, and the classes with $R=1.0$. This test suggests that an effort for keeping highly optimized sequences on the machines can be worth, if a clever search strategy can be developed in order to limit the growth of the CPU time