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

F Della Croce, Thierry Garaix, A. Grosso

ITERATED LOCAL SEARCH AND VERY LARGE NEIGHBORHOODS FOR THE PARALLEL-MACHINES TOTAL TARDINESS PROBLEM

F. DELLA CROCE, T. GARAIX, A. GROSSO

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.

1. Introduction

In the $Pm|\ |\sum_j w_jT_j$ problem a set of jobs $N = \{1, 2, \ldots, n\}$ are given, with their processing times p_j, weights w_j and due dates d_j. 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] and [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 $Pm|\ |\sum_j w_j T_j$ 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 $1||\sum w_j T_j$ 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 $Pm|\ |\sum_j w_j T_j$ 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 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* $\alpha i \pi j \omega \to \alpha j \pi i \omega$ (π may be empty);
(2) *Forward insertion* $\alpha i \pi j \omega \to \alpha \pi j i \omega$;
(3) *Backward insertion* $\alpha i \pi j \omega \to \alpha j i \pi \omega$.
(4) *Twist* $\alpha i \pi j \omega \to \alpha j \bar{\pi} i \omega$ with $\bar{\pi} = \pi$ 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]

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

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 1 and 2 respectively.

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.

We considered a batch of 125 randomly instances with $n = 100$, $m = 4$; we refer to Section 3.3 for details on the generation scheme. Two parameters R, T determine the practical difficulty of the instances. 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 one hour 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. 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

2.3. **A parallel-machines neighborhood.** A simple notion of independent moves arises if, instead of using the dispatch rule on a sequence, one works directly on the jobs-to-machines assignment. Assume a schedule is given. We consider, for a given pair of machines m_1, m_2, the respective job working sequences σ_1, σ_2 and the following moves:

(a) extract a job j from σ_1 and insert it into σ_2;
(b) extract a job j from σ_2 and insert it into σ_1;
(c) extract jobs $i \in \sigma_1$, $j \in \sigma_2$ and insert i in $\sigma_2 - \{j\}$, and j in $\sigma_1 - \{i\}$.

The insertion position for a job j in a sequence σ is chosen in linear time, so that $\sigma = \alpha j \omega$ and $T(\alpha j \omega)$ is as small as possible. We note that if such moves are executed on disjoint pairs of machines (m_1, m_2), (m_3, m_4), ..., they are *independent* moves, in the sense that their contributions to the schedule's cost combine additively. Hence a neighborhood whose size is non-polynomial in the number of machines can be defined as follows, for a given schedule.

(1) For each pair of machines (m_1, m_2), compute the maximum decrease in tardiness Δ_{m_1,m_2} that can be obtained by applying moves (a), (b) and (c) to σ_1, σ_2, for all $j \in \{\sigma_1\} \cup \{\sigma_2\}$.

(2) Build a weighted improvement graph $G(M, E)$ where
$E = \big\{\{m_1, m_2\} \colon \Delta_{m_1,m_2} > 0\big\}$.

(3) A neighbor schedule is generated by taking a matching on G and executing the moves associated to the matching edges.

Note that, for each pair of machines:

- no more than $\mathcal{O}(n)$ jobs have to be considered for moves (a) and (b);
- no more than $\mathcal{O}(n^2)$ pairs i, j have to be considered for move (c);
- evaluating each move of type (a), (b), (c) requires $\mathcal{O}(n)$ operations for each job (or job pair) — note that the schedule defines the machine to which the job is assigned.

Hence evaluating all the possible moves (a), (b), (c) requires $\mathcal{O}(n^3)$ operations. The best neighbor schedule can be computed by taking a maximum weighted matching in G; the whole process for building G and selecting the matching can be implemented with running time

$$\mathcal{O}(n^3) + \mathcal{O}(m^3) \approx \mathcal{O}(n^3) \qquad (\text{as } m < n \text{ in non trivial instances}).$$

Simple testing showed that the solutions provided by the GPI ILS technique of [13] are often not locally optimal with respect to the parallel machines neighborhood. Particularly, applying one search of the parallel machine neighborhood on the best solution provided by Algorithm A1 after a 1-hour run, we found 11 improvements (over 125 solutions) for the instances in the $n = 100, m = 10$ batch; the number of improvement rises to 96 for the $n = 300, m = 20$ batch.

The exploration of the parallel neighborhood is usually fast (less than 0.6 seconds on the $n = 300$ instances). Hence, we keep the parallel machines neighborhood as a cheap and useful tool whose impact becomes more and more important on instances with many machines.

3. Combining different neighborhoods

3.1. **Neighborhoods and refinements.** In view of the experimental observations reported in the previous section, a careful combination of GPI moves, dynasearch optimization and the parallel machines neighborhood can be the key instrument for handling larger instances of the $Pm| \mid \sum_j w_j T_j$ problem. We used three neighborhoods called N_1, N_2, N_3.

- Neighborhood N_1 is the GPI neighborhood of Rodrigues et al. [13]. Note that in [13] a somewhat sophisticated rule is used for breaking ties in choosing the best neighbor. We avoided it in favor of a random tie-breaking rule.
- Neighborhood N_2 is the parallel-machines neighborhood described in Section 2.
- Neighborhood N_3 is a GPI neighborhood where every neighbor schedule generated by a GPI operator is improved by a dynasearch descent applied on each machine schedule.

In order to reduce the computational effort, we applied the following refinements to $N1$ and N_3. Following [13], the neighborhood search proceed by first-improve, hence the first improving neighbor is adopted as new solution. We observed that often profitable moves happen between jobs that appear in relatively "close" positions. Hence the GPI operators are applied in "stages"; each stage applies the GPI operators between jobs in positions

$$i \text{ and } i + \gamma \mod n, \qquad \text{for } i = 1, \ldots, n, \ \gamma \text{ fixed.}$$

In exploring N_1, N_3, at each successive stage γ is set to $1, 2, \ldots, n-1$. The improving neighbor is often found at early stages.

Accordingly with [13], a full exploration of a N_1 neighborhood is accomplished in $\mathcal{O}(n^3 \log m)$ operations. An exploration of N_2 requires $\mathcal{O}(n^3)$ operations (see Section 2). Exploring N_3 takes up to $\mathcal{O}(n^4)$ operations.

3.2. **The algorithm.** We now describe the complete proposed algorithm. It uses all three neighborhoods N_1, N_2, N_3. Among these neighborhoods, the local search phase for N_1, N_2 is exploited to a full descent, while for N_3 it is limited to a single neighborhood exploration because of the higher computational cost of such procedure. This is denoted in the pseudocode by the operations called FullDescent and Search.

The search starts from the usual EDD (dispatched) sequence, which is often accepted in literature as a reasonable quick-and-dirty starting point. A full descent of N_1 performs the first optimization in the main loop, then N_2 and one exploration of N_3 are invoked iteratively one after the other, as long as the iteration is profitable. The Kick phase — i.e. perturbation of the best-known solution — consists of a limited number of random swaps, removal and insertions performed among different machines.

The algorithm uses a time-limit as stopping criterion, since we considered it the most simple and tunable one.

Algorithm A3

Set $S^* := S :=$ ⟨The EDD dispatched sequence⟩;
Set IterCount := 0;
repeat
    Improve S by executing dynasearch on each machine;
    $S^1 :=$ FullDescent(S, N_1);
    repeat
        $S^2 :=$ FullDescent(S^1, N_2);
        $S^3 :=$ Search(S^2, N_3);
        if $T(S^3) < T(S^2)$ **then**
            Set $S^1 := S^3$;
        end if
    until $\langle N_3$ failed improving $S^2\rangle$;
    if $T(S^3) \geq T(S^*)$ **then**
        Set IterCount := IterCount + 1;
    else
        Set $S^* := S^3$;
        Set IterCount := 0;
    end if
    if $\langle N_2$ and N_3 failed to improve $S^1\rangle$ **then**
        if IterCount > MaxNoImprove **then**
            $S :=$ Kick(S^*);
            IterCount := 0;
        else
            $S :=$ Kick(S^3);
        end if
    else
        $S := S^3$;
    end if
until ⟨Time-limit exceeded⟩

Following [13], a kick is executed each MaxNoImprove non-improving iterations, with MaxNoImprove = 5. Also, as a further refinement, the number of stages in the exploration on N_3 was fixed to a maximum of $\gamma_{\max} = 5$; note that

a higher value of $\gamma_{\max}$ causes more time to be spent in exploring N_3 and, correspondingly, a lower number of kicks executed in the allowed time limit. The values MaxNoImprove $= 5$, $\gamma_{\max} = 5$ gave the best results in some preliminary tests — only a modest amount of testing was needed to identify this value, without need for extensive calibration. The value $\gamma_{\max} = 5$ actually lowers the time spent for exploring N_3 to $\mathcal{O}(\gamma_{\max} n^3)$.

3.3. **Evaluation of the algorithm.** We tested the hybrid ILS algorithm on batches of random instances 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 a uniform distribution whose bounds are determined by two parameters R, T called *due date range* and *tardiness factor* — see for example References [6, 12]. Specifically, the d_i values are randomly drawn from

$$[(1 - T - R/2) \textstyle\sum_{i=1}^{n} p_i, (1 - T + R/2) \sum_{i=1}^{n} p_i].$$

For fixed n, m, we considered five instances for each (R, T) pair, with $R, T \in \{0.2, 0.4, 0.6, 0.8, 1.0\}$ — 25 (R, T) pairs, and 125 instances. For $n = 40, 50, 100$ we used times, weights, and due dates from the 375 single-machine OR-library instances. For $n = 300$ we used the instances from Tanaka et al. [15]. The number of machines m was set to 2, 4 and 10, and pushed up to 20 for the largest instances. The due dates are adapted to the parallel machines case by scaling the due dates by $\frac{1}{m}$ (rounding down the obtained values).

Tables 2–4 focus on the comparison of Algorithm A1 by Rodrigues et al. [13] and Algorithm A3. Accordingly with [13] we allowed at most one hour CPU time to each test and report an aggregated comparison in Table 2 [2]; the results obtained within shorter CPU times are also presented in Tables 3 and 4. The total number of instances tested was 1625. For both A1 and A3 the results of a single run are reported; athough the kick phase accounts for some nondeterminism in A3, we did not observe it delivering significantly different tadiness values in different runs, as far as the time limits reported in the tables are allowed.

In Table 2 we report information on the behaviour of the algorithms running with a 3600 seconds time limit. The "dev" columns report the percentage average and maximum deviations of the objective function delivered by A1 and A3 with respect to the best found value — the "best" value is defined as the minimum between the tardiness value obtained by the two algorithms within the allowed 1-hour run. The column $\#_{\text{best}}$ counts the number of instances (out of the 125) where Algorithm A1 or Algorithm A3 delivered a better solution. CPU_{best} reports the average time-to-best for both algorithms and N_{desc} the average number of descent performed. The performances of the two algorithms are basically comparable for "small" instances (say for $n = 40, 50$); with such limited problem sizes, both A1 and A3 often find an optimal solution.

Tables 3 and 4 compare the behaviour of the two algorithms for different values of the time limits enforced. The $n = 300$ cases are not reported for time limits ≤ 30 secs (Table 3) because in several instances such time limits where not enough to perform a full execution of A3 — this is due to the heavier computational requirements of the dynasearch component of A3. Aside from this limitation, A3 is seen to strongly outperform A1 in terms of solution quality.

A3 becomes apparently the best option for large n ($n = 100, 300$), and especially for instances with a large number of machines ($m = 10, 20$). On the latter

[2] per-instance results are available as a compressed file at
`www.di.unito.it/~grosso/solution.tar.7z`

instances the key factors for the success of A3 are the ability to exploit the parallel-machines "very large" neighborhood, and the powerful dynasearch neighborhood for optimizing each machine sequence.

References

[1] Ahuja R. K., Ergun Ö., Orlin J. B., Punnen A. P., A survey of very large-scale neighborhood search techniques, *Discrete Applied Mathematics* **123**, 75–102 (2002).

[2] Anghinolfi D., Paolucci M., Parallel machine total tardiness scheduling with a new hybrid metaheuristic approach, *Computers and Operations Research* **34**, 3471–3490 (2007).

[3] Bilge U., Kyraç S., Kurtulan F., Pekgun M., A tabu search algorithm for parallel machine total tardiness problem, *Computers and Operations Research* **31**, 397–414 (2004)

[4] Bigras L. Ph., Gamache M., Gilles S. Time-Indexed Formulations and the Total Weighted Tardiness Problem, *INFORMS Journal on Computing* **20**, 133–142 (2008).

[5] Congram R., Potts C. N., van de Velde S., An iterated dynasearch algorithm for the single machine total weighted tardiness problem, *INFORMS Journal on Computing,* **14**, 52–67 (2002).

[6] Crawuels H. A. J., Potts C. N., Van Wassenhove L. N., Local search heuristics for the single machine total weighted tardiness scheduling problem, *INFORMS Journal on Computing* **10**, 341–350 (1998).

[7] Ergun Ö., Orlin J. B., Fast neighborhood search for the single machine total weighted tardiness problem, *Operations Research Letters* **34**, 41–45 (2006).

[8] Grosso A., Della Croce F., Tadei R., An enhanced dynasearch neighborhood for the single machine total tardiness problems, *Operations Reasearch Letters,* **32**, 68–72 (2004).

[9] Koulamas C., Decomposition and hybrid simulated annealing heuristics for the parallel-machine total tardiness problem, *Naval Research Logistics* **44**, 109–125 (1997).

[10] Lourenco H. R., Stützle T., Iterated local search, in *Handbook of MetaheuRistics,* F. Glover and G. Kochenberger, ISORMS 57, pp. 321–253 (2002).

[11] Pan Y., Shi L., On the equivalence of the max-min transportation lower bound and the time-indexed lower bound for single machine scheduling problems, *Mathematical Programming,* **100**, 543–559 (2007).

[12] Potts C. N., Van Wassenhove L. N., A branch and bound algorithm for the total weighted tardiness problem, *Operations Research,* **33**, 363–377 (1985).

[13] Rodrigues R., Pessoa A., Uchoa E., Poggi de Aragão M. Heuristic algorithm for the parallel machine total weighted tardiness scheduling problem, internal report 10/2008, Universidad Federal Fluminense `http://www.producao.uff.br/conteudo/rpep/volume82008/RelPesq_V8_2008_10.pdf`

[14] Rodrigues R., Pessoa A., Uchoa E., Poggi de Aragão M., Algorithms over Arc-time Indexed Formulations for Single and Parallel Machine Scheduling Problems, Optimization-Online, `http://www.optimization-online.org/DB_HTML/2008/06/2022.html`.

[15] Tanaka S., Fujikuma S., Araki M. An exact algorithm for single-machine scheduling without machine idle time, *Journal of Scheduling,* **12**, 575–593 (2009).

		Algorithm 2			Algorithm 1		
R	T	CPU_{avg}	$N\text{desc}_{\text{avg}}$	#Bests	CPU_{avg}	$N\text{desc}_{\text{avg}}$	#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

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

n	m	$\text{dev}_{\%}$				# best		CPU_{best}(sec)		N_{desc}	
		A3_{avg}	A1_{avg}	A3_{max}	A1_{max}	A3	A1	A3	A1	A3	A1
40	2	0.0	0.0	0.0	0.0	1	0	0.0	0.2	464	4819
40	4	0.0	0.0	0.0	5.5	2	0	0.9	0.1	2174	8072
40	10	0.0	0.0	0.0	1.5	8	0	0.3	0.2	34780	64940
50	2	0.0	0.0	0.0	0.0	0	0	0.1	0.1	486	4414
50	4	0.0	0.0	0.0	0.6	4	1	2.5	1.5	3077	9885
50	10	0.0	0.1	0.0	8.1	13	0	0.2	29.6	75898	111079
100	2	0.0	0.0	0.0	0.0	14	0	52.3	35.3	5480	44630
100	4	0.0	0.0	0.0	0.1	55	4	835.7	1030.2	8958	23856
100	10	0.0	0.3	0.0	9.0	86	1	1525.6	1139.7	11113	11562
300	2	0.0	0.0	0.1	0.3	72	5	2800.9	3308.3	183	1192
300	4	0.0	0.2	0.0	11.3	89	3	3290.4	1148.8	279	628
300	10	0.0	0.8	0.0	21.3	102	0	2819.9	3022.6	346	287
300	20	0.0	0.8	0.0	28.3	104	1	1663.1	2149.4	336	207

TABLE 2. Overall results (1 hour time limit).

n	m	dev% 5 s.				dev% 10 s.				dev% 30 s.			
		avg		max		avg		max		avg		max	
		A3	A1	A3	A1	A3	A1	A3	A1	A3	A1	A3	A1
40	2	0.0	0.0	0.0	0.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
40	4	0.1	0.1	0.0	11.3	0.1	0.1	0.0	6.1	0.0	0.1	0.2	6.1
40	10	0.3	0.6	0.2	22.2	0.2	0.5	0.1	22.2	0.2	0.4	0.1	13.9
50	2	0.0	0.0	0.0	0.3	0.0	0.0	0.0	0.2	0.0	0.0	0.0	0.0
50	4	0.1	0.1	0.0	3.1	0.0	0.0	0.0	3.1	0.0	0.0	0.0	1.8
50	10	0.2	0.7	0.0	35.0	0.1	0.6	0.0	35.0	0.1	0.4	0.0	20.3
100	2	0.0	0.0	0.0	2.0	0.0	0.0	0.0	0.2	0.0	0.0	0.0	0.2
100	4	0.1	0.4	0.0	29.1	0.1	0.4	0.0	29.1	0.1	0.3	0.0	29.1
100	10	0.9	2.0	0.1	45.3	0.6	1.5	0.0	44.1	0.4	1.2	0.0	36.4

TABLE 3. Comparison for small time limits ($\leq$ 30 sec).

n	m	dev% 60 s.				dev% 120 s.				dev% 180 s.				240 s.				dev% 300 s.			
		avg		max		avg		max		avg		max		avg		max		avg		max	
		A3	A1	A3	A1	A3	A1	A3	A1	A3	A1	A3	A1	A3	A1	A3	A1	A3	A1	A3	A1
40	2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
40	4	0.0	0.0	0.0	5.5	0.0	0.0	0.0	5.5	0.0	0.0	0.0	5.5	0.0	0.0	0.0	5.5	0.0	0.0	0.0	5.5
40	10	0.1	0.3	3.2	11.1	0.1	0.2	2.8	5.6	0.0	0.2	2.8	5.6	0.0	0.2	2.8	5.6	0.0	0.2	0.0	5.0
50	2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
50	4	0.0	0.0	0.0	1.8	0.0	0.0	0.0	1.8	0.0	0.0	0.0	0.6	0.0	0.0	0.0	0.6	0.0	0.0	0.0	0.6
50	10	0.1	0.3	3.3	16.3	0.0	0.2	1.8	16.3	0.0	0.2	1.8	16.3	0.0	0.2	1.4	16.3	0.0	0.2	1.0	16.3
100	2	0.0	0.0	0.1	0.1	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.1
100	4	0.0	0.3	2.1	29.1	0.0	0.2	2.1	13.3	0.0	0.2	0.2	13.3	0.0	0.1	0.2	3.5	0.0	0.1	0.2	3.5
100	10	0.3	1.0	6.7	29.7	0.2	0.9	4.2	29.7	0.1	0.6	3.5	9.6	0.1	0.6	3.5	9.0	0.1	0.6	2.6	9.0
300	2	0.1	0.1	1.7	1.8	0.1	0.1	0.9	0.9	0.0	0.0	0.7	0.9	0.0	0.0	0.5	0.9	0.0	0.0	0.4	0.9
300	4	0.5	1.3	16.4	76.2	0.3	1.3	16.4	76.2	0.3	1.2	16.4	76.2	0.3	0.7	13.6	18.1	0.3	0.6	13.6	17.9
300	10	0.7	3.2	17.3	72.6	0.6	2.1	17.2	41.9	0.6	1.6	17.2	37.5	0.5	1.5	15.8	37.5	0.4	1.5	15.8	37.5
300	20	0.7	4.3	11.4	96.1	0.6	2.2	11.4	50.4	0.5	2.0	11.4	50.4	0.5	1.9	11.4	50.4	0.4	1.8	10.3	50.4

TABLE 4. Comparison for large time limits