

Comparative Analysis of Heuristic Approaches to $P||C_{max}$

Dragutin Ostojić¹^a, Tatjana Davidović²^b, Tatjana Jakšić Krüger²^c and Dušan Ramljak³^d

¹Faculty of Science, Department of Mathematics and Informatics, University of Kragujevac, Serbia

²Mathematical Institute, Serbian Academy of Sciences and Arts, Belgrade, Serbia

³School of Professional Graduate Studies at Great Valley, The Pennsylvania State University, Malvern, PA, U.S.A.

Keywords: Scheduling Problems, Identical Processors, Stochastic Heuristics, Solution Transformation.

Abstract: Cloud computing, new paradigms like fog, edge computing, require revisiting scheduling and resource allocation problems. Static scheduling of independent tasks on identical processors, one of the simplest scheduling problems, has regained importance and we aim to find stochastic iterative heuristic algorithms to efficiently deal with it. Combining various actions to define solution transformations to improve solution quality, we created 35 heuristic algorithms. To investigate the performance of the proposed approaches, extensive numerical experiments are performed on hard benchmark instances. Among the tested variants, we identified the best performing ones with respect to the solution quality, running time, and stability.

1 INTRODUCTION

Driving online decision making requires generating huge amounts of data and its expeditious analysis - thus high performance computing resources and their efficient use are needed. This yields reconsideration of already extensively investigated practical, usually complex, scheduling/resource allocation problems.

We revisit a problem of scheduling independent tasks on parallel processors (Graham, 1969; Davidović et al., 2012; Frachtenberg and Schwegelshohn, 2010; Pinedo, 2012) as a result of an increased interest in efficient exploration of high performance computing resources, cloud computing and massively parallel multiprocessor systems. We are considering a case study of $P||C_{max}$ - static scheduling of independent tasks on identical processors. The expression *static* means that the number of tasks and their duration (lengths, processing times) are known *a priori*. The problem objective is to minimize the time required to complete the execution of all tasks, i.e., *makespan* C_{max} . $P||C_{max}$ is known to be NP-hard in a strong sense (Fanjul-Peyro and Ruiz, 2010). Numerous exact (Mrad and Souayah, 2018), heuristic (Paletta and Ruiz-Torres, 2015), and metaheuristic

(Davidović et al., 2012; Alharkan et al., 2018; Laha and Gupta, 2018; Kamaraj and Saravanan, 2019) algorithms have been developed for $P||C_{max}$.


To efficiently handle $P||C_{max}$, we analyze stochastic iterative heuristic algorithms based on random transformations of the current solution with an aim to improve its quality. Transformations are applied repeatedly until some predefined stopping criterion is met. Each transformation consists of several steps (different actions could be performed within each). Combining various actions, we created 35 heuristic algorithms and compared them on hard benchmark instances. The diversity in performance with respect to the solution quality, running time, and stability can be significant. Our experimental evaluation enabled to identify the best performing variants.


In the remainder of this paper: Section 2 contains $P||C_{max}$ problem description, and a brief relevant literature overview, the proposed heuristic algorithms are presented in section 3, the experimental evaluation in section 4, and concluding remarks in section 5.


2 $P||C_{max}$ BACKGROUND

Let m be the total number of available identical processors, n the number of tasks to be executed. The $P||C_{max}$ problem consists of assigning tasks to processors, and determining their starting times. All the

^a <https://orcid.org/0000-0001-6704-353X>

^b <https://orcid.org/0000-0001-9561-5339>

^c <https://orcid.org/0000-0001-6766-4811>

^d <https://orcid.org/0000-0001-7477-1973>

tasks should be allocated for execution, each to exactly one processor. Task execution is performed in a non-preemptive way: once the task starts it will continue without interruption until completion. We denote by $T = \{1, 2, \dots, n\}$ a given set of independent tasks, and by $M = \{1, 2, \dots, m\}$ the set of identical processors. Each processor can engage only one task at a time. Let p_i denote the processing time of task i ($i = 1, 2, \dots, n$), which is *a priori* known and fixed, and let y_j ($j = 1, 2, \dots, m$) represent the load of processor j calculated as the sum of processing times of all tasks assigned to processor j . The goal is to find a schedule of tasks on processors such that the corresponding completion time of all tasks (*makespan*) is minimized (Davidović et al., 2012). The makespan is usually denoted as C_{max} and calculated as: $C_{max} = \max_{j \in M}(y_j)$. $P||C_{max}$ can be formulated as integer linear program (ILP) based on the assignment variables (Mokotoff, 2004), on the arc-flow model (Mrad and Souayah, 2018) or in some other ways (Unlu and Mason, 2010). Although the optimality of provided solutions is guaranteed, these formulations have limitations in practical use: they require a lot of time and memory, even for small-size instances. Therefore, after briefly reviewing some of the relevant results below, we propose heuristic approaches that can be more efficient in practice.

$P||C_{max}$ is by far the most studied among all completion time related criteria and an enormous body of knowledge, technical results, and connections to real-world problems has been accumulated in this area. We could think of production lines where several machines with the same speed have to perform a certain amount of jobs (Mokotoff, 2004), or minimizing the overall CPU-time for identical processors by efficiently assigning tasks (Graham, 1969; Frachtenberg and Schwegelshohn, 2010; Davidović et al., 2012).

(Lawrinenko, 2017) contains an extensive survey of the $P||C_{max}$, in-depth explanations of exact algorithms, while (Walter and Lawrinenko, 2017) presented a depth-first branch-and-bound algorithm with symmetry-breaking dominance criteria. Exact algorithms state-of-the-art application (Mrad and Souayah, 2018) presents an arc-flow based ILP model inspired by the duality between the bin-packing and the $P||C_{max}$ and discuss the hardness of test instances expressed by n and m ratio.

(Della Croce and Scatamacchia, 2020) revisited Longest Processing Time (LPT (Graham, 1969)) and derived an $O(n \log n)$ time complexity constructive heuristic. Two procedures of constructive heuristic approach, (Paletta and Ruiz-Torres, 2015), are constructing a feasible solution, and then Many Times Multifit (MTMF) procedure tightening the initial so-

lution by iteratively using a bin-packing based procedure on different job sets. Iterative heuristic approach, (Paletta and Vocaturo, 2011), also builds upon initially constructed feasible solution, and use local search techniques where single jobs or sets of jobs are exchanged between different machine pairs (i_1, i_2). Approximation heuristic approach (Mnich and Wiese, 2015) show that there is an Fixed-parameter tractability (FPT) algorithm for this problem when parameterized by p_{max} (the largest job processing time). Several Efficient Polynomial Time Approximation Schemes (EPTAS) for the $P||C_{max}$ exist (Jansen et al., 2020).

Among the first metaheuristic approaches, Tabu Search (TS) is proposed in (Thesen, 1998). A rather simple implementation of Variable Neighborhood Search (VNS) is proposed in (Davidović and Janićijević, 2009). The authors of (Alharkan et al., 2018) extended its study to more general variant of VNS, investigating the effect of including new neighborhood structures and changing the order in which the neighborhoods are explored. Bee Colony Optimization (BCO) metaheuristic was developed in (Davidović et al., 2012) exploring stochastic LPT rule to construct feasible solutions. (Laha and Gupta, 2018) improved Cuckoo Search Algorithm (CSA) and explored LPT construction scheme, but included the pairwise exchange neighborhood in the improvement phase. Grey Wolf Optimiser (GWO) algorithm in (Kamaraj and Saravanan, 2019) starts from randomly generated population and explores GWO rules attempting to improve selected subset of solutions.

3 THE PROPOSED HEURISTIC ALGORITHMS FOR $P||C_{max}$

Our approach consists of building heuristics through transformations that have a potential to improve the quality of solutions. We present the process of building the proposed 35 variants of the heuristic algorithms for $P||C_{max}$ and discuss their implementation, including dependence on initial solution and complexity of the implementation. The latter is reduced by carefully defined solution representation and performed pre-processing. The variants of heuristic algorithms differ by the type of transformation applied to the given initial solution and by the way initial solution is obtained. All the variants can be described by the steps presented in the remainder of this section.

3.1 Description of Transformations

The quality of a solution is defined by C_{max} that represents the execution time of the most heavily loaded

processor (k). The only way to improve the current solution is to reduce its load by moving a task to some other processor l . At the same time, the new load of the processor l should not exceed the previous C_{max} . However, constant improvement of the solution quality could result in undesired outcome: getting stuck in a local optimum. For $P||C_{max}$, in majority of the cases, the only way to improve the current solution is to degrade its quality first. Therefore, we define stochastic transformations that do not necessarily improve the current solution.

We define two classes of transformations depending on selection strategies for reallocation task and destination processors for them. Transformations are composed of steps that could be realized in different ways, i.e., by performing different actions on the given input solution. Different actions are coded by different indicators combined in the input vector Ind . Even though two classes of transformations have some similar steps, their number is different, thus we present two different algorithms. Each algorithm starts with an initial solution (Sol) and performs the corresponding transformation as it is defined by the input parameters from the Ind vector.

First class transformations' five steps are in Algorithm 1. First step of each transformation ($SProc$): the most heavily loaded processor k is identified. $SProc$ is also the source of the task i to be moved.

Algorithm 1: Class 1 transformation.

```

1 Transform(Sol, Ind)
2   k ← SProc(Sol, Ind)
3   i ← TaskID(Sol, Ind, k)
4   l ← DProc(Sol, Ind, k)
5   Move(Sol, k, i, l)
6   Swap(Sol, Ind, k, l)
7   return(Sol)

```

The second step ($TaskID$): selecting the task i to be moved from processor k . Three different actions we used to perform it are 1. random selection (denoted by MR), 2. roulette wheel with the higher probability to select longer tasks, (marked by MRL), 3. roulette wheel with the higher probability to select shorter tasks, (denoted by MRS).

The third step ($DProc$): selecting the destination processor for task i . The five different actions are as follows. The first is completely random ($DRand$), while the remaining four assume the roulette wheel application involving a subset of promising processors and giving the higher probability to the less loaded processors. In this step the following parameters are used: $\underline{y} = \min_{h \in [1, m]} y_h$ - load of the least

loaded processor, \hat{y} - load of the most loaded processor k , \bar{y} - constant defined as average load of all processors, and $LB = \lceil \bar{y} \rceil$ - constant defining theoretically the best objective function value of $P||C_{max}$. In the first roulette action (DROpt), the considered subset is composed of the processors with load belonging to the interval $[LB, \underline{y}]$. The second (DRYmax) considers the complete set of processors. The third (DROpt1) the processor loads are limited by $LB + 1$ and \underline{y} , while the corresponding interval for processor load in the fourth roulette action (DRUpper) is bounded by $\lceil \bar{y} + 1 \rceil$ and \underline{y} .

In the fourth step task i is moved from processor k to processor l , performed by procedure $Move(Sol, k, i, l)$. The fifth step, not mandatory, considers the possibility of swapping tasks. More precisely, for task i that was moved from processor k to processor l , a number of shorter tasks is reallocated from l to k , in such a way that the load of k does not exceed LB value. This process starts with the longest possible task from processor l that satisfies the above mentioned condition - procedure is performed as long as possible and denoted as $Swap(Sol, Ind, k, l)$.

Algorithm 2: Class 2 transformation.

```

1 Transform(Sol, Ind)
2   k ← SProc(Sol, Ind)
3   l ← DProc(Sol, Ind, k)
4   Mix(Sol, k, l)
5   return(Sol)

```

Second class transformations contain three steps (Algorithm 2). The first is the same as the first step in class 1, while the second is same as the third step in class 1. In the third step, denoted as $Mix(Sol, k, l)$, tasks from processors k and l of the current solution Sol are rearranged in a deterministic way as follows. First, a subset of tasks with the sum of loads as close as possible to LB is selected with the knapsack 0-1 algorithm and allocated to processor k . Then the remaining tasks are allocated to processors l . Table 1 summarizes both classes transformations notation.

3.2 Complexity Evaluation

To efficiently perform the transformations, solutions are represented by four structures:

- *tasksOnProcessor* - 2D vector (k, i) representing index of i -th task scheduled on processor k .
- *tasksOnProcessorSet* - k -th value of 1D vector represents set of tasks scheduled to processor k . The set is balanced binary tree, each

Table 1: Transformations nomenclature.

Step	Action	Indicator
SProc()	deterministic	
TaskID()	random	MR
	roulette (longest)	MRL
	roulette (shortes)	MRS
DProc()	random	DRand
	roulette $[LB, \bar{y}]$	DROpt
	roulette $[\hat{y}, \bar{y}]$	DRYmax
	roulette $[LB + 1, \bar{y}]$	DROpt1
	roulette $[UB(\bar{y} + 1), \bar{y}]$	DRUpper
Move()	deterministic	
Swap()		NONE Sw
Mix()		mix

Table 2: Transformation complexity.

Indicator	Time	Memory
SProc	$O(1)$	$O(m)$
MR	$O(1)$	$O(n)$
MRL	$O(1)$	$O(n)$
MRS	$O(1)$	$O(n)$
DRand	$O(1)$	$O(m)$
DROpt	$O(1)$	$O(m)$
DRYmax	$O(1)$	$O(m)$
DROpt1	$O(1)$	$O(m)$
DRUpper	$O(\log m)$	$O(m)$
Move	$O(\log n)$	$O(n)$
Sw	$O(n \log n)$	$O(n)$
mix	$O(n \cdot LB + \log m)$	$O(n \cdot LB + m)$

node is 2-tuple $(p_i, \text{index of task } i \text{ in vector } tasksOnProcessor_k)$. The sets are sorted in non-decreasing order, by p_i first and then by the second value of a tuple.

- *processorLoad* - 1D vector with k -th value representing the load of processor with id k . More precisely, $processorLoad_k = \sum_{j \in tasksOnProcessor_k} p_j$.
- *processorLoadSet* - Set of processors. The set is a balanced binary tree, where l -th node is 2-tuple $(processorLoad_l, \text{id of processor with } l\text{-th load})$.

3.3 Iterative Heuristics Approach

Having all the transformations characterized by actions taken to transform a solution, we defined 35 heuristic algorithms for $P||C_{max}$. Each heuristic repeatedly calls the corresponding transformation to

Algorithm 3: Iterative heuristic.

```

1 Heuristic (Sol, Ind, StopCrit)
2   BestSol  $\leftarrow$  Sol
3   while StopCrit is not met do
4     Transform (Sol, Ind)
5     if ( $y(Sol) < y(BestSol)$ ) then
6       BestSol  $\leftarrow$  Sol
7     end
8   end
9   return BestSol
    
```

modify the current solution until the stopping criterion is met. After the transformation is completed, the quality of the obtained solution is compared to the best so far and the corresponding update is performed. At the end, the best obtained solution is reported. The main components of our iterative heuristics approach are presented in Algorithm 3. Inputs to our algorithm are an initial solution *Sol*, transformation indicator *Ind*, and the stopping criterion for heuristics. The most common stopping criteria are the maximum number of iterations n_{it} and the maximum allowed CPU time t_{max} .

4 EXPERIMENTAL EVALUATION

Comparative analysis of our iterative heuristics approaches is performed in a simulator we built in C++ and compiled with gcc version 7.5.0. All the runs were performed on Intel(R) Core(TM) i5-6400 CPU @ 2.70GHz with 8GB RAM under Ubuntu 7.5.0-3ubuntu1~18.04. In the remainder of this section we explain all details of the performed experiments.

4.1 Simulation Environment

In previous section we defined 35 heuristic algorithms for $P||C_{max}$ based on various solution transformations performed iteratively aiming to lead towards optimal solution. Due to the stochastic nature of proposed transformations, all the factors influencing their variability need to be explored. We present our heuristic algorithms effectiveness with respect to solution quality and convergence, depending on the initial solution and number of transformations required to reach the solution of good quality. In addition, we examined the stability of transformations, i.e., the variations of results depending on the random number generator seed value initialization. Our generated testing environment is presented in Algorithm 4.

The initial solution we obtained applying the random

Algorithm 4: Testing environment.

```

1 Run( $n_{run}, n_{it}, Ind$ )
2   for  $i \leftarrow 1$  to  $n_{run}$  do
3      $Seed \leftarrow i$ 
4      $Sol \leftarrow Init()$ 
5      $Heuristic(Sol, Ind, n_{it})$ 
        $Update(SolBase)$ 
6   end
7   return  $SolBase$ 

```

and greedy approaches. The random initial solution is determined by allocating randomly selected task i to the randomly selected processor k . Time complexity of this random procedure is $O(n \log n + m \log m)$. For greedy algorithm, we selected list scheduling procedure based on LPT rule used for ordering tasks and Earliest Start (ES) heuristic to determine the most appropriate processor. This implies sorting the jobs in non-increasing order of processing times before they are scheduled to the least loaded processor (Graham, 1969; Davidović et al., 2012). LPT+ES constructive heuristic is in fact an approximation algorithm with the worst-case approximation ratio $4/3 - 1/(3m)$. Its time complexity is $O(n \log n + n \log m)$ and it is performed in preprocessing phase. Each run starts from either different random solution or from the same LPT+ES solution.

Testing Environment. input values for our heuristics are the number of repetitions n_{run} (the stopping criterion of the corresponding test), the number of iterations (transformation applied) n_{it} , and the vector Ind containing indicators which transformation to be considered. In each run, an initial solution of $P||C_{max}$ is generated and then the transformation defined by the vector Ind is performed repeatedly, until the stopping criterion is satisfied, i.e., until n_{run} repetitions are completed. All obtained solutions are saved in $SolBase$, together with the required number of transformations and the corresponding running times.

Test Instances. We evaluated the proposed heuristic algorithms on the known set $logra$ of hard benchmark problem instances with the known optimal value of the objective function used in (Davidović et al., 2012). The number of tasks in these instances range is $n \in \{50, 100, 200, 250\}$ and there are 28 instances in total. Several studies have shown that hardness of instances increase with increasing the value of n/m (Mrad and Souayah, 2018), but this doesn't apply for $logra$ set. In fact, for fixed number of machines, the hardness of problem instances decreases as the number of tasks increases. In addition, for fixed number of tasks the problem instances become harder as number of machines increases, regardless the de-

crease in n/m . The hardest test instances are with 50 tasks (Jakšić Krüger, 2017). Therefore, comparative analysis of heuristics is depicted on 7 test instances with 50 tasks.

4.2 Comparative Analysis Methodology

We set $n_{run} = 100$ to repeat algorithm's execution for different values of random number generator seed due to stochastic nature of presented heuristic algorithms. After each run we record: solution, solution's quality, number of iterations and time required to generate the best (optimal) solution for the first time. In order to capture central tendencies of these results we employ descriptive statistics such as mean, median and the corresponding dispersion statistics. The goal is to gauge the effectiveness of our algorithms, i.e., to estimate an effort needed to reach the optimal solution.

Performance Measures. Due to high efficiency of our algorithms, the dominant performance measure is number of times the optimal solution is reached (n_{opt}) within n_{run} repetitions. Other performance measures: average solution quality, average or maximal runtime and the corresponding number of iterations. The average quality of solutions \bar{y}_{max} is important in case when the optimal solution is never reached. We also employ \bar{y}_{max} in graphical representations of our results. An important concept for our comparison study is the definition of poor performance. We define a transformation's poor performance if an average quality of solutions is worse than in case of transformations that obtained the optimum. Measure of time is important due to the fact that duration of one iteration is different w.r.t. to type of transformation. The iteration count may be misleading and overly optimistic for methods that invoke more work within one iteration. For some transformation A we determine runtime t_A as $t_A = \max_{1 \geq seed \geq n_{run}} t_{seed}$, where t_{seed} is the first-hit time of the best solution for the given $seed$.

Stopping Criteria Evaluation. A high value for n_{it} might lead to a better solution quality, but for heuristic approaches it is hard to predict how far from the optimum we are. The most suitable value for n_{it} is when algorithms reach stagnation, i.e., there is no solution improvements anymore. After performing initial evaluation, we set $n_{it} = 10000$ as a stopping criterion for empirical analysis of the proposed heuristic algorithms. As each transformation needs different time to execute n_{it} iterations, we report the time required to complete all of them and the number of iterations required to find the best reported solution.

Methodology of Selecting Good Transformations. Our comparison study of heuristics on $logra$ benchmark set is based on n_{opt} , \bar{y}_{max} , and t_A . To represent

whether optimal solution was found and how often, our results are two-dimensional scatter plots with n_{opt} represented by the size of the circle (see Fig. 1). The ranking procedure is performed in two phases. The first phase criterion awards the transformations that reached an optimal solution within n_{run} repetitions. All such transformations are removed from the ranking pool and used to define the performance of the remaining transformations. The second phase criterion awards the transformations that have a better average solution quality than the removed transformations. Whenever the heuristic algorithm achieves one of these goals we appoint 1 to its rank. We repeat this process for all problem instances from I_{ogra} set.

4.3 Results

Selection of Good Heuristics. In Fig. 1 we present scatter plot for all 70 tested variants (IR random initial solution, IL greedy initial solution) for problem instance I_{ogra50} and variable number of processors. The position of the point on the graph is determined by number of iterations needed to either reach optimum or a given average solution. Comparing heuristics grouped by IR and IL, we do not detect significant changes in the performance w.r.t. average solution quality and average number of iterations indicating that the initial solution does not influence significantly. Obviously, the transformations positioned at the lower left corner of graphs perform the best. Then, we clearly demonstrate that transformations depicted as points in the upper left region of the plot in Fig. 1 for instances with $m \geq 6$ characterised by small runtime and low average solution quality performed the worst. They exhibit premature convergence property, reaching their best solution early in the execution stages and mainly call MRS action. Next, a group of heuristics at the right parts of the plots in Fig. 1 (mostly Drand with MR or MRL strategies) for $m \geq 4$ also performed poorly, due to the fact that the average required number of iterations is large. Finally, from the same figure we see that combination of DRYmax with MR or MRL strategies perform badly.

Performance Discussion. Good heuristics are ranked based on their performance on all testing instances. The largest rank is appointed to all heuristics presented in Table 3. It contains the results for I_{ogra50} organized as follows. Each heuristic variant is identified by the acronym (first column). The indicators of different initial solution (IR, IL) are in the second column and the percentage of found optimal solutions is in the third column. The fourth column contains \bar{y}_{max} . In the last two columns, the average number of iterations required to obtain the best solution and the

corresponding average CPU times are given. For all average values, standard deviations are also provided.

From Table 3 we note the following. MRL-DROpt-Sw and MRL-DROpt1-Sw were not as successful in finding an optimal solution as the other three methods, but the variability of the results is low - they produce the best average quality of solutions. In addition, all heuristics with Mix() step belong to the so-called *good* heuristics that produce optimal solutions for I_{ogra} test set in the largest percentage of runs. Including the action Swap(), significantly increases the performance w.r.t. n_{opt} . Action DProc() is the most efficient when combined with actions such as DROpt and DRUpper.

The initial solution has significant influence in the case of the last two (MRL-DROpt-Sw and MRL-DROpt1-Sw) where random initial solutions prevent reaching the optimal solution. We can note that all heuristics are very stable in reaching high-quality solutions as the average values of \bar{y}_{max} are close to the optimums and the standard deviations are small (except for heuristics that used Mix() action where in some sporadic cases the low quality solutions are provided). Based on the average number of iterations, it can be concluded that our selected heuristics reach their best solutions very fast, except in some isolated cases. This is confirmed by the CPU time data provided in the last column. Our analysis shows that the selected heuristics are very fast in providing high-quality solutions for the hard benchmark instances of $P||C_{max}$ and thus can be an useful tool in more general frameworks, such as metaheuristics.

5 CONCLUSION

In this paper we proposed several stochastic iterative heuristic algorithms to efficiently deal with the problem of static scheduling of independent tasks on homogeneous multiprocessors. We created 35 heuristic algorithms by combining various actions to define solution transformations aimed at improving solution quality. Testing was performed for 2 variants of choosing the initial solution which resulted in 70 variants to be tested. We have conducted an extensive numerical experiments on hard benchmark instances from the literature. Among the tested 70 variants, we identified 10 the best performing heuristics with respect to the solution quality, running time, and stability. As the future work we plan to evaluate our heuristics on other sets of test instances and to incorporate the best performing heuristics in some metaheuristic frameworks and examine their influence on the performance of the resulting approaches.

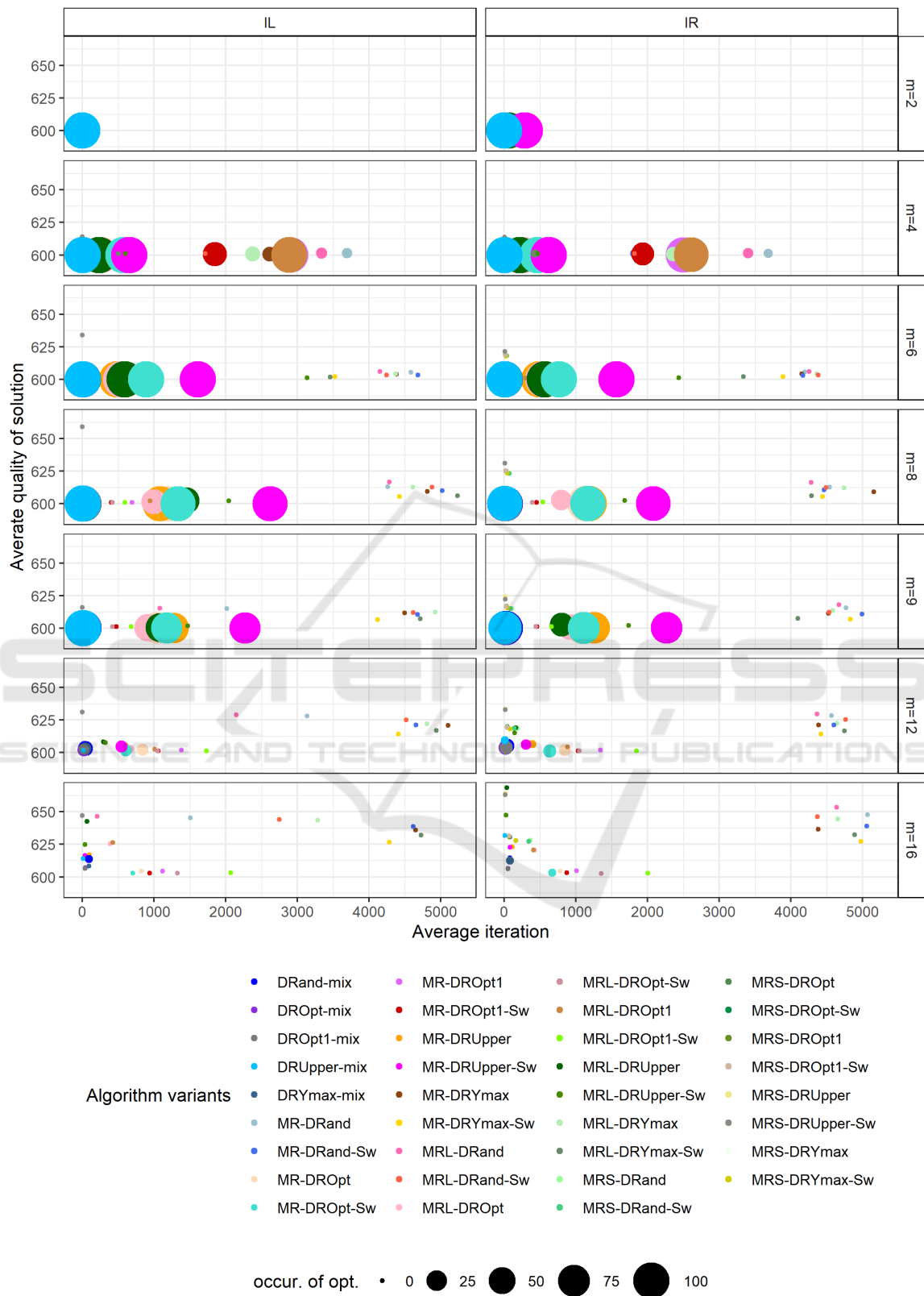


Figure 1: Results of comparisons between 35 transformations on 7 problem instances from Iogra50 set.

Table 3: 10 best heuristics on Iogra50, $m = 2, 4, 6, 8, 9, 12, 16$. Each average result has the standard deviation.

Heuristic	Init()	sum.opt[%]	\bar{y}_{max}	Av. iter	Av. t_A [$10^{-3}sec$]
DRand-Mix	IR	70	602.91 ± 5.57	25.85 ± 27.94	0.90 ± 0.46
	IL	71	602.45 ± 5.11	27.69 ± 32.08	0.98 ± 0.86
MR-DROpt-Sw	IR	68.4	600.66 ± 1.16	729.38 ± 320.19	2.44 ± 2.13
	IL	66.8	600.66 ± 1.08	754.44 ± 439.13	2.80 ± 2.26
DRYmax-Mix	IR	68.3	602.47 ± 4.73	23.76 ± 28.53	0.80 ± 0.54
	IL	70.7	601.63 ± 3.12	24.41 ± 31.14	0.71 ± 0.57
MRL-DROpt-Sw	IR	0.6	601.3 ± 0.63	501.21 ± 509.95	1.59 ± 1.91
	IL	14.3	601.20 ± 0.91	499.6 ± 501.94	1.18 ± 1.07
MRL-DROpt1-Sw	IR	0.6	601.32 ± 0.69	762.56 ± 830.41	2.63 ± 3.35
	IL	14.3	601.23 ± 0.99	771.03 ± 816.83	2.02 ± 2.41

ACKNOWLEDGEMENTS

This work was partially supported by the Science Fund of Republic of Serbia AI4TrustBC project and by the Serbian Ministry of Education, Science and Technological Development, Agreement No. 451-03-9/2021-14/200029. The authors thank Penn State GV IT team for the support.

REFERENCES

- Alharkan, I. et al. (2018). An order effect of neighborhood structures in variable neighborhood search algorithm for minimizing the makespan in an identical parallel machine scheduling. *Math. Problems in Engineering*.
- Davidović, T. et al. (2012). Bee colony optimization for scheduling independent tasks to identical processors. *Journal of Heuristics*, 18(4):549–569.
- Davidović, T. and Jančićević, S. (2009). VNS for scheduling independent tasks on identical processor. In *Proc. 36th Symp. on Operational Research, SYM-OP-IS 2009*, pages 301–304, Ivanjica.
- Della Croce, F. and Scatamacchia, R. (2020). The longest processing time rule for identical parallel machines revisited. *Journal of Scheduling*, 23(2):163–176.
- Fanjul-Peyro, L. and Ruiz, R. (2010). Iterated greedy local search methods for unrelated parallel machine scheduling. *European Journal of Operational Research*, 207(1):55–69.
- Frachtenberg, E. and Schwiegelshohn, U. (2010). Preface. In *15th International Workshop, JSSPP 2010, Job Scheduling Strategies for Parallel Processing*, pages V–VII.
- Graham, R. L. (1969). Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429.
- Jakšić Krüger, T. (2017). *Development, implementation and theoretical analysis of the bee colony optimization meta-heuristic method*. PhD thesis, Faculty of technical sciences, University of Novi Sad.
- Jansen, K. et al. (2020). Closing the gap for makespan scheduling via sparsification techniques. *Mathematics of Operations Research*, 45(4):1371–1392.
- Kamaraj, S. and Saravanan, M. (2019). Optimisation of identical parallel machine scheduling problem. *Int. Journal of Rapid Manufacturing*, 8(1-2):123–132.
- Laha, D. and Gupta, J. N. (2018). An improved cuckoo search algorithm for scheduling jobs on identical parallel machines. *Computers & IE*, 126:348–360.
- Lawrinenko, A. (2017). *Identical parallel machine scheduling problems: structural patterns, bounding techniques and solution procedures*. PhD thesis, Friedrich-Schiller-Universität Jena.
- Mnich, M. and Wiese, A. (2015). Scheduling and fixed-parameter tractability. *Mathematical Programming*, 154(1):533–562.
- Mokotoff, E. (2004). An exact algorithm for the identical parallel machine scheduling problem. *European Journal of Operational Research*, 152(3):758–769.
- Mrad, M. and Souayah, N. (2018). An arc-flow model for the makespan minimization problem on identical parallel machines. *IEEE Access*, 6:5300–5307.
- Paletta, G. and Ruiz-Torres, A. J. (2015). Partial solutions and multifit algorithm for multiprocessor scheduling. *Journal of Mathematical Modelling and Algorithms in Operations Research*, 14(2):125–143.
- Paletta, G. and Vocaturo, F. (2011). A composite algorithm for multiprocessor scheduling. *Journal of Heuristics*, 17(3):281–301.
- Pinedo, M. L. (2012). *Scheduling: theory, algorithms, and systems*. Springer Science & Business Media.
- Thesen, A. (1998). Design and evaluation of a tabu search algorithm for multiprocessor scheduling. *J. Heuristics*, 4(2):141–160.
- Unlu, Y. and Mason, S. J. (2010). Evaluation of mixed integer programming formulations for non-preemptive parallel machine scheduling problems. *Computers & Industrial Engineering*, 58(4):785–800.
- Walter, R. and Lawrinenko, A. (2017). Lower bounds and algorithms for the minimum cardinality bin covering problem. *European Journal of Operational Research*, 256(2):392–403.