Communication Papers of the 18th Conference on Computer Science and Intelligence Systems pp. 157–164
DOI: 10.15439/2023F9294
ISSN 2300-5963 ACSIS, Vol. 37

Mushroom Picking Framework with Cache Memories for Solving Job Shop Scheduling Problem

Piotr Jedrzejowicz, Izabela Wierzbowska
0000-0001-6104-1381
0000-0003-4818-4841
Department of Information Systems,
Gdynia Maritime University,
ul. Morska 81-87, 81-225 Gdynia, Poland;
Email: {p.jedrzejowicz, i.wierzbowska}@wznj.umg.edu.pl

***Abstract*—Applying population-based metaheuristics is a known method of solving difficult optimization problems. In this paper the search for the best solution is conducted by decentralized, self-organized agents, working in parallel threads, in the so called mushroom-picking method. The search is enhanced by remembering in which part of the recently improved solution the last successful change took place and intensifying the search in this part. A computational experiment shows that introducing the component for remembering the most recent changes may improve the results obtained by the model in the case of JSSP problems.**

I. Introduction

POPULATION-BASED methods belong to the most effective approaches when dealing with computationally difficult optimization problems, including combinatorial optimization ones. A population, in a broad term, represents here solutions of the problem, potential solutions, parts of solutions, or some constructs that can be somehow transformed into solutions.

The main focus while using population-based methods is to find a proper mechanism controlling their three fundamental components - intensification, diversification, and learning [1]. Diversification is understood as the method of identifying diverse promising regions over the whole search space, while intensification is the method of finding a solution by exploring some promising regions. Learning is gaining and using the knowledge of where and how applying intensification and diversification operations. Population-based methods belong to a wider class of metaheuristics. Metaheuristics differ between themselves by using different intensification, diversification and learning rules.

Pioneering population-based methods included genetic programming – (GP) [2], genetic algorithms (GA) [3], evolutionary computations - EC [4], [5], ant colony optimization (ACO) [6], particle swarm optimization (PSO) [7] and bee colony algorithms (BCA) [8]. Since the nineties a massive number of metaheuristics using the population-based paradigm have been proposed (see reviews [9], [10]), some of them named after biological, physical, chemical, and other natural sciences phenomena. Generally, the discussed metaheuristics have achieved a certain level of perfection in finding solutions to many computationally difficult problems. None of them, however, could be considered a champion and their performances differ depending on the problem characteristics and specifications. Moreover, a vast majority of well-performing population-based metaheuristics require fine-tuning or even adaptation to produce high-quality solutions to difficult computational problems.

An important step towards increasing the effectiveness of the population-based methods was the emergence, during the last few decades, of commonly accessible technologies enabling parallelization of search over the solution space. Among the successful attempts to parallelize searching for the best solution among the population of solutions was parallel GA on MapReduce framework using the Hadoop cluster [11]. The method was designed for solving instances of the travelling salesman problem (TSP). A similar approach for implementing a parallel genetic algorithm with the Hadoop MapReduce for TSP can be found in [12]. Spark-based ant colony optimization algorithm for solving the TSP was proposed in [13]. A Spark-based version of the population learning metaheuristic applied, among others, to job-shop scheduling problem (JSSP) can be found in [14]. Some extensive reviews of developments in the field of parallel metaheuristics can be found in [15], [16], and [17].

Job-shop scheduling problem (JSSP) is one of the "classic" computationally hard problems. In recent years several approaches to solving the JSSP using population-based metaheuristics have been proposed. Some examples of such approaches include:

- Specialized cuckoo search algorithm [18].
- A hybrid particle swarm optimization (PSO) and neural network algorithm [19].
- An improved whale optimization algorithm [20].
- Genetic Algorithm for JSSP [21].
- Wolf pack algorithm for JSSP [22].

Thematic track: Computational Optimization

- Biomimicry hybrid bacterial foraging optimization algorithm for JSSP [23].
- Hybrid harmony search algorithm for JSSP [24].
- Discrete particle swarm optimization (PSO) algorithm for JSSP [25].
- Hybrid PSO optimization algorithm with nonlinear inertial weight and Gaussian mutation for JSSP. [26]

To take advantage of the expected speed-up several attempts of using parallel computational environments for solving the JSSP have been also recently reported. MapReduce coral reef algorithm for solving JSSP instances was proposed by [27]. The distributed Evolutionary Algorithm for scheduling large-scale problems was suggested by [28]. An efficient parallel tabu search for the blocking job shop scheduling problem was suggested by [29].

Another metaheuristic designed for parallel environments named Mushroom Picking Algorithm (MPA) was proposed by the authors in [30]. The approach proved successful in solving the JSSP instances. Motivated by the good performance of the MPA when solving the JSSP we proposed in [31] an extension of the MPA in the form of the software framework named Mushroom Picking Framework (MPF). The MPF provides a generic functionality of parallel searching for the best solution among population members and is implemented as a multiple-agent system. MPF can be adopted by a user to fit particular combinatorial problem requirements.

In this paper, we further extend the MPF by equipping each solution with a cache memory where recent changes that occur during the search process are stored. The extended MPF is denoted as MPF+. The rest of the paper is organized as follows. Section 2 contains a description of the Mushroom Picking Framework with cache memories. Section 3 recalls briefly the Job Shop Scheduling Problem. Section 4 explains our implementation of the proposed MPF with cache memories for solving JSSP instances. Section five presents the results of the computational experiment held to validate the approach. The final section contains conclusions and suggestions for future research.

II. Mushroom Picking Framework with cache memories

A. Mushroom Picking Algorithm (MPA)

Mushroom Picking Algorithm was introduced in [30] for solving difficult optimization problems. It is characterized by the following features:

- It operates on a population of individuals representing solutions to the given combinatorial optimization problem.
- It uses a set of agents, that may improve a solution or solutions from the population
- Randomly selected agents try to improve randomly selected solutions. If successful, the resulting solution may replace a solution from the population.

Each agent reads a solution or solutions, depending on its improvement method and requirements as to the number of the input solutions (here one or two). Each agent then runs its internal improvement algorithm creating a new solution. If the fitness of a newly generated solution is better than the fitness value of the solution drawn from the population, or is better than the worse fitness of the two initially drawn solutions, it replaces the worse solution.

In order to avoid obtaining a population with a very low diversity of solutions, whenever two solutions drawn from the population are similar to one another, one of them is replaced by a new, random solution. The similarity of solutions is decided through a comparison of the respective fitness values. If these values are identical or differ by a predefined value, solutions are considered identical. The procedure allows for the maintenance of the required diversity of solutions in the population.

B. Mushroom Picking Framework (MPF)

Mushroom Picking Framework works in the following manner:

- The initial population of solutions is generated.
- The search for the best solution is performed in cycles.
- In each cycle, the population of solutions is divided into several subpopulations of equal size.
- Using the Apache Spark functionality subpopulations are independently and in parallel explored by improvement agents. Each subpopulation is processed in a separate thread.
- In each subpopulation the Mushroom Picking Algorithm is used to improve solutions.
- After each cycle, all the solutions from the subpopulations are drawn back into the common memory and shuffled.
- A predefined number of the worst solutions may be replaced by the currently best one. Then the next cycle begins.

The process of searching for the best solution is iterative and runs as described by Algorithm 1. The stopping criterion is defined by the maximum number of consecutive cycles in which the best solution in the population does not improve (the process ends when the best makespan of the solutions has not changed for predefined number of consecutive cycles).

What happens within the method $optimize$ is shown as Algorithm 2. In each subpopulation, the process of applying improvement agents to solutions - MPA - is represented by the $p.applyOptimizations$ in Algorithm 2. Attempted improvements are executed by improvement agents. The set of agents in each subpopulation is identical and consists of an equal number of agents of the same type.

Algorithm 1 Mushroom Picking Framework operation

$solutions \leftarrow$ set of random solutions;
while ! stoppingCriterion **do**
  $solutions \leftarrow solutions.optimize$;
  $bestSolution \leftarrow$ the best solution chosen from $solutions$;
end while
return $bestSolution$;

Algorithm 2 Method *optimize*

Require:
$solutions$ = set of solutions;
k = number of parallel threads in which the solutions will be processed;
Ensure:
$populations \leftarrow solutions$ divided to a list of k-element subpopulations;
$populationsRDD \leftarrow populations$ parallelized in Apache Spark;
$populationsRDD \leftarrow populationsRDD.map(p => p.applyOptimizations)$;
$solutions \leftarrow$ solutions collected from $populationsRDD$;
$solutions \leftarrow solutions$ with l worst solutions replaced with the best solution;

The framework may be used for all problems, for which a task, a solution with a method that returns the fitness value, and a set of agents working as in MPA is defined.

C. Cache memories

For the JSSP instances, we use the MPF implementation proposed in [31] extended by adding the cache memory to each population member. The extended framework will be denoted as MPF+.

The general assumption is that solutions are encoded in the form of lists. In our framework improvement agents try to improve current solutions by moving, swapping, or modifying parts of the lists that encode solutions. The proposed cache memory is used to record and store for each solution the position (index in the list representing the solution), in which the last successful improvement move or change took place. The above feature helps to intensify the search for successful moves in the vicinity of the recent change. The idea of using information stored in the cache memory is to improve the synergistic effects of agent interactions, by providing the current information on which part of the solution they should focus it the next step.

Since we consider solutions that are represented as a list, changing an element in such a list (for example moving or swapping it with another element), results in saving its position together with the respective weight which is allocated by the user. In the next iteration of improvement in $ApplyOptimizations$, the new starting position for a change is drawn at random from the close neighborhood of the element at the saved position, and either the weight's value is reduced by one or - if the next change successfully led to a solution with better makespan - a new position with the maximum weight is remembered. When after several iterations the weight reaches value 0, the saved position receives a random value.

The process described above is shown as Algorithm 3. For simplicity's sake, Algorithm 3 covers the case of a single argument agent. The cache memory is represented by $solution.position$ and $solution.weight$ values. When creating a random solution, its $position$ and $weight$ are set to random position and maximum weight. The radius and the maximum weight are both predefined as the algorithm parameters.

III. Job Shop Scheduling Problem (JSSP)

Job shop scheduling problem (JSSP) is a well-known NP-hard optimization problem in which n jobs $(J_1, \ldots J_n)$ must be processed on m machines $(m_1, \ldots m_m)$.

In JSSP each job consists of a list of operations, the operations must be processed in the exact order as in the given list, and only after all preceding operations have been completed. Also every operation has to be processed on a specific machine in the given time. The operations cannot be interrupted and each machine can process only one operation at a time.

The makespan is defined as the length of the schedule, or the time in which all operations of all jobs will be processed. The JSSP objective is to find such schedule, that its makespan is minimal.

In JSSP a solution may be represented as sequence of jobs' numbers of the length $\leq n \times m$. In this sequence each job j appears at most m times, and the i-th occurence of the job corresponds to the i-th operation of this job. The algorithm in this paper uses this representation to find the solution with the smallest makespan.

Fig. 1 presents a solution of JSSP problem that may be represented by, for example, list (1,2,0,1,1,2,0,0) or (2,1,0,1,2,1,0,0).

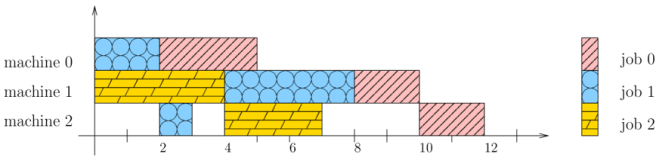

Fig. 1: Solution of JSSP task with makespan 12

IV. MPF+ Implementation for solving JSSP instances

In the Mushroom Picking Framework, one has to define the task, the solution and the agents. We implement the solutions as lists of job numbers - as it has been described in the previous section. All solutions in the initial population are randomly generated. The agents that try to improve solutions transform the lists by changing the order of elements or moving the elements to different positions. In the MPF+ for JSSP, the following agents are used:

Algorithm 3 *ApplyOptimizations* with cache memory of recent changes

Require:
W =maximum weight;
R =radius;
Ensure:
for *iteration* in the given range **do**
  $agent \leftarrow$ agent drawn at random according to some probability set by the user;
  $solution \leftarrow$ random solution;
  if $solution.weight > 0$ **then**
    $newWeight \leftarrow solution.weight - 1$
    $newPosition \leftarrow random(solution.position - R, solution.position + R)$;
  else
    $newWeight \leftarrow solution.weight$;
    $newPosition \leftarrow random(0, solution.size)$;
  end if
  $optimizedSolution$ with $optPosition$ and $optWeight \leftarrow agent(solution, start = newPosition)$;
  if $optimizedSolution$ is better than $solution$ **then**
    return $optimizedSolution$ with $optPosition$ and $optWeight$;
  else
    return $solution$ with $newPosition$ and $newWeight$;
  end if
end for

- RandomSwap - replaces pairs representing jobs on two random positions in the list of pairs. If successful, the position of the first swapped element is remembered as the base for future exploring.
- RandomMove - moves one element representing the job and moves it to another, random position. If successful, the original position of the element is remembered as the base for future exploration.
- RandomOrder - takes a random slice of the list and shuffles the elements in this slice (the order of the slice' elements changes at random). If successful, the middle element of the slice is remembered as the base for future exploration.
- RandomCrossover - requires two randomly drawn solutions. A slice from the first solution is extended with the missing elements in the order as in the second solution. If successful, the middle element of the slice is remembered as the base for future exploration.

Each agent stores in the solution's memory index of one element of the list representing solution. When the solution is again sent to an agent (in the next iteration in the *ApplyOptimizations* method), the agent will draw the starting point for the transformation from the part of the solution given by the range of indices: $(solution.position - R, solution.position + R)$, where R is given as the algorithm parameter.

Each iteration of the *ApplyOptimizations* method starts with drawing at random an agent. The agents are drawn with the following probabilities: 0.28 for each one-argument agent and 0.14 for RandomCrossover agent, to maintain the empirically identified required frequency of calling a single and double argument agents.

V. Results

A. Computational experiment

To validate the proposed approach, we have carried out several computational experiments. Experiments were run on a benchmark dataset for the JSSP problem: the set of 40 instances proposed by Lawrence [32], that have sizes from 5x10 to 15x15. All computations have been run on the Spark cluster at the Centre of Informatics Tricity Academic Supercomputer and Network (CI TASK) in Gdansk. In all experiments 240 subpopulations have been used, each consisting of 3 solutions. These subpopulations have not been processed literally in parallel due to a varying temporary constraint on the number of available nodes. Using a cluster with more allocated nodes would lead to shorter computation times, as demonstrated in [31].

The use of our cache memory has been controlled by two parameters R and W. R is used to define the range of solution elements from which the next starting point for an agent will be drawn. The starting point is drawn from $(position - R, position + R)$. R parameter has been set to 5 in all experiments. W is the weight assigned to a solution after an agent performs a change. Its value was set to 0, 10, or 15, where 0 value results in not using the cache memory at all. For tasks from la01 to la15 and task la31, if the solution was calculated using the cache memory, the weight of 10 was used. For the remaining solutions, their initial weight was set as 15.

The time of computations mainly depends on the number of iterations in one cycle, and the stopping criterion, which

is the maximum number of cycles in which the best solution does not change, denoted mwc. The number of iterations was set to 1000, 3000 or 6000 iterations in one cycle. The mwc was set to 2 or 5. The values of parameters that were used in the experiment are described in the Table I.

TABLE I: Parameters used at the experiments

Task	R **and** W if cache used	**Number of iterations**	
la01	5, 10	1000	2
la02	5, 10	3000	2
la03	5, 10	3000	5
la04	5, 10	3000	2
la05-la15	5, 10	3000	2
la16-la27	5, 15	3000	5
la28-la30	5, 15	6000	5
la31	5, 10	3000	2
la32	5, 15	3000	5
la33	5, 15	3000	2
la34-la35	5, 15	3000	5
la36-la40	5, 15	6000	5

Average computation times and average errors are shown in Table II. BKS stands for the best-known solution (in terms of the solution makespan) and the errors have been calculated for BKS values. The average errors and times have been calculated from at least 30 results.

Tables III and IV contain a comparison of results obtained by MPF+ with results obtained by other recently published algorithms. Table III contains Q-Learning Algorithm (QL, [33]) and a hybrid EOSMA algorithm [34] that mixes the strategies of Equilibrium Optimizer (EO) and Slime Mould Algorithm (SMA). Table IV shows results for the Coral Reef Optimization (CROLS, [35]). The average errors for CROLS, QL and EOSMA algorithms have been calculated based on average results given in the original papers. In the case of the Coral Reef Optimization results reported in [35] were given for only chosen instances of the problem. The Coral Reef algorithm was run for three different reef sizes. For each task in the table, the best Coral Reef Optimization result was chosen from among the three available results in [35]. In the case of QL0 and QL1 [33] running times of algorithms were not given. Algorithm EOSMA needed from over 10 seconds to 10^3 seconds of running time.

Figures 2 and 3 present convergence for six different runs of the algorithm for tasks ls03 and la26 respectively. For both figures such runs of the algorithm have been chosen for which solutions with the best known makespan were found. In three of the runs the cache memories were used (red lines with triangles), and three runs did not use the cache memories (blue lines with circles).

VI. Discussion

From Table II it can be seen, that in many cases intensifying the search using data stored in the proposed cache memory leads to better results obtained in comparable and even occasionally shorter times. To gain better knowledge of the performance of the proposed MPF+ implementation as compared with its earlier version (MPF) we have carried out a pairwise comparison using the Wilcoxon matched pairs tests.

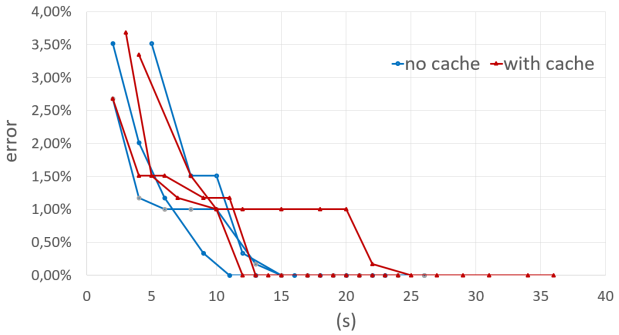

Fig. 2: Convergence for runs with and without cache on la03

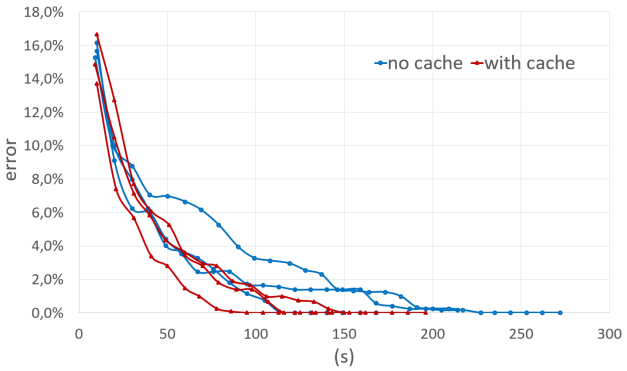

Fig. 3: Convergence for runs with and without cache on la26

The null hypothesis in such a case states that results produced by two different methods are drawn from samples with the same distribution. With T statistics equal to 53.00, Z statistics equal to 2.771429, and a p-value equal to 0.005581, the null hypothesis has to be rejected at the significance level of 0.05.

Analysis of results from Tables III and IV allows observing that MPF+ implementation for solving the JSSP instances performs well as compared with several other approaches offering for numerous instances better performance or shorter computation time.

From Fig. 2 and Fig. 3 it can be noticed that most runs in which the cache was used required less time to finish. The markers show the error of the best makespan found at the time when a cycle ends, and after most of the cycles the best makespan value found so far was better in cases when the cache was used.

VII. Conclusion

The main contribution of the paper is extending the earlier proposed Mushroom Picking Framework by incorporating the, so-called, cache memory. It serves to store recent changes to solutions effected by improvement agents. A novel version of the framework referred to as MPF+ can be used for solving a variety of computationally hard combinatorial optimization problems. The approach takes advantage of better controlling the intensification part of searching for the best solution. The

TABLE II: Average computation times and average errors obtained in the experiment

		MPF		MPF+	
Task	**BKS**	**Avg err**	**Avg time (s)**	**Avg err**	**Avg time (s)**
la01	666	0.00%	3.53	0.00%	**3,07**
la02	655	0.00%	11.33	0.00%	**9,93**
la03	597	0.45%	23.67	**0.25%**	**22,37**
la04	590	0.07%	10.97	**0.02%**	13.33
la05	593	0.00%	8.23	0.00%	8.40
la06	926	0.00%	11.93	0.00%	13.10
la07	890	0.00%	14.33	0.00%	**13.20**
la08	863	0.00%	10.23	0.00%	10.40
la09	951	0.00%	10.03	0.00%	10.10
la10	958	0.00%	12.23	0.00%	**10.63**
la11	1222	0.00%	13.20	0.00%	13.43
la12	1039	0.00%	13.50	0.00%	**13.27**
la13	1150	0.00%	13.57	0.00%	**13.20**
la14	1292	0.00%	13.23	0.00%	13.60
la15	1207	0.00%	18.23	0.00%	20.27
la16	945	0.47%	60.23	**0.34%**	**43.07**
la17	784	0.39%	42.83	**0.27%**	46.43
la18	848	0.44%	45.85	**0.33%**	48.63
la19	842	1.15%	54.13	1.34%	**47.23**
la20	901	0.65%	38.60	**0.63%**	**37.43**
la21	1046	3.94%	121.70	**3.57%**	125.93
la22	927	2.64%	152.43	2.80%	**113.73**
la23	1032	0.05%	96.23	0.07%	**87.07**
la24	935	4.14%	103.90	**4.12%**	111.63
la25	977	4.13%	126.53	**3.75%**	**122.73**
la26	1218	2.18%	199.20	**1.96%**	**197.83**
la27	1235	5.60%	171.40	**5.40%**	200.10
la28	1216	3.23%	321.67	3.31%	419.83
la29	1152	7.95%	363.13	**7.43%**	428.90
la30	1355	0.92%	375.63	**0.60%**	**347.87**
la31	1784	0.00%	146.73	0.00%	155.53
la32	1850	0.00%	227.83	0.00%	235.50
la33	1719	0.00%	222.00	0.00%	**176.17**
la34	1721	0.58%	366.90	0.62%	388.87
la35	1888	0.11%	274.83	**0.06%**	298.97
la36	1268	4.46%	290.15	4.51%	311.85
la37	1397	4.95%	372.18	**4.83%**	380.53
la38	1196	6.42%	434.65	**6.11%**	**411.63**
la39	1233	4.25%	375.90	**3.15%**	436.53
la40	1222	4.37%	350.20	**4.36%**	390.07
avg		1.59%	138.08	1.50%	143.81

mechanism helps enhance the synergetic effects of interactions between agents by providing constantly updated information pointing directly at a part of the solution in which the recent change caused some improvement of the fitness function value. As a test-bed for validation purposes, we have selected one of the classic computationally hard combinatorial optimization problems – the job shop scheduling problem. While incorporating the proposed cache memory has not caused a dramatic improvement in the quality of results, it helped nevertheless to improve some of them, and in many cases has led to a shortening of the computation time. For the JSSP results obtained in the experiments are competitive, even if the cluster environment that has served as the platform to run the programs has not been able to provide fully parallel computations for all threads.

We believe that the MPF+ could be further improved. Future research should focus on finding mechanisms for the automatic setting of weights values depending on the scale of improvements. Another possibility is to take advantage of reinforcement learning techniques for controlling and managing the course of computations.

ACKNOWLEDGMENT

This research was funded by Gdynia Maritime University, grant number WZNJ/2023/PZ/03

REFERENCES

[1] F. Glover and M. Samorani, "Intensification, diversification and learning in metaheuristic optimization," *Journal of Heuristics*, vol. 25, 03 2019. doi: 10.1007/s10732-019-09409-w

[2] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992. ISBN 0-262-11170-5

[3] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA: Addison-Wesley, 1989.

[4] D. B. Fogel, "On the relationship between the duration of an encounter and the evaluation of cooperation in the iterated prisoner's dilemma," *Evol. Comput.*, vol. 3, no. 3, pp. 349–363, 1995. doi: 10.1162/evco.1995.3.3.349. [Online]. Available: https://doi.org/10.1162/evco.1995.3.3.349

TABLE III: Comparison of results obtained by MPF+ with other recently published results (average error and average running time)

Task	**MPF+**		**QL01** [33]	**QL02** [33]	**EOSMA** [34]
	err	**time (s)**	**err**	**err**	**err**
la01	0.00%	3.07	0.00%	0.15%	0.00%
la02	0.00%	9.93	4.58%	3.21%	0.00%
la03	0.25%	22.37	3.69%	5.03%	2.35%
la04	0.02%	13.33	5.08%	3.05%	0.00%
la05	0.00%	8.40	0.00%	0.00%	0.00%
la06	0.00%	13,10	0.00%	0.00%	0.00%
la07	0.00%	13,20	8.65%	0.11%	0.00%
la08	0.00%	10.40	1.51%	0.00%	0.00%
la09	0.00%	10.10	0.00%	0.00%	0.00%
la10	0.00%	10.63	0.00%	0.00%	0.00%
la11	0.00%	13.43	0.00%	0.00%	0.00%
la12	0.00%	13.27	0.00%	0.00%	0.00%
la13	0.00%	13.20	0.00%	0.00%	0.00%
la14	0.00%	13.60	0.00%	0.00%	0.00%
la15	0.00%	20.27	7.87%	4.06%	0.00%
la16	0.34%	43.07	4.02%	5.08%	3.24%
la17	0.27%	46.43	2.04%	2.55%	1.13%
la18	0.33%	48.63	2.95%	3.07%	2.22%
la19	1.34%	47.23	3.92%	6.29%	4.17%
la20	0.63%	37.43	4.22%	4.55%	1.67%
la21	3.57%	125.93	5.83%	12.81%	7.07%
la22	2.80%	113.73	10.25%	18.99%	5.79%
la23	0.07%	87.07	0.58%	6.59%	1.45%
la24	4.12%	111.63	6.95%	15.19%	7.91%
la25	3.75%	122,73	9.93%	14.23%	7.11%
la26	1.96%	197.83	6.90%	17.65%	4.82%
la27	5.40%	200.10	10.45%	18.95%	9.02%
la28	3.31%	419.83	11.68%	15.79%	7.26%
la29	7.43%	428.90	18.32%	24.91%	11.83%
la30	0.60%	347.87	2.58%	14.10%	3.88%
la31	0.00%	155.53	4.09%	6.84%	0.08%
la32	0.00%	235.50	3.46%	8.22%	0.19%
la33	0.00%	176.17	5.70%	6.92%	0.17%
la34	0.62%	388.87	6.22%	12.38%	2.16%
la35	0.06%	298.97	5.14%	11.55%	0.42%
la36	4.51%	311.85	11.59%	16.72%	6.38%
la37	4.83%	380.53	9.74%	14.96%	9.14%
la38	6.11%	411.63	11.54%	19.98%	11.19%
la39	3.15%	436.53	10.14%	18.09%	8.43%
la40	4.36%	390.07	7.28%	22.09%	8.93%
avg	1.37%	126.96	5.17%	8.35%	3.20%

[5] Z. Michalewicz, "Genetic algorithms + data structures = evolution programs," in *Springer Berlin Heidelberg*, 1996. doi: 10.1007/978-3-662-03315-9

[6] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: optimization by a colony of cooperating agents," *IEEE transactions on systems, man, and cybernetics. Part B, Cybernetics : a publication of the IEEE Systems, Man, and Cybernetics Society*, vol. 26 1, pp. 29–41, 1996. doi: 10.1109/3477.484436

[7] R. Poli, J. Kennedy, and T. M. Blackwell, "Particle swarm optimization," *Swarm Intelligence*, vol. 1, pp. 33–57, 1995. doi: 10.1109/icnn.1995.488968

[8] T. Sato and M. Hagiwara, "Bee system: finding solution by a concentrated search," *1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation*, vol. 4, pp. 3954–3959 vol.4, 1997.

[9] H. Ma, S. Shen, M. Yu, Z. Yang, M. Fei, and H. Zhou, "Multi-population techniques in nature inspired optimization algorithms: A comprehensive survey," *Swarm Evol. Comput.*, vol. 44, pp. 365–387, 2019. doi: 10.1016/j.swevo.2018.04.011

[10] P. Jedrzejowicz, "Current trends in the population-based optimization," in *Computational Collective Intelligence*, N. T. Nguyen, R. Chbeir, E. Exposito, P. Aniorté, and B. Trawiński, Eds. Cham: Springer International Publishing, 2019. doi: 10.1007/978-3-030-28377-3_4343. ISBN 978-3-030-28377-3 pp. 523–534.

[11] H. R. Er and N. Erdogan, "Parallel genetic algorithm to solve traveling salesman problem on mapreduce framework using hadoop cluster," *JSCSE*, 01 2014. doi: 10.7321/jscse.v3.n3.57

[12] E. Alanzi and H. Bennaceur, "Hadoop mapreduce for parallel genetic algorithm to solve traveling salesman problem," *International Journal of Advanced Computer Science and Applications*, vol. 10, 01 2019. doi: 10.14569/IJACSA.2019.0100814

[13] Y. Karouani and Z. Elhoussaine, "Efficient spark-based framework for solving the traveling salesman problem using a distributed swarm intelligence method," in *2018 International Conference on Intelligent Systems and Computer Vision (ISCV)*, 2018. doi: 10.1109/ISACV.2018.8354075 pp. 1–6.

[14] P. Jedrzejowicz and I. Wierzbowska, "Apache spark as a tool for parallel population-based optimization," in *Intelligent Decision Technologies 2019*, I. Czarnowski, R. J. Howlett, and L. C. Jain, Eds. Singapore: Springer Singapore, 2020. ISBN 978-981-13-8311-3 pp. 181–190.

[15] E. Alba, G. Luque, and S. Nesmachnow, "Parallel metaheuristics: Recent advances and new trends," *International Transactions in Operational Research*, vol. 20, pp. 1–48, 08 2012. doi: 10.1111/j.1475-3995.2012.00862.x

[16] P. González, X. Pardo, R. Doallo, and J. Banga, "Implementing cloud-based parallel metaheuristics: an overview," *Journal of Computer Science and Technology*, vol. 18, p. e26, 12 2018. doi: 10.24215/16666038.18.e26

TABLE IV: Comparison of results obtained by MPF+ with other recently published results (average error and average running time for chosen la instances)

Task	**MPF+**		**CROLS1** [35]		**CROLS2** [35]	
	err	**time (s)**	**err**	**time (s)**	**err**	**time (s)**
la01	0.00%	3.07	0.00%	39,91	0.00%	15.64
la02	0.00%	9.93	0.00%	40.91	0.00%	15.27
la06	0.00%	13,10	0.00%	151.82	0.00%	92.45
la07	0.00%	13,20	0.08%	148.18	0.00%	88.73
la11	0.00%	13.43	0.00%	224.09	0.00%	136.55
la12	0.00%	13.27	0.03%	228.55	0.00%	149.91
la16	0.34%	43.07	0.29%	125.91	0.39%	132.55
la17	0.27%	46.43	0.17%	198.55	0.39%	130.09
la21	3.57%	125.93	0.27%	269.36	0.63%	165.55
la22	2.80%	113.73	0.62%	185.18	0.56%	265.55
la26	1.96%	197.83	0.98%	281.73	1.01%	440.36
la27	5.40%	200.10	0.33%	260.18	0.34%	447.36
la32	0.00%	235.50	0.16%	453.45	0.12%	418.45
la33	0.00%	176.17	0.08%	643.09	0.29%	617.27
la39	3.15%	436.53	0.70%	675.45	0.37%	502.82
la40	4.36%	390.07	1.33%	585.45	2.16%	495.55
avg	1.37%	126.96	0.32%	281.99	0.39%	257.13

[17] M. Essaid, L. Idoumghar, J. Lepagnot, and M. Brévilliers, "GPU parallelization strategies for metaheuristics: a survey," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 34, pp. 1–26, 01 2018. doi: 10.1080/17445760.2018.1428969

[18] H. Hu, W. Lei, X. Gao, and Y. Zhang, "Job-shop scheduling problem based on improved cuckoo search algorithm," *International Journal of Simulation Modelling*, vol. 17, pp. 337–346, 06 2018. doi: 10.2507/IJSIMM17(2)CO8

[19] Z. Zhang, Z. Guan, J. Zhang, and X. Xie, "A novel job-shop scheduling strategy based on particle swarm optimization and neural network," *International Journal of Simulation Modelling*, vol. 18, pp. 699–707, 12 2019. doi: 10.2507/IJSIMM18(4)CO18

[20] J. Zhu, Z. Shao, and C. Chen, "An improved whale optimization algorithm for job-shop scheduling based on quantum computing," *International Journal of Simulation Modelling*, vol. 18, pp. 521–530, 09 2019. doi: 10.2507/IJSIMM18(3)CO13

[21] X. Chen, B. Zhang, and D. Gao, "Algorithm based on improved genetic algorithm for job shop scheduling problem," in *2019 IEEE International Conference on Mechatronics and Automation (ICMA)*. IEEE Press, 2019. doi: 10.1109/ICMA.2019.8816334. ISBN 978-1-7281-1698-3 p. 951–956. [Online]. Available: https://doi.org/10.1109/ICMA.2019.8816334

[22] F. Wang, Y. Tian, and X. Wang, "A discrete wolf pack algorithm for job shop scheduling problem," in *2019 5th International Conference on Control, Automation and Robotics (ICCAR)*, 2019. doi: 10.1109/ICCAR.2019.8813444 pp. 581–585.

[23] A. Vital-Soto, A. Azab, and M. Baki, "Mathematical modeling and a hybridized bacterial foraging optimization algorithm for the flexible job-shop scheduling problem with sequencing flexibility," *Journal of Manufacturing Systems*, vol. 54, pp. 74–93, 01 2020. doi: 10.1016/j.jmsy.2019.11.010

[24] H. Piroozfard, K. Y. Wong, and A. D. Asl, "A hybrid harmony search algorithm for the job shop scheduling problems," in *2015 8th International Conference on Advanced Software Engineering & Its Applications (ASEA)*, 2015. doi: 10.1109/ASEA.2015.23 pp. 48–52.

[25] R. Krishnaswamy and C. Rajendran, "A novel discrete PSO algorithm for solving job shop scheduling problem to minimize makespan," *IOP Conference Series: Materials Science and Engineering*, vol. 310, p. 012143, 02 2018. doi: 10.1088/1757-899X/310/1/012143

[26] H. Yu, Y. Gao, L. Wang, and J. Meng, "A hybrid particle swarm optimization algorithm enhanced with nonlinear inertial weight and gaussian mutation for job shop scheduling problems," *Mathematics*, vol. 8, no. 8, p. 1355, Aug 2020. doi: 10.3390/math8081355. [Online]. Available: http://dx.doi.org/10.3390/math8081355

[27] C.-W. Tsai, H.-C. Chang, K.-C. Hu, and M.-C. Chiang, "Parallel coral reef algorithm for solving JSP on spark," in *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2016. doi: 10.1109/SMC.2016.7844511 pp. 001 872–001 877.

[28] L. Sun, L. Lin, H. Li, and M. Gen, "Large scale flexible scheduling optimization by a distributed evolutionary algorithm," *Computers & Industrial Engineering*, vol. 128, pp. 894–904, 2019. doi: https://doi.org/10.1016/j.cie.2018.09.025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S036083521830442X

[29] A. Dabah, A. Bendjoudi, A. AitZai, and N. Nouali-Taboudjemat, "Efficient parallel tabu search for the blocking job shop scheduling problem," *Soft Computing*, vol. 23, p. 13283–13295, 12 2019. doi: 10.1007/s00500-019-03871-1

[30] P. Jedrzejowicz and I. Wierzbowska, "Parallelized swarm intelligence approach for solving TSP and JSSP problems," *Algorithms*, vol. 13, no. 6, p. 142, Jun 2020. doi: 10.3390/a13060142. [Online]. Available: http://dx.doi.org/10.3390/a13060142

[31] P. Jedrzejowicz, E. Ratajczak-Ropel, and I. Wierzbowska, "A population-based framework for solving the job shop scheduling problem," in *Computational Collective Intelligence: 13th International Conference, ICCCI 2021, Rhodes, Greece, September 29 – October 1, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021. doi: 10.1007/978-3-030-88081-1_26. ISBN 978-3-030-88080-4 p. 347–359. [Online]. Available: https://doi.org/10.1007/978-3-030-88081-1_26

[32] S. Lawrence, "Resource constrained project scheduling - technical report," Carnegie-Mellon University: Pittsburgh, PA, USA, Tech. Rep., 1984.

[33] M. A. Belmamoune, L. Ghomri, and Z. Yahouni, "Solving a job shop scheduling problem using Q-learning algorithm," in *Service Oriented, Holonic and Multi-Agent Manufacturing Systems for Industry of the Future*, T. Borangiu, D. Trentesaux, and P. Leitão, Eds. Cham: Springer International Publishing, 2023. doi: 10.1007/978-3-031-24291-5_16. ISBN 978-3-031-24291-5 pp. 196–209.

[34] Y. Wei, Z. Othman, K. Mohd Daud, S. Yin, and Q. Luo, "Equilibrium optimizer and slime mould algorithm with variable neighborhood search for job shop scheduling problem," *Mathematics*, vol. 10, p. 4063, 11 2022. doi: 10.3390/math10214063

[35] C.-S. Shieh, T.-T. Nguyen, W.-W. Lin, D.-C. Nguyen, and M.-F. Horng, "Modified coral reef optimization methods for job shop scheduling problems," *Applied Sciences*, vol. 12, no. 19, p. 9867, Sep 2022. doi: 10.3390/app12199867. [Online]. Available: http://dx.doi.org/10.3390/app12199867