Variable Neighborhood Programming for Job Shop Scheduling Problems

Michael-Alexandros Triantoglou[1][0009−0005−4851−4291], Angelo Sifaleras[1][0000−0002−5696−7021], and Rachid Benmansour[2,3][0000−0003−2553−4116]

[1] Department of Applied Informatics, University of Macedonia, School of Information Sciences, 156 Egnatias Str., 54636, Thessaloniki, Greece
mtriantoglou@uom.edu.gr, sifalera@uom.gr

[2] Research Laboratory in Information Systems, Intelligent Systems and Mathematical Modeling, National Institute of Statistics and Applied Economics, Rabat, Morocco
r.benmansour@insea.ac.ma

[3] LAMIH CNRS UMR 8201, INSA Hauts-de-France, Polytechnic University of Hauts-de-France (UPHF), Campus Mont Houy, F-59313 Valenciennes Cedex 9, France

Abstract. The Job Shop Scheduling Problem is a classic combinatorial optimization problem and one of the most well-studied scheduling problems. Several methodologies, both exact and metaheuristic, have already been proposed for the solution of this computationally difficult problem. This work presents for the first time a solution approach based on Variable Neighborhood Programming for the Job Shop Scheduling Problem. Variable Neighborhood Programming is a recent methodology which constitutes a combination of Genetic Programming and Variable Neighborhood Search. In addition, some encouraging comparative computational results are also shown against the state-of-the-art Gurobi optimization solver using medium- and large-scale benchmark instances. The findings of this work have a plethora of modern applications in Manufacturing-as-a-Service online platforms. All experimental evaluations were performed on the Google Cloud Platform.

Keywords: Variable Neighborhood Programming · Job Shop Scheduling Problem · Mixed-Integer Programming · Manufacturing-as-a-Service · Google Cloud Platform.

1 Introduction

Scheduling problems constitute a large area of optimization problems [1]. The Job Shop Scheduling Problem (JSSP) is one of the most well-studied scheduling problems [14]. Thus, several researchers have tried various solution approaches including exact methodologies, metaheuristic, Constraint Programming (CP), and others [26].

In JSSP, there are a set of jobs and a set of machines. Each job consists of a sequence of activities (otherwise, tasks or operations). Each activity must be processed on a specific machine and has a fixed processing time. Activities of

the same job must follow the given order, and each machine can process at most one operation at a time. A schedule assigns starting times to all activities so that the activities on the same machine do not overlap. The goal is usually to finish all jobs as early as possible, which means minimizing the total makespan subject to precedence and machine capacity constraints [18].

The JSSP is also $\mathcal{NP}$-hard since it consists of a generalization of the $\mathcal{NP}$-hard Traveling Salesman Problem (TSP), if we consider a single job (as a TSP salesman) and machines (as TSP cities). Therefore, finding the exact optimal solution becomes very difficult when the problem size increases. For this reason, several researchers have developed parallel implementations that significantly improve performance. Taillard presented a parallel Taboo Search metaheuristic for the solution of the JSSP [22]. Also, machine learning techniques are now widely used for the solution of the JSSP. Zhang & Zhu recently compiled a survey of reinforcement learning methods applied to the JSSP [28].

Scheduling problems also arise in modern Manufacturing-as-a-Service (MaaS) online platforms. Such distributed manufacturing ecosystems usually provide a decomposition of scheduling requests and a composition of supply chains using advanced artificial intelligence and optimization techniques. Recently, Lagos et al. proposed a CP model for the efficient solution of flow shop scheduling problems with multiple objectives in the plastic injection molding domain [13]. The research methodology of this paper will also be applied in similar real-world optimization problems that arise on the Tec4MaaSEs platform `http://tec4maases.eu`.

The research contribution of this paper is the development, for the first time, of a Variable Neighborhood Programming method for the efficient solution of large-scale JSSP instances. The remainder of this paper is organized as follows. Section 2 gives the mathematical formulation of the JSSP. In Section 3, we present the research methodology and an illustrative example. The proposed VNP approach was tested using well-known benchmark results against the state-of-the-art Gurobi optimization solver, and the experimental results are shown in Section 4. Finally, conclusions and future work are discussed in Section 5.

2 Mathematical formulation of the JSSP

Several mathematical formulations have already been proposed for the solution of the JSSP, such as the time-indexed formulation [5], the rank-based formulation [25], and the disjunctive formulation [14]. This section will provide the disjunctive formulation proposed by Mann [14] in 1960, which is considered the best performing Mixed Integer Programming (MIP) model for the JSSP according to theoretical [17] and experimental comparisons [12] in the literature.

Indices and Sets

- $i, k \in \{1, \ldots, n\}$: Index for jobs
- $j \in \{1, \ldots, m\}$: Index for machines

Parameters

- t_{ij}: Processing time of job i on machine j.
- M: A large positive number (big-M).

Decision Variables

- x_{ij}: The start time of the job i on machine j.
- y_{ikj}: A binary variable, where $y_{ikj} = 1$ if job i precedes job k on machine j, and 0 otherwise.
- $C_{\max}$: The makespan (i.e., the completion time of the last job).

Mathematical Formulation

The objective is to minimize the makespan:

$$\text{Minimize} \quad C_{\max} \tag{1}$$

Subject to the following constraints:

$$x_{i,j} + t_{ij} \leq x_{i,j+1} \qquad \forall i \in \{1, \ldots, n\}, \quad \forall j \in \{1, \ldots, m-1\} \tag{2}$$

$$x_{ij} + t_{ij} \leq x_{kj} + M \cdot (1 - y_{ikj}) \quad \forall j \in \{1, \ldots, m\}, \quad \forall i, k \in \{1, \ldots, n\}, i < k \tag{3}$$

$$x_{kj} + t_{kj} \leq x_{ij} + M \cdot y_{ikj} \qquad \forall j \in \{1, \ldots, m\}, \quad \forall i, k \in \{1, \ldots, n\}, i < k \tag{4}$$

$$x_{in} + t_{in} \leq C_{\max} \qquad \forall i \in \{1, \ldots, n\} \tag{5}$$

$$x_{ij} \geq 0 \qquad \forall i \in \{1, \ldots, n\}, \quad \forall j \in \{1, \ldots, m\} \tag{6}$$

$$y_{ikj} \in \{0, 1\} \qquad \forall j \in \{1, \ldots, m\}, \quad \forall i, k \in \{1, \ldots, n\}, i < k \tag{7}$$

The precedence constraints (2) ensure that for any given job, the activities are performed in the correct sequence. This formulation assumes that the technological order of the machines is $1, 2, \ldots, n$ for all jobs. The pair of disjunctive constraints (3) and (4) ensures that no two jobs can be processed on the same machine simultaneously. If $y_{ikj} = 1$, job i must finish before job k starts on machine j (constraint (4) becomes redundant). If $y_{ikj} = 0$, job k must be completed before job i starts (constraint (3) becomes redundant). In addition, the constraint (5) ensures that the makespan must be greater than or equal to the completion time of the last operation of every job. Finally, the constraints (6) and (7) define the domains of the decision variables.

3 Research methodology

Variable Neighborhood Programming (VNP) is a more recent variant of the VNS metaheuristic framework [10, 15] for Automatic Programming (AP). The idea is to treat rules (expressed as AP trees) as programs and then systematically modify them by exploring different neighborhoods. Essentially, it is a combination of Variable Neighborhood Search (VNS) and Genetic Programming (GP). GP is an evolutionary algorithm [19] with several applications in scheduling [27]. Genetic Programming works with "AP trees" that represent small programs or rules. These AP trees have operators (for example $+$, $-$, $\times$, min, max) and terminals, which are the problem's features, such as the processing time or remaining operations.

One simple way to build schedules is by using dispatching rules [23], which are simple heuristics that decide which job should be processed next when a machine becomes free. Examples include FIFO (First In First Out) where we process jobs in the order they arrive, SPT (Shortest Processing Time) where we choose the job with the shortest processing time, LPT (Longest Processing Time) where we choose the job with the longest processing time, MWKR (Most Work Remaining) where we choose the job with the largest amount of work left, EDD (Earliest Due Date), and others. These rules are often used as constructive heuristic methods. In [16]. The SPT rule is one of the oldest and most important. In 1956, Smith proved that ordering jobs by increasing processing time is the best way to minimize the total completion time for a single machine problem [20]. Because of this, SPT is often used as an initial solution in more advanced methods. Dispatching rules can be expressed as priority functions based on job and operation attributes, also called terminals. Commonly used attributes are the following [27]:

- pt: processing time of the current activity
- rt: total remaining processing time of the job
- ro: number of remaining operations of the job
- dd: due date of the job
- wt: waiting time before the activity can start
- idx: index of the current activity in its job sequence (shows the position of the activity inside the job)

By combining these attributes with the operators described above (i.e., $+$, $-$, $\times$, min, max), more complex rules can be created. For example, the expression $\min(pt \times 1.0, ro \times 1.0) + rt \times 1.0$ is a composite rule that can be used to assign priorities. The interval $[-1, 1]$ is often used for the coefficient values at the terminal nodes [15]. This choice makes the search process simpler and faster while still giving enough flexibility to adjust the solution. Using this limited range has been reported to improve convergence speed compared to larger intervals, which is why it is considered a suitable option in practice.

In this work, the SPT priority rule, according to which jobs are executed in order of increasing processing time, is used to initialize the decision AP tree in the

VNP. This rule [20] was first presented by Smith in 1956 as an optimal strategy for minimizing the mean flow time in single-stage scheduling environments.

VNS is a strategy that improves solutions by systematically applying different types of changes, called "neighborhoods". If one neighborhood does not give an improvement, the search moves to the next one, in order to avoid getting stuck in local optima. The VNS metaheuristic also finds a plethora of applications in various fields [4, 6] and also in scheduling [2, 3, 7].

In the VNP context, a neighborhood is defined by a small change in the rule structure, such as changing a weight, replacing an operator, or adding a new subtree. By moving from smaller to larger neighborhoods, VNP can escape local optima and generate more effective dispatching rules. In this way, VNP is used to evolve scheduling heuristics for the JSSP with the objective of minimizing the makespan. The starting point is an initial rule (for example, SPT), and then various neighborhoods can be applied.

3.1 Solution representation

The solution representation, (i.e., AP tree) of this problem is shown below in Figure 1.

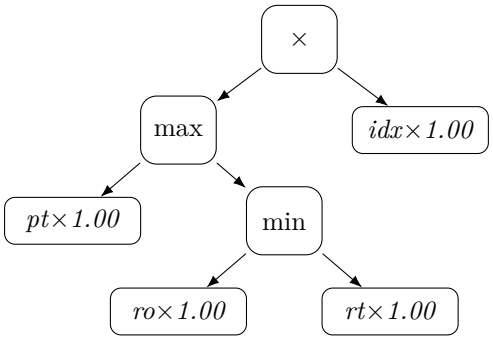


Fig. 1: Solution representation using AP tree

At the root there is a × operator, which multiplies two branches. On the left branch, there is a *max* operator that compares two values. The first value is obtained by multiplying *pt* by the weight 1.00, while the second value comes from another activity: a *min* between $ro \times 1.00$ and $rt \times 1.00$. On the right branch, the value is simply $idx \times 1.00$. Therefore, the solution corresponds to the following function:

$$(\max((pt \times 1.00), \min((ro \times 1.00), (rt \times 1.00)))) \times (idx \times 1.00)$$

3.2 Variable Neighborhood Programming

In the proposed study, a General VNP scheme was used in a manner similar to that of General VNS. Thus, the General VNP consists of the exploitation phase (i.e., Variable Neighborhood Descent - VND) and the exploration phase (i.e., shaking process). The proposed VNP algorithm employs the set $\mathcal{N}$ of the following four neighborhoods that are applied to the rule tree.

- N_1: change terminal/weight (mutate terminal),
- N_2: replace operator,
- N_3: add subtree, and
- N_4: replaces subtree.

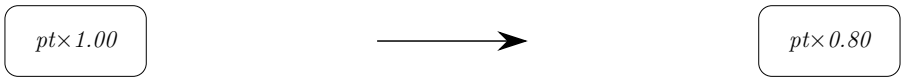
These four neighborhoods correspond to classic GP mutations (terminal / function / subtree mutation), as described by Koza in [11]. In addition, recent work on VNP study specific neighborhood structures for AP trees [9]. The first three neighborhoods as shown in Figure 2 are used in the intensification phase (i.e., in the VND part). The fourth neighborhood as shown in Figure 3 is used in the diversification phase of the VNP (i.e., the shake procedure of the VNP).

Since three neighborhoods are used during the local search, the cardinality number of the neighborhoods (i.e., l_{max}) will be equal to three. The VND algorithm is shown below in the pseudocode 1. The neighborhood structure is sequentially changed at each iteration once there is no other improvement using the current neighborhood structure.

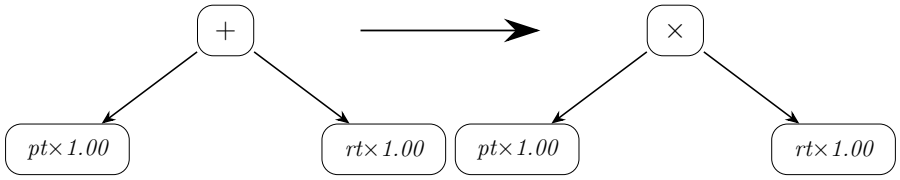
Algorithm 1 VND

Input: $T, l_{max}, \mathcal{N}$
Output: T'
repeat
  stop $\leftarrow$ **true**
  $l \leftarrow 1$
  while $l \leq l_{max}$ **do**
    $T' \leftarrow \arg\min_{y \in \mathcal{N}_l(T)} f(y)$ {Find the best makespan in the current neighborhood}
    if $f(T') \leq f(T)$ **then**
      $T \leftarrow T'$ {make a move}
      $l \leftarrow 1$
      stop $\leftarrow$ **false**
    else
      $l \leftarrow l + 1$ {next neighborhood}
    end if
  end while
until stop = **true**
return T'

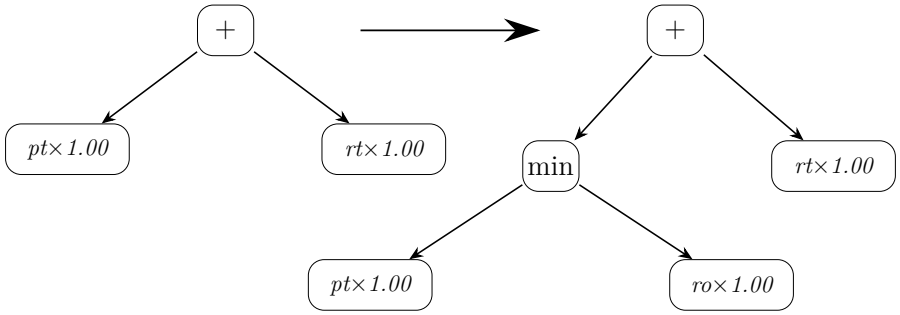
The shaking process in VNP follows the same idea as in the VNS method. It is an adaptive shaking using a different neighborhood (N_4) that replaces subtrees



(a) The first neighborhood operator (*mutate_terminal*) changes the weight of the initial node.



(b) The second neighborhood operator (*replace_operator*) changes the operator of the root from + into ×.



(c) The third neighborhood operator (*add_subtree*) adds a new left subtree $\min(pt, ro)$.

Fig. 2: The three neighborhoods used for local search in the VND algorithm.

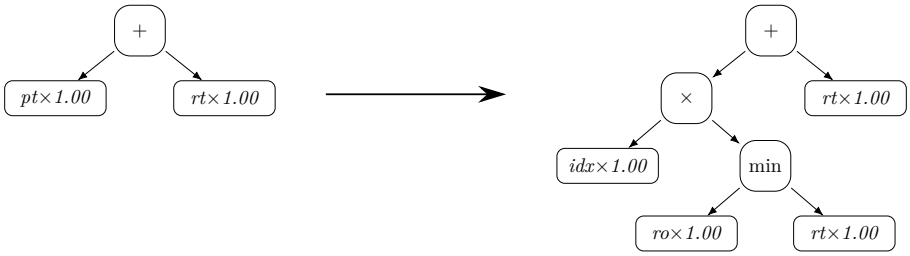


Fig. 3: The fourth neighborhood operator (*replace_subtree*) which replaces a subtree, is used in the shaking phase.

for scheduling rules based on GP. The intensity level of the shaking process (that is, how many times we repeat a non-improved solution to escape from the local optimum) is denoted by k_{max}. Its purpose is to escape from the current local optimum by applying a different neighborhood, which randomly changes the priority rule tree. In the present implementation, shaking allows replacing parts of the AP tree with randomly generated structures, while keeping a restriction on the AP tree size to avoid excessive growth in complexity and k_{max} was set equal to three. In this way, VNP avoids premature convergence and explores new areas of the solution space for the JSSP. The shaking procedure is shown below in the pseudocode 2.

Algorithm 2 Shake

Input: T, k
Output: T'
for $i \leftarrow 1$ **to** k **do**
    Choose randomly $T' \in N_4(T)$
    $T \leftarrow T'$
end for
return T

The complete General VNP algorithm is shown below in the pseudocode 3.

Algorithm 3 General VNP for JSSP

Input: jobs, machines, activities, k_{max}, l_{max}, $\mathcal{N}$
Output: schedule, makespan
{Initialization}
Create an AP tree solution (T) using the SPT rule
repeat
    $k \leftarrow 1$
    while $k \leq k_{max}$ **do**
        $T' \leftarrow \text{Shake}(T, k)$
        $T'' \leftarrow \text{VND}(T', l_{max}, \mathcal{N})$
    end while
until maximum CPU time limit

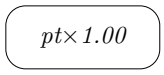
3.3 Illustrative example

To better understand how the three neighborhoods work, let us assume the following small 3 × 3 instance in the common Taillard format [21]:

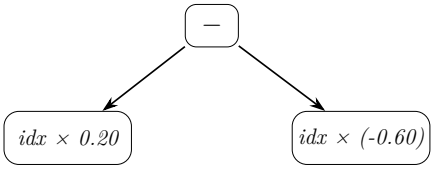
```
3 3
0 3 1 2 2 2
1 2 2 1 0 4
2 4 0 3 1 1
```

As can be seen from the header line, there are three jobs and three machines. Each job consists of ordered sequences of activities in the following lines. Thus, according to the first row (Job 0), the first activity must be on machine 0 and takes 3 units of time, the second activity must be on machine 1 and takes 2 units of time, and finally the third activity must be on machine 2 and takes 2 units of time.

Figure 4a shows the initial solution (i.e., terminal node). The initial solution is generated using the dispatching rule: Processing time (pt) $\times$ 1.00. This means that the priority of each activity is determined solely by its processing time.



(a) At first, SPT generates an initial AP tree solution (root terminal), which corresponds to a pt node with weight 1 and a makespan of 15.

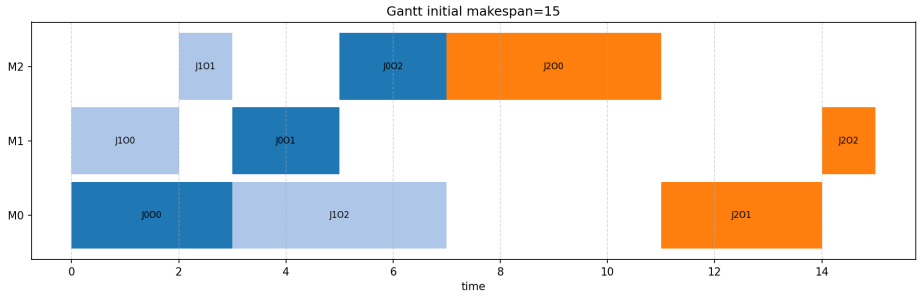


(b) Final AP tree solution with a makespan of 11.

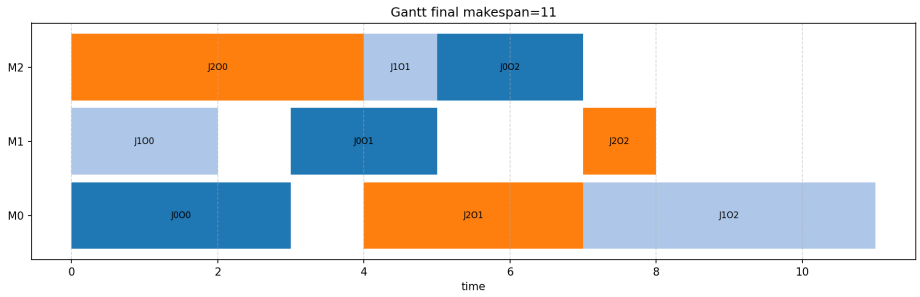
Fig. 4: Initial and final AP tree solutions.

Figure 4b shows the final solution after several iterations for this specific example. The final solution is generated using the dispatching rule: $(idx \times 0.20) - (idx \times (-0.60)) = 0.80 \times idx$. Therefore, the priority score is simply: Priority $= 0.80 \times idx$. We use the rule $Priority = 0.80 \times idx$ and always pick the job with the smallest priority. Because 0.80 is positive, this is the same as sorting by idx: all first activities ($idx = 0$) are chosen before any second ones ($idx = 1$), then the third ones ($idx = 2$), and so on. If two jobs have the same idx, we break ties by the job's index (lower id first). After choosing an activity, we place it at the earliest time when both its job and the required machine are free. In short, the schedule is built layer by layer across stages: the first stage of every job, then the second stage of every job, then third, etc.

Figure 5 shows the two Gantt charts corresponding to the initial and final solutions, respectively. As can be seen, the initial makespan is equal to 15 and the final makespan was equal to 11.



(a) Initial solution.



(b) Final solution.

Fig. 5: Initial and final solutions.

4 Experimental results and comparison

The computational experiments were performed on a virtual machine on the Google Cloud Platform running 64-bit Windows Server 2022 Datacenter with an Intel Xeon CPU at 3.10 GHz featuring 1 socket with 8 cores and 16 threads (2 threads per core) and 64 GB main memory. Also, both VNP and Gurobi were implemented in Python 3.13.7.

Efforts were made to solve each one of them using the latest version of the state-of-the-art Gurobi optimizer v12.0.3 within a reasonable amount of time set to 10 minutes. The time limit (stopping condition) of our VNP approach was 30 seconds. However, due to the increased computational difficulty, Gurobi was unable to solve any instance to optimality. Thus, Gurobi has only computed an upper bound of the optimal objective value. Therefore, Table 1 reports the comparative numerical results regarding the incumbent solutions found by Gurobi (time limit = 600 secs) and the proposed implementation of the VNP (time limit = 30 secs).

Each implementation was tested using well-known benchmark instances with dimensions ranging from 20 to 50 machines and jobs ranging from 15 to 100. The data set is publicly available online at `https://www.jobshoppuzzle.com/`

`benchmarks.html`. These instances belong to the standard general JSSP and not in any other special case; thus, the machine order is not the same for all jobs.

Each instance was solved 10 times using the VNP method, and the results of the comparative computational study are shown in Table 1, where the best values per instance are indicated in bold font. The name and dimension of each instance are depicted in the first two columns. Columns three and four show the average makespan computed by the proposed VNP method and the makespan found by the Gurobi optimizer, respectively.

Table 1: Comparative computational study VNP Vs Gurobi

Instance	Dimension	Gurobi Makespan (600 secs)	VNP Makespan (30 secs)
Ta54	50x15	3,354	3,303.9
Ta55	50x15	3,498	3,497.0
Ta56	50x15	3,690	3,348.2
Ta60	50x15	3,265	3,198.4
Ta61	50x20	4,098	3,659.2
Ta62	50x20	3,949	3,792.8
Ta65	50x20	4,269	3,557.5
Ta66	50x20	3,704	3,573.9
Ta67	50x20	3,996	3,662.1
Ta70	50x20	4,629	3,899.8
Ta71	100x20	69,198	6,900.1
Ta73	100x20	68,300	6,656.5
Ta74	100x20	69,133	6,103.4
Ta75	100x20	57,903	6,571.0
Ta76	100x20	69,335	6,406.4
Ta77	100x20	78,561	6,249.0
Ta80	100x20	72,573	6,203.6

In addition, a maximum depth of three levels was used for any new subtree in the proposed implementation, and also a size limit of 200 nodes was set for the entire tree. In case the tree exceeds this limit, then it is pruned by replacing a subtree with a terminal node.

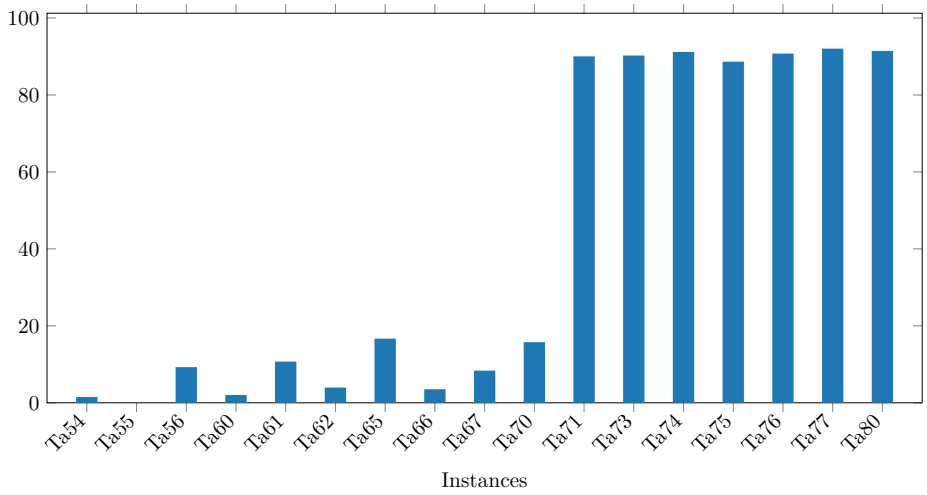


Fig. 6: Improvement of VNP over Gurobi (%)

Figure 6 shows the percentage improvement (%) of the proposed VNP approach compared to the Gurobi optimizer. As can be seen, the computational benefits of VNP increase as the dimension of the problem increases. Although only the Gurobi solver utilized the 16 threads, since the VNP code was not parallelized, VNP found better solutions than Gurobi in a short amount of time.

4.1 Discussion

The JSSP is $\mathcal{NP}$-hard with a massive search space, and the classical disjunctive model has a weak linear relaxation, making branch-and-bound inefficient. Therefore, Gurobi often finds it difficult to obtain good incumbent solutions within the time limit. From the above Table 1 it can be seen that the VNP compared to Gurobi found a smaller makespan in all these instances. Also, in some larger instances with 20 machines and 100 jobs, VNP in just a few seconds outperformed Gurobi that was running with 16 threads even in 600 seconds and found a much better solution. The largest difference is in large instances, where VNP gives up to 90% better solutions. This shows very clearly that VNP works better when the problem size grows, and thus constitutes a very competitive method for other complex scheduling problems.

5 Conclusions and Future Work

The results show that the VNP approach is promising for the JSSP. The proposed VNP method outperformed the Gurobi optimizer in situations requiring a quick solution. Therefore, the proposed VNP solution method can be a useful tool for

managers in competitive real-world Manufacturing-as-a-Service ecosystems such as the Tec4MaaSEs online platform.

For future research, it is suggested to investigate more neighborhoods within the VNP framework and try to apply the method to larger instances of JSSP [24]. Also, a computational comparison with other efficient solution methods, such as CP, would be very interesting. In addition, other researchers have suggested more advanced operators, such as the Elementary Tree Transformation Local Search (ETTLS) [8]. Adding ETTLS and comparing it with the proposed method could also be a future extension of this study. Finally, another direction is to analyze the performance of commercial solvers when combined with VNP as a warm-start strategy.

Acknowledgements

This research was funded by the European Health and Digital Executive Agency, Project: 101138517, Tec4MaaSEs, HORIZON-CL4-2023-TWIN-TRANSITION-01.

Google Cloud Platform (GCP) resources were provided by the Greek National Infrastructures for Research and Technology GRNET and funded by the EU Recovery and Resiliency Facility.

References

1. Agnetis, A., Billaut, J.C., Pinedo, M., Shabtay, D.: Fifty years of research in scheduling — Theory and applications. European Journal of Operational Research **327**(2), 367–393 (2025)
2. Benbrik, O., Benmansour, R., Elidrissi, A., Sifaleras, A.: Advanced algorithms for the reclaimer scheduling problem with sequence-dependent setup times and availability constraints. In: Metaheuristics International Conference. MIC 2024. Lecture Notes in Computer Science. vol. 14753, pp. 291–308. Springer, Cham (2024)
3. Benmansour, R., Sifaleras, A.: Scheduling in parallel machines with two servers: the restrictive case. In: Variable Neighborhood Search. ICVNS 2021. Lecture Notes in Computer Science. vol. 12559, pp. 71–82. Springer (2021)
4. Benmansour, R., Sifaleras, A., Mladenović, N. (eds.): Variable Neighborhood Search. 7th International Conference, ICVNS 2019, Rabat, Morocco, October 3-5, 2019, Revised Selected Papers, LNCS, vol. 12010. Springer, Cham (2020)
5. Bowman, E.H.: The schedule-sequencing problem. Operations Research **7**(5), 621–624 (1959)
6. Brimberg, J., Salhi, S., Todosijević, R., Urošević, D.: Variable neighborhood search: The power of change and simplicity. Computers & Operations Research **155**, 106221 (2023)
7. Elidrissi, A., Benmansour, R., Sifaleras, A.: A general variable neighborhood search for the parallel machine scheduling problem with two common servers. Optimization Letters **17**(9), 2201–2231 (2023)
8. Elleuch, S., Hansen, P., Jarboui, B., Mladenović, N.: New VNP for automatic programming. Electronic Notes in Discrete Mathematics **58**, 191–198 (2017)

9. Elleuch, S., Jarboui, B.: Analysis of six different GP-tree neighborhood structures. In: International Conference on Intelligent Systems Design and Applications. pp. 1237–1249. Springer (2021)
10. Elleuch, S., Jarboui, B., Mladenović, N., Pei, J.: Variable neighborhood programming for symbolic regression. Optimization Letters **16**(1), 191–210 (2022)
11. Koza, J.R.: Genetic programming: On the programming of computers by means of natural selection cambridge. MA: MIT Press (1992)
12. Ku, W.Y., Beck, J.C.: Mixed integer programming models for job shop scheduling: A computational analysis. Computers & Operations Research **73**, 165–173 (2016)
13. Lagos, A.I., Avgerinos, I., Zois, G., Mourtos, I., Casla, P.: Optimising a manufacturing-as-a-service platform through mathematical modeling. In: IFIP International Conference on Advances in Production Management Systems. pp. 478–493. Springer (2025)
14. Manne, A.S.: On the job-shop scheduling problem. Operations Research **8**(2), 219–223 (1960)
15. Mladenović, N., Jarboui, B., Elleuch, S., Mussabayev, R., Rusetskaya, O.: Variable neighborhood programming as a tool of machine learning. In: Black Box Optimization, Machine Learning, and No-Free Lunch Theorems, pp. 221–271. Springer (2021)
16. Nguyen, S., Zhang, M., Johnston, M., Tan, K.C.: Genetic programming for job shop scheduling. In: Bansal, J.C., Singh, P.K., Pal, N.R. (eds.) Evolutionary and Swarm Intelligence Algorithms, pp. 143–167. Springer (2018)
17. Pan, C.H.: A study of integer programming formulations for scheduling problems. International Journal of Systems Science **28**(1), 33–41 (1997)
18. Pinedo, M.L.: Scheduling: Theory, Algorithms, and Systems, Sixth Edition. Springer (2022)
19. Sette, S., Boullart, L.: Genetic programming: principles and applications. Engineering applications of artificial intelligence **14**(6), 727–736 (2001)
20. Smith, W.E.: Various optimizers for single-stage production. Naval Research Logistics Quarterly **3**(1-2), 59–66 (1956)
21. Taillard, E.: Benchmarks for basic scheduling problems. European Journal of Operational Research **64**(2), 278–285 (1993)
22. Taillard, E.D.: Parallel taboo search techniques for the job shop scheduling problem. ORSA Journal on Computing **6**(2), 108–117 (1994)
23. Đurasević, M., Jakobović, D.: Evolving dispatching rules for optimising many-objective criteria in the unrelated machines environment. Genetic Programming and Evolvable Machines **19**(1), 9–51 (2018)
24. Van Hoorn, J.J.: The current state of bounds on benchmark instances of the job-shop scheduling problem. Journal of Scheduling **21**(1), 127–128 (2018)
25. Wagner, H.M.: An integer linear-programming model for machine scheduling. Naval research logistics quarterly **6**(2), 131–140 (1959)
26. Xiong, H., Shi, S., Ren, D., Hu, J.: A survey of job shop scheduling problem: The types and models. Computers & Operations Research **142**, 105731 (2022)
27. Zhang, F., Nguyen, S., Mei, Y., Zhang, M.: Genetic programming for production scheduling. Springer (2021)
28. Zhang, X., Zhu, G.Y.: A literature review of reinforcement learning methods applied to job-shop scheduling problems. Computers & Operations Research **175**, 106929 (2025)