XLVIII Symposium on Operational Research
SYM-OP-IS 2021
Banja Koviljača, 20-23. September 2021.
MATΦ University of Belgrade Faculty of Mathematics
Mathematical Institute SANU

GENERAL VARIABLE NEIGHBORHOOD SEARCH FOR THE WEIGHTED SCHEDULING PROBLEM WITH DEADLINES AND RELEASE TIMES

LUKA MATIJEVIĆ[1], UNA STANKOVIĆ[1], TATJANA DAVIDOVIĆ[1]

[1] Mathematical Institute, Kneza Mihaila 36, Belgrade, Serbia
{luka,una,tanjad}@mi.sanu.ac.rs

Abstract: *We consider a non-preemptive scheduling problem with deadlines and release times, where each task is associated also a nonnegative weight, representing price to be rewarded when task is scheduled for execution. The problem consists of finding a subset of tasks that are going to be scheduled and ordering them in such a way to maximize the price (sum of weights) of all scheduled tasks. Since the problem is NP-hard, we propose a metaheuristic approach, using the General Variable Neighborhood Search (GVNS). This approach is tested on a set of randomly generated instances, and the results obtained this way suggest that GVNS is able to find a high-quality solutions in a short amount of time.*

Keywords: *Combinatorial optimization, scheduling problem, identical machines, metaheuristics*

1. INTRODUCTION

The scheduling problem and its variations are among the most frequently investigated problems in the field of combinatorial optimization. Its basic variants were formulated by Graham [2] in 1966 and remained relevant till today. The variations are usually classified according to some properties of the problem [6]: task data (such as task length, release time, deadline, etc.), task characteristics (such as precedence relations, preemption, etc.), machine environment (single or multiple machines, homogeneity of machines, etc.), and optimality criteria.

Here we consider a non-preemptive scheduling problem with multiple identical machines, where each of the tasks is characterized by a deadline, release time, weight, and length. This variant of scheduling problem can have many practical applications, and we are primarily interested in is the use of the developed scheduling algorithm as a part of *Proof-of-Useful-Work* (**PoUW**) in blockchain systems [6]. A review of the relevant literature is presented in the first part of our research [7].

In [7], we proposed two MILP formulations for the aforementioned problem, and used them within the GLPK solver. Since the solver was unable to give us satisfactory solutions within a limited amount of time, we naturally switch to metaheuristics. The main contribution of this paper is the development of the GVNS algorithm, which already proved itself successful for dealing with other scheduling problems [3, 5, 9, 10]. A set of five neighborhood structures is identified, all five are used in the shaking step, while only two of them are used in the local search step. We also present a greedy procedure for finding a high-quality initial solution. This approach is tested on a set of randomly generated instances, differing in the number of tasks and machines.

This paper is organized as follows. In Section 2 we briefly describe the considered problem. Section 3 is devoted to the implementation of the proposed GVNS, description of neighborhood structures, and generation of the initial solution. In Section 4 we present the results of the experimental analysis of comparing the results obtained by GVNS and GLPK using MILP formulations. Finally, concluding remarks are presented in Section 5.

2. PROBLEM DESCRIPTION

The problem can be stated as follows. We are given a set of tasks to be scheduled on a set of identical machines. Tasks are defined by their lengths, weights, deadlines and release times. We consider a non-preemptive version of the problem in which tasks cannot be interrupted before they are completely finished. Deadline constraints prevent a task to complete after a certain time unit, while the release time specifies how many time units have to pass before the task can begin its execution. In our version, empty time slots in which a machine is idle are not permitted. If a task can't be assigned to any of the machines without violating some constraints, it is not going to be scheduled. This is different from the much more common versions of this problem, which allow (with a certain penalty) to schedule tasks missing their deadlines. We can fill all the potential empty timeslots with some dummy unit tasks with zero weights. The reasoning behind this is explained in more detail in our paper [7]. The objective is to maximize the total sum of weights for scheduled tasks, taking into account constraints

imposed by deadlines and release times. More formal description of our problem, together with a mathematical formulation can be found in [7].

3. GVNS IMPLEMENTATION

Variable neighborhood search (VNS) is a metaheuristic method proposed by Mladenović and Hansen [8]. In its basic variant, it employs a systematic change of neighborhoods, together with an intensified search within a single neighborhood. VNS consists of three main steps: shaking, local search, and neighborhood change. Shaking step perturbs the incumbent solution, to prevent being stuck in a local optimum. This is done by randomly choosing a neighbor from the current neighborhood. The perturbed solution is then improved by some local search method. If the the obtained local optimum is better than the incumbent solution, it becomes the new incumbent solution, and the search continues from the first neighborhood, otherwise, the search moves to the next neighborhood. There are various versions of VNS [4]. Here, we explore the *General Variable Neighborhood Search* (**GVNS**), which uses *Variable Neighborhood Descent* (**VND**) as its local search procedure. In this section, we present in detail some important parts of this method.

3.1. Solution representation

A solution is represented as an array of $(K+1)$ double-linked lists, where K is the number of available machines. In each list, we store indices of tasks that should be executed on the corresponding machine, in the order in which they should be executed. The $(K+1)$-th list stores unscheduled tasks, that is, tasks that do not contribute to the value of the objective function. Therefore, constraints do not apply to the tasks in this list. The structure used for representing a solution can be seen in Figure 1.

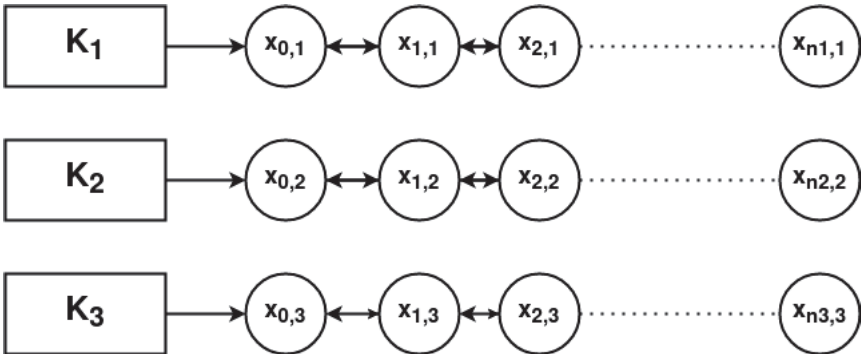

Figure 1 Representation of a solution

3.2. Determination of the initial solution

The initial solution is constructed in a greedy way. First, we sort tasks according to their deadlines in ascending order. Then, we iterate through all of the tasks trying to assign them to one of the machines. If a task cannot be assigned to a machine without violating some of the constraints, we try to assign it to the next machine. In order to balance the load over all of the machines, for each task, we choose a different machine as a starting point, in a round-robin manner. If a task cannot be assigned to any of the machines, we assign it to the dummy machine. The pseudo-code for this approach can be seen in Algorithm 1. A solution generated this way is always feasible.

3.3. Neighborhood structures

Properly chosen neighborhood structures can greatly influence the overall performance of GVNS. We consider five different neighborhood structures, classified into two categories: *active* and *inactive*. Active neighborhoods work only with tasks that are already scheduled, moving them to different time slots. On the other hand, inactive neighborhoods work with unscheduled tasks, trying to insert them into the schedule.

1. **Swap_active(k)** ($N_1(r,s)$) - This neighborhood selects a pair of blocks consisting of r and s consecutive tasks assigned to the same machine, and exchanges their places if possible;

Algorithm 1 Solution initialization

```
1: procedure INITSOLUTION(number of tasks N, number of machines K, tasks)
2:     solution ← emptySolution()
3:     tasks ← sortTasksByDeadlinesAscending(tasks)
4:     i ← 0
5:     for task ∈ tasks do
6:         inserted ← false
7:         j ← i
8:         while j < i + K do
9:             m ← j mod K
10:            if task doesn't violate constraints for machine m then
11:                solution[m].add(task)
12:                inserted ← true
13:                break
14:            end if
15:            j ← j + 1
16:        end while
17:        if not inserted then
18:            solution[K].add(task)
19:        end if
20:        i ← (i + 1) mod K
21:    end for
22:    return solution
23: end procedure
```

2. **Exchange_active(k)** ($N_2(r,s)$) - This neighborhood selects a pair of blocks consisting of r and s consecutive tasks assigned to two different machines, and exchanges their places if possible;
3. **Relocate_active(k)** ($N_3(r)$) - This neighborhood moves a block of r consecutive tasks from one machine to the other if possible;
4. **Exchange_inactive(k)** ($N_4(r,s)$) - This neighborhood selects a pair of blocks consisting of r and s consecutive tasks assigned to two different machines, one of which is the dummy machine, and exchanges their places if possible;
5. **Relocate_inactive(k)** ($N_5(r)$) - This neighborhood moves a block of r consecutive tasks from the dummy machine to some other machine if possible.

Considering the fact that active neighborhoods cannot contribute to the value of the objective function, they are used only during the shaking step. The logic behind this is that even though they cannot provide a better solution, changing the ordering of the tasks may potentially open a time slot for an unscheduled task.

3.4. Shaking

The shaking step of our procedure consists of finding a neighbor at the distance l from the current solution x. It is done by repeating l times the following steps: randomly chose one of the neighborhood structures and select a random neighbor of the incumbent solution. The pseudo-code of this procedure is presented in Algorithm 2.

3.5. Variable Neighborhood Descent

VND procedure explores neighborhoods $N_4(1,1)$ and $N_5(1)$, using the First Improvement principle. The ordering of neighborhoods was determined experimentally. At the beginning, the procedure searches for an improvement in the neighborhood $N_4(1,1)$. This continues as long as an improvement can be found. In case that procedure cannot find any improvement in the neighborhood $N_4(1,1)$, it moves to the next neighborhood - $N_5(1)$. If the improvement is obtained in $N_5(1)$, the search continues from $N_4(1,1)$, otherwise, the VND procedure ends. The pseudo-code of this procedure is given in Algorithm 3.

3.6. The structure of the proposed GVNS

Our GVNS can be presented by the pseudo-code given in Algorithm 4. After the instance is read, the first feasible solution is constructed using the described greedy procedure. GVNS then iterates its steps until the

Algorithm 2 Shaking

```
1: procedure SHAKING(solution x, value of neighborhood index l)
2:     i ← 0
3:     x' ← x
4:     while i < l do
5:         i ← i + 1
6:         Randomly select a value p ∈ {0,1,2,3,4}
7:         switch (p) do
8:             case 0 :
9:                 x' ← find a random neighbor from N1(1,1) of x'
10:            case 1 :
11:                x' ← find a random neighbor from N2(1,1) of x'
12:            case 2 :
13:                x' ← find a random neighbor from N3(1) of x'
14:            case 3 :
15:                x' ← find a random neighbor from N4(1,1) of x'
16:            case 4 :
17:                x' ← find a random neighbor from N5(1) of x'
18:    end while
19:    return x'
20: end procedure
```

Algorithm 3 Variable Neighborhood Descent

```
1: procedure VND(solution x)
2:     imp ← true
3:     x' ← x
4:     while imp do
5:         imp ← LS(N4(1,1), x')
6:         if not imp then
7:             imp ← LS(N5(1), x')
8:         end if
9:     end while
10:    return x'
11: end procedure
```

stopping criterion is met, which is a time limit in our case. More precisely, the inner loop, depending on the neighborhood index for shaking, is performed. The algorithm perturbs the solution and then tries to improve it with VND. If the newly found solution is better than the incumbent solution, it is accepted as the new incumbent solution and the search continues from the first neighborhood l_{min}. Otherwise, the neighborhood index for shaking is increased.

4. EXPERIMENTAL EVALUATION

GVNS presented in this paper was written in the C++ programming language and executed on a laptop with an Intel i7-10750H processor and 32GB of RAM. The obtained results are compared to results obtained by *GNU Linear Programming Kit* (GLPK), executed on the same machine. For more information about the mathematical model used by GLPK, please refer to [7].

For the purpose of testing our algorithm, we randomly generated a set of ten test instances, with the number of machines varying between 2 and 4, and the number of tasks ranging between 10 and 45. When testing these instances with GLPK, we limited CPU time to 1800 seconds, while the time limit imposed on GVNS was 300 seconds. In order to prove the stability of the results obtained by GVNS, for every instance we repeated the test 30 times, with a different seed value for the random number generator. The parameters l_{min} and l_{max} were

Algorithm 4 GVNS

```
1: procedure GVNS(Instance to be solved,l_min,l_max,runtime)
2:     x ← initSolution()
3:     while Time < runtime do
4:         l ← l_min
5:         while l < l_max do
6:             x' ← shake(x,l)
7:             x'' ← vnd(x')
8:             if f(x'') > f(x) then
9:                 x ← x''
10:                l ← l_min
11:                t_min ← Time
12:            else
13:                l ← l + 1
14:            end if
15:        end while
16:    end while
17:    return Solution x, minimal time t_min
18: end procedure
```

determined by using the iRace[1] package for the R programming language. With the budget of 200 tests, the algorithm determined that the best possible values are $l_{min} = 2$ and $l_{max} = 19$.

The results obtained by our GVNS are shown side by side with the results from GLPK in Table 1. The first three columns in this table describe instances, where N is the number of tasks to be scheduled, and K is the number of available machines. The second part of Table 1 (columns 4-7) shows the results from GLPK, namely, the objective function value (with the gap in parentheses) and the corresponding CPU time in seconds. The third part (columns 8 and 9) shows the results obtained from GVNS. The average objective value obtained over 30 independent executions is shown together with the corresponding standard deviation in column 8, while the average t_{min} (minimum CPU time) and its standard deviation are displayed in column 9. The last column in the table represents the objective function value of the initial GVNS solution, generated by the greedy algorithm.

As we can see from Table 1, GVNS outperformed GLPK for all instances, either by finding a better solution or by finding the same solution in a shorter time. Compared to MODEL I (MODEL II), GVNS was able to find a better solution than GLPK in eight (six) out of ten instances, within a negligible CPU time. We can also see that GVNS had no problem with finding a good solution even for larger instances (Ex9 and Ex10), for which GLPK was unable to find even the first feasible solution. What we have found especially interesting is that the greedy procedure used in GVNS for generating an initial solution usually provides a nearly optimal solution, solely outperforming GLPK in 7(6) out of 10 cases in the case of MODEL I (MODEL II). For instance Ex3, it was even able to find the best-known solution. Because of this, we can assume that providing the results of this greedy procedure to GLPK as an initial solution would greatly improve its performance.

5. CONCLUSION

We proposed GVNS for the non-preemptive version of scheduling tasks to identical machines. The considered version of the problem involves multiple machines and a set of tasks, each characterised by deadline, release time, and weight. The main objective is to maximize the total sum of weights for the scheduled tasks, having in mind that not all of the tasks have to be scheduled. Our GVNS uses five neighborhood structures, as well as a new greedy procedure for generating an initial solution.

We tested GVNS on a set of randomly generated instances and compared the results to the ones obtained by an exact solver GLPK exploring two different MILP formulation of the problem. The experimental evaluation shows that GVNS significantly outperforms GLPK, both in terms of solution quality and the running time. Future research should include testing the algorithm on some larger (and preferably real-life) instances, developing other metaheuristic methods, and generating hybrids between (meta)heuristic methods and exact solvers.

[1] https://cran.r-project.org/web/packages/irace/index.html

Table 1: A comparison between GLPK and GVNS

Example	K	N	GLPK (MODEL I)		GLPK (MODEL II)		GVNS		Greedy
			Obj. (% gap)	CPU time	Obj. (% gap)	CPU time	Av. obj. (σ)	Av. CPU time (σ)	Obj.
Ex1	2	10	**27 (0%)**	3.64	**27(0%)**	2.98	**27 (0)**	0.000072 (0.000006)	26
Ex2	2	15	23 (69%)	1800	**34** (14.7%)	1800	**34 (0)**	0.000816 (0.000630)	24
Ex3	2	15	33.0 (21.2%)	1800	33 (21.2%)	1800	**36 (0)**	0.000070 (0.000000)	**36**
Ex4	3	15	35 (8.6%)	1800	**37** (2.7%)	1800	**37 (0)**	0.000056 (0.000010)	32
Ex5	2	20	30 (56.7%)	1800	N/A	1800	**38 (0)**	0.001220 (0.000888)	34
Ex6	3	20	42 (21.4%)	1800	N/A	1800	**50 (0)**	0.000718 (0.000438)	47
Ex7	3	20	44.0 (11.4%)	1800	N/A	1800	**49 (0)**	0.000209 (0.000011)	46
Ex8	4	20	**59.0 (0%)**	1.18	**59.0 (0%)**	12.56	**59 (0)**	0.000210 (0.000016)	56
Ex9	4	40	N/A	1800	N/A	1800	**106 (0)**	0.016765 (0.010869)	97
Ex10	4	45	N/A	1800	N/A	1800	**102 (0)**	0.197937 (0.202902)	86

Acknowledgement

This work was supported by the Serbian Ministry of Education, Science and Technological Development, Agreement No. 451-03-9/2021-14/200029 and by the Science Fund of the Republic of Serbia, Grant AI4TrustBC: Advanced Artificial Intelligence Techniques for Analysis and Design of System Components Based on Trustworthy BlockChain Technology.

REFERENCES

[**1**] Ball, M., Rosen, A., Sabin, M., & Vasudevan, P. N. (2017). *Proofs of Useful Work*. IACR Cryptol. ePrint Arch., 2017, 203.

[**2**] Graham, R. L. (1966). *Bounds for certain multiprocessing anomalies*. Bell system technical journal, 45(9), 1563-1581.

[**3**] Guo, P., Chen, W., & Wang, Y. (2013). *A general variable neighborhood search for single-machine total tardiness scheduling problem with step-deteriorating jobs*. arXiv preprint arXiv:1301.7134.

[**4**] Hansen, P., Mladenovic, N., Todosijevic, R., & Hanafi, S. (2017). *Variable neighborhood search: basics and variants*. EURO Journal on Computational Optimization, 5(3), 423-454.

[**5**] Komaki, M., & Malakooti, B. (2017). *General variable neighborhood search algorithm to minimize makespan of the distributed no-wait flow shop scheduling problem*. Production Engineering, 11(3), 315-329.

[**6**] Lawler, E. L., Lenstra, J. K., Kan, A. H. R., & Shmoys, D. B. (1993). *Sequencing and scheduling: Algorithms and complexity*. Handbooks in operations research and management science, 4, 445-522.

[**7**] Matijević, L., Stanković, U., Davidović, T., *Mathematical models for the weighted scheduling problem with deadlines and release times* , Proc. XLVIII Symposium on Operational Research, SYMOPIS 2021, Banja Koviljača, Sept. 20-23, 2021 (submitted)

[**8**] Mladenovic, N., & Hansen, P. (1997). *Variable neighborhood search*. Computers & operations research, 24(11), 1097-1100.

[**9**] Tasgetiren, M. F., Buyukdagli, O., Pan, Q. K., & Suganthan, P. N. (2013, December). *A general variable neighborhood search algorithm for the no-idle permutation flowshop scheduling problem*. In International Conference on Swarm, Evolutionary, and Memetic Computing (pp. 24-34). Springer, Cham.

[**10**] Wen, Y., Xu, H., & Yang, J. (2011). *A heuristic-based hybrid genetic-variable neighborhood search algorithm for task scheduling in heterogeneous multiprocessor system*. Information Sciences, 181(3), 567-581.