

INTRODUCTION TO VNS TYPE OPTIMIZATION

Luka Matijević, Mathematical Institute SASA

luka@mi.sanu.ac.rs

May 1, 2020

Abstract

Variable Neighborhood Search (VNS) is a stochastic metaheuristic for discrete and continuous optimization problems, based on a systematic change of neighborhoods. In recent years, many variations of this method have been proposed, being applied to many different optimization problems. In this paper, we review the basic VNS scheme, as well as some of its variations, including matheuristics and continuous VNS. In addition, we describe parallelization strategies and parameter tuning.

Contents

1	Introduction	3
2	Variable Neighbourhood Search	4
2.1	Shaking	7
2.2	Local Search	7
2.3	Move or Not	8
3	Variations	11
3.1	Variable neighborhood descent	11
3.2	Reduced VNS	13
3.3	General VNS	13
3.4	Skewed VNS	14
3.5	Nested VNS	14
3.6	Variable neighborhood decomposition search	16

4	VNS Based Matheuristics	16
4.1	MIP Notation and definitions	17
4.2	Variable Neighborhood Branching	17
4.3	Variable Neighborhood Decomposition Search for 0-1 MIPs	19
4.4	Variable Intensity Neighborhood Search for 0-1 MIPs	21
5	Continuous VNS	24
5.1	Unconstrained optimization	25
5.2	Constrained optimization	26
6	Parallelization strategies	28
6.1	Parallel local search	29
6.1.1	Multiple-walk parallelism	29
6.1.2	Single-walk parallelism	30
6.2	Parallel VNS strategies	30
6.2.1	Independent executions	31
6.2.2	Parallel local search	32
6.2.3	Distributive VNS	32
6.2.4	Cooperative VNS	34
6.3	Existing parallel algorithms	35
7	Parameter Tuning	35
7.1	The Brute-Force Approach	36
7.2	The Racing Approach	37
8	Conclusion	40

1 Introduction

Many problems that arise from industry and theoretical science deal with finding the *best* possible values for a given set of parameters to achieve some goal. We call these problems *optimization problems*. Optimization problems can be seen as a generalization of decision problems, involving additional evaluation of solution with some *objective function* [1]. In the case of *minimization*, we try to find a solution that corresponds to the minimum value of its objective function, or in case of *maximization*, maximal value is searched for.

We can define optimization problems as follows.

$$\min_{x \in S} f(x) \quad (1)$$

where $f : R^n \rightarrow R$ is an *objective function* and $S \subseteq R^n$ is a set of all feasible solutions. It is easy to see that maximization problems can be very easily reduced to minimization problems and vice versa, since

$$\max_{x \in S} f(x) = -\min_{x \in S} (-f(x)) \quad (2)$$

If the point x^* satisfies the equation

$$f(x^*) \leq f(x), \quad \forall x \in S \quad (3)$$

we call that point the *global minimum* (or more generally *global optimum*). On the other hand, if there exists some neighborhood $\mathcal{N}(\bar{x}, \rho)$ of $\bar{x}$ such that

$$f(\bar{x}) \leq f(x), \quad \forall x \in \mathcal{N} \cap S \quad (4)$$

then we call the point $\bar{x}$ a *local minimum* (or *local optimum*) in the neighborhood N . A graphic representation of global and local minimum can be seen in Fig. 1.

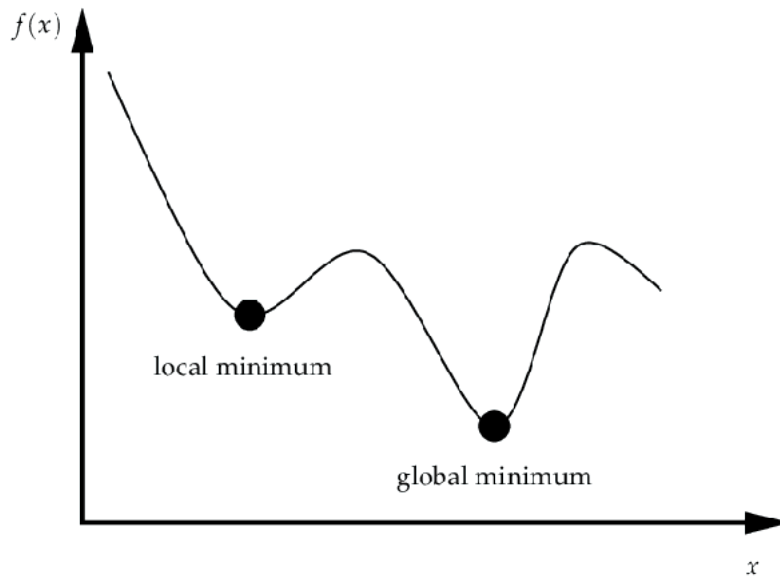


Figure 1: The difference between local and global minimum

We can generally distinguish between three types of optimization problems: discrete, continuous and mixed [2]. In problems of discrete optimization, all of the variables have discrete

values, opposed to continuous optimization where all of the variables must take values from some continuous set, such as the set of all real numbers. In the mixed optimization problems, some of the variables are discrete, while others are continuous. In some literature, mixed optimization problems are classified as discrete.

Examples of discrete optimization problems include many well-known problems such as *traveling salesman problem* [3], *vehicle routing problem* [4], *set cover problem* [5], *scheduling problem* [6], *minimum Steiner tree* [7], and many others. Examples of continuous optimization include *maximization of the general yield of differential amplifiers*, *optimization of the mechanical system of shock-absorber*, *estimation of the parameters of non-linear regression of an immunological model* and others [8]. What all of these problems have in common is that they are very difficult to solve. Most of the discrete optimization problems are at least NP-hard, while finding the global optimum for continuous problems is proven to have exponential complexity in general case [9]. It is worth mentioning that in special cases of global optimization, such as convex optimization problems, there are very efficient methods for solving these problems to a specified accuracy in polynomial time, for example, *Interior point methods* [10].

In the most general sense, there are three types of algorithms that we can use on optimization problems. The first type is *exact* algorithms which, given enough time and memory, can find the globally optimal solution and **prove** that it is optimal. Examples of such algorithms are *cutting-planes* [11], *branch-and-bound* [12] and *dynamic programming* [13]. The second type is the *approximation* algorithms, which can find a suboptimal solution in polynomial time, with a guarantee that a distance between suboptimal and optimal solution is not greater than some predetermined constant α . The third type of algorithms are heuristic (and metaheuristic) methods, which can find a suboptimal solution relatively fast, but without any guarantee about the quality of solution or execution time.

We are going to present one metaheuristic approach to solving a given optimization problem. A metaheuristic is a high-level problem-independent algorithmic framework that provides a set of guidelines or strategies to develop heuristic optimization algorithms that efficiently produce solutions of high quality [14][15]. There are many metaheuristics, which can be classified according to several different criteria [16]. Examples of metaheuristics that are most commonly researched include *simulated annealing*, *tabu search*, *ant colony optimization*, *bee colony optimization*, *particle swarm optimization*, *genetic algorithm* and many others.

The metaheuristic that we are going to present here is called *variable neighborhood search* - **VNS**. It is a mathematically founded, single solution stochastic method, that is based on the improvement principle (it transforms a given solution in search of a better one). We will present the main idea behind the algorithm, some of its variations, parallelization strategies and give some ideas on how to configure the algorithm.

2 Variable Neighbourhood Search

Variable Neighborhood Search was first proposed by Mladenović and Hansen [17][18]. In its basic form, VNS is a very simple, yet effective method, for approximately solving hard optimiza-

tion problems. It uses the systematic change of neighborhoods to explore the search space of the problem. It is based on three facts:

- A local optimum for one neighborhood structure is not necessarily a local optimum for some other neighborhood structure.
- The global optimum is a local optimum for every neighborhood structure.
- For many problems, all or most of the local optima are close to each other [19].

The first fact justifies the introduction of several neighborhoods. The second fact says that if a solution is not a local optimum for some neighborhood, it can not possibly be the global optimum, which also justifies the introduction of several neighborhoods. It does not guarantee that a solution which is a local optimum in every neighborhood used in algorithm necessarily has to be the global optimum [2]. The third fact is not theoretical but empirical. It justifies the extensive exploration of the closest neighborhood of the local optimum because there is a high probability that we can find an even better solution in that neighborhood. In case this third fact is not true for a specific problem, we can use some VNS variations that do not rely too much on this fact.

From the use of several different neighborhoods, a few questions can arise [20]:

- *Selection of neighborhood structures* - it depends on the metrics that can be used for a given problem.
- *Ordering neighborhoods* - it should correspond to an increasing distance between a current solution x and solutions x_k that belong to $\mathcal{N}_k(x)$, $k = 1, \dots, k_{max}$.
- *Neighborhood sizes* - depending on constraints, we should determine and examine only those neighbors that can potentially improve the current solution.
- *Search strategies* - either *first improvement* or *best improvement*, both of which will be described in section 2.2.
- *Descent and descent-ascent* - if the better solution is not found, we need to make a decision whether to accept an ascend move, where neighborhood and/or solution are changed.
- *Search directions* - specifies the order in which the neighborhoods will be searched. In the *forward* VNS k takes the values $k = 1, \dots, k_{max}$, while in the *backward* VNS it's in reverse: $k = k_{max}, \dots, 1$.
- *Intensification and diversification* - in local search method we want to intensify the search in some promising areas (intensification) and find some previously unexplored regions (diversification).

Pseudocode for the VNS algorithm is given in algorithm 1. As we can see from the pseudocode, VNS metaheuristic is composed of three main parts:

1. Shaking

2. Local Search

3. Move or Not

Algorithm 1 Variable Neighborhood Search

```
1: procedure VNS( $K_{max}$ , stopping criterion)
2:    $x_{best} \leftarrow \text{initialSolution}()$ 
3:   while stopping criterion is not met do
4:      $k \leftarrow 1$ 
5:     while  $k \leq K_{max}$  do
6:        $x' \leftarrow \text{shake}(x_{best})$ 
7:        $x'' \leftarrow \text{localSearch}(x')$ 
8:        $x \leftarrow \text{moveOrNot}(x, x'', k)$ 
9:     end while
10:  end while
11: end procedure
```

In Figure 2 we can see an intuitive representation of the algorithm. In this figure, we start from the initial solution that is automatically declared the current best solution x_{min} . We do the shaking in the neighborhood $N_1(x_{min})$, which basically means that we choose a random solution x' on the circle labeled as N_1 . We then improve that solution with the local search procedure in order to get x'' , but since we are unable to find a new best solution for which $f(x'') < f(x_{min})$, we continue the search in the next neighborhood. We can see that in N_2 neighborhood we also couldn't find a new best solution but in the neighborhood N_3 we find the solution that is better than x_{min} so that newly found solution becomes the new x_{min} . The whole process is now repeated for the new x_{min} .

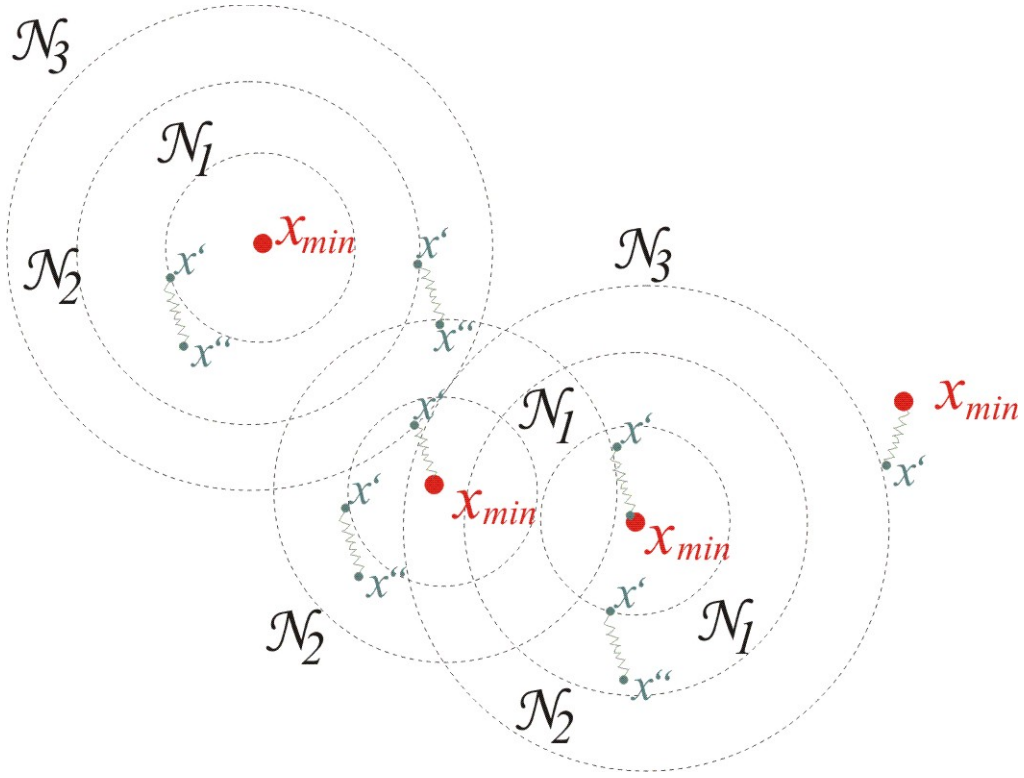


Figure 2: Graphic representation of the VNS algorithm.

2.1 Shaking

The big problem with local search is that it can easily get stuck in one of the local optima. In order to prevent this, we use the *Shaking* procedure, which perturbs the solution in regards to one or more neighborhoods. Consider the set of k neighborhood structures $N_1, N_2, \dots, N_{k_{max}}$. Generally, we assume that these neighborhood structures are sorted by their size, so that $|N_k| \leq |N_{k+1}|$. The shaking procedure chooses one solution at random from the current neighborhood N_k (Algorithm 2), which becomes the starting point of the local search algorithm.

Algorithm 2 Shaking

- 1: **procedure** SHAKE(x, k, N)
 - 2: Choose $x' \in N_k(x)$ at random
 - 3: **return** x'
 - 4: **end procedure**
-

2.2 Local Search

Local search is a heuristics method that starts from the given solution and systematically explores neighborhood $N(x)$ of that solution. Starting from the current solution x , in every iteration of the algorithm, we choose a solution from the neighborhood $N(x)$ which is better than the incumbent solution (if there is such solution) and make that the new incumbent solution. The algorithm continues until it finds the local optimum for the $N(x)$ neighborhood structure. It can

utilize different strategies, the most common ones being *First improvement* and *Best improvement*. In the *First improvement* approach, the new incumbent solution is set as soon as the better solution is detected in the neighborhood $N(x)$ (Algorithm 3), while in the *Best improvement* approach the new incumbent solution is the best of all improving solutions in the neighborhood $N(x)$ (Algorithm 4).

Algorithm 3 Local search (first improvement)

```

1: procedure LS_FI( $x, N$ )
2:   repeat
3:     Let  $N(x) = \{x^1, x^2, \dots, x^p\}$ 
4:      $i \leftarrow 0$ 
5:      $x' \leftarrow x$ 
6:     repeat
7:        $i \leftarrow i + 1$ 
8:       if  $f(x^i) < f(x)$  then
9:          $x \leftarrow x^i$ 
10:      break
11:    end if
12:    until  $i = p$ 
13:  until  $f(x') \leq f(x)$ 
14:  return  $x'$ 
15: end procedure

```

Algorithm 4 Local search (best improvement)

```

1: procedure LS_FI( $x, N$ )
2:   repeat
3:      $x' \leftarrow x$ 
4:      $x \leftarrow \operatorname{argmin}_{y \in N(x')} f(y)$ 
5:   until  $f(x') \leq f(x)$ 
6:
7:   return  $x'$ 
8: end procedure

```

2.3 Move or Not

In the *Move or not* procedure the algorithm decides if the newly found solution should be accepted as the new best solution and whether to change to another neighborhood structure. There are a few different variations of this method:

- *Sequential neighborhood change step* - the most common procedure, which accepts new solution x' if the value of the objective function of x' is better than the value of the objective

function of the incumbent solution x . In that case, the search is continued from the first neighborhood structure, otherwise, the next neighborhood is explored (Algorithm 5).

- *Plateau neighborhood change step* - it allows the new solution with the same value of the objective function to be accepted (Algorithm 6). It can be combined with other neighborhood change steps.
- *Cyclic neighborhood change step* - similar to the *Sequential neighborhood change step* with a difference that search is always continued from the next neighborhood $k + 1$ (Algorithm 7). This is in direct contradiction with the third fact described in section 2 which says that local optima are close to each other, but since this fact is not theoretical but empirical, this variation works better for problems in which this observation is not true.
- *Pipe neighborhood change step* - similar to the *Sequential neighborhood change step* but in case that better solution is found in neighborhood k , the search is continued from the same neighborhood k (Algorithm 8).
- *The skewed neighborhood change step* - in this case, it is not required that the newfound solution has an exclusively better value of the objective function than the current solution. This procedure allows for the worse solution to be accepted, but only if the new solution is distant from the current solution (Algorithm 9). It can be combined with some of the previously explained neighborhood change steps.

Algorithm 5 Sequential neighborhood change step

```

1: procedure NEIGHBORHOOD CHANGE SEQUENTIAL( $x, x', k$ )
2:   if  $f(x') < f(x)$  then
3:      $x \leftarrow x'$ 
4:      $k \leftarrow 1$ 
5:   else
6:      $k \leftarrow k + 1$ 
7:   end if
8: end procedure

```

Algorithm 6 Plateau neighborhood change step

```

1: procedure NEIGHBORHOOD CHANGE PLATEAU( $x, x', k$ )
2:   if  $f(x') \leq f(x)$  then
3:      $x \leftarrow x'$ 
4:      $k \leftarrow 1$ 
5:   else
6:      $k \leftarrow k + 1$ 
7:   end if
8: end procedure

```

Algorithm 7 Cyclic neighborhood change step

```
1: procedure NEIGHBORHOOD CHANGE CYCLIC( $x, x', k$ )
2:    $k \leftarrow k + 1$ 
3:   if  $f(x') < f(x)$  then
4:      $x \leftarrow x'$ 
5:   end if
6: end procedure
```

Algorithm 8 Pipe neighborhood change step

```
1: procedure NEIGHBORHOOD CHANGE PIPE( $x, x', k$ )
2:   if  $f(x') < f(x)$  then
3:      $x \leftarrow x'$ 
4:   else
5:      $k \leftarrow k + 1$ 
6:   end if
7: end procedure
```

Algorithm 9 Skewed neighborhood change step

```
1: procedure SKEWED NEIGHBORHOOD CHANGE( $x, x', k, \alpha$ )
2:   if  $f(x') - f(x) < \alpha d(x', x)$  then
3:      $x \leftarrow x'$ 
4:      $k \leftarrow 1$ 
5:   else
6:      $k \leftarrow k + 1$ 
7:   end if
8: end procedure
```

3 Variations

There are many variations of the VNS algorithm, which can differ in some (or all) basic steps. This difference can be in the way that local search is performed, how we define neighborhoods for shaking or how we choose the next incumbent solution. In this section, we are going to present some of the most common variations.

3.1 Variable neighborhood descent

Variable neighborhood descent (VND) is a group of procedures that use a deterministic change of the neighborhoods during the search. It is based upon an observation that a solution that is the local optimum for several neighborhoods has a higher probability of being the global optimum than a solution that is the local optimum with respect to only one neighborhood. Same as local search, it can use the *first improvement* and the *best improvement* approach. There are few variations of this procedure:

- **Sequential VND**

- *Basic VND (B-VND)* - in its most basic form, the procedure sequentially explores neighborhoods $N = \{N_1, N_2, \dots, N_{k_{max}}\}$, starting from an initial solution x . When a better solution x' is found, it becomes the new incumbent solution and the procedure starts over from N_1 neighborhood. The procedure ends when all k_{max} neighborhoods are explored, without finding a better solution (Algorithm 10).
- *Pipe VND (P-VND)* - it uses the *Pipe neighborhood change step* described in section 2.3.
- *Cyclic VND (C-VND)* - it uses the *Cyclic neighborhood change step* described in section 2.3.
- *Union VND (U-VND)* - all of the k_{max} neighborhoods are united into one neighborhood N , which is then explored in every iteration. If this union does contain a better solution x' than incumbent x , x' is taken as the new incumbent, otherwise, the procedure ends.

- **Nested VND** - lets assume that $N = \{N_1, N_2, \dots, N_{k_{max}}\}$ is a set of operators that defines neighborhood structures $N_k(x), 1 \leq k \leq k_{max}$. Nested VND explores a neighborhood defined by the composition of these operators $N^* = N_1 \circ N_2 \circ \dots \circ N_{k_{max}}$. In other words, local search is performed with respect to the first neighborhood N_1 in every point of the second neighborhood N_2 , which is generated with the points from N_3 and so on, until $N_{k_{max}}$ (Algorithm 11).

- **Mixed procedures** - it is a mixture of sequential and nested VND. It uses a set of operators $N' = \{N_1, \dots, N_b\}$ to define a nested neighborhood, but each time it explores a solution from this neighborhood, a sequential VND is started to further explore neighborhoods of that solution, using operators $N'' = \{N_{b+1}, \dots, N_{k_{max}}\}$ (Algorithm 12).

Algorithm 10 Variable Neighborhood Descent

```
1: procedure VND( $x, k_{max}$ )
2:   repeat
3:      $k \leftarrow 1$ 
4:     repeat
5:        $x' \leftarrow \arg \min_{y \in N'_k(x)} f(y)$  ▷ Find best solution in  $N'_k$ 
6:       NeighborhoodChange( $x, x', k$ )
7:     until  $k = k_{max}$ 
8:   until No improvement of  $x$  has been found
9: end procedure
```

Algorithm 11 Nested Variable Neighborhood Descent

```
1: procedure NEST-VND( $x, k_{max}$ )
2:    $x_{opt} \leftarrow x$ 
3:    $f_{opt} \leftarrow f(x)$  ▷  $f(x)$  is a objective function
4:    $k \leftarrow k_{max}$ 
5:   repeat
6:     if  $\forall x_k \in N_k(x)$  is visited then
7:        $k \leftarrow k + 1$ 
8:     end if
9:     if  $\exists x_k \in N_k(x)$  that is not visited  $\wedge k \geq 2$  then
10:       $x_{cur} \leftarrow x_k$ 
11:       $k \leftarrow k - 1$ 
12:    end if
13:    if  $k = 1$  then
14:       $f_{cur} = f(x_{cur})$ 
15:      if  $f_{cur} < f_{opt}$  then
16:         $x_{opt} \leftarrow x_{cur}$ 
17:         $f_{opt} \leftarrow f_{cur}$ 
18:      end if
19:    end if
20:  until  $k = k_{max} + 1$  ▷ No more points in the last neighborhood
21: end procedure
```

Algorithm 12 Mixed Variable Neighborhood Descent

```
1: procedure MIXED_VND( $x, b, k_{max}, N', N''$ )
2:    $N^* = N_1 \circ N_2 \circ \dots \circ N_b$ 
3:    $x' \leftarrow x$ 
4:   repeat
5:      $stop \leftarrow true$ 
6:      $x' \leftarrow x$ 
7:     for  $\forall y \in N^*(x)$  do
8:        $x'' \leftarrow B - VND(y, k_{max} - b, N'')$ 
9:       if  $f(x'') < f(x')$  then
10:         $stop \leftarrow false$ 
11:         $x' \leftarrow x''$ 
12:       end if
13:     end for
14:   until  $stop = true$ 
15:
16:   return  $x'$ 
17: end procedure
```

3.2 Reduced VNS

Reduced VNS is a variation of VNS which consists of shaking procedure and neighborhood change step, while *local search* step is omitted (Algorithm 13). Use of this method can be seen in the article [21].

Algorithm 13 Reduced Variable Neighborhood Search

```
1: procedure REDUCED_VNS( $x, k_{max}, N$ )
2:   repeat
3:      $k \leftarrow 1$ 
4:     while  $k < k_{max}$  do
5:        $x' \leftarrow Shake(x, k, N)$ 
6:        $Neighborhood_{change}(x, x', k)$ 
7:     end while
8:   until  $Stoppingcriterionismet$ 
9:
10:  return  $x$ 
11: end procedure
```

3.3 General VNS

General VNS is similar to basic VNS, with a difference that GVNS uses some form of VND as an improvement step of the procedure (Algorithm 14). However, neighborhoods used in the

shaking step and VND do not have to be the same, as can be seen in the paper [22], where shaking procedure uses only a subset of the neighborhood set that VND uses. GVNS is a very common variation and some examples of its use can be found in [22][23][24][25][26] and many other papers.

Algorithm 14 General Variable Neighborhood Search

```

1: procedure GVNS( $x, k_{max}, l_{max}, N', N''$ )
2:   repeat
3:      $k \leftarrow 1$ 
4:     while  $k < k_{max}$  do
5:        $x' \leftarrow Shake(x, k, N')$ 
6:        $x'' \leftarrow VND(x', l_{max}, N'')$ 
7:        $Neighborhoodchange(x, x'', k)$ 
8:     end while
9:   until Stoppingcriterionismet
10:
11:   return  $x$ 
12: end procedure

```

3.4 Skewed VNS

As the name suggests, *Skewed VNS* uses the skewed neighborhood change step, described in subsection 2.3, in order to prevent the procedure to get stuck in a local optimum. This approach takes into account the distance between solutions and sometimes accepts a little bit worse solution if it is relatively far from the incumbent solution. Some of the examples include [27][28][29][30].

Generally, we make a move between solution x to a solution x'' if the following condition is met:

$$\frac{f(x'')}{f(x)} < 1 + \alpha \times d(f(x), f(x''))$$

where $d(x, y)$ represents the distance between solutions x and y , and α is a predefined parameter. It's important to choose the value of α carefully, since if α is too big, the algorithm is going to accept the new solution too often, while if α is too small, the algorithm is going to behave in the same way as *BVNS*.

A variation of this approach is that a worse solution is accepted only temporarily. In that case, the procedure will not reset parameter k to 1, but will continue the search. If a better solution is not found while $k \leq k_{max}$, the incumbent solution is reverted to the best-known solution.

3.5 Nested VNS

Nested VNS [31][32] is a variation of VNS procedure with a difference that if a better solution is not found after shaking and improvement step, a second (nested) VNS is started with this worse

solution as its initial solution (Algorithm 15). Since this nested VNS is going to run very often, it is important to limit its execution with either CPU time limit or the number of iterations. The set of neighborhoods explored in the outer VNS and nested VNS doesn't have to be the same. An example of this approach can be found in the paper [33].

Algorithm 15 Nested Variable Neighborhood Search

```

1: procedure NESTED_VNS( $k_{max}, t_{max}, N', N''$ )
2:    $x \leftarrow InitialSolution$ 
3:    $k \leftarrow 1$ 
4:   repeat
5:      $x' \leftarrow SHAKE_1(x, k)$   $\triangleright x' \in N'_k(x)$ 
6:      $x'' \leftarrow IMPROVE(x')$ 
7:     if  $f(x'') < f(x)$  then
8:        $x \leftarrow x''$ 
9:        $k \leftarrow 1$ 
10:    else  $\triangleright$  Nested VNS
11:       $t \leftarrow 1$ 
12:      repeat
13:         $y' \leftarrow SHAKE_2(x'', t)$   $\triangleright y' \in N''_t(x'')$ 
14:         $y'' \leftarrow IMPROVE(y')$ 
15:        if  $f(y'') < f(x'')$  then
16:           $x'' \leftarrow y''$ 
17:           $t \leftarrow 1$ 
18:        else
19:           $t \leftarrow t + 1$ 
20:          if  $t > t_{max}$  then
21:             $t \leftarrow 1$ 
22:          end if
23:        end if
24:      until  $stopping\_criterion_2$  is not met
25:      if  $f(x'') < f(x)$  then
26:         $x \leftarrow x''$ 
27:         $k \leftarrow 1$ 
28:      else
29:         $k \leftarrow k + 1$ 
30:        if  $k > k_{max}$  then
31:           $k \leftarrow 1$ 
32:        end if
33:      end if
34:    end if
35:  until  $stopping\_criterion_1$  is not met
36:
37:  return  $x$ 
38: end procedure

```

3.6 Variable neighborhood decomposition search

The idea behind *Variable neighborhood decomposition search (VNDS)* [34] is to reduce the main problem into smaller subproblems, which are then explored in the improvement step. More precisely, for a given solution x , all but k variables are fixed during the improvement step. During the shaking step, we create a solution x' that is on a distance k from solution x . Then, we create a decomposed solution y , which contains only k variables that differ from x . That decomposed solution is further improved during the *improvement step*, which gives us an improved decomposed solution y' . Finally, we combine solution y' with the partial solution $x' \setminus y$, and use that combined solution in the *neighborhood change* step (Algorithm 16). Some of the interesting papers that use this algorithm are [35, 36, 37, 38, 39, 40].

Algorithm 16 Variable neighborhood decomposition search

```
1: procedure VNDS( $x, k_{max}, N$ )
2:   repeat
3:      $k \leftarrow 1$ 
4:     repeat
5:        $x' \leftarrow Shake(x, k, N)$ 
6:        $y \leftarrow x' \setminus x$ 
7:        $y' \leftarrow Improve(y)$ 
8:        $x'' \leftarrow (x' \setminus y) \cup y'$ 
9:        $NeighborhoodChange(x, x'', k)$ 
10:    until  $k = k_{max}$ 
11:  until Stoppingcriterionismet
12:
13:  return  $x$ 
14: end procedure
```

4 VNS Based Matheuristics

Note

This section was completely taken from my other paper. At the point of writing this, this paper has not yet been published, so if you plan on citing some of its parts, please send me an email so that I can send you the proper reference.

Matheuristics are heuristic algorithms made by the interoperation of metaheuristics and mathematic programming technique [41]. They are also called *model-based metaheuristics*, since they use a mathematical model of a problem inside metaheuristic algorithm. In this section, we will describe three matheuristics based on variable neighborhood search.

4.1 MIP Notation and definitions

Matheuristics usually operate exclusively on binary variables as they can take only two possible values, zero and one. On the other hand, it is straightforward to reformulate a model containing integer variables into a model with binary variables having an additional index.

In order to introduce neighborhood structures in the space of binary variables, we need to define the distance between two solutions $x = (x_1, \dots, x_n)$ and $x' = (x'_1, \dots, x'_n)$. Hamming distance is obviously the natural choice.

$$\delta(x, x') = \sum_i |x_i - x'_i|, \quad (5)$$

where i refers to the subset of binary variables.

If the original MIP is denoted by P , let $LP(P)$ represent its linear relaxation (obtained when in P the integer requirements on variables are released. The solution of $LP(P)$ is usually denoted by y . The distance (5) could be generalized to $\delta(x, y)$, where $|x_i - y_i|$ can take any value from the interval $[0, 1]$.

$(P|C)$ is used to mark the subproblem of P obtained by adding the set of constraints C , while $P(k, x) = (P | \delta(x, x') \leq k)$, for $k = 1, 2, \dots$ denotes the solution subspace of P containing all the solutions whose distance from a given solution x is at most k , i.e., all the solutions that have at most k different values for binary variables. $P(k, x)$ could be used to define neighborhoods:

$$\mathcal{N}_k(x) = \{x' \in X | \delta(x, x') \leq k\}. \quad (6)$$

More precisely, neighborhood $\mathcal{N}_k(x)$ contains all the solutions differing from x in the values of at most k binary variables. It is easy to see that $\mathcal{N}_k(x) \subset \mathcal{N}_{k+1}(x)$, and therefore, if neighborhood $\mathcal{N}_{k+1}(x)$ is completely explored, it is not necessary to search in neighborhood $\mathcal{N}_k(x)$.

An alternative way to define neighborhoods is using variable states. Each binary variable can be in one of the two possible states: fixed and relaxed. The fixed state means that the exact MIP solver is not allowed to change its value during the optimization process. On the other hand, we want the optimizer to find the best value of the variables in relaxed state. Neighborhood structures are then defined with respect to the indices and number of relaxed variables.

Local search in the space of binary variables is realized by invoking exact solver on a subproblem $(P|C)$ obtained by adding the set of neighborhood defining constraints C to the original problem P . The exact solver is allowed to run for a given time limit to avoid long executions in the cases when the defined neighborhoods are too large. The decision about the next step is then made based on the return status of the exact solver and is specific for each particular matheuristic.

All matheuristic methods described in this section apply VNS rules for creating initial solutions and subproblems and CPLEX exact solver as a Local Search procedure to improve the current best solution.

4.2 Variable Neighborhood Branching

Variable neighborhood branching (VNB) [42] is a GVNS inspired matheuristic developed for solving 0-1 MIP problems. The input parameters for VNB are 0-1 MIP problem P , VNS param-

eters k_{min} , k_{step} , k_{max} , maximum allowed CPU time T_{max} , and the time for subproblems t_{lim} . The pseudo-code of VNB is illustrated by Algorithm 17.

Algorithm 17 Pseudo-code for VNB

```

procedure VNB( $P, T_{max}, t_{lim}, k_{min}, k_{max}, k_{step}$ )
   $IntSolLim \leftarrow 1; TL \leftarrow T_{max}$ 
   $status \leftarrow MIPSolve(P, TL, IntSolLim, x)$ 
  if ( $status = no\_feasible$ ) then
    Print("The problem cannot be solved within the given time limit.")
    Exit
  else
     $x_{best} \leftarrow x; f_{best} \leftarrow f(x); x' \leftarrow x; f' \leftarrow f(x)$ 
  end if
   $t \leftarrow 0$ 
  while ( $t < T_{max}$ ) do ▷ Main VNB loop
     $cont \leftarrow True; k \leftarrow 1; k_1 \leftarrow k_{min}; IntSolLim \leftarrow MaxSol$ 
    while ( $cont \wedge t < T_{max}$ ) do ▷ VND
       $TL \leftarrow \min\{t_{lim}, T_{max} - t\}$ 
       $AddConstraint(d(x', x) \leq k)$ 
       $status \leftarrow MIPSolve(P|C, TL, IntSolLim, x'')$ 
       $RemoveConstraint(d(x', x) \leq k)$ 
      switch ( $status$ ) do
        case  $no\_feasible$  :
           $cont \leftarrow False$ 
        case  $opt\_sol$  :
           $AddConstraint(d(x', x) \geq k + 1)$ 
           $x' \leftarrow x''; f' \leftarrow f(x''); k \leftarrow 1$ 
        case  $feasible\_sol$  :
           $AddConstraint(d(x', x) \geq 1)$ 
           $x' \leftarrow x''; f' \leftarrow f(x''); k \leftarrow 1$ 
        case  $infeasible$  :
           $k \leftarrow k + 1$ 
      end switch
       $t \leftarrow GetUserTime()$ 
    end while
    if ( $f' < f_{best}$ ) then ▷ // Move or Not
       $x_{best} \leftarrow x'; f_{best} \leftarrow f(x'); k_1 \leftarrow 1$ 
    else
       $k_1 \leftarrow k_1 + k_{step};$  if ( $k_1 > k_{max}$ ) then  $k_1 \leftarrow k_{min}$ 
    end if
     $cont \leftarrow True; k_1 \leftarrow k_{min}; IntSolLim \leftarrow 1$ 
    while ( $cont \wedge t < T_{max}$ ) do ▷ // Shaking
       $TL \leftarrow T_{max} - t$ 
       $AddConstraint(k_1 \leq d(x_{best}, x) \leq k_1 + k_{step})$ 
       $status \leftarrow MIPSolve(P|C, TL, IntSolLim, x')$ 
      if ( $status = infeasible \vee status = no\_feasible$ ) then
         $k_1 = k_1 + k_{step}$ 
      else
         $cont \leftarrow False$ 
      end if
       $RemoveConstraint(k_1 \leq d(x_{best}, x) \leq k_1 + k_{step})$ 
       $t \leftarrow GetUserTime()$ 
    end while
  end while
  return ( $x_{best}, f_{best}$ )
end procedure

```

To generate subproblems, VNB adds constraints to the original problem P and generates subproblems explored in both Local Search (VND) and Shaking phase. The initial solution is obtained as the first feasible solution found by CPLEX. It becomes also the current best solution. Neighborhoods are defined using Hamming distance (5) and adding constraints used in (6) to the original model. The value for k is updated properly with respect to the CPLEX return status. Actually, VNB uses two indices to count neighborhoods, k and k_1 . Index k counts neighborhoods in VND, while in Shaking k_1 is used.

The diversification in VNB (Shaking) is performed by invoking CPLEX to find the first feasible solution within the disk defined by radii k_1 and $k_1 + k_{step}$. If it is necessary, the disk size is increased until a feasible solution is found. VND is realized by executing CPLEX on the defined subproblems until the pre-specified time limit t_{lim} is reached. Subproblems are controlled by changing the value for the current neighborhood index k . At the beginning of VND, $k = 1$ is set. If CPLEX proves that no feasible solution exists in the current neighborhood k , the value for k is increased by 1. In the case when in $\mathcal{N}_k(x)$ CPLEX finds an optimal solution or better feasible solution, an improvement is recorded and k is set to 1 again. Finally, if no feasible solution is found within a t_{lim} time limit, VND procedure is terminated and the control is given to VNB. Move or Not step is realized in the standard way.

4.3 Variable Neighborhood Decomposition Search for 0-1 MIPs

VNDS for 0-1 MIPs appeared in [43]. It applies VNDS scheme on the set of binary variables to decompose the original problem. At each step, the original problem P is decomposed into two parts: in the first part, the variable values are fixed and this part should remain unchanged; the second part consists of relaxed variables whose values are to be determined by the exact MIP solver within the given time limit. The pseudo-code of the VNDS-MIP procedure is provided in Algorithm 18.

The first step of VNDS-MIP is to find an integer feasible solution x of the considered optimization problem P and an optimal solution y of $LP(P)$. The current best solution x_{best} is initialized by x . Using generalized definition for the distance $\delta(x, y)$, binary variables are sorted in a non-decreasing order with respect to $|x_i - y_i|$. More precisely, if $p = |\mathcal{B}|$ and $\delta_j = |x_j - y_j|$, $j \in \mathcal{B}$, then x_j , $j = 1, 2, \dots, p$ are ordered such that $\delta_1 \leq \delta_2 \leq \dots \leq \delta_p$. Subproblems in VNDS-MIP are generated by using states of binary variables, i.e., by fixing values for a subset of ordered binary variables to their values in the current best solution x_{best} . The variables to be fixed are the ones with values closest to their linear relaxation counterparts (see Fig. 3). The remaining variables are considered relaxed, and their number defines the size of the subproblem (neighborhood) to be solved by CPLEX within the given CPU time limit. In Fig. 3, variables on the left side (including shaded part) are fixed, while the remaining variables (towards the right side of the figure) are relaxed.

After the binary variables are sorted, $q = |\{j \in \mathcal{B}, s.t. \delta_j = 0\}|$ is determined and it is used to calculate values for k_{min} , k_{max} , and k_{step} as it is shown in Algorithm 18. Then the appropriate subsets of variables are fixed/released and time limited CPLEX is invoked to solve the obtained subproblem. If a new best solution is obtained, an attempt of further improvement

Algorithm 18 Pseudo-code for VNDS-MIP

procedure VNDS-MIP(P, T_{max}, t_{lim}) $IntSolLim \leftarrow 1; TL \leftarrow T_{max}$ $lpstat \leftarrow LPSolve(P, TL, y)$ $status \leftarrow MIPSolve(P, TL, IntSolLim, x)$ **if** ($status = no_feasible$) **then**

Print("The problem cannot be solved within the given time limit.")

Exit

else $x_{best} \leftarrow x; f_{best} \leftarrow f(x); x' \leftarrow x; f' \leftarrow f(x); p \leftarrow |\mathcal{B}|$ **end if** $t \leftarrow 0; execute \leftarrow True$ **while** ($execute \wedge t < T_{max}$) **do**

▷ Main VNDS-MIP loop

 $Calculate(\delta); x \leftarrow Sort(x_{best}, y); q \leftarrow |\{j \in \mathcal{B}, s.t. \delta_j = 0\}|$ $k_{max} \leftarrow p - q; k_{min} \leftarrow k_{max}/10; k_{step} \leftarrow k_{min}; k \leftarrow k_{min}$ $cont \leftarrow True; IntSolLim \leftarrow MaxSol$ **while** ($cont \wedge t < T_{max}$) **do** $TL \leftarrow \min\{t_{lim}, T_{max} - t\}; improved \leftarrow True$

▷ Local Search

 $FixAndRelease(P, k)$ $AddConstraint(f(x), L, f_{best})$ $status \leftarrow MIPSolve(P|C, TL, IntSolLim, x'')$ $ReleaseAll(P|C)$ **if** ($status = no_feasible \vee status = infeasible$) **then** $improved \leftarrow False$ **end if** **if** ($improved$) **then**

▷ Move or Not

 $VND(P, T_{max}, t_{lim}, k, x'', x', f')$ $x_{best} \leftarrow x'; f_{best} \leftarrow f(x'); k \leftarrow k_{min}; cont \leftarrow False$ **else** **if** ($k + k_{step} > k_{max}$) **then** $k_{step} \leftarrow \max\{\lfloor k/2 \rfloor, 1\}$ **if** ($k \leq p$) **then** $k \leftarrow k + k_{step}$ **else** $execute \leftarrow False$ **end if** **end if** $t \leftarrow GetUserTime()$ **end while****end while** **return** (x_{best}, f_{best})**end procedure**

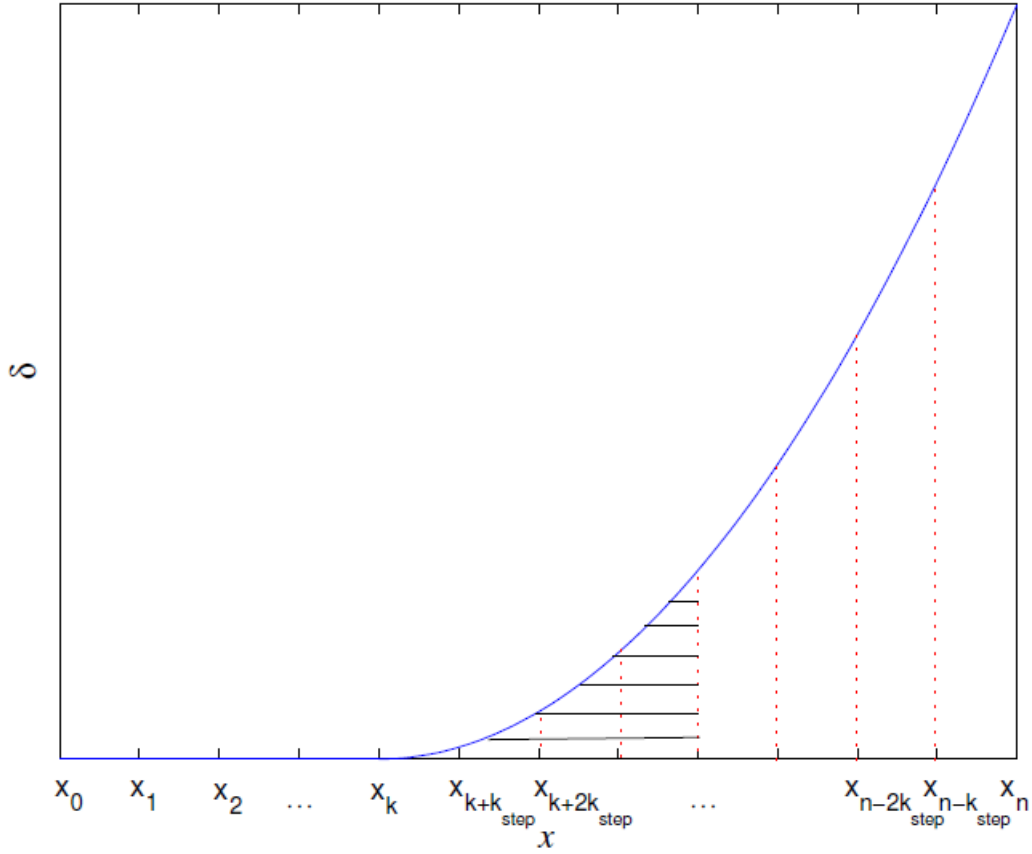


Figure 3: Illustration of subproblem generation in VNDS-MIP (Figure is provided by Dr. Jasmina Lazić)

is made by applying VND (the same one used in VNB) to the x_{best} , otherwise, the subproblem is increased by properly changing neighborhood size for searching. In the case when the subproblem involves all variables with positive distance from the linear relaxation counterparts ($k + k_{step} > k_{max}$), the search is extended (half by half) to the remaining binary variables. If all binary variables were already relaxed during the search (which actually means that the whole problem was considered as a subproblem and treated by CPLEX), the execution of VNDS-MIP ends (flag execute is set to False). This means that either CPLEX was able to solve the original problem P (optimal solution has been found) or the maximum time limit was reached and the best feasible solution should be reported to user. As VNDS-MIP is able to prove the optimality, given enough time and memory, we can consider it as an exact solution method. therefore, it is legitimate to compare its performance with CPLEX on the original problem P within the same maximum time limit.

4.4 Variable Intensity Neighborhood Search for 0-1 MIPs

VINS [44] is inspired by the VNDS-MIP matheuristic and Variable Intensity Local Search (VILS), a hybrid heuristic developed for multi-resource generalized assignment problem [45]. It generalizes the idea of fixing and relaxing subsets of binary variables in such a way that, contrary to VNDS-MIP which uses a single pattern, VINS exploits different patterns as it is the case in VILS.

As in VNDS-MIP, VINS solves the linear relaxation of the MIP first, and then finds the first feasible solution. The corresponding solutions are marked by y and x , respectively. Having these two sets of variable values, the distances between x and y based on the generalized form of Eq. (5) are calculated and the variables are sorted in the non-decreasing order according to these distance values. In VNDS-MIP, variables having similar values in x and y were considered to offer less space for improvement, however, this may not be true. Therefore, 10 different patterns (referred to as neighborhoods) for generating subproblems are defined in VINS.

The following ten types of neighborhoods (defined in the space of variable states) are used:

- N1: $\alpha\%$ of the worst variables are released;
- N2: variable set is divided into 10 bins and $\alpha/10\%$ worst variables are released in each bin;
- N3: starting at random position $\alpha\%$ variables are released;
- N4: at 10 random positions, $\alpha/10\%$ variables are released;
- N5: $\alpha\%$ of the best variables are released;
- N6: in 10 equal bins $\alpha/10\%$ best variables are released;
- N7: $\alpha/2\%$ of best and $\alpha/2\%$ of worst variables are released;
- N8: in 10 equal bins the same pattern as in N7 is applied;
- N9: random $\alpha\%$ variables are released;
- N10: in 10 equal bins, random $\alpha/10\%$ variables are released.

Terms "the best" and "the worst" are used here to mark variables with the lowest and the biggest distance between the values in current best solution and in the linear relaxation solution, respectively. The value of parameter α defines the neighborhood size, i.e., the percentage of variables that will be released and given to time limited CPLEX solver for improving. In addition to changing neighborhood types, VINS allows changes in the size of neighborhoods. As the search advances, the size of neighborhoods increases, as well as the corresponding time to limit CPLEX engagement in the resulting subproblems. Therefore, with respect to VNDS-MIP, VINS has two new input parameters: an array of neighborhood sizes *alphas* and the corresponding array of times for subproblems *time_limits*.

The neighborhoods are searched in round robin fashion, starting from N1 till N10, in each of the available sizes. In the case an improvement is made, the current best solution is updated, the corresponding constraint limiting the objective function value is added to the model, the variables are re-sorted by the distance between values in current best solution and the linear relaxation solution, and the search continues in the next neighborhood of the same size. The size never decreases as it is the case in other two methods. When all neighborhoods are explored, the size is increased and the search continues in N1. If there is enough time and all sizes are explored, CPLEX is invoked to additionally improve the current best solution acting on the whole problem P . The proposed procedure is illustrated by Algorithm 19.

Algorithm 19 VINS

```
procedure VINS( $P, T_{max}, alphas, time\_limits$ )
   $IntSolLim \leftarrow 1; TL \leftarrow T_{max}$ 
   $lpstat \leftarrow LPSolve(P, TL, y)$ 
   $status \leftarrow MIPSolve(P, TL, IntSolLim, x)$ 
  if ( $status = no\_feasible$ ) then
    Print("The problem cannot be solved within the given time limit.")
    Exit
  else
     $x_{best} \leftarrow x; f_{best} \leftarrow f(x_{best})$ 
  end if
   $t \leftarrow 0; cont \leftarrow True$ 
   $N \leftarrow N1$ 
   $\alpha \leftarrow Alpha1$ 
   $TL \leftarrow TimeLimit1$ 
   $IntSolLim \leftarrow \infty$ 
   $x \leftarrow Sort(x_{best}, y)$ 
  while ( $cont \wedge t < T_{max}$ ) do ▷ Main VINS loop
     $TL \leftarrow \min\{t_{lim}, T_{max} - t\}$ 
     $FixAndRelease(N, \alpha)$ 
     $status \leftarrow MIPSolve(P|C, TL, IntSolLim, x)$ 
     $improvement \leftarrow Improved(x)$ 
    if ( $improvement$ ) then ▷ Checking for Improvement
       $x_{best} \leftarrow x; f_{best} \leftarrow f(x)$ 
       $AddConstraint(f(x), L, f_{best})$ 
       $x \leftarrow Sort(x_{best}, y)$ 
    end if
     $N \leftarrow NextNeighborhood$  ▷ Move Step
     $t \leftarrow GetUserTime()$ 
    if ( $N = null$ ) then
       $N \leftarrow N1$ 
       $\alpha \leftarrow NextAlpha$ 
       $t_{lim} \leftarrow NextTimeLimit$ 
      if ( $\alpha = null$ ) then ▷ Final improvement of the whole problem
         $TL \leftarrow \min\{t_{lim}, T_{max} - t\}$ 
         $status \leftarrow MIPSolve(P, TL, IntSolLim, x)$ 
         $cont \leftarrow False$ 
      end if
    end if
  end while
  return ( $x_{best}, f_{best}$ )
end procedure
```

5 Continuous VNS

Up to now, we talked about discrete optimization, which can be applied to various problems in which variables that define a search space have discrete values. Such problems include *Traveling salesman problem - TSP*, *Vehicle routing problem - VRP*, *Scheduling problem*, *Minimum spanning tree - MST* and many others. On the other hand, continuous optimization deals with problems in which values of the variables belong to some subset of R^n . Because of the fact that the variables have values from a continuous interval (Fig. 4), we can use some calculus techniques which are not applicable to discrete values.

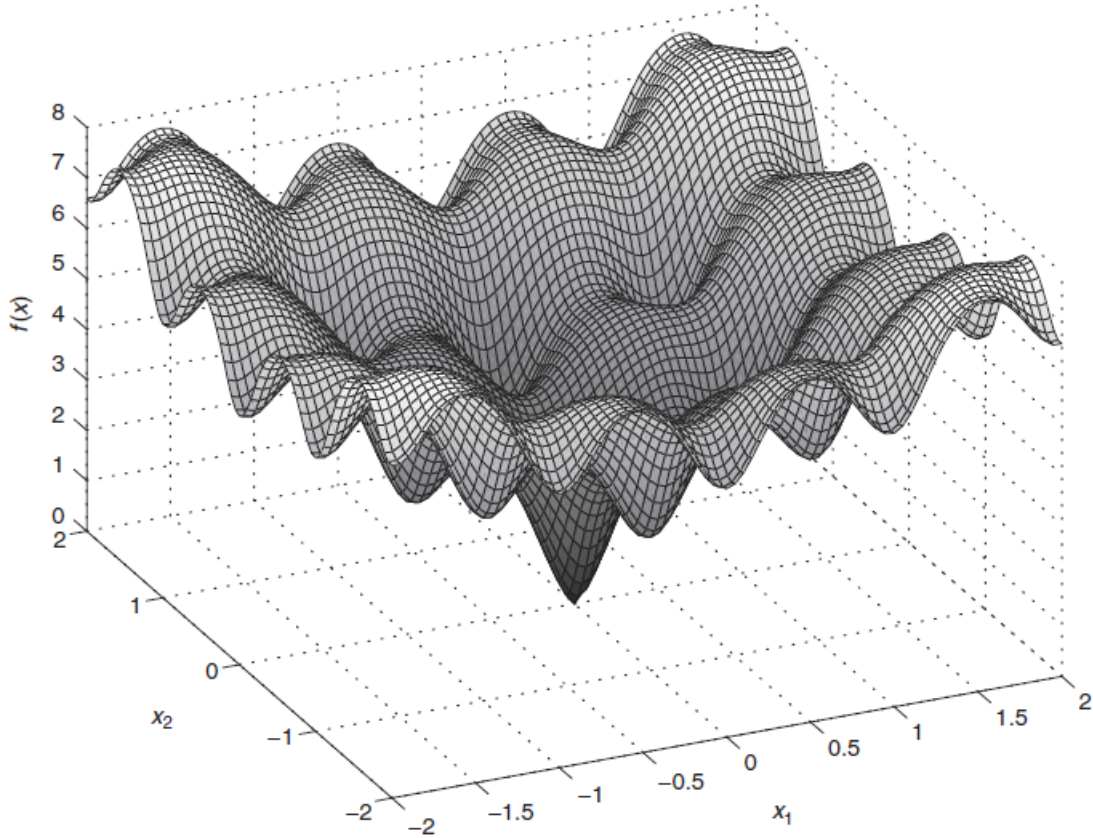


Figure 4: An example of continuous function (Ackley test function over $[-2, 2] \times [-2, 2]$). The minimum of this function is in $(0, 0)$

The formulation for this problem can be written as such:

$$\text{global} \min_{x \in S} f(x) \quad (7)$$

with a feasible set S defined by:

$$g_i(x) \leq 0, \quad i = 1, 2, \dots, m \quad (8)$$

$$h_i(x) = 0, \quad i = 1, 2, \dots, r \quad (9)$$

$$a \leq x \leq b \quad (10)$$

where $x \in R^n$, $f : R^n \rightarrow R$, $g_i : R^n \rightarrow R$, $h_i : R^n \rightarrow R$ are possibly nonlinear continuous functions, and $a, b \in R^n$ are the variable bounds [46].

5.1 Unconstrained optimization

In order to effectively apply VNS to global continuous optimization, we first need to redefine our neighborhood structures to reflect the continuous nature of the problem. We can define neighborhoods in one of the two ways:

$$N_k(x) = \{y \in S | \rho_k(x, y) \leq r_k\} \quad (11)$$

or

$$N_k(x) = \{y \in S | r_{k-1} \leq \rho_k(x, y) \leq r_k\} \quad (12)$$

where r_k is the radius (size) of the neighborhood structure N_k , which increases with k , while $\rho_k(x, y)$ is a metric based on ℓ_p distance.

$$\rho_k(x, y) = \begin{cases} (\sum_{i=1}^n |x_i - y_i|^p)^{\frac{1}{p}} & 1 \leq p < \infty \\ \max_{1 \leq i \leq n} |x_i - y_i| & p = \infty \end{cases} \quad (13)$$

The other thing we have to consider is how we choose a random point from a neighborhood. In order to do that, we generate a random direction d , using one of the distributions given below. After that, we generate a random radius $r \in [0, r_k]$ (or $r \in [r_{k-1}, r_k]$). In the end, d is scaled to r in metric defining the geometry of $N_k(x)$ [46].

- *Distribution $\mathcal{D}_1$* - The direction d is generated using the uniform distribution in ℓ_∞ unit ball.
- *Distribution $\mathcal{D}_2$* - The direction d is generated using the uniform distribution on ℓ_1 unit sphere.
- *Distribution $\mathcal{D}_3$* - The direction d is generated using specially designed distribution on ℓ_1 unit sphere as follows
 - The coordinate d_1 is taken uniformly on $[-1, 1]$, d_k is taken uniformly from $[-A_k, A_k]$ where $A_k = 1 - |d_1| - \dots - |d_{k-1}|$, $k = 2, \dots, n-1$ and d_n takes A_n with random sign.
 - Coordinates of d are permuted randomly.
- *Distribution $\mathcal{D}_4$*
 - The direction d is generated using uniform distribution in ℓ_∞ unit ball.
 - With the current point x and box-constraint vectors a and b , all of the coordinates d_i are scaled
 - * If $d_i \geq 0$ the scaling factor is $\frac{(b_i - x_i)}{r_{kmax}}$.
 - * If $d_i < 0$ the scaling factor is $\frac{(x_i - a_i)}{r_{kmax}}$.

There are also many different methods for local search. We will give a few examples:

- First-order methods
 - Steepest descent
 - FletcherReeves
 - Variable metric
- Direct search methods
 - NelderMead
 - HookJeves
 - Rosenbrock

Considering all of this, we can now construct a VNS-based algorithm called *Glob-VNS* for unconstrained global optimization problems, as shown in Algorithm 20.

Algorithm 20 Glob-VNS algorithm for unconstrained optimization

```

1: procedure GLOB-VNS( $x, k_{max}, t_{max}, N, D$ )
2:    $f^* \leftarrow f(x)$ 
3:    $t \leftarrow 0$ 
4:   while  $t < t_{max}$  do
5:      $k \leftarrow 1$ 
6:     repeat
7:       for all distribution types  $D_j \in D$  do
8:          $x' \leftarrow Shake(x, k)$  ▷ Get  $x' \in N_k(x)$  at random with  $D_j$  distribution
9:          $x'' \leftarrow LocalSearch(x')$ 
10:        if  $f(x'') < f^*$  then
11:           $x \leftarrow x''$ 
12:           $f^* \leftarrow f(x'')$ 
13:          go to 5
14:        end if
15:      end for
16:       $k \leftarrow k + 1$ 
17:    until  $k = k_{max}$ 
18:     $t \leftarrow CpuTime()$ 
19:  end while
20: end procedure

```

5.2 Constrained optimization

To solve the constrained optimization problem described earlier, we will use the penalization method. Penalization methods can be classified into two groups: *exterior point penalty* methods, which impose a penalty for violating a constraint, and *interior-point penalty* methods, which impose a penalty for reaching the boundary of an inequality constraint [46]. Since the problem

we described earlier (7-10) has both equality and inequality constraints, exterior point penalty method is better suited for the problem.

Algorithm 21 The exterior point CGVNS

```

1: procedure CGVNS( $x, k_{max}, q, \mu_0, \mu_{min}, N, D$ )
2:    $x^* \leftarrow x$ 
3:    $f^* \leftarrow F_{\mu,q}(x)$ 
4:    $f_F^* \leftarrow \infty$ 
5:    $\mu \leftarrow \mu_0$ 
6:   repeat
7:      $k \leftarrow 1$ 
8:     repeat
9:       for all distribution types  $D_j \in D$  do
10:        Generate random point  $y \in N_k(x^*)$ 
11:         $y' \leftarrow localSearch(y)$ 
12:        if  $P_{\mu,q}(y') \leq \varepsilon \wedge f(y') < f_F^*$  then
13:           $x_F^* \leftarrow y'$ 
14:           $f_F^* \leftarrow f(y')$ 
15:        end if
16:        if  $F_{\mu,q}(y') < F_{\mu,q}(x^*)$  then
17:           $x^* \leftarrow y'$ 
18:        go to 7
19:        else
20:           $\mu \leftarrow \max\{\mu_{min}, \alpha\mu\}$ 
21:        end if
22:      end for
23:       $k \leftarrow k + 1$ 
24:    until  $k > k_{max}$ 
25:  until stopping criterion is met
26:  if  $f_F^* = \infty$  then  $\triangleright x_F^*$  is not found
27:     $x_F^*$  is acquired by a local minimizer applied to  $F_{\mu,q}(x)$ , with  $q = 1, \mu = \mu_{min}$ , starting
    from  $x^*$ 
28:  end if
29:  return  $x_F^*$ 
30: end procedure

```

The unconstrained problem that we get from (7-10) with the introduction of penalty function can be written as follows:

$$\min_{a \leq x \leq b} F_{\mu,q}(x) = f(x) + \frac{1}{\mu} P_q(x) \quad (14)$$

where $P_q(x)$ is a penalty function:

$$P_q(x) = \sum_{i=1}^m (\max\{0, g_i(x)\})^q + \sum_{i=1}^r |h_i(x)|^q \quad (15)$$

where μ is a positive penalty parameter and penalty exponent $q \geq 1$. An algorithm that uses VNS to solve this problem is given in Algorithm 21. In this algorithm, parameter α is some constant factor $0 \leq \alpha \leq 1$, while ε is a tolerance factor.

6 Parallelization strategies

Because metaheuristics are usually used to solve hard optimization problems and can take up a lot of time, we naturally try to speed up that process by taking the advantage of parallel systems such as clusters, GPU-s or just multicore CPU-s. This way, we can increase search speed, find a better solution or solve larger problems. It comes as no surprise that many papers on the topic of parallelization of VNS have already been published, such as [47, 48, 49, 50, 51].

There are three main aspects to consider when talking about parallelization of metaheuristics [52]:

- *Search Control Cardinality* - describes how many processes control search. If one process is in charge of others (like master-slave architecture) than we call that *1-control (1C)*, otherwise, if several processes jointly control search, either by cooperating with each other or not, we call that architecture *p-control (pC)*.
- *Communication* - it describes how information is exchanged between processes. There are four basic types of communication:
 - *Rigid Synchronization (RS)* - strict synchronous communication
 - *Knowledge Synchronization (KS)* - synchronous communication with modification of the transferred data
 - *Collegial (C)* - Asynchronous communication
 - *Knowledge Collegial (KC)* - Asynchronous communication with learning from transferred data
- *Search Differentiation* - it describes whether the threads use the same or different starting solution and whether they use the same search method. We can distinguish four different cases:
 - *Same initial point, same search strategy (SPSS)*
 - *Same initial point, different search strategies (SPDS)*
 - *Multiple initial point, same search strategy (MPSS)*
 - *Multiple initial point, different search strategies (MPDS)*

We can classify different parallelization strategies based on the values which describe the aforementioned aspects, as it is shown in column 3. of table 1.

Considering that metaheuristics are stochastic methods, which means that for the same input different executions of the same algorithm don't have to provide the same solution, we have to redefine quality measures for parallel algorithms. Also, the parallelization of such an algorithm can change its structure and properties.

We generally consider that a parallel version of a metaheuristic is better than sequential in case it can generate a solution of the same quality in less time, or it can generate a solution of better quality in the same time. We can express this with the following equation [53]:

$$U_q = S \frac{f_1}{f_q} \quad (16)$$

where S represents the acceleration factor of parallelization, f_1 is the value of the objective function in the sequential algorithm and f_q is the value of the objective function achieved by parallel algorithm on q processors.

6.1 Parallel local search

Considering the fact that local search is usually the most time-consuming part of VNS, it makes sense that we first explore some parallelization strategies for local search. We can distinguish between two main approaches: *single-walk* and *multiple-walk* (Fig. 5). In a *single-walk* approach, we explore only a part of the search space in depth, unlike a *multiple-walk* approach, where we explore multiple parts of the search space at the same time. Furthermore, we can divide multiple-walk algorithms in those that interact with each other and those that run independently.

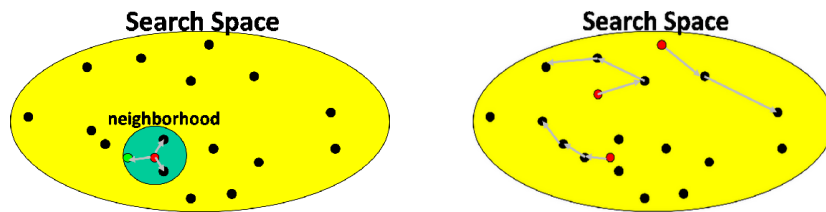


Figure 5: Left: single-walk parallelism, Right: multiple-walk parallelism

6.1.1 Multiple-walk parallelism

Multiple-walk algorithms distribute search over several processors and then perform them at the same time. The most common type of multiple-walk algorithms consists of multiple independent walks, without any communication between them. It is possible to achieve linear speed-up with multiple independent walks, if the probability to find an optimal solution in a given amount of time is distributed exponentially [54]. It is important to note that in order to achieve good results with this approach, search space should be divided into disjunctive parts. On the other hand, we can use interactions between walks to direct the search into promising areas of the search space. We can direct this search by exchanging information about occurrences of local optimums, as well as information about good parts of the solution. Also, local optimums often have similar

properties, which can be identified and used to reduce the size of the search space [55]. We refer to this information as *hints*. At each step, each process writes down hints into centralized shared memory, along with probability p . When a process needs to make a decision, it reads hint from memory with its probability. If the probability is zero, the algorithm behaves the same as the algorithm with independent walks.

6.1.2 Single-walk parallelism

Single-walk methods use parallelism within a local search procedure by doing the most expensive functions in parallel. In other words, one or more consecutive steps of local search are executed in parallel. One of the most resource-consuming parts of local search is neighborhood evaluation, which is why we can divide a neighborhood into different partitions and explore them independently in parallel. This strategy is known as *neighborhood decomposition*. Some of the neighborhood decomposition strategies for scheduling problem are given in [56]. We can also differentiate between two different variants of single-walk parallelism: *single step* and *multiple-step*. The idea behind a single-step approach is to evaluate all of the neighbors in parallel and then make a single step, while in a multiple-step approach, the algorithm makes several consecutive steps in the solution, as can be seen in figure 6.

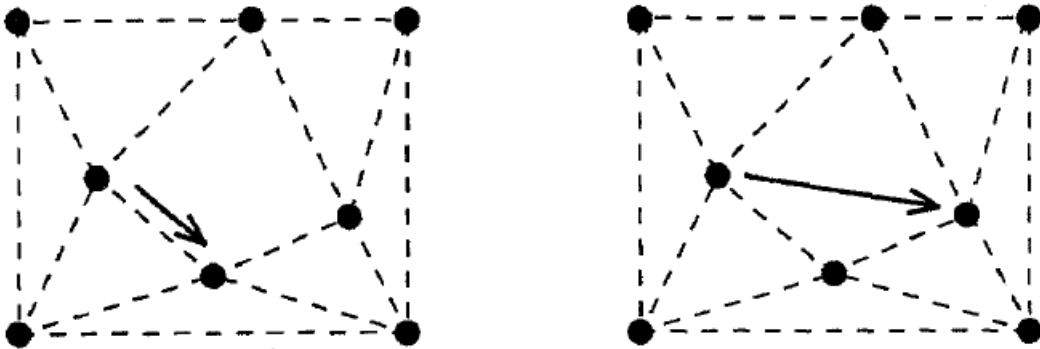


Figure 6: Left: local search with single-step parallelism, Right: local search with multiple-step parallelism

6.2 Parallel VNS strategies

In this subsection, we will briefly discuss some of the parallelization strategies for VNS. In figure 7 we can see some of the topology types at our disposal.

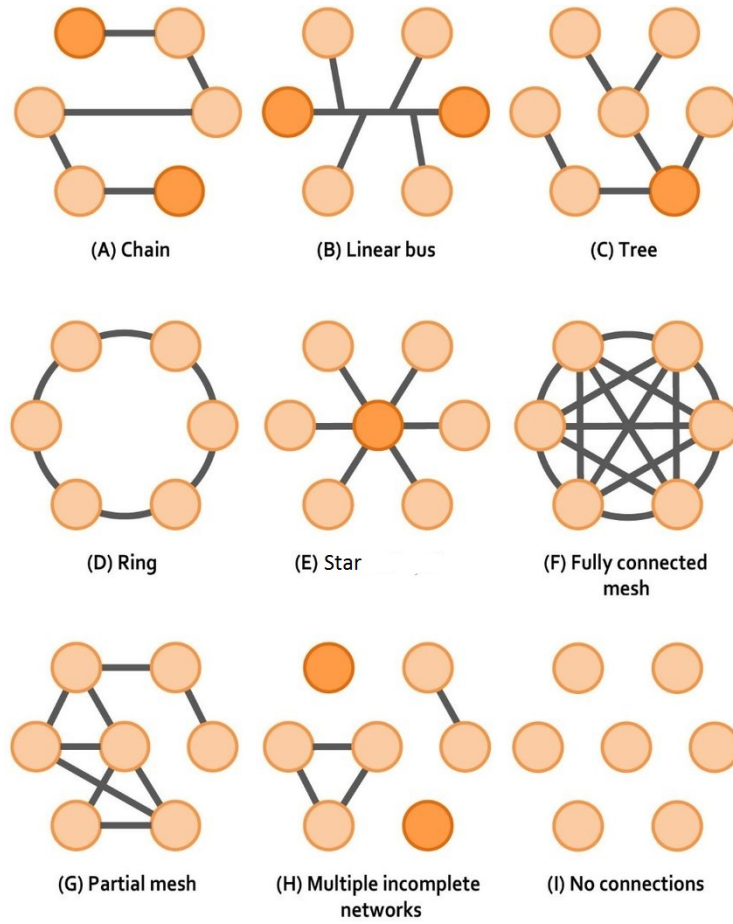


Figure 7: Different types of network topologies [57]. Highlighted nodes have a special roles, such as *endpoints* in (A) and (B), *root* in (C), or *hub* in (E)

6.2.1 Independent executions

The first parallelization strategy that logically comes to mind is to execute multiple independent instances of VNS on different processors. In some papers, this algorithm is referred to as **IVNS** (*Independent VNS*) [48], while in others it's called **RPVNS** (*Replicated parallel VNS*) [58]. The pseudocode of **RPVNS** is given in Algorithm 22;

Algorithm 22 Replicated parallel VNS

```
1: procedure RPVNS
2:   for each processor  $r$  do
3:     Initialize  $S_r$ 
4:      $k_r \leftarrow 1$ 
5:     while  $k \leq k_{max}$  do
6:        $S'_r \leftarrow shake(S_r, k_r)$ 
7:        $S''_r \leftarrow local\_search(S'_r)$ 
8:       if  $S''_r$  is better than  $S_r$  then
9:          $S_r \leftarrow S''_r$ 
10:         $k_r \leftarrow 1$ 
11:      else
12:         $k_r \leftarrow k_r + 1$ 
13:      end if
14:    end while
15:  end for
16:   $S'' \leftarrow S_1$ 
17:  for  $r = 2; r \leq number\_of\_processors; r++$  do
18:    if  $S_r$  is better than  $S''$  then
19:       $S'' \leftarrow S_r$ 
20:    end if
21:  end for
22: end procedure
```

} in parallel

6.2.2 Parallel local search

In subsection 6.1 we already discussed different parallelization strategies for local search procedure. These strategies can be easily incorporated into VNS algorithm, thus speeding up the whole algorithm. It is considered *low-level* parallelization since only the local search is executed in parallel, while the rest of the algorithm is executed sequentially. For this strategy, the most suitable topology would be the *star topology*, though in some cases of cooperative parallel local search other topologies can be considered, such as a hybrid topology combined of both star and ring topology.

6.2.3 Distributive VNS

Another strategy for VNS parallelization is to explore different neighborhoods (or different parts of the same neighborhood) in parallel. In order to accomplish this, in each tread, we run a shaking procedure and local search procedure, with different neighborhoods. The rest of the algorithm is run sequentially, with the master process in control. Same as in the previous strategies, this algorithm uses star topology.

In case that the number of available processors does not match the number of neighborhoods (which is reasonable to expect), we are left with two possibilities. Let us denote the number of available processors as q and number of neighborhoods as k_{max} . In case that $q \leq k_{max}$, only the q first neighborhoods are used for shaking at the same time. In case that $q > k_{max}$, we can assign neighborhoods cyclically, in the sense that processor $(k*q+i)$ gets the neighborhood N_i , $1 \leq i \leq q$, $0 \leq k \leq (q \bmod k_{max})$.

Pseudocode for a distributive VNS, as proposed in [48], can be seen in Algorithm 23.

Algorithm 23 Pseudocode for DVNS

```

1: procedure MASTER PROCESSOR
2:    $x \leftarrow generate\_initial\_solution()$ 
3:    $x_{best} \leftarrow local\_search(x)$ 
4:   Send  $x_{best}$  and the corresponding neighborhood  $k$  to all slaves
5:   while stopping condition is not met do
6:     Receive  $x''$  from a slave
7:     if  $x''$  is better than  $x_{best}$  then
8:        $x_{best} \leftarrow x''$ 
9:        $k \leftarrow 1$ 
10:    else
11:       $k \leftarrow k + 1$ 
12:    end if
13:  end while
14:  Send STOP message to all slaves
15:  Collect time measuring data from all slaves and print output
16: end procedure
17:
18: procedure SLAVE PROCESSOR
19:  Receive  $x_{best}$  and  $k$  from master processor
20:  while true do
21:     $x' \leftarrow shake(x_{best}, k)$ 
22:     $x'' \leftarrow local\_search(x')$ 
23:    Send  $x''$  to the master
24:    Receive a message from master
25:    if received message is STOP then
26:      break
27:    else
28:      Read  $x_{best}$  and  $k$ 
29:    end if
30:  end while
31:  Send measured time variables to the master
32: end procedure

```

6.2.4 Cooperative VNS

The last strategy that we are going to examine consists of several VNS instances that run in parallel, with an asynchronous exchange of some of the information between them. In order to ensure communication between processors, central memory is used, and every VNS instance communicates exclusively with this central memory or master.

Pseudocode for one of the versions of cooperative VNS called *CNVNS* ([59]) is given in Algorithm 24, but there are other variations of it (*CVNSall*, *CVNSallK*, *CVNSring*)[48]. Some of these variants use a star topology, while others use a ring topology.

Algorithm 24 Cooperative Neighbourhood VNS

```
1: procedure MASTER PROCESSOR
2:   parallel_RVNS(init_solution[p]) ▷ p = 1..number_of_processors
3:   initialize(joint_best_sol)
4:   for p = 1; p < number_of_processors; p ++ do
5:     send(init_solution[p] : p = 1..number_of_processors)
6:   end for
7:   while stopping criterion is not met do
8:     get(improved_solution[p]);
9:     if improved_solution[p] is better than joint_best_sol then
10:      joint_best_sol = improved_solution[p]
11:      init_solution[p] = joint_best_sol
12:      send(init_solution[p] : p = 1..number_of_processors)
13:    end if
14:  end while
15: end procedure
16:
17: procedure SLAVE PROCESSOR
18:   get(best_solution_p)
19:   k = 0
20:   while k < kmax do
21:     k = k + 1
22:     x' = shake(best_solution_p, k)
23:     x'' = local_search(x')
24:     if x'' is better than best_solution_p then
25:       best_solution_p = x''
26:       k = 0
27:     end if
28:   end while
29:   return best_solution_p
30: end procedure
```

6.3 Existing parallel algorithms

In table 1 we give examples of existing parallel versions of VNS algorithm and their classification in accordance with three main aspects described at the beginning of Section 6. Some of these algorithms have been previously explained, while for the others, we refer the interested reader to the original papers. These are by no means the only existing parallel versions of VNS, there are many more papers that deal with the same subject.

Table 1: Some of the existing parallelization strategies for VNS

Ref.	Strategies	Classification	Details
[58]	PLS in seq VNS IVNS SH+LS in parallel	1C/RS/SPSS pC/RS/MPSS 1C/KS/SPSS	low level independent execution for same k synchronous
[59]	IVNS SH+LS in parallel	pC/RS/MPSS pC/C/MPSS	independent execution for same k asynchronous
[60]	SH+LS in parallel SH+LS in parallel SH+LS non-centralized ring SH+LS non-centralized mesh	1C/KS/SPSS pC/C/MPSS pC/C/MPSS pC/C/MPSS	for same k synchronous for random k asynchronous asynchronous asynchronous
[61]	SH+VND in parallel several iterations of SH+VND	pC/C/MPSS pC/KC/MPSS	fine grained cooperation coarse grained cooperation
[62]	SH+RVND in parallel	1C/KS/SPDS	random neighborhood subset
[63]	several VNS in parallel	pC/KS/MPDS	synchronous VNS multisearch
[64]	SH+LS in parallel	1C/RS/SPDS	different neighborhoods for LS
[48]	PVNSPLS DVNS CVNS CVNSall CVNSallK	1C/C/SPSS 1C/C/MPSS pC/C/MPDS pC/C/SPDS pC/C/SPDS	parallel LS in sequential VNS, low level in sequential VNS, SH+LS in parallel, (different k, same LS) SH+LS in parallel, non-centralized (different k, different LS) SH+LS in parallel, (different k, different LS) VNS iteration in parallel, centralized

7 Parameter Tuning

In order to ensure the best possible performance of any metaheuristic, we must carefully choose its parameters. The process of choosing those parameters is usually referred to as *tuning of an algorithm*. When it comes to VNS, the obvious parameters that we have to choose are a minimal and maximal value of k . Neighborhood structures used in shaking and local search process can also be seen as parameters, the same as the ordering of those neighborhood structures. For some variants of VNS we have additional parameters, such as parameter α in *skewed VNS*. The value of these parameters can greatly influence the performance of our algorithm, therefore, we

will review some of the usual tactics for choosing the values of these parameters.

7.1 The Brute-Force Approach

Let us denote overall available time for the tuning process as T , where every experiment runs for a time t . With these constraints, we can perform $M = \lfloor \frac{T}{t} \rfloor$ experiments. Now, let us consider a set of candidate configurations Θ and a set of instances I . Each of the candidates from Θ is tested $N = \lfloor \frac{M}{|\Theta|} \rfloor$ times, which gives us performance estimates $\hat{\mu}(\Theta_1), \hat{\mu}(\Theta_2), \dots, \hat{\mu}(\Theta_{|\Theta|})$. This way, we can define a brute-force algorithm named *Brutus*. In this algorithm, we use a distribution P_I in order to randomly sample N instances, and then we test all given candidate configurations on these selected instances. Using these results, we can estimate performance $\hat{\mu}(\Theta_i)$ for each candidate $\Theta_i \in \Theta$, with respect to distribution P_I . In the end, we select the candidate Θ_i with the best value of $\hat{\mu}(\Theta_i)$. The pseudocode for this algorithm is given in Algorithm 25.

The downsides of the brute-force approach can be summed up as follows:

- The size of the training set has to be defined before any computations. Too few instances can lead to unreliable estimates, while too many instances would result in lots of unnecessary computations.
- The number of runs for each configuration and each instance, necessary to overcome the stochastic nature of metaheuristics is not well defined.
- Each configuration gets the same amount of resources, whether it's a promising configuration or not.

To overcome these downsides, we can use the *racing approach*.

Algorithm 25 Brutus algorithm

```

1: procedure BRUTUS( $M, \Theta$ )
2:    $N = \text{floor}(M / |\Theta|)$ 
3:    $A = \text{allocate\_array}(|\Theta|)$  ▷  $A$  stores performance estimates
4:   for ( $k = 1; k \leq N; k++$ ) do
5:      $i = \text{sample\_instance}()$  ▷ According to  $P_I$ 
6:     for  $\theta \in \Theta$  do
7:        $s = \text{run\_experiment}(\theta, i)$ 
8:        $c = \text{evaluate\_solution}(s)$ 
9:        $A[\theta] = \text{update\_mean}(A[\theta], c, k)$ 
10:    end for
11:  end for
12:   $\tilde{\theta} = \arg \arg \min(A)$  ▷ Returns  $\theta$  with minimal value of  $A[\theta]$ 
13:  return  $\tilde{\theta}$ 
14: end procedure

```

7.2 The Racing Approach

This family of algorithms was first proposed in [65], inspired by the *Hoeffding race* algorithm, introduced in [66]. The idea behind them is to better assign available resources among candidate configurations. Racing algorithms evaluate candidate configurations and eliminate those that are not promising enough, based on statistical analysis, which allows the algorithm to test promising configuration on a larger number of instances. Pseudocode for generic racing algorithm can be seen in Algorithm 26. The difference in the number of computations between brute-force and racing approaches can be seen in Fig. 8.

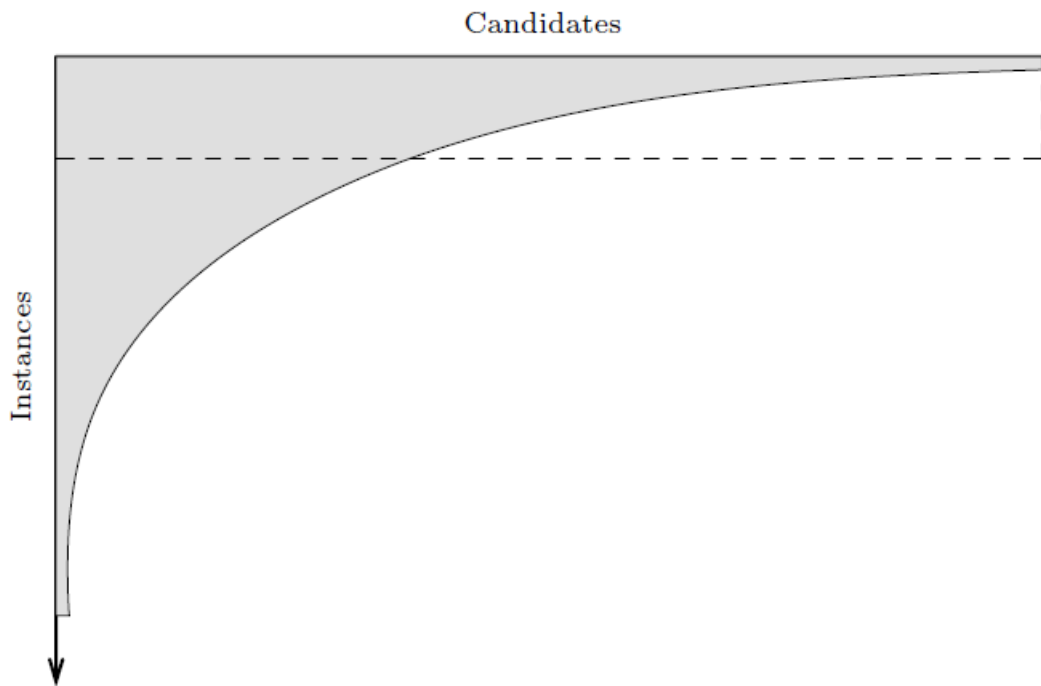


Figure 8: The difference in the number of computations between brute-force and racing approaches. The surface of the dashed rectangle represents the number of computations needed in the brute-force approach, while the colored area represents the number of computations for the racing approach.

Different statistical tests can be utilized in racing algorithms, classified by three criteria:

- *Parametric vs. nonparametric tests* - parametric tests expect the data with some underlying distribution, while nonparametric tests do not rely on any distribution. Therefore, nonparametric tests can be applied on a wider range of problems, while parametric tests have the advantage of being more powerful, in the sense that they are more able to reject the null hypothesis.
- *Blocking vs. non-blocking design* - in statistics, blocking refers to grouping experiments that are similar to one another into clusters (groups). In the problem of metaheuristics tuning, blocking design is usually adopted [67].
- *Family-wise vs. pair-wise tests* - in the family-wise approach we try to determine whether all of the results, no matter to which candidate they correspond, are extracted from the

Algorithm 26 Generic racing approach

```
1: procedure RACE( $M, \Theta, use\_test$ )
2:    $experimentsSoFar \leftarrow 0$ 
3:    $instancesSoFar \leftarrow 0$ 
4:    $C \leftarrow allocate\_array(max\_instances, |\Theta|)$ 
5:    $\mathcal{S} \leftarrow \Theta$  ▷ Surviving instances
6:   while  $experimentsSoFar + |\mathcal{S}| \leq M \wedge instancesSoFar + 1 \leq max\_instances$  do
7:      $i = sample\_instance()$  ▷ According to  $P_I$ 
8:      $instancesSoFar \leftarrow instancesSoFar + 1$ 
9:     for  $\theta \in \mathcal{S}$  do
10:       $s = run\_experiment(\theta, i)$ 
11:       $experimentsSoFar \leftarrow experimentsSoFar + 1$ 
12:       $C[instancesSoFar, \theta] \leftarrow evaluate\_solution(s)$ 
13:     end for
14:      $\mathcal{S} \leftarrow drop\_candidates(\mathcal{S}, C, use\_test)$ 
15:   end while
16:    $\tilde{\theta} \leftarrow select\_best\_survivor(\mathcal{S}, C)$ 
17:   return  $\tilde{\theta}$ 
18: end procedure
```

same distribution. Only if a family-wise test shows that candidates differ we can proceed to perform *post-tests*, in which we compare candidates with one another, to determine which one is better. On the other hand, with pair-wise tests, we compare a pair of candidates.

We will here present four different tests that can be used in the racing algorithm:

- *tNo-Race* - a statistical test used in this method is the classical paired *t-test*. This test is parametric with a blocking design. Also, it does not perform correction of the *p-value* for multiple comparisons.
- *tBo-Race* - it's the same as *tNo-Race* with the difference that correction of *p-values* is performed using a Bonferroni correction for multiplicity.
- *tHo-Race* - similar to *tNo-Race* but with proper multiplicity correction proposed by Holm [68].
- *F-Race* - the statistical test used in this method is *Friedman's two-way analysis of variance by ranks*. This test is nonparametric, with blocking design, and it uses a family-wise approach. If this test shows any statistical significance in the data, it proceeds with *post-hoc tests*, which can test configurations pairwise, in order to find a more promising configuration. There are many ways of conducting post-hoc tests, some of which are explained in [69].

With these tests defined, we can proceed to formulate a method for dropping candidates, referred in Algorithm 26. The pseudocode for this method is given in Algorithm 27.

Algorithm 27 Function for dropping candidates

```
1: procedure DROP_CANDIDATES( $\mathcal{S}, C, use\_test$ )
2:   switch  $use\_test$  do
3:     case  $friedman\_test$ 
4:       if  $is\_significant(friedman\_family\_wise(\mathcal{S}, C))$  then
5:          $\tilde{\theta} \leftarrow select\_best\_survivor(\mathcal{S}, C)$ 
6:         for  $\theta \in \mathcal{S}$  do
7:           if  $is\_significant(friedman\_post\_test(\tilde{\theta}, \theta, C))$  then
8:              $\mathcal{S} \leftarrow \mathcal{S} \setminus \{\theta\}$ 
9:           end if
10:        end for
11:      end if
12:     case  $t\_test$ 
13:        $\tilde{\theta} \leftarrow select\_best\_survivor(\mathcal{S}, C)$ 
14:       for  $\theta \in \mathcal{S}$  do
15:         if  $is\_significant(t\_test(\tilde{\theta}, \theta, C))$  then
16:            $\mathcal{S} \leftarrow \mathcal{S} \setminus \{\theta\}$ 
17:         end if
18:       end for
19:     case  $t\_test\_bonferroni$ 
20:        $\tilde{\theta} \leftarrow select\_best\_survivor(\mathcal{S}, C)$ 
21:       for  $\theta \in \mathcal{S}$  do
22:         if  $is\_significant(t\_test\_bonferroni(\tilde{\theta}, \theta, C))$  then
23:            $\mathcal{S} \leftarrow \mathcal{S} \setminus \{\theta\}$ 
24:         end if
25:       end for
26:     case  $t\_test\_holm$ 
27:        $\tilde{\theta} \leftarrow select\_best\_survivor(\mathcal{S}, C)$ 
28:       for  $\theta \in \mathcal{S}$  do
29:         if  $is\_significant(t\_test\_holm(\tilde{\theta}, \theta, C))$  then
30:            $\mathcal{S} \leftarrow \mathcal{S} \setminus \{\theta\}$ 
31:         end if
32:       end for
33:   return  $\mathcal{S}$ 
34: end procedure
```

8 Conclusion

In this paper, we presented several variations of the VNS algorithm, as well as a few different parallelization strategies. The natural question that remains is which one of these methods is better than the others and when it should be used. The answer to this question is simple: it depends on lots of factors, such as a problem at hand, properties of instances, the hardware we use. The same algorithm can perform poorly on a certain set of instances, while giving satisfactory results on some other set of instances. On a set of small-sized instances, exact solvers usually give us better results quicker, so this approach only makes sense when trying to solve a problem on a set of large instances, which would be impractical for an exact solver. In the end, the best approach would be to test several versions on a subset of instances, and then deciding which one is the most promising one.

References

- [1] H.H. Hoos and T. Stützle. *Introduction*. Stochastic Local Search: Foundations and Applications, The Morgan Kaufmann Series in Artificial Intelligence, Pages 13-59 (2005)
- [2] T. Davidović. *Scheduling Tasks to Multiprocessor Systems by Applying Metaheuristics*. Ph.D. Thesis (2006) (in Serbian)
- [3] R. Matai, S. Singh and M.L. Mittal. *Traveling Salesman Problem: an Overview of Applications, Formulations, and Solution Approaches*. In: Traveling Salesman Problem, IntechOpen (2010). <https://doi.org/10.5772/12909>
- [4] P. Toth and D. Vigo (Eds.). *Vehicle Routing: Problems, Methods, and Applications, Second Edition*. (MOS-SIAM Series on Optimization; No. 18). Philadelphia: SIAM. (2014)
- [5] A. Caprara, P. Toth and M. Fischetti. *Algorithms for the Set Covering Problem*. Annals of Operations Research. 98. 353-. 10.1023/A:1019225027893. (2000)
- [6] V. Tkindt and J.C. Billaut. *Introduction to scheduling*. In: Multicriteria Scheduling. Springer, Berlin, Heidelberg (2002)
- [7] G. Robins and A. Zelikovsky. *Minimum steiner tree construction*. In Alpert, C.J., Mehta, D.P. and Sapatnekar, S.S. (eds), The Handbook of Algorithms for VLSI Physical Design Automation, CRC Press. (2009)
- [8] J. Mockus, W. Eddy, A. Mockus, L. Mockus and Reklaitis. *Examples of Continuous Optimization*. Bayesian Heuristic Approach to Discrete and Global Optimization. Nonconvex Optimization and Its Applications, vol 17. Springer, Boston, MA (1997)
- [9] M. Gaviano and D. Lera. *Complexity of general continuous minimization problems: a survey*. Optimization Methods and Software, 20:4-5, 525-544, DOI: 10.1080/10556780500139872 (2005)

- [10] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.
- [11] J.E. Mitchell. *Integer programming: cutting plane algorithms*. In: Floudas C., Pardalos P. (eds) *Encyclopedia of Optimization*. Springer, Boston, MA (2008)
- [12] K. Kianfar. *Branch-and-Bound Algorithms*. Wiley Encyclopedia of Operations Research and Management Science (2011)
- [13] R. Bellman and S. Dreyfus. *Applied Dynamic Programming*. Princeton, New Jersey: Princeton University Press. doi:10.2307/j.ctt183pq99 (1962)
- [14] S. Voß, S. Martello, I.H. Osman, and C. Roucairol. *Meta-heuristics: Advances and trends in local search paradigms for optimization*. Springer Science & Business Media, 2012.
- [15] K. Sörensen. *Metaheuristics - the metaphor exposed*. *International Transactions in Operational Research*, vol. 22(1): pp. 318, 2013.
- [16] E.G. Talbi. *Metaheuristics: From Design to Implementation*. Wiley, (2009)
- [17] N. Mladenović and P. Hansen. *Variable neighborhood search*. *Computers & Operations Research*, vol. 24(11): pp. 1097-1100, 1997.
- [18] N. Mladenović. *A variable neighborhood algorithm-a new metaheuristic for combinatorial optimization*. In papers presented at Optimization Days, p. 112. 1995.
- [19] S. Kirkpatrick and G. Toulouse. *Configuration space analysis of traveling salesman problems*. *Journal de physique* 46, 1277-1292 (1985)
- [20] P. Hansen and N. Mladenović. *Developments of the variable neighborhood search*. In C. Ribeiro and P. Hansen, editors, *Essays and Surveys in Metaheuristics*, pages 415-439. Kluwer Academic Publishers, 2002.
- [21] Y. Xiao, I. Kaku, Q. Zhao and R. Zhang. *A reduced variable neighborhood search algorithm for uncapacitated multilevel lot-sizing problems*. *European Journal of Operational Research*, Volume 214, Issue 2, 16 October 2011, Pages 223-231
- [22] L. Matijević, T. Davidović, V. Ilin and P.M. Pardalos. *General variable neighborhood search for asymmetric vehicle routing problem*. XLVI International Symposium on Operational Research SYM-OP-IS 2019 : Proceedings, Kladovo 15-18. September 2019. ISBN: 978-86-7680-363-7
- [23] B. Menéndez, M. Bustillo, E.G. Pardo and A. Duarte. *General Variable Neighborhood Search for the Order Batching and Sequencing Problem*. *European Journal of Operational Research*, Volume 263, Issue 1, 16 November 2017, Pages 82-93
- [24] A. Sifaleras and I. Konstantaras. *General variable neighborhood search for the multi-product dynamic lot sizing problem in closed-loop supply chain*. *Electronic Notes in Discrete Mathematics*, Volume 47, February 2015, Pages 69-76

- [25] S. Frifita, M. Masmoudi and J. Euchi. *General variable neighborhood search for home healthcare routing and scheduling problem with time windows and synchronized visits*. Electronic Notes in Discrete Mathematics, Volume 58, April 2017, Pages 63-70
- [26] A. Ilić, D. Urošević, J. Brimberg and N. Mladenović. *A general variable neighborhood search for solving the uncapacitated single allocation p-hub median problem*. European Journal of Operational Research, Volume 206, Issue 2, 16 October 2010, Pages 289-300
- [27] N. Mladenović, D. Urošević and D. Pérez-Brit. *Variable neighborhood search for minimum linear arrangement problem*. Yugoslav Journal of Operations Research, 26 (2016), Number 1, 316 DOI: 10.2298/YJOR140928038M
- [28] P. Hansen, B. Jaumard, N. Mladenović and A. Parreira. *Variable neighborhood search for Weighted maximum satisfiability problem*. Les Cahiers du GERAD G200062, HEC Montréal, Canada
- [29] N. Smiti, M.M. Dhiaf, B. Jarboui and S. Hanafi. *Skewed general variable neighborhood search for the cumulative capacitated vehicle routing problem*. Intl. Trans. in Op. Res. 27 (2020) 651664, DOI: 10.1111/itor.12513
- [30] H. Derbel, B. Jarboui and R. Bhiri. *A skewed general variable neighborhood search algorithm with fixed threshold for the heterogeneous fleet vehicle routing problem*. Ann Oper Res 272, 243272 (2019). <https://doi.org/10.1007/s10479-017-2576-2>
- [31] J. Brimberg, N. Mladenović, R. Todosijević and D. Urošević. *A general framework for nested variable neighborhood search*. Electronic Notes in Discrete Mathematics, Volume 58, April 2017, Pages 159-166
- [32] J. Brimberg, N. Mladenović, R. Todosijević and D. Urošević. *Nested variable neighborhood search*. Proc. of XLII International Symposium on Operations Research (SYM-OP-IS), Serbia, 2015.
- [33] R. Todosijević, R. Benmansour, S. Hanafi, N. Mladenović and A. Artiba. *Nested general variable neighborhood search for the periodic maintenance problem*. European Journal of Operational Research, Volume 252, Issue 2, 16 July 2016, Pages 385-396
- [34] P. Hansen, N. Mladenović and D. Perez-Britos. *Variable Neighborhood Decomposition Search*. Journal of Heuristics 7, 335350 (2001). <https://doi.org/10.1023/A:1011336210885>
- [35] M. Costa, F. Monclar and M. Zrikem. *Variable neighborhood decomposition search for the optimization of power plant cable layout*. Journal of Intelligent Manufacturing 13, 353365 (2002). <https://doi.org/10.1023/A:1019980525722>
- [36] Q. Zhao, C. Xie and Y. Xiao. *A variable neighborhood decomposition search algorithm for multilevel capacitated lot-sizing problems*. Electronic Notes in Discrete Mathematics, Volume 39, 1 December 2012, Pages 129-135, <https://doi.org/10.1016/j.endm.2012.10.018>

- [37] D. Džamić, D. Aloise and N. Mladenović. *Ascentdescent variable neighborhood decomposition search for community detection by modularity maximization*. Ann Oper Res 272, 273287 (2019). <https://doi.org/10.1007/s10479-017-2553-9>
- [38] M.A. Lejeune. *A variable neighborhood decomposition search method for supply chain management planning problems*. European Journal of Operational Research, Volume 175, Issue 2, 1 December 2006, Pages 959-976, <https://doi.org/10.1016/j.ejor.2005.05.021>
- [39] F. Seeanner, B. Almada-Lobo and B. Almada-Lobo. *Combining the principles of variable neighborhood decomposition search and the fix & optimize heuristic to solve multi-level lot-sizing and scheduling problems*. Computers & Operations Research, Volume 40, Issue 1, January 2013, Pages 303-317, <https://doi.org/10.1016/j.cor.2012.07.002>
- [40] D. Urošević, J. Brimberg and N. Mladenović. *Variable neighborhood decomposition search for the edge weighted k-cardinality tree problem*. Computers & Operations Research, Volume 31, Issue 8, July 2004, Pages 1205-1213, [https://doi.org/10.1016/S0305-0548\(03\)00073-X](https://doi.org/10.1016/S0305-0548(03)00073-X)
- [41] M.A. Boschetti, V. Maniezzo, M. Roffilli and A.B. Röhrer. *Matheuristics: Optimization, Simulation and Control*. Blesa M.J., Blum C., Di Gaspero L., Roli A., Sampels M., Schaerf A. (eds) Hybrid Metaheuristics. HM 2009. Lecture Notes in Computer Science, vol 5818. Springer, Berlin, Heidelberg (2009)
- [42] P. Hansen, N. Mladenović and D. Urošević. *Variable neighbourhood search and local branching*. Comput. Oper. Res. 33(10), 3034-3045 (2006)
- [43] J. Lazić, S. Hana, N. Mladenović and D. Urošević. *Variable neighbourhood decomposition search for 0-1 mixed integer programs*. Oper. Res. 37(6), 1055-1067 (2010)
- [44] P. Jovanović, T. Davidović, J. Lazić and S. Mitrović-Minić. *The variable intensity neighborhood search for 0-1 mip*. In: Proc. 42nd Symposium on Operations Research, SYM-OP-IS 2015. pp. 229-232. Srebrno jezero, Serbia (2015)
- [45] S. Mitrović-Minić and A.P. Punnen. *Local search intensified: Very large-scale variable neighborhood search for the multi-resource generalized assignment problem*. Discrete Optimization 6(4), 370-377 (2009)
- [46] N. Mladenović, M. Dražić, V. Kovačević-Vujčić and M. Čangalović. *General variable neighborhood search for the continuous optimization*. European Journal of Operational Research 191 (2008) 753770
- [47] T. Davidović and T.G. Crainic. *MPI Parallelization of Variable Neighborhood Search*. Electronic Notes in Discrete Mathematics, 39(2012), 241-248. Special Issue devoted to EURO Mini Conference XXVIII dedicated to Variable Neighborhood Search, (EUROmC-XVIII-VNS), Herceg-Novi, Montenegro, Oct. 04-07, 2012.

- [48] T. Davidović and T.G. Crainic. *Parallelization Strategies for Variable Neighborhood Search*. 2013. CIRRELT Research Report CIRRELT-2013-47.
- [49] P. Kalatzantonakis, A. Sifaleras and N. Samaras. *On a Cooperative VNS Parallelization Strategy for the Capacitated Vehicle Routing Problem*. Learning and Intelligent Optimization. LION 2019. Lecture Notes in Computer Science, vol 11968. Springer, Cham
- [50] A. Duarte, J.J. Pantrigo, E.G. Pardo, J. Sánchez-Oro. *Parallel variable neighbourhood search strategies for the cutwidth minimization problem*. IMA Journal of Management Mathematics, vol. 27, no. 1, pp. 55-73, Jan. 2016.
- [51] B. Menendez, E.G. Pardo, J. Sánchez-Oro, A. Duarte. *Parallel variable neighborhood search for the minmax order batching problem*. International Transactions in Operational Research. 2017 May;24(3):635-62.
- [52] T. G. Crainic and N. Hail. *Parallel meta-heuristics applications*. In E. Alba, editor, Parallel Metaheuristics, pages 447-494. John Wiley & Sons, Hoboken, NJ., 2005.
- [53] T. Davidović. *Parallelization of Metaheuristic Methods*. (in Serbian), Proc. 44th Symposium on Operations Research, SYM-OP-IS 2017, Zlatibor, Sept. 25-28, 2017, pp. 140-145.
- [54] M.G.A. Verhoeven and E.H.L. Aarts. *Parallel local search*. J Heuristics 1, 4365 (1995). <https://doi.org/10.1007/BF02430365>
- [55] S. Kirkpatrick and G. Toulouse. *Configuration space analysis of travelling salesman problems*. Journal Physique, 46, 1277-1292. (1985)
- [56] T. Davidović and T.G. Crainic. *Parallel local search to schedule communicating tasks on identical processors*. Parallel Computing, Volume 48, October 2015, Pages 1-14
- [57] M. Gladden. *From Virtual Teams to Hive Minds: Developing Effective Network Topologies for Neuroprosthetically Augmented Organizations*. Neuroprosthetic Supersystems Architecture, pp. 253-71. Indianapolis: Synthynion Academic, 2017.
- [58] F. García-López, B. Melián-Batista, J. A. Moreno-Pérez and J.M. Moreno-Vega. *The parallel variable neighborhood search for the p-median problem*. J. Heur., 8(3): 375-388, May 2002.
- [59] T.G. Crainic, M. Gendreau, P. Hansen and N. Mladenović. *Cooperative parallel variable neighborhood search for the p-median*. J. Heur., 10(3):293-314, 2004.
- [60] M. Sevkli and M.E. Aydin. *Parallel variable neighbourhood search algorithms for job shop scheduling problems*. IMA Journal of Management Mathematics, 18(2):117-133, 2007.
- [61] M. Polacek, S. Benkner, K.F. Doerner, and R.F. Hartl. *A cooperative and adaptive variable neighborhood search for the multi depot vehicle routing problem with time windows*. Business Research, 1(2):1-12, 2008.

- [62] M. Knausz. *Parallel variable neighbourhood search for the car sequencing problem*. Technical report, Fakultät für Informatik der Technischen Universität Wien, 2008.
- [63] S. Pirkwieser and G. Raidl. *Multiple variable neighborhood search enriched with ilp techniques for the periodic vehicle routing problem with time windows*. Hybrid Metaheuristics, pages 45-59, 2009.
- [64] M. Yazdani, M. Amiri, and M. Zandieh. *Flexible job-shop scheduling with parallel variable neighborhood search algorithm*. Expert Systems with Applications, 37(1):678-687, 2010.
- [65] M. Birattari, T. Stützle, L. Paquete and K. Varrentrapp. *A racing algorithm for configuring metaheuristics*. Langdon, W.B., Cantú-Paz, E., Mathias, K., Roy, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., Bull, L., Potter, M.A., Schultz, A.C., Miller, J.F., Burke, E., Jonoska, N. (eds.) Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1118. Morgan Kaufmann Publishers, San Francisco (2002)
- [66] O. Maron and A.W. Moore. *Hoeffding races: Accelerating model selection search for classification and function approximation*. Cowan, J.D., Tesauro, G., Alspector, J. (eds.) Advances in Neural Information Processing Systems, vol. 6, pp. 5966. Morgan Kaufmann, San Francisco (1994)
- [67] D.C. Montgomery. *Design and Analysis of Experiments, 5th edn*. John Wiley & Sons, New York (2000)
- [68] S. Holm. *A simple sequentially rejective multiple test procedure*. Scandinavian Journal of Statistics 6, 6570 (1979)
- [69] M. Birattari, Z. Yuan, P. Balaprakash and T. Stützle. *F-Race and Iterated F-Race: An Overview*. Experimental Methods for the Analysis of Optimization Algorithms. Springer, Berlin, Heidelberg (2010)
- [70] J. Brimberg, P. Hansen, and N. Mladenović. *Convergence of Variable Neighborhood Search*. Les Cahiers du GERAD, pp. 115, 2004.
- [71] P. Hansen, N. Mladenović, R. Todosijević, and S. Hanafi. *Variable neighborhood search: basics and variants*. Article in EURO Journal on Computational Optimization, August 2016. DOI: 10.1007/s13675-016-0075-x
- [72] J. Brimberg, P. Hansen and N. Mladenović. *Continuous Optimization by Variable Neighborhood Search*. Wiley Encyclopedia of Operations Research and Management Science, (2015) 10.1002/9780470400531.eorms1107.
- [73] M. Birattari. *Tuning Metaheuristics: A Machine Learning Perspective*. Studies in Computational Intelligence, Volume 197