SOLVING A $P||C_{max}$ SCHEDULING PROBLEM USING GENERAL VARIABLE NEIGHBORHOOD SEARCH

IMPLEMENTED IN *Python* AND *C*

MOHAMED HMINI

Achieved as part of a set of Optimization Techniques Projects
Master's degree of Information Systems and Intelligent Systems

Supervised by : RACHID BENMANSOUR
COMPUTER SCIENCE DEPARTMENT
INSEA
MOROCCO
December 15, 2019

Abstract

SOLVING A SCHEDULING PROBLEM USING GVNS

Abstract

meta heuristics are much flexible and efficient than exact methods, at least for the moment we are seeing an emergence of new optimization techinques which allows the exploitation of the solution space in a very weird yet effective way.

most of these meta heuristics are nature-inspired, common but unfortunate name for any stochastic optimization algorithm intended to be the last resort before giving up and using random or brute-force search. Such algorithms are used for problems where you don't know how to find a good solution, but if shown a candidate solution, you can give it a grade. The algorithmic family includes genetic algorithms, hill-climbing, simulated annealing, ant colony optimization, particle swarm optimization, and so on.

Keywords Scheduling problems, meta-heuristics, operations research, optimization

Thanks to our beloved teacher **RACHID BENMANSOUR**

Contents

Listings

LIST OF ALGORITHMS

Chapter 1

Overview

1.1 Scheduling problem classification :

In general scheduling problems are one of the most essential yet harsh to solve problems in computer-science, ergo each scheduling problem have to be defined by its classification which we'll tackle right now.

This classification is based on a classification scheme widely used in the literature, which makes the scheduling problem apparent in three forms $\alpha||\beta||\gamma$, α simply refers to **the machine environment** which characterizes the nature of the set of processing machines which have been given to tackle the problem, in our use-case example we are going to consider only **identical parallel machines** which means that each job J_i can be processed in any of these machines with the same amount of processing time, the letter β refers to **the job characteristics** which in our example is the necessary loading time for each job J_i to be loaded (prepared for processing phase) to a machine M_j, last but not least the γ refers to **the optimality criterion** which describes the objective function as well as the logic behinds the comparison function (min/max) to solve our problem, in our use-case we are confronted with a C_{max} criterion which means the overall goal is the reduce the amount of time needed to deliver the last job or simply the completion time C_{max}.

the use-case we are going to tackle is a **transportation scheduling problem**, it can be seen simply as a standard scheduling problem, the general description is as follows, let's consider an entity in possession of multiple machines (**means of transportation**) as well as a single loading machine to load a set of **client demands**, the problem is to minimize the overall time of transporting every single request, and from the last paragraph we denote that the our problem is a $P||C_{max}$ classified scheduling problem where **P** refers to **identical parallel machines** and C_{max} the **the completion time of the last delivery**.

1.2 VNS essential parameters :

VNS is a meta-heuristic based optimization technique which is widely used in the field of operations research, it's a single-instance based optimization which means it doesn't consider more than one possible solution at a time which gets evolved to the final stage returned optimal solution, the basic idea of this method makes it so efficient only when choosing optimal hyper-parameters, in general there are two phases for a VNS-extension algorithm **EXPLOITATION** and **EXPLORATION** which we shall see next.

the famous three facts which VNS is based upon sits on the heart of our approach to solve this problem :

Fact 1 : A local minimum w.r.t. one neighborhood structure is not necessarily so for another;

Fact 2 : A global minimum is a local minimum w.r.t. all possible neighborhood structures;

Fact 3 : For many problems, local minima w.r.t. one or several Nk are relatively close to each other;

1.2.1 Local search :

as i said earlier, in VNS we have an **exploitation** phase which is embedded within the **exploration** phase or the global optimization algorithm, exploitation simply means doing local search until we satisfy a stopping condition which is the most trivial case reaching the local optimum or as in our case $argmin_{x \in N_k(x')} f(x)$. i'm going to introduce three local search algorithms which have been implemented in that program :

- Best-Improvement
- First-Improvement
- Stochastic-Hill-Climbing

these algorithms are to be found detailed and discussed in the next chapter.

1.2.2 Neighborhood structures :

a neighborhood structure is what feeds the local search algorithm the needed data (related points in the space =¿ neighbors of a point x) to do the search, there a ton of variations which can be applied to these structures yet i'll only consider which i've broaden them to be formed as neighborhood structures generators (each time with different params). the structures i have used are the following :

- Block-Swapping
- Block-Reversing
- Shuffling (not implemented)

Chapter 2

Implementation of a GVNS solution to a $P||C_{max}$ scheduling problem

2.1 Pseudo-code Implementation

2.1.1 Defining the Objective function

driver scheduling problems are characterized with two essential things, firstly the loading time of each commodity, goods or passengers which we shall denote s_j the step up time of job J_j by the loading machine *loader*, secondly the processing time of these goods which we denote p_j the processing time of job J_j by the processing machine (driver) M_i. from the previous paragraph we consider the following symbols :

J_j : is the job number j (so-called a commodity).

M_i : is the machine number i (so-called a driver).

s_j : the set-up time of job J_j. (so-called the loading time)

p_j : the processing time of job J_j. (so-called the delivery time)

the problem is of minimization type cause we want to delivery every single job at an optimal overall processing time considering the available drivers as well as the available loading machine (which is only one of a type), now we have to define two types of functions, the first one will allow us the schedule a specific job J_j, and the second will be considered as the objective function to be minimized, we shall notice that the first one is embedded in the second one.

$$\mathscr{S}(J_j) = max \begin{cases} min_{0<i<m} Availability(M_i) \\ \\ Availability(Loader) \end{cases} \tag{2.1}$$

$$Availability(R) = \begin{cases} \mathscr{S}(J_{j-1}) + C(J_{j-1}), & \text{if } R \in M_{0<i<m} \\ \mathscr{S}(J_{j-1}) + s_{j-1}, & \text{if } R \text{ is } Loader \end{cases} \tag{2.2}$$

as clear as it appears we defined the scheduling function for our problem, which is the earlier one denoted $\mathscr{S}$, whereas the later is characterizing the time of availability of the machine denoted $Availability(R)$, now we have to look at the objective function which we'll try to minimizing, let's put $C(J_j)$ the objective function where $C(J_{last})$ (so-called C_max) is what we are trying to minimize or in other words the completion time of the last job that has been delivered.

$$C(J_j) = \mathscr{S}(J_j) + s_j + p_j \tag{2.3}$$
$$C_{max}(x) = C(J_{x[last]}) \tag{2.4}$$

$$argmin f(x) \tag{2.5}$$
$$f(x) = C_{max}(x) \tag{2.6}$$

the variable x defines the current scheduling of jobs from $j = 0$ to $x = n$, now that we have defined our objective function we need to set up the GVNS meta-heuristic to do the job for us but first let's look at the pseudo-code of our objective function which shall be defined below 1.

Algorithm 1: ObjectiveFunction(X, n, p, s, m, Loader)

Input: current solution (X), total # of commodities (n), processing time per job (p) loading time per job (s) total # of drivers (m), loading machine (Loader)

Result: the completion delivery time of the last commodity C_{max}

```
1  Loader ← 0
2  M_{0<i<m} ← 0
3  for j ← 0 to n do
4      M_min ← min_{0<i<m}(M_i)
5      S(X_j) ← max(M_min, Loader)
6      Loader ← S(X_j) + s_j
7      C(X_j) ← Loader + p_j
8      M_min ← C(X_j)
9  End for
10 C_max = max_{0<i<m}(M_i)
```

2.1.2 Local-Search pseudo-code Implementation :

as we saw earlier in the previous chapter we have to define some essential parameters for our VNS, so let's start with local search algorithms, we'll define three types of local search algos which can be used later in our test phase, down below we shall see the exact implementation of these local optimization techniques.

the first type of local-search functions and one of the most trivial ones is the **Best-Improvement** way, it considers every single possible neighbor of our current solution X and tries to get the best out of the neighborhood, this technique is useful in smaller data-sets but it's a time consuming and an ineffective technique, the implementation of it is given in 2.

Algorithm 2: Best-Improvement(X, N_k)

Input: current solution (X), neighborhood structure (N_k)
Result: local optima solution $X^{'}$

```
do
    X' ← X
    for X" ∈ N_k(X') do
        if f(X") < f(X') then
            X' ← X"
        End if
    End for
while f(X') < f(X);
```

the best-improvement guarantees the local optima yet it takes a lot of time to do so, in the next methods that we are supposed to be using we'll only consider a shallow deterministic search called **First-Improvement** and a complete stochastic-based search called **Stochastic-ill-Climbing**.

first improvement is based on two other facts, firstly, local minima isn't that good, secondly, it takes too much time to get to that not good local minima, therefor the idea of descending from the first observation that an available neighbor point X^{i} is better than our current solution makes it a lot easier to find optimums in a perfect time interval, the cood of such algo is given below in 3.

an extreme case of **first-improvement** is the stochastic search, where we don't care about visiting all neighbors or even visiting them in order, the function below 4 uses a **shake** function to randomly generate a random neighbor of a given point.

Algorithm 3: First-Improvement(X, N_k)

Input: current solution (X), neighborhood structure (N_k)
Result: local optima solution $X^{'}$

```
do
    i ← 0
    X' ← X
    while f(X') >= f(X) do
        X' ← N_k(X)_i
        i ← i + 1
    End while
while f(X') < f(X);
```

Algorithm 4: Stochastic-ill-Climbing(X, N_k, failurePoint)

Input: current solution (X), neighborhood structure (N_k), nbr of iterations (failurePoint)
Result: local optima solution $X^{'}$

```
counter ← failurePoint
while counter > 0 do
    X' ← shake(X, N_k)
    if f(X') < f(X) then
        X ← X'
        counter ← failurePoint
    else
        counter ← counter − 1
    End if
```

the shake function simply generates a random neighbor from a given neighborhood structure $N_k(X)$, the related pseudo-code can be expressed as in 5.

Algorithm 5: shake(X, N_k)

Input: current solution (X), neighborhood structure (N_k)
Result: random neighbor $X^{'}$

```
X' ∈ N_k(X)
```

2.1.3 Neighborhood Structures pseudo-code Implementation:

the other part of a variable neighborhood search is the **neighborhood generator** itself, which can be called a neighborhood structure, in our usecase we shall use two different approaches to generate neighborhoods, firstly we shall look at **Block-Swapping** which takes two parameters the size of the blocks and the distance between the two blocks to be swapped, the code is given in **Algorithm 6**.

Algorithm 6: BlockSwapping(X, BlockSize, SwapFactor, neiIndex)

Input: current solution (X), block size (BlockSize), swap factor(SwapFactor) neighbor index (neiIndex)
Result: neighbor point $X^{'}$

$bound \leftarrow |X| - (2 * BlockSize + SwapFactor) + 1$
if $neiIndex < bound$ **then**
  $lowerbound1 \leftarrow neiIndex$
  $upperbound1 \leftarrow neiIndex + BlockSize$
  $lowerbound2 \leftarrow neiIndex + BlockSize + SwapFactor$
  $upperbound2 \leftarrow neiIndex + 2 * BlockSize + SwapFacto$
  $X^{'} \leftarrow X$
  swap($X^{'}[from\text{lowerbound1}to\text{upperbound1}], X^{'}[from\text{lowerbound2}to\text{upperbound2}]$)
End if

the previous neighborhood structure has a definite amount of neighbors of each given parameter this is way we should specify the *bound* constant, it's the same for the next structure which is given in **Algorithm 7** where the only parameter is the size of the block to be reversed.

Algorithm 7: BlockReversing(X, BlockSize, neiIndex)

Input: current solution (X), block size (BlockSize), neighbor index (neiIndex)
Result: neighbor point $X^{'}$

$bound \leftarrow |X| - BlockSize + 1$
if $neiIndex < bound$ **then**
  $lowerbound \leftarrow neiIndex$
  $upperbound \leftarrow neiIndex + BlockSize$
  $X^{'} \leftarrow X$
  reverse($X^{'}[from\text{lowerbound}to\text{upperbound}]$)
End if

2.1.4 GVNS pseudo-code Implementation

the General Variable Neighborhood search is a combination of VND or variable neighborhood descent and reduced VNS, the later is just a use of the shake method we defined whereas the earlier is an optimized **local-search** algorithm which make use of different types of neighborhood structures.

Algorithm 8: changeNeighborhood($X^{'}$, X, K)

Input: current solution ($X^{'}$), earlier best solution (X), current neighborhood structure (K)
Result: (best solution ($X^{"}$), next neighborhood structure (K))

if $f(X^{'}) < f(X)$ **then**
    $X^{"} \leftarrow X^{'}$
    $K \leftarrow 1$
else
    $X^{"} \leftarrow X$
    $K \leftarrow K + 1$
End if

when doing VND or GVNS we are constantly changing neighborhood structures, the nature of the change depend on the evolution we made in the last step of the search, the given pseudo-code is found above **Algorithm 8**.

now to perform better on the local-search we need to optimize our algorithms with the ability to switch to multiple neighborhood structures to allow the diversification of neighborhoods during the search, this idea is presented in the variable neighborhood descent underneath this paragraph **Algorithm 9**.

Algorithm 9: VND(X, K_{max})

Input: current solution (X), neighborhood structures (K_{max})
Result: neighbor point $X^{'}$

$K \leftarrow 1$
$X^{'} \leftarrow X$
while $K < K_{max}$ **do**
    $X^{"} \leftarrow LocalSearch(X^{'}, N_K)$
    $(X^{'}, K) \leftarrow changeNeighborhood(X^{"}, X^{'}, K)$
End while

we've already built the needed building blocks for our end-product, the GVNS is based on everything we talked about earlier, from local search to neighborhood structures considering

the shaking phase as well, it has the random functionality of a **reduced VNS** and the optimized local search of **VND**, our final algorithm is presented in **Algorithm 10**.

the difference between the **basic VNS** and **general VNS** can be represented in two things, firstly in BVNS we are only using one set of neighborhood structures whereas in GVNS we can use two sets, one for the **shake** phase the the other for the local search, secondly, BVNS uses a basic local search algorithm which means it doesn't exploit the local neighborhoods with different neighborhood structures, as we noticed before GVNS uses VND as a local search algorithm.

Algorithm 10: GVNS(X, K_{max}, L_{max}, StoppingCondition)

Input: current solution (X),
first set of neighborhood structures (K_{max}),
second set of neighborhood structures (L_{max}),
the stopping condition (StoppingCondition)

Result: neighbor point $X^{'}$

```
1 K ← 1
2 X' ← X
3 while isReached(StoppingCondition) == False do
4     while K < K_max do
5         X" ← shake(X', K)
6         X^3rd ← VND(X", L_max)
7         (X', K) ← changeNeighborhood(X^3rd, X', K)
8     End while
9 End while
```

2.2 Py-code Implementation

2.2.1 Setting up the environment :

due to the readability of the python code and its easiness in implementation i suggest to start looking at the source code in this section which will be completely in **Python** and then we can move on to the **C** which is a bit harder to conventionalize due to nature of the VNS code, i created the VNS header for a **general purpose reason**, it can be used to solve any optimization problem.

as usual we start by defining the **objective function**, let's look at the related source code of it down below in **listing 2.1**.

```python
# OBJECTIVE FUNCTION :
def f(policy):
    loader = 0
    Machines = [0 for i in range(ds["m"])]
    for i in policy:
        mnv = min(Machines)
        mni = Machines.index(mnv)
        starting_pt = max(mnv, loader)
        loader = (starting_pt + ds["s"][i])
        Machines[mni] = (loader + ds["p"][i])
    return max(Machines)
```

Listing 2.1: Py Objective Function

2.2.2 Local Search Implementation in Py :

unfortunately python is slow in execution time, so i'll only consider **first improvement local search** or even the **stochastic hill climbing search**, the source code which represents the first improvement local search is given in **listing 2.2** down below.

```python
# first improvement :
def first_improvement(bx, struct):
    while True:
        x = bx
        i = 0
        while True:
            xc = struct(bx, i)
            if xc == None:
                break;

            if f(xc) < f(x):
                x = xc
                break;
            else:
                i += 1
        if f(x) >= f(bx):
            break;
        else:
            bx = x
    return bx
```

Listing 2.2: Py First Improvement

the second method that we shall use if python is the way to go, is **stochastic hill climbing search**, the idea is described earlier but for a refresher we can say that we are blindly jumping from a neighbor to a neighbor hopping for a better solution, the source code is found in **listing 2.3**.

```
def stochastic_hill_climbing(bx, struct, stopping_condition = 10):
    # the stopping condition allows the exit if no evolution is found after
    some amount of iterations :
    counter = stopping_condition

    while counter > 0:
        x = struct(bx, -1)
        if x is not None:
            if f(x) < f(bx):
                bx = x
                counter = stopping_condition
            else:
                counter -= 1
    return bx
```

Listing 2.3: Py Stochastic Hill Climbign

2.2.3 Neighborhood Structures Implementation in Py :

both block swapping and block reversing are bounded, which means we can calculate the number of neighbors in a given neighborhood structure starting from a given initial point **X**, will see that the implementation is the very same thing in **C**, the source code of the used neighborhood structures is given below.

the block swapping entry **listing 2.4**, given below contains an instruction to calculate the bound which is essential to proceed the operation.

```
def block_swapping(x, block_size = 1, swap_factor = 0, index = -1):
    bound = len(x) - (2*block_size + swap_factor) + 1;
    xc = None

    if index == -1 :
        index = random.randint(0, bound)

    if index < bound:
        xc = __block_swapping(x, block_size, swap_factor, index);

    return xc
```

Listing 2.4: Py Block Swapping

the logic behind the swapping is given in **listing 2.5**.

```
def __block_swapping(x, block_size = 1, swap_factor = 0, index = 0):
    lb = index+block_size + swap_factor;
    ub = index+2*block_size + swap_factor;

    xc = x.copy()
    xc[index:index+block_size], xc[lb:ub] = xc[lb:ub], xc[index:index+
    block_size]

    return xc
```

Listing 2.5: Py Block Swapping Logic

the block reversing entry **listing 2.6**, given below contains an instruction to calculate the bound which is essential to proceed the operation.

```
def block_reversing(x, block_size = 1, index = -1):
    bound = len(x) - block_size + 1
    xc = None

    if index == -1 :
        index = random.randint(0, bound)

    if index < bound:
        xc = __block_reversing(x, block_size, index);

    return xc
```

Listing 2.6: Py Block Reversing

the logic behind the reversing is given in **listing 2.7**.

```
def __block_reversing(x, block_size = 1, index = 0):
    xc = x.copy()
    rb = xc[index:index + block_size]
    xc[index:index + block_size] = rb[::-1]
    return xc
```

Listing 2.7: Py Block Reversing Logic

2.2.4 GVNS Py-code Implementation :

as usual we need a change neighborhood function to switch from a neighborhood to another when a certain condition is met, the logic is simple and it's previously discussed in the earlier section and will be detailed in the coming section, the code is given in **listing 2.8**.

```
def change_neighborhood(x, bx, k):
    if f(x) < f(bx):
        bx,k = x, 0
    else:
        k += 1
    return bx,k
```

Listing 2.8: Py Change Neighborhood

the optimized local search algorithm used by the General VNS is **VND**, therefor the need for VND is essential, we implement VND in py in the following way **listing 2.9**, notice that we used **stochastic search** within in VND algo, this makes the algorithm behaves irrationally or randomly this type of VNDs is called **Random VND**.

```
# variable neighborhood descent :
def VND(bx, Ns):
    k = 0
    while k < len(Ns):
        x = stochastic_hill_climbing(bx, Ns[k]) # local search
        print(f(x))
        bx,k = change_neighborhood(x, bx, k)
    return bx
```

Listing 2.9: Py VND

as said GVNS uses VND as local search and a phase to generate a random solution each time before the local search, this logic is appeared clearly in the **listing 2.10** down below.

```
# RVNS (N1) + VND (N2) :
def GVNS(bx, N1, N2, stopping_condition = 10):
    counter = stopping_condition

    while counter > 0:
        print(f(bx))
        x_start = bx.copy()
        k = 0
        while k < len(N1):
            x = N1[k](bx, -1)
            x = VND(x, N2)
            bx, k = change_neighborhood(x, bx, k)

        if f(x_start) <= f(bx):
            counter -= 1
        else:
            counter = stopping_condition
    return bx
```

Listing 2.10: Py GVNS

2.3 C-code Implementation

2.3.1 VNS Header file :

the project's code is built upon **HLCio** an earlier project of mine which can be found in the following repo https://github.com/MohamedHmini/HLCio, it helps me to parse data from files and to structure it in a neat way.

we will first look at the overall nature of the **VNS** project which can be broaden to solve any type of problems just by defining the exact **configurations**, let's explore the header file first, which shall be found in.

```
1   // the optimal value shall be stored in a var of type :
2   typedef DF_ELEMENT OPT_VAL;
3
4   // possible solution :
5   typedef DF_ELEMENT POLICY;
6
7   // each neighborhood structure is this type :
8   typedef POLICY* (*NEIGHBORHOOD_STRUCT)();
9
10  // each set of neighborhood structures is of type :
11  typedef POLICY* (**NEIGHBORHOOD_STRUCTURES)();
12
13  // each local search function shall be of type LOCAL_SEARCH
14  typedef POLICY (*LOCAL_SEARCH)();
15
16  // after a local search is done we should return the results using the
    following type
17  typedef struct LOCAL_SEARCH_RESULT{
18      POLICY bx; // neighborhood structure optimal policy
19      int k; // neighborhood structure index
20  }LOCAL_SEARCH_RESULT;
21
22  // the returned value type after using VNS
23  typedef struct VNS_RESULT{
24      POLICY policy;
25      OPT_VAL opt_val;
26  }VNS_RESULT;
27
28  // OBJECTIVE FUNCTION :
29  typedef OPT_VAL (*OPT_FUNC)(POLICY);
30
31  // optimality comparison between two instances :
32  typedef CMP_RESULT (*CMP_OPTIMALITY)(POLICY, POLICY, int, int);
33
34  // this structure is made to allow customization of VNS :
35  typedef struct VNS_CONFIG{
36      OPT_FUNC f;
37      CMP_OPTIMALITY cmp_optimality;
38      LOCAL_SEARCH local_search;
39      int STOCHASTIC_LR_FAILURE_LIMIT;
40      DATAFRAME *ds;
41      int* other_params;
42  }VNS_CONFIG;
43
```

```
44    extern VNS_CONFIG vns_config;
```

Listing 2.11: Types in VNS Header File

2.3.2 Local Search Implementation in C :

the VNS is implemented not only to solve this very problem, but also to be used to solve any kind of problems, due to the nature of the program which is characterized by configuring the VNS before using it so it can be **projected** unto the domain of application.

we shall find the best-improvement implementation in **Listing 2.12**, the idea is simple as discussed in the pseudo-code.

```
// DETERMINISTIC LOCAL SEARCH :
POLICY best_improvement(POLICY bx, NEIGHBORHOOD_STRUCT nei_struct){
    POLICY bxc = df_element_copy(bx);
    POLICY *x = NULL;
    CMP_RESULT cr;

    // local search exploitation :
    // the first loop won't stop until convergence :
    do{
        POLICY started_with = df_element_copy(bxc);
        int i = 0;
    // brute-force search for argmin in the neighborhood :
        while((x = nei_struct(bxc,i)) != NULL){

    // here the value 0 will be returned if x is optimal than bxc :
            cr = vns_config.cmp_optimality(*x, bxc, 0, 1);

            if(cr.index == 0){
                arrfree(&bxc);
                bxc = df_element_copy(*x);
            }

            arrfree(&cr.best);
            arrfree(x);
            free(x);
            x = NULL;
            i++;
        }

        cr = vns_config.cmp_optimality(bxc, started_with, 0, 1);
        arrfree(&cr.best);
        arrfree(&started_with);
    }while(cr.index == 0);

    return bxc;
}
```

Listing 2.12: C Best-Improvement

the next algoritm will be the first improvement which differs from the best improvement by the exit condition from a neighborhood, the code is found in **Listing 2.13**, this is what we'll use mainly to solve the driver scheduling problem, it's more effective than the **stochastic method** for medium size datasets.

```
POLICY first_improvement(POLICY bx, NEIGHBORHOOD_STRUCT nei_struct){
    POLICY bxc = df_element_copy(bx);
    CMP_RESULT cr;

    // local search exploitation :
    do{
        POLICY *x = NULL;
        POLICY started_with = df_element_copy(bxc);
        int i = 0;

        while((x = nei_struct(bxc,i)) != NULL){
            cr = vns_config.cmp_optimality(*x, bxc, 0, 1);

            if(cr.index == 0){
                arrfree(&bxc);
                bxc = df_element_copy(*x);
                arrfree(&cr.best);
                arrfree(x);
                free(x);
                x = NULL;
  // this instruction makes a best-improvement algorithm a first-
improvement one :
                break;
            }

            arrfree(&cr.best);
            arrfree(x);
            free(x);
            x = NULL;
            i++;
        }

        cr = vns_config.cmp_optimality(bxc, started_with, 0, 1);
        arrfree(&cr.best);
        arrfree(&started_with);
    }while(cr.index == 0);

    return bxc;
}
```

Listing 2.13: C First-Improvement

a stochastic local search shall be so efficient in larger datasets, the code i given i **listing 2.14**

```
// STOCHASTIC LOCAL SEARCH :
POLICY stochastic_hill_climbing(POLICY bx, NEIGHBORHOOD_STRUCT
nei_struct){
     int stop_condition = vns_config.STOCHASTIC_LR_FAILURE_LIMIT;
     int counter = stop_condition;
     POLICY bxc = df_element_copy(bx);

     while(counter > 0){
// instead of searching in the ordinary way, we can randomize the
search as follows :
// the -1 value means random :
          POLICY *x = nei_struct(bxc, -1);
          CMP_RESULT cr = vns_config.cmp_optimality(*x, bxc, 0, 1);


          if(cr.index == 0){
               arrfree(&bxc);
               bxc = df_element_copy(*x);
               counter = stop_condition;
          }
          else{
               counter--;
          }
          arrfree(&cr.best);
          arrfree(x);
          free(x);
          x = NULL;
     }


     return bxc;
}
```

Listing 2.14: C Stochastic-Hill-Climbing

2.3.3 Neighborhood Structures Implementation in C :

both BlockSwapping and BlockReversing can be used stochastically or deterministically allowing all types of search algorithms to make use of them, from each starting point **X** we can extract a neighbor by its index (only of the index is less or equal to the structure bound).

the block swapping entry is given in **listing 2.15**, it's just a pre-operational functions which defines the bound and set a value for the **index** if it's equal to -1 (stochastic).

```
POLICY* block_swapping(POLICY x, int block_size, int swap_factor, int
i){
    // this is a mathematically proven bound :
    int bound = x.node.Arr->size - (2*block_size + swap_factor) + 1;
    POLICY* nx = NULL;
    // we use the structure in a stochastic manner only if the index is
 equal to -1:
    if(i == -1){
        srand(time(NULL));
        i = randint(0, bound);
    }
    // if we met our conditions than we proceed the operation :
    if(i < bound)
        nx = __block_swapping(x, block_size, swap_factor, i);

    return nx;
}
```

Listing 2.15: C Block-Swapping

the main logic behind the block swapping is given below in **listing 2.16**, we just define a lower bound for the second block and start swapping element by element.

```
POLICY* __block_swapping(POLICY x, int block_size, int swap_factor, int i){
    int lb = i+block_size + swap_factor;

    POLICY* nx = malloc(sizeof(DF_ELEMENT));
    *nx = df_element_copy(x);

    // swapping :
    for(int j = 0;j<block_size;j++){
        int tmp;
        tmp = nx->node.Arr->data[i + j].node.Int;
        nx->node.Arr->data[i + j].node.Int = nx->node.Arr->data[lb + j].
    node.Int;
        nx->node.Arr->data[lb + j].node.Int = tmp;
    }

    return nx;
}
```

Listing 2.16: C Block-Swapping-Logic

the block reversing entry is given in **listing 2.17**, it's just a pre-operational functions which defines the bound and set a value for the **index** if it's equal to -1 (stochastic), same as the previously viewed neighborhood structure, in this structure the bound is easily defined and can be understood by intuition.

```
    POLICY* block_reversing(POLICY x, int block_size,  int i){
        int bound = x.node.Arr->size - block_size + 1;
        POLICY* nx = NULL;

        if(i == -1){
            srand(time(NULL));
            i = randint(0, bound);
        }

        if(i < bound)
            nx = __block_reversing(x, block_size, i);

        return nx;
    }
```

Listing 2.17: C Block-Reversing

the main logic behind the block reversing is given below in **listing 2.18**.

```
POLICY* __block_reversing(POLICY x, int block_size, int i){

    POLICY* nx = df_element_create();
    *nx = df_element_copy(x);

    // reversing :
    for(int j = 0;j< (int)((block_size-2)/2);j++){
        int tmp;
        int first_element = 1 + j;
        int second_element = block_size - 2 - j;
        tmp = nx->node.Arr->data[i + first_element].node.Int;
        nx->node.Arr->data[i + first_element].node.Int = nx->node.Arr->data
    [i + second_element].node.Int;
        nx->node.Arr->data[i + second_element].node.Int = tmp;
    }

    return nx;
}
```

Listing 2.18: C Block-Reversing-Logic

2.3.4 GVNS C-code Implementation :

as in the previous section we shall start by basics and then build up the logic to the general VNS, our next move will be to impelement the relative C code for the **change of neighborhood** logic, we shall find the source code in **listing 2.19**.

we should always keep in mind that the logic by which the whole VNS code is written isn't a specific problem driven, the very same code can be reused everywhere with any type of optimization problems due to the availability of the **VNS-CONFIG** structure.

```
LOCAL_SEARCH_RESULT change_neighborhood(POLICY x, POLICY bx, int k){
    LOCAL_SEARCH_RESULT lsr;
    // cmp_optimality is part of vns_config and it defines the comparison
    function :
    CMP_RESULT cmp_r = vns_config.cmp_optimality(x, bx, 0, 1);

    if(cmp_r.index == 0)
        k = 0;
    else
        k++;

    lsr.k = k;
    lsr.bx = df_element_copy(cmp_r.best);
    arrfree(&cmp_r.best);

    return lsr;
}
```

Listing 2.19: C Change-Neighborhood

now it's time to look at the VND, as we encountered before, this algorithm allows the exploitation of multiple types of neighborhoods of a given starting point **X** called neighborhood structures, in the source code given **listing 2.20** we start by looping the structures then we make use of a **local search algorithm** which can be defined by the developer or just using one of the previously seen algos, we then shall make a change of neighborhood using **2.19**

```
POLICY VND(POLICY bx, NEIGHBORHOOD_STRUCTURES structs, int kmax){
    int k = 0;
    POLICY bxc = df_element_copy(bx);

    while(k < kmax){
        POLICY x = vns_config.local_search(bxc, structs[k]);
        LOCAL_SEARCH_RESULT lsr = change_neighborhood(x, bxc, k);
        // change of neighborhood
        k = lsr.k;
        // change of the best solution
        arrfree(&bxc);
        arrfree(&x);
        bxc = df_element_copy(lsr.bx);
        arrfree(&lsr.bx);
    }

    return bxc;
}
```

Listing 2.20: C VND

in this function we expect two sets of neighborhood structures of type **NEIGHBORHOOD-STRUCTURES** where the first one is used in the shaking phase and the second for the VND search phase, another thing to notice is the **stopping condition**, in this implementation i didn't broaden the concept, the algorithm will stop when there is no evolution (stagnation) from the last found optimum till the last iteration of the specified stopping condition value, the algorithm is found underneath this paragraph in **listing 2.21**.

```
POLICY GVNS(POLICY bx, NEIGHBORHOOD_STRUCTURES N1, NEIGHBORHOOD_STRUCTURES
    N2, int kmax, int lmax, int stopping_condition){

    POLICY bxc = df_element_copy(bx);
    int counter = stopping_condition;

    while(counter > 0){
        POLICY x_start = df_element_copy(bxc);
        int k = 0;
        while(k < kmax){
            printf("%d\n", vns_config.f(bxc).node.Int);
            // the shake phase :
            POLICY* x = N1[k](bxc, -1);
            // in case of memory allocation problem :
            if(x != NULL){
                // local search :
                POLICY x_2 = VND(*x, N2, lmax);

                arrfree(x);
                free(x);
                x = NULL;

                // changing the neighborhood :
                LOCAL_SEARCH_RESULT lsr = change_neighborhood(x_2, bxc, k);
                k = lsr.k;

                arrfree(&x_2);
                arrfree(&bxc);
                bxc = df_element_copy(lsr.bx);
                arrfree(&lsr.bx);
            }

        }

        // change the status of stopping condition :
        CMP_RESULT cr = vns_config.cmp_optimality(bxc, x_start, 0, 1);
        if(cr.index == 0)
            counter = stopping_condition;
        else
            counter--;
        arrfree(&cr.best);
        arrfree(&x_start);
    }

    return bxc;
}
```

Listing 2.21: C GVNS

Chapter 3

Solving a $P||C_{max}$ scheduling problem

3.1 Setting up VNS configurations :

before using the GVNS algorithm we'll define certain configurations, firstly we need an **objective function** for our problem, it's the same thing as the previously seen one in **algorithm 1**, the implementation of this function in our environment is represented undernearth this paragraph in **listing 3.1**.

```
OPT_VAL f(POLICY x){
    OPT_VAL ov;

    DF_ELEMENT loader;
    loader.node.Int = 0;
    DF_ELEMENT initval;
    initval.type = DF_ELEMENT_TInt;
    initval.node.Int = 0;
    DF_ELEMENT machines = arrinit(vns_config.other_params[1], initval);

    for(int i = 0;i<x.node.Arr->size; i++){
    // this part returns the min of the machines :
        CMP_RESULT mn = arrcmp(&machines, min);
    // this one returns the max between the min machine and the loader :
        CMP_RESULT mx = max(mn.best, loader, 0, 0);

        loader.node.Int = mx.best.node.Int + vns_config.ds->data[1][x.node.Arr->data[i].node.Int].node.Int;
        machines.node.Arr->data[mn.index].node.Int = loader.node.Int + vns_config.ds->data[0][x.node.Arr->data[i].node.Int].node.Int;
    }
    // returning the related C max :
    CMP_RESULT mx = arrcmp(&machines, max);

    ov = mx.best;
    arrfree(&machines);
    return ov;
}
```

Listing 3.1: Objective Function

the second thing to build is the comparison function of type **CMP OPTIMALITY**, our optimization is a minimization kind so the CMP OPTIMALITY function is as follows **listing 3.2**.

```
CMP_RESULT cmp(POLICY x, POLICY bx, int i, int j){
    CMP_RESULT r;

    if(vns_config.f(x).node.Int < vns_config.f(bx).node.Int){
        r.best = df_element_copy(x);
        r.index = i;
    }
    else{
        r.best = df_element_copy(bx);
        r.index = j;
    }
    // returing the best input with its related index (i or j)
    return r;
}
```

Listing 3.2: Compare Optimality

the third point is defining the neighborhood structures, we can redefine the existing neighborhood structures by injecting different parameters each time, let's look at the source code of our structures in **listing 3.3**

```
// the first set of neighborhood structures :
POLICY* s1(POLICY x, int i){
    return block_swapping(x, 10, 0, i);
}

POLICY* s2(POLICY x, int i){
    return block_swapping(x, 1, 10, i);
}

POLICY* s3(POLICY x, int i){
    return block_swapping(x, 30, 0, i);
}

// the second set of neighborhood structures:
POLICY* s4(POLICY x, int i){
    return block_reversing(x, 36, i);
}

POLICY* s5(POLICY x, int i){
    return block_reversing(x, 47, i);
}

POLICY* s6(POLICY x, int i){
    return block_reversing(x, 10, i);
}
```

Listing 3.3: Neighborhood Structures

now all what is left is the main code to inject all of the created configurations, let's look at the source code as batches this time to make it easier and understandable, **VNS Header File** contains a global VNS CONFIG variable called vns config which may call after importing the VNS.h file, as we see down below we simply assign each objective function, comparison function and the used local search function.

```
vns_config.f = f;
vns_config.cmp_optimality = cmp;
vns_config.local_search = first_improvement;
```

we need to inject the dataset now, and we do so by parsering the related file as follows.

```
char *filename = args[1];
FILE *fds = fopen(filename, "r");
// start reading from row 1 :
// load the commodities processing time / set up time :
vns_config.ds = csv_to_df(fds, 1, "\t");
// the first row (n,m) :
// load the number of commodities and the number of drivers :
rewind(fds);
vns_config.other_params = (int*)malloc(sizeof(int) * 2);
char* fline = get_line(fds);
vns_config.other_params[0] = atoi(strtok(fline, "\t"));
vns_config.other_params[1] = atoi(strtok(NULL, "\t"));
free(fline);
```

we should retype our dataset to **Int** as follows using dataframe.h.

```
// retyping the data to int :
df_retype(vns_config.ds, DF_ELEMENT_TInt, 0);
```

now let's initiate a starting point :

```
// initiating the solution with LONGEST PROCESSING TIME first :
POLICY bx = LPT();
```

what left is creating the two sets of neighborhood structures and the defining the stopping condition value, we do so as follows :

```
// creating the first neighborhood structure set :
NEIGHBORHOOD_STRUCTURES N1 = neistructs(3);
N1[0] = s1;
N1[1] = s4;
N1[2] = s3;

// creating the second neighborhood structure set :
NEIGHBORHOOD_STRUCTURES N2 = neistructs(3);
N2[0] = s4;
N2[1] = s2;
N2[2] = s6;
```

running the GVNS algorithm on the given configurations :

```
// running GVNS :
int STOPPING_CONDITION = 20;
POLICY x = GVNS(bx, N1, N2, 3, 3, STOPPING_CONDITION);
```

3.2 Bench-Marking the GVNS implementation :

the code related to the benchmarking is found on the **evaluation.c** file in the given project, the repo is found in here https://github.com/MohamedHmini/VARIABLE-NEIGHBORHOOD-SEARCH.

let's consider the **c500t3** dataset, which is composed of **three drivers** and **500 commodities to be delivered**, the idea is to **evaluate** the algorithm so the definite way to go is **statistics**, we will use three operations **MIN**, **MAX** and **AVG** to benchmark our solution, these operations are going to be applied to **OBTIMAL VALUES** which have been found in each instance, and **EXECUTION TIME** of each instance.

let's remind ourselves of the configurations, we are testing on any dataset with a **First Improvement local search** with exactly **6 neighborhood structures** , for the **shake phase** we assign (3 block reversing based) whereas the **VND phase** is getting (3 block swapping based), the GVNS **stopping condition** is **15 failure points**.

- **Local Search :** First Improvement.
- **Neighborhood Structures :**
  - + **Block Swapping :** $blocksize \leftarrow 10$ and $blockfactor \leftarrow 0$
  - + **Block Swapping :** $blocksize \leftarrow 4$ and $blockfactor \leftarrow 30$
  - + **Block Swapping :** $blocksize \leftarrow 1$ and $blockfactor \leftarrow 100$
  - + **Block Reversing :** $blocksize \leftarrow 20$
  - + **Block Reversing :** $blocksize \leftarrow 30$
  - + **Block Reversing :** $blocksize \leftarrow 10$
- **GVNS Stopping condition :** 15 times failure to improve.
- **Running the GVNS for :** 20 times for each dataset.

	MIN	**MAX**	**AVG**
OPTIML VALUE	8946	8952	8948
EXECUTION TIME	28s	95s	47s

Table 3.1:
c500t3 dataset results

for **20** instances (running the algorithm for 20 times) of the same dataset we get the following results **table 3.1**, you can always check them yourself by reruning the given code yet it's not fully deterministic due to the reduced form of GVNS.

Conclusion

Meta-heuristics are the most recent development in approximate search methods for solving complex optimization problems, that arise in business, commerce, engineering, industry, and many other areas. A meta-heuristic guides a subordinate heuristic using concepts derived from artificial intelligence, biological, mathematical, natural and physical sciences to improve their performance. We shall present brief overviews for the most successful meta-heuristics. The paper concludes with future directions in this growing area of research.

Bibliography

[1] Peter Brucker [*Scheduling Algorithms*]

[2] Michel Gendreau · Jean-Yves Potvin [*Handbook of Metaheuristics*]