PROJECT : SOLVING A MULTI-MACHINES SCHEDULING PROBLEM USING A METAHEURISTIC APPROACH : VARIABLE NEIGHBORHOOD SEARCH

A PREPRINT

CHARHBILI Moad mcharhbili@insea.ac.ma

NAJAH Ismail inajah@insea.ac.ma

BOUGADRE Achraf abougadre@insea.ac.ma

KABAROUSEE Azzeddine akabarousse@insea.ac.ma

December 15, 2019

ABSTRACT

In this project, we will study a multi-machines scheduling problem, a factory that has **m** transport vehicles for delivering **n** commands, and one loading machine (a cargo) to load it into the **m** transport vehicles, this loading can't load more than one command in the same time, each command has a delivery time, and loading time, delivery time it's the time that the transport vehicle will take to deliver the command and get back to the factory, the loading time is the time the loading machine will take to load the command into a transport vehicle. Our objective is to minimize the time T the factory will take to deliver all the commands. As this problem is a NP-Hard problem, an exact method will not help as solving the problem, but we will use a meta heuristic method, The General Variable Neighborhood Search to get an approached solution in reasonable time.

***K*eywords** Meta-heuristics · Variable · Neighborhood · Search · Optimization

1 The Variable Neighborhood Search:

Variable neighborhood search combines local search strategies with dynamic neighborhood structures subject to the search progress. The local search is an intensification step focusing the search in the direction of high-quality solutions. Diversification is a result of changing neighborhoods. By changing neighborhoods, the method can easily escape from local optima. With an increasing cardinality of the neighborhoods, diversification gets stronger as the shaking steps can choose from a larger set of solutions and local search covers a larger area of the search space.

The intensification and diversification are the main components of any meta-heuristic algorithms.
Diversification means to generate diverse solutions so as to explore the search space on the global scale, while intensification means to focus on the search in a local region by exploiting the information that a current good solution is found in this region. This is in combination with the selection of the best solutions.

The variable neighborhood search is capable of finding a feasible solution for too large problems, which the exact methods couldn't extract the determinate solution at all or not in a reasonable time. The variable neighborhood search is based on the observation that a local minimum tend to clutter in one or more areas of the searching space. Therefore when a local optimum is found, one can get advantage of the information it contains, because the value of several variables might be equal or close to their values at the global optimum, it exploits systematically the following facts:

- **FACT 1 :** A local minimum with respect to one neighborhood structure is not necessarily so for another;

- **FACT 2 :** A global minimum is a local minimum with respect to all possible neighborhood structures.
- **FACT 3 :** For many problems local minimum with respect to one or several neighborhoods are relatively close to each other.

1.1 The different algorithms of the Variable Neighborhood Search:

1.1.1 The Basic Variable Neighborhood Search

The variable neighborhood search searches for a better solution at every iteration executed, its start by exploring (diversification) first the neighborhoods of its current solution, then it explores the more distant ones.

At first, the variable neighborhood search initialize a solution S_i in the first neighbor $N_n(S_i), n = 1$, and at every iteration, two steps are executed :

- Generate a neighbor solution of S_i, named $S_j \epsilon N_n(S_i), n = 1$.
- The application of a local search procedure on S_j, that leads to a new solution S_k, if S_k improves the current solution S_i, the searching procedure will restart from S_k using n=1, if it's not the case, the n is incremented and the procedure is repeated from S_i.

The algorithm stops after a certain number of times that the complete exploration sequence $N_1, N_2, ..., N_{kmax}$ is performed.

Algorithm 1: Basic variable neighbourhood search

```
initialization: find an initial solution x,k ← 1
repeat
    shake: generate a point x' at random from the neighbourhood N_k(x)
    local search: apply a local search procedure starting from the solution x to find a solution x
until k=k_max
if x" is better than x' then
    x ← x'' and k ← 1
    (centre the search around x" and search again with neighbourhood 1)
else
    k ← k + 1 (enlarge the neighbourhood)
end
```

1.1.2 The Basic Variable Neighborhood Descent

The basic variable neighborhood descent explore a set of neighborhoods, and search the best solution in every neighbor of the solution x, and take it as solution, if no solution found at the neighbor k, the algorithm will search in the next neighbor and so on, the stopping criteria in the basic VND is when there is no improvement obtained:

1.1.3 The General Variable Neighborhood Search

The variable neighborhood search (VNS) is a simple and effective meta-heuristic proposed by Mladenović and Hansen. It has the advantage of avoiding entrapment at a local optimum by systematically swapping neighborhood structures during the search. The basic VNS combines two search approaches: a stochastic approach in the shaking step that finds a random neighbor of the incumbent, and a deterministic approach which applies any type of local search algorithm. Serval VNS variants have been proposed and successively applied to numerous combinatorial optimization problems. Among which, the variable neighborhood descent (VND) as the best improved deterministic method is the most frequently used variant. The VND initiates from a feasible solution as the incumbent one and then carries out a series of neighborhood searches through operators $N_k(k = 1, ...,k_{max}$). If a better solution is found in a neighborhood, the incumbent solution is replaced by the better one and the search continues within the current neighborhood; otherwise it will explore the next neighborhood. The obtained solution at the end of the search is a local optimum with respect to all neighborhood structures. If the deterministic local search ap-

Algorithm 2: Basic variable neighbourhood descent

initialization: Select the set of neighborhood structures N_k, for $k = 1, .., k_{max}$ that will be used in the descent; find an initial solution x (or apply the rules to a given x);
repeat
  set $k \leftarrow 1$
  repeat
    Find the best neighbor x' of x ($x' \epsilon N_k(x)$);
    if *the solution x' is better than x* **then**
      set x $\leftarrow$ x' and k $\leftarrow 1$
    else
      set k $\leftarrow k + 1$
    end
  until $k = k_{max}$
until *no improvement is obtained*

proach of the basic VNS is replaced by the VND search, the resulted algorithm is called a general VNS (GVNS).

Algorithm 3: General Variable Neighbourhood Search

Initialize the parameters of the algorithm;
Define a set of neighborhood structures $N_k (k = 1, ..., k_{max})$, that will be used in the shaking phase and the local search phase;
Generate the initial solution x_0 by the EDD rule
Calculate the objective value $f(x_0)$ for the solution x_0 ;
Set the current best solution x = x_0 ;
Choose the stopping criterion; initialize the counter: $iter_1$ = 0, $iter_2$ = 0 and $iter_3$ = 0;
Set the first neighborhood structure for the shaking procedure to be k $\leftarrow 1$;
repeat
  Shaking: Generate a point x' at random in the N_k neighborhood of x;
  Produce a random sequence K of applying the k_{max} neighborhood structures;
  Local Search: Apply the VND scheme in the order specified by K, update x if a better local optimum is obtained;
  if $f(x') < f(x)$ **then**
    $x \leftarrow x'$
    reset the counter $iter_2 = 0$;
    reset the counter $iter_3 = 0$;
  else
    Increment the counters: $iter2 = iter2 + 1, iter3 = iter3 + 1$;
  end
  Update the counter of total number of iterations $iter1 = iter1 + 1$;
  if *x has not improved for a given number of iterations* $\lambda, thatis, wheniter_3 > \lambda, andifiter_2 < iter_{nip}$ **then**
    use the 3-opt perturbation procedure to generate a new starting solution x; and reset $iter_3 = 0$;
  Set k = k mod 5 + 1 to cycle through the neighborhood structures for shaking.
until *the stopping criterion is met, that is,* $iter_1 > Iter_{max}$ *or* $iter_2 > Iter_{nip}$

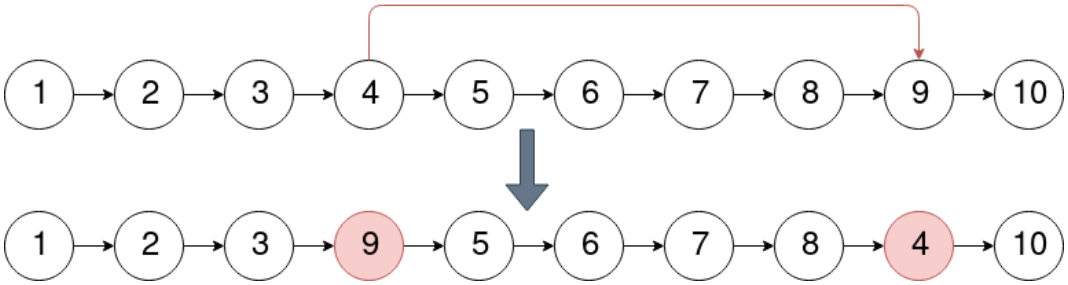
To see the implementation of this algorithm : go to 3.7 in page 12

1.2 The neighborhood structures

1.2.1 Swap structure

The Swap neighborhood contains the solutions that can be obtained by swapping two item of the solution's sequence.

Figure 1: Swap Structure

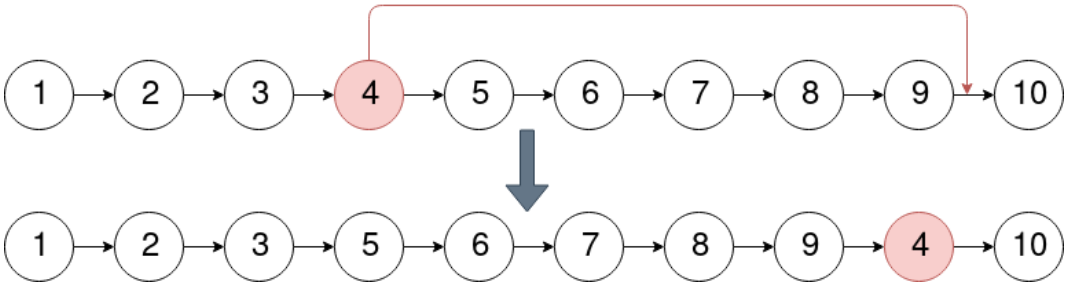


To see the implementation of this algorithm : go to 3.3.1 in page 7

1.2.2 Insertion structure

The Swap neighborhood contains the solutions that can be obtained by changing the position of an item of the sequence.

Figure 2: Insertion Structure

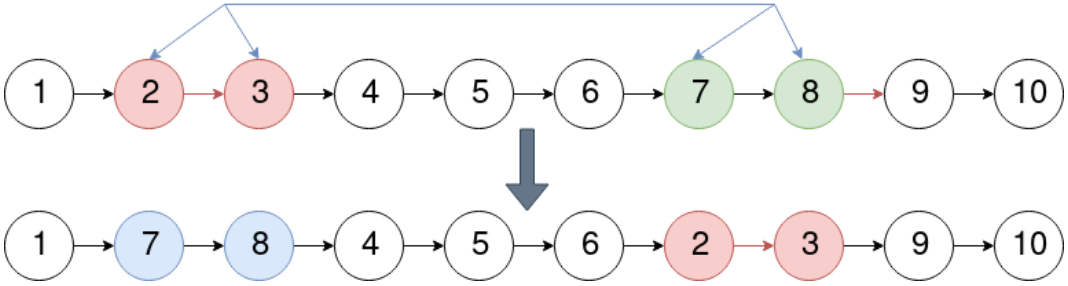


To see the implementation of this algorithm : go to 3.3.2 in page 8

1.2.3 Double Swap structure

The Double Swap neighborhood contains the solutions that can be obtained by swapping two successive items of the sequence by other two successive items in the sequence.

Figure 3: Double Swap Structure

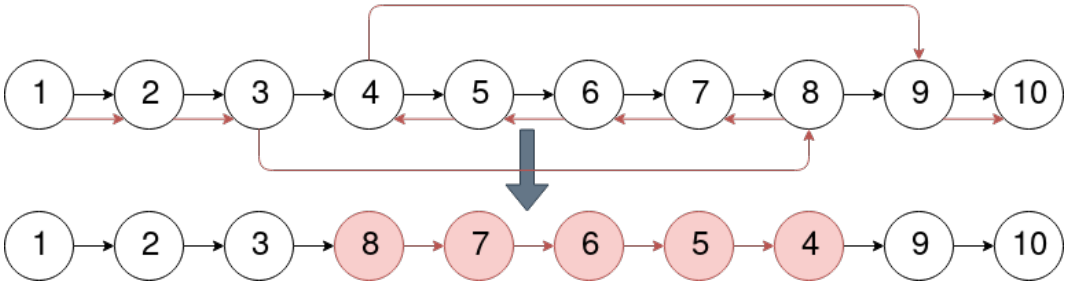


To see the implementation of this algorithm : go to 3.3.4 in page 9

1.2.4 2-opt structure

The 2-opt neighborhood contains the solutions that can be obtained by reversing the sub-sequence between two items of the sequence.

Figure 4: 2-opt Structure



To see the implementation of this algorithm : go to 3.3.3 in page 9

1.3 Local Search

The local search is the process of searching a feasible solution that improves our objective function in a neighborhood, the local search can be applied using two methods : the best improvement, or the first improvement:

1.3.1 Best improvement

consists of choosing an initial solution s, finding a direction of steepest descent from X, within a neighborhood N(x), and moving to the minimum of f(x) within N(x) along that direction; if there is no direction of descent, the heuristic stops, and otherwise it is iterated.

Algorithm 4: Best improvement

initialization: Choose f, X, neighborhood structure N(x), initial solution x;
repeat
  x' $\leftarrow x$
  Find x'=$argmin_{y \epsilon N(x)} f(y)$
  if *f(x')<f(x)* **then**
    set x $\leftarrow x'$
    iterate
  else
    stop;
  end
until *f(x')>f(x)*

1.3.2 First improvement

consists of choosing an initial solution s, finding the first solution that improves our objective function.

Algorithm 5: First improvement

initialization: Choose f, X, neighborhood structure N(x), initial solution x;
repeat
  (1) Find first solution x' ϵ N(x)
  (2)
  if *f(x')>f(x)* **then**
    find the next solution $x'' \epsilon N(x)$,
    set x' $\leftarrow x''$
    iterate (2)
  else
    set x $\leftarrow x'$
    iterate (1)
  end
until *All solutions of N(x) have been considered*

Since we're not using any constructive heuristic, (we're chosen a solution at random), in our implementation of the General Variable neighborhood search, in the local search we are using first improvement method for exploring the neighborhoods.
To see the implementation of this algorithm : go to 3.6 in page 11

2 Definition of the problem in hands

A factory have **m** transport vehicles, one loading machine (cargo) used to deliver **n** commands, the objective is to synchronize the work of those **m** transport vehicles and the loading machine in a way that all the time it takes to deliver all the **n** commands is minimum.
First of all we have to calculate the cost of delivering a bench of commands considering its order:
Lets consider:
p_j is the time that a transport vehicle will take to deliver la $command_j$.
l_j is the time that the loading machine will take to load la $command_j$ into a transport vehicle.
s_j the time the loading machine will be available after loading the $command_j$
M_i the time the transport vehicle i will be available.

$$s_j = l_j + max \begin{cases} min_{0 \leq i \leq m} M_i \\ s_{j-1} \end{cases} \tag{1}$$

$$M_{ik} = M_{ik-1} + p_j + l_j \tag{2}$$

$$Cost(J_j) = l_j + p_j + max(s_{j-1}, min(M_i)) \tag{3}$$

So, now we need to order the commands in an arrangement such that the cost of it is minimum, so our objective function is :

$$argminf(J)f(J) = Cost(J) \tag{4}$$

Algorithm 6: Cost Function

```
initialize J: an array of commands, Machine : array of size m, available : when the loading machine will be
  available
available ← 0
for i from 0 to size(J) do
    available = max(available,min(Machine)) + delivery_time[J[i]]
    Machine[index_of_minimum(Machine)] = available + loading_time[J[i]]
end
return max(Machine)
```

To see the implementation of this algorithm : go to 3.2 in page 7

3 Implementation of the General Variable Neighborhood Search

3.1 Data representation

3.1.1 Structure "Data"

We created a structure Data to store the loading time and delivery time for the commands in an instance.
We're defining also methods to manipulate the structure (memory allocation, free up memory, get instance's data...)

Listing 1: Structure Data

```
struct Data{
    int n;
    int m;
    int *duration;
```

```
    int *loading;
};
typedef struct Data* Data;

Data getData(FILE*);
void showData(Data);
void freeData(Data);
```

3.1.2 Structure "Sequence"

We created a structure "Sequence" to represent the order of the commands and store the cost of that sequence.

We're defining also methods to manipulate the structure (memory allocation, free up memory, get instance's data...)

Listing 2: Structure Sequence

```
struct Sequence{
    int size;
    int *values;
    int cost;
};
typedef struct Sequence* Sequence;

Sequence new_Sequence(int length);
void show_Sequence(Sequence sequence);
void initialize_Sequence(Sequence sequence,Data data);
void free_Sequence(Sequence);
void copy_sequence(Sequence source,Sequence cible);
```

3.2 Cost function:

The objective function is represented as follows :

Listing 3: Cost function

```
int cost(Sequence policy,Data data){
    Sequence m = new_Sequence(data->m);
    int cost = 0;
    int index;
    for(int i=0;i < policy->size; i++){
        index = index_of(m,min(m));
        cost = data->loading[ policy->values[i] ] + MAX(cost,m->values[index]);
        m->values[index] = cost + data->duration[ policy->values[i] ];
    }
    int return_value = max(m);
    free_Sequence(m);
    return return_value;
}
```

3.3 Neighborhood Structures

3.3.1 Swap structure

Listing 4: Swap structure

```
Sequence Swap(Sequence policy,Data data){
    int n=policy->size;
    Sequence temp=new_Sequence(n);
    copy_sequence(policy,temp);
```

```
    int temp1=-1;
    int temp2=-1;
    for(int k=0;k<n;k++){
        if(temp1>=0){
            swap(temp->values+temp1,temp->values+temp2);
        }
        int j=rand()%n;
        while(k==j)
            j=rand()%n;
        temp1=k;
        temp2=j;
        swap(temp->values+k,temp->values+j);
        temp->cost=cost(temp,data);
        temp->cost=cost(temp,data);
        if(temp->cost<policy->cost){
            copy_sequence(temp,policy);
            free_Sequence(temp);
            return policy;
        }
    }
    free_Sequence(temp);
    return policy;
}
```

3.3.2 Insertion structure

Listing 5: Insertion structure

```
Sequence Insertion(Sequence policy,Data data){
    int n=policy->size;
    Sequence temp=new_Sequence(n);
    copy_sequence(policy,temp);
    int temp1=-1;
    int temp2=-1;
    for (int i = 0; i < n-1; ++i) {
        if(temp1>=0){
            for(int k=temp2;k>temp2-temp1;k--){
                swap(temp->values+k,temp->values+k-1);
            }
        }
        int j=(i+1+rand())%n;
        while(i==j)
            j=(i+1+rand())%n;
        temp2=MAX(i,j);
        temp1=MIN(i,j);
        for(int k=temp1;k<temp2;k++){
            swap(temp->values+k,temp->values+k+1);
        }
        temp->cost=cost(temp,data);
        if(temp->cost<policy->cost){
            copy_sequence(temp,policy);
            free_Sequence(temp);
            return policy;
        }
    }
    free_Sequence(temp);
    return policy;
}
```

3.3.3 2_opt structure

Listing 6: 2_opt structure

```
Sequence Two_opt(Sequence policy,Data data){
    int n=policy->size;
    int e=0;
    Sequence temp=new_Sequence(n);
    copy_sequence(policy,temp);
    int temp1=-1;
    int temp2=-1;
    for (int k = 0; k < n-3; ++k) {
        if(temp1>=0){
            for(int i=temp1+1,j=temp2-1;i<j;i++,j--){
                swap(temp->values+i,temp->values+j);
            }
        }
        e = k+3+rand()%(n-k-3);
        temp1=k;
        temp2=e;
        for(int i=k+1,j=e-1;i<j;i++,j--){
            swap(temp->values+i,temp->values+j);
        }
        temp->cost=cost(temp,data);
        if(temp->cost<policy->cost){
            copy_sequence(temp,policy);
            free_Sequence(temp);
            return policy;
        }
    }
    free_Sequence(temp);
    return policy;
}
```

3.3.4 Double swap structure

Listing 7: Double swap structure

```
Sequence Two_swap(Sequence policy,Data data){
    int n=policy->size;
    Sequence temp=new_Sequence(n);
    copy_sequence(policy,temp);
    int temp1=-1;
    int temp2=-1;
    for(int k=0;k<n-3;k++){
        if(temp1>=0){
            swap(temp->values+temp2,temp->values+temp1);
            swap(temp->values+temp2+1,temp->values+temp1+1);
        }
        int j=k+1+rand()%(n-2-k);
        while(k==j)
            j=k+1+rand()%(n-2-k);
        temp1=k;
        temp2=j;
        swap(temp->values+temp2,temp->values+temp1);
        swap(temp->values+temp2+1,temp->values+temp1+1);
        temp->cost=cost(temp,data);
        if(temp->cost<policy->cost){
```

```
            copy_sequence(temp,policy);
            free_Sequence(temp);
            return policy;
        }}
    free_Sequence(temp);
    return policy;
}
```

3.4 Neighborhood feed:

Listing 8: Neighborhood function

```
Sequence Neighborhood(Sequence sequence,Data data,int k){
    if(k==0){
        return Swap(sequence,data);
    }
    if(k==1){
        return Insertion(sequence,data);
    }
    if(k==2){
        return Two_opt(sequence,data);
    }
    if(k==3){
        return Two_swap(sequence,data);
    }
}
```

3.5 Shaking phase

The shaking phase is applied by choosing a random sequence given by applying a specific neighborhood structure k.

Listing 9: Shaking function

```
void Shaking(Sequence policy,int k,Sequence out,Data data){
    if(k==0){
        Sequence temp=new_Sequence(policy->size);
        int i=rand()%policy->size;
        int j=rand()%policy->size;
        while(i==j)
            j=rand()%policy->size;
        copy_sequence(policy,temp);
        swap(temp->values+i,temp->values+j);
        temp->cost=cost(temp,data);
        copy_sequence(temp,out);
        free_Sequence(temp);
    }
    if(k==1){
        Sequence temp=new_Sequence(policy->size);
        copy_sequence(policy,temp);
        int i=rand()%policy->size;
        int j=(i+1+rand())%(policy->size);
        while(i==j)
            j=(i+1+rand())%(policy->size);
        int mx,mn;
        mx=MAX(i,j);
        mn=MIN(i,j);
        for(int u=mn;u<mx;u++){
            swap(temp->values+u,temp->values+u+1);
```

```
        }
        temp->cost=cost(temp,data);
        copy_sequence(temp,out);
        free_Sequence(temp);
    }
    if(k==2){
        Sequence temp=new_Sequence(policy->size);
        copy_sequence(policy,temp);
        int u=rand()%(policy->size - 4);
        int e = u+3+rand()%(policy->size - u - 3);
        for(int i=u+1,j=e-1;i<j;i++,j--){
            swap(temp->values+i,temp->values+j);
        }
        temp->cost=cost(temp,data);
        copy_sequence(temp,out);
        free_Sequence(temp);
    }
    if(k==3){
        Sequence temp=new_Sequence(policy->size);
        int i=rand()%(policy->size - 3);
        int j= i+1+rand()%(policy->size - 2 - i);
        while(i==j)
            j= i+1+rand()%(policy->size - 2 - i);
        copy_sequence(policy,temp);
        swap(temp->values+i,temp->values+j);
        swap(temp->values+i+1,temp->values+j+1);
        temp->cost=cost(temp,data);
        copy_sequence(temp,out);
        free_Sequence(temp);
    }
}
```

3.6 Local Search

Using first improvement method:

Listing 10: Local Search

```
Sequence LocalSearch(Sequence sequence,Data data,Sequence neighborhood){
    bool improvement=true;
    Sequence sequence1=new_Sequence(sequence->size);
    Sequence sequence2=new_Sequence(sequence->size);
    int kmax=neighborhood->size;
    while(improvement==true){
        int k=0;
        improvement = false;
        while(k<kmax && improvement==false){
            copy_sequence(sequence,sequence2);
            copy_sequence(Neighborhood(sequence2,data,neighborhood->values[k]),sequence1)
            if (sequence1->cost<sequence->cost){
                copy_sequence(sequence1,sequence);
                k=0;
                improvement=true;
            } else{
                k++;
            }
        }
    }
    free_Sequence(sequence1);
    free_Sequence(sequence2);
```

```
    return sequence;
}
```

3.7 General variable search algorithm

Listing 11: GVNS

```
Sequence GVNS(Sequence sequence,Data data,Sequence neighborhood,int iter_max,int iter_nip
    Sequence solution=new_Sequence(sequence->size);
    bool changed=false;
    Sequence sequence1=new_Sequence(sequence->size);
    int iter_1=0;
    int iter_2=0;
    int iter_3=0;
    int nip=(int)(0.5*iter_nip);
    int k=0;
    while(iter_1<iter_max && iter_2<iter_nip) {
        Shaking(sequence, neighborhood->values[k],sequence1,data);
        Shuffle(neighborhood,data);
        sequence1 = LocalSearch(sequence1, data, neighborhood);
        if (sequence1->cost < sequence->cost) {
            copy_sequence(sequence1, sequence);
            iter_2 = 0;
            iter_3 = 0;
        } else {
            iter_2++;
            iter_3++;
        }
        k=(k+1)%neighborhood->size;
        iter_1++;
        if(iter_2<iter_nip && iter_3>nip){
            if(changed){
                if(sequence->cost<solution->cost){
                    copy_sequence(sequence,solution);
                }
            }else{
                copy_sequence(sequence,solution);
            }
            Shaking(sequence,rand()%sequence->size,sequence,data);
            changed=true;
            iter_3=0;
        }
    }
    if(!changed){
        copy_sequence(sequence,solution);
    }
    free_Sequence(sequence1);
    return solution;
}
```

4 Test results

For the test of our algorithm, we run the program 5 times with the same max iterations and nip (non improvement) iterations, and we mark our results for the followed instances.

4.1 Instance of 500 commands and 3 transport vehicles

Max Iterations	NIP Iterations	AVG valeur	MIN valeur	MAX valeur	AVG time	MIN time	MAX time
100	20	8950.4	8948	8954	5.5609968	3.70935	7.818501
200	40	8948.8	8946	8950	6.0606086	4.465601	7.193939
300	60	8948.6	8947	8952	12.4105902	7.993017	15.885967
400	80	8946.88	8945	8948	15.6319656	9.470777	22.027352
500	100	8946.4	8946	8947	16.2271306	10.193229	26.084402

4.2 Instance of 1000 commands and 5 transport vehicles

Max Iterations	NIP Iterations	AVG valeur	MIN valeur	MAX valeur	AVG time	MIN time	MAX time
100	20	11135	11128	11148	49.350025	27.236761	66.970054
200	40	11127	11127	11133	100.1591378	83.77956	122.637944
300	60	11128.8	11122	11133	84.0406778	59.394823	124.016818
400	80	11125.6	11122	11129	165.6081154	59.394823	207.293305
500	100	11123.4	11116	11128	245.6552698	124.784401	383.051134

5 Conclusion

As it shown in the test of two instances, one with 500 commands and 3 transport vehicles, and other with 1000 commands and 5 transport vehicles, the higher the number of max iterations gets, the more accurate results will be.

For more large problems, the time will be an important factor, so the general variable neighborhood search algorithm will be a good choice, because as we learned from this work, its speed is much better than using an exact method such as dynamic programming .

References

[1] Beatriz Bernabé Loranca et al. "A statistical comparative analysis of simulated annealing and variable neighborhood search for the geographic clustering problem". In: (2011).

[2] Jason Brownlee. *Clever algorithms: nature-inspired programming recipes*. Jason Brownlee, 2011.

[3] Ke-Lin Du and MNS Swamy. "Search and optimization by metaheuristics". In: *Techniques and Algorithms Inspired by Nature; Birkhauser: Basel, Switzerland* (2016).

[4] Peng Guo, Wenming Chen, and Yi Wang. "A general variable neighborhood search for single-machine total tardiness scheduling problem with step-deteriorating jobs". In: *arXiv preprint arXiv:1301.7134* (2013).

[5] Aleksandar Ilić et al. "A general variable neighborhood search for solving the uncapacitated single allocation p-hub median problem". In: *European Journal of Operational Research* 206.2 (2010), pp. 289–300.

[6] Gokhan Kirlik and Ceyda Oguz. "A variable neighborhood search for minimizing total weighted tardiness with sequence dependent setup times on a single machine". In: *Computers & Operations Research* 39.7 (2012), pp. 1506–1520.

[7] Hongtao Lei, Gilbert Laporte, and Bo Guo. "A generalized variable neighborhood search heuristic for the capacitated vehicle routing problem with stochastic service times". In: *Top* 20.1 (2012), pp. 99–118.

[8] Nenad Mladenović, Dragan Urošević, Aleksandar Ilić, et al. "A general variable neighborhood search for the one-commodity pickup-and-delivery travelling salesman problem". In: *European Journal of Operational Research* 220.1 (2012), pp. 270–285.

[9] Angelo Sifaleras, Said Salhi, and Jack Brimberg. *Variable Neighborhood Search*. Vol. 11328. Springer, 2019.

[10] Xin-She Yang, Suash Deb, and Simon Fong. "Metaheuristic Algorithms: Optimal Balance of Intensification and Diversification". In: *Applied Mathematics Information Sciences* 8 (Aug. 2013). DOI: 10.12785/amis/080306.