# ACT

Pierre Lague & Romain Lecleire

Compte Rendu

TP4

**Abstract**

L'objectif de ce TP est la conception, l'implémentation et l'étude d'heuristiques pour un problème d'ordonnancement simple mais appartenant à la classe des problèmes NP-dur. Le langage de programmation utilisé est `Python 3`.

# Section 1: Travail Préliminaire

## Lecture des données d'instance

Pour pouvoir lire les données d'instance, nous proposons le programme suivant qui retourne un tableau de 4-uplets de la forme :

$$[(p_i, w_i, d_i, i)...]$$

- $p_i$ : le temps d'éxécution d'une tâche
- $w_i$ : le poids d'une tâche (son importance)
- $d_i$ : sa limite d'éxécution dans le temps
- $i$ : indice de la tâche $i$

```python
def readFile(filePath) :
    file = open(filePath, 'r')
    nb_data = int(file.readline())
    ordonnancement = []
    for i in range(nb_data) :
        data = file.readline().strip().split()
        ordonnancement.append((int(data[0]),int(data[1]),int(data[2]),i))
    file.close
    return ordonnancement
```

Ce programme retourne, par exemple, l'instance suivante :

```
[(1, 9, 2012, 0), (7, 10, 2134, 1), (3, 3, 2177, 2) ......] pour le fichier de test
↪ n100_15_b.txt
```

## Programme retournant la valeur $f(O)$

Notons que la fonction $f(O)$ est la fonction objectif que nous devons minimiser. Cette dernière calcule la somme totale des retards pondérés :

$$f(O) = \sum_{j=1}^{n} w_j * T_j$$

avec $T_j = max\{C_j - d_j, 0\}$, le retard de la tâche j.

Nous proposons le programme suivant pour calculer la somme totale des retards pondérés :

```
def retard_ordonnancement(ordonnancement) :
Cj = 0
retard_total = 0
for elt in ordonnancement :
    Cj += elt[0]
    Tj = max(Cj - elt[2],0)
    retard_total += Tj * elt[1]
return retard_total
```

Ce programme retourne, par exemple, un retard total de **285015** pour l'instance issue du fichier n100_15_b.txt (suivant une heuristique décrite en Section 2).

## Generation de solution aléatoire et évaluation de qualité

Afin de générer une solution aléatoire, et par mesure de simplicité, nous proposons un programme qui prend en entrée un nom de fichier de test dont l'instance sera mélangée de manière aléatoire, puis évaluée par la fonction précédente :

```
def random_heuristic(filePath) :
    ordonnancement = readFile(filePath)
    random.shuffle(ordonnancement)
    return retard_ordonnancement(ordonnancement), [elt[3] for elt in ordonnancement]
```

## Programme de calcul du pourcentage d'erreur

Le programme suivant nous permet de savoir notre pourcentage de d'erreur par rapport a la solution optimale qui nous a été fournie.

```
def calcul_pourcent_erreur(func,filepath) :
    res = func(filepath)
    opt = opt_res[filepath[13:19]]
    print("une erreur de : " + str(((res[0] - opt) / opt) * 100) + " %")
```

## Section 2: Heuristiques constructives et recherche locales simples

## Quelques heuristiques simples

Voici quelques heuristiques simples qui effectuent un tri simple sur les taches à effectuer.

La première fait un mélange aléatoire dans la liste des taches. Les résultats liés à cette heuristique peuvent être très bons, comme très mauvais.

```
def random_heuristic(filePath) :
    ordonnancement = readFile(filePath)
    random.shuffle(ordonnancement)
    return retard_ordonnancement(ordonnancement), [elt[3] for elt in ordonnancement])
```

La seconde trie la liste avec en priorisant les taches avec la date limite la plus proche puis les tâches avec le temps d'execution le plus court.

```
def naive_heuristic(filePath) :  # ordre date limite et temps execution
    ordonnancement = readFile(filePath)
    ordonnancement = sorted(ordonnancement, key= lambda x : (x[2], x[0]))
    return retard_ordonnancement(ordonnancement), [elt[3] for elt in ordonnancement]
```

La troisieme trie la liste des ttaches en utilisant le ratio de la date limite sur le temps d'execution.

```python
def ratio_heuristic(filePath) :
    ordonnancement = readFile(filePath)
    ordonnancement = sorted(ordonnancement, key= lambda x : x[2]/x[0], reverse=True)
    return retard_ordonnancement(ordonnancement), [elt[3] for elt in ordonnancement]
```

La quatrième heuristique effectue le même processus que la troisième tout en prenant en compte le poids de chaque tache.

```python
def ratio_heuristic_times_weight(filePath) :
    ordonnancement = readFile(filePath)
    ordonnancement = sorted(ordonnancement, key= lambda x : (x[2]/x[0])*x[1], reverse=True)
    return retard_ordonnancement(ordonnancement), [elt[3] for elt in ordonnancement]
```

A l'aide de ces heuristiques simples, nous avons pu déterminer que les poids d'une tache affecte nos résultats. Dans l'optique de nous rapprocher de la solution optimale, nous devrons donc éxécuter les taches de poids plus important avec un temps d'éxécution plus court.

Les heuristique suivantes reprennent l'ordre de rangement vu au dessus, mais elles éxécutent les taches qui sont deja en retard pour avoir une chance d'executer les taches qui ne sont pas encore en retard apres celle-ci. Ces methodes retrournent le retard final suivie de l'ordre d'execution des taches.

**Pour notre première heuristique** 1, nous proposons une heuristique naïve qui trie l'ordonnancement par la date limite et le temps d'éxécution :

**Algorithm 1** Constructive Next Not Retard Heuristic Naive

**function** ConstructiveNextNotRetardHeuristicNaive($filePath$)
  $retardTotal \leftarrow 0$
  $resOrdonnancement \leftarrow []$
  $alreadyRetard \leftarrow []$
  $ordonnancement \leftarrow \mathrm{readFile}(filePath)$
  $ordonnancement \leftarrow \mathrm{sorted}(ordonnancement, key = \lambda x : (x[2], x[0]))$ ▷ Naive sorting
  **for** $elt$ **in** $ordonnancement$ **do**
    **if** $elt[2] > retardTotal$ **then** ▷ Comparaison de la date limite de l'élément et du retard total
      $\mathrm{append}(resOrdonnancement, elt)$
      $retardTotal \leftarrow retardTotal + elt[0]$
    **else**
      $\mathrm{append}(alreadyRetard, elt)$
    **end if**
  **end for**
  $resOrdonnancement \leftarrow resOrdonnancement + alreadyRetard$
  **return** $\mathrm{retardOrdonnancement}(resOrdonnancement)$, [$elt[3]$ for $elt$ in $resOrdonnancement$]
**end function**

**Notre deuxième heuristique** 2 est un peu plus complexe, elle se base sur le ratio

$$\frac{d_i}{p_i}$$

. Nous allons donc évaluer une instance de problème triée par ordre décroissant de ce ratio.

**Notre troisième heuristique** 3 est en partie basée sur la seconde. Nous multiplions le ratio $\frac{d_i}{p_i}$ par $w_i$. Nous allons donc évaluer une instance de problème triée par ordre décroissant de cette multiplication.

**Notre quatrième heuristique** 4 est à caractère aléatoire. Au début, l'ordonnancement est mélangé de façon aléatoire.

**Algorithm 2** Constructive Next Not Retard Heuristic Ratio

```
function ConstructiveNextNotRetardHeuristicRatio(filePath)
    retardTotal ← 0
    resOrdonnancement ← []
    alreadyRetard ← []
    ordonnancement ← readFile(filePath)
    ordonnancement ← sorted(ordonnancement, key = λx : x[2]/x[0], reverse = True)    ▷ Ratio sorting
    for elt in ordonnancement do
        if elt[2] > retardTotal then    ▷ Comparaison de la date limite de l'élément et du retard total
            append(resOrdonnancement, elt)
            retardTotal ← retardTotal + elt[0]
        else
            append(alreadyRetard, elt)
        end if
    end for
    resOrdonnancement ← resOrdonnancement + alreadyRetard
    return retardOrdonnancement(resOrdonnancement), [elt[3] for elt in resOrdonnancement]
end function
```

**Algorithm 3** Constructive Next Not Retard Heuristic Ratio Times Weight

```
function ConstructiveNextNotRetardHeuristicRatioTimesWeight(filePath)
    retardTotal ← 0
    resOrdonnancement ← []
    alreadyRetard ← []
    ordonnancement ← readFile(filePath)
    ordonnancement ← sorted(ordonnancement, key = λx : (x[2]/x[0]) ∗ x[1], reverse = True)    ▷ Ratio times weight sorting
    for elt in ordonnancement do
        if elt[2] > retardTotal then
            append(resOrdonnancement, elt)
            retardTotal ← retardTotal + elt[0]
        else
            append(alreadyRetard, elt)
        end if
    end for
    resOrdonnancement ← resOrdonnancement + alreadyRetard
    return retardOrdonnancement(resOrdonnancement), [elt[3] for elt in resOrdonnancement]
end function
```

**Algorithm 4** Constructive Next Not Retard Heuristic Random

```
function ConstructiveNextNotRetardHeuristicRandom(filePath)
    retardTotal ← 0
    resOrdonnancement ← []
    alreadyRetard ← []
    ordonnancement ← readFile(filePath)
    shuffle(ordonnancement)    ▷ Random shuffling
    for elt in ordonnancement do
        if elt[2] > retardTotal then
            append(resOrdonnancement, elt)
            retardTotal ← retardTotal + elt[0]
        else
            append(alreadyRetard, elt)
        end if
    end for
    resOrdonnancement ← resOrdonnancement + alreadyRetard
    return retardOrdonnancement(resOrdonnancement), [elt[3] for elt in resOrdonnancement]
end function
```

# Résultats

| **Heuristic** | **Retard** |
|---|---|
| Random | 535434 |
| Naive | 471807 |
| **Ratio** | **285015** |
| Ratio*Weight | 215031 |
| **Constructive ratio*weight** | **233906** |
| Constructive ratio | 296546 |
| Constructive naive | 445331 |
| Constructive random | 454996 |

**Table 1:** Performance de nos heuristiques, résultats basé sur le fichier $n100_15_b.txt$

1. **Heuristique du Ratio:**
    - **Avantage :** Tente d'optimiser en fonction du rapport entre le retard et le temps d'exécution de la tâche.
    - **Considération :** Peut bien fonctionner lorsque les tâches ont des tailles variées.
2. **Heuristique du Ratio*Weight :**
    - **Avantage :** Optimise le rapport entre le retard et le temps d'exécution, tout en prenant en compte le poids des tâches.
    - **Considération :** Utile lorsque le poids des tâches est important dans la prise de décision.
3. **Heuristique Constructive Ratio*Weight :**
    - **Avantage :** Propose une approche constructive basée sur le Ratio*Weight, ce qui peut être efficace pour minimiser le retard total.
    - **Considération :** Peut être plus complexe que les heuristiques simples.
4. **Heuristique Constructive Ratio :**
    - **Avantage :** Approche constructive basée sur le Ratio, visant à minimiser le retard total.
    - **Considération :** Moins complexe que le Ratio*Weight, mais peut être moins précis dans certains scénarios.
5. **Heuristique Constructive Naïve :**
    - **Avantage :** Une approche simple mais constructive qui pourrait fonctionner dans certains cas.
    - **Considération :** Peut ne pas être aussi performante que des heuristiques plus sophistiquées.
6. **Heuristique Constructive Aléatoire:**
    - **Avantage :** Peut explorer différents ordonnancements de manière aléatoire.
    - **Considération :** Moins déterministe, les résultats peuvent varier d'une exécution à l'autre.

Les résultats du tableau nous indiquent que les heuristiques basées sur plusieurs facteurs comme le poids, la limite dans le temps ainsi que le temps d'éxécution proposent des performances plus intéréssantes que les heuristiques basiques.

# Hill Climbing

Pour la conception d'heuristiques par recherche locale simple, nous avons décidé d'implémenter l'algorithme du Hill Climbing. Pour rappel cet algorithme fonctionne de la manière suivante : **Algorithme HillClimbing :**

- 1. Evaluation de la configuration de base
- 2. Perturabation dans la configuration
- 3. Boucle dans le voisinnage tant que:
  - a. Une meilleure solution a été trouvée
    - 1. Appliquer cette meilleure configuration
  - b. Aucune solution a été trouvée

Nous proposons une implémentation qui utilise une fonction de "swapping" pour chercher des solutions voisines, et notre fonction objectif (retard total de l'ordonnancement) que nous allons vouloir minimiser :

```
def hill_climbing_get_better(ordonnancement,func): #peut d'amelioration
    continuer = True
    new_ord_list = []
    while continuer :
        new_ord_list = func(ordonnancement,retard_ordonnancement(ordonnancement))
        if retard_ordonnancement(ordonnancement) > retard_ordonnancement(new_ord_list[0]):
            ordonnancement = new_ord_list[0]
        else :
            continuer = False
    return ordonnancement
```

En utilisant la fonction de swapping (les voisins de plus faible retards sont positionnés au début de la liste), nous obtenons le résultat suivant : **176832**

```
def all_swap_get_better (ordonnancement,retard) :
    voisin = []
    for i in range(len(ordonnancement)) :
        for j in range(len(ordonnancement)):
            if(i != j):
                ord = deepcopy(ordonnancement)
                tmp = ord[i]
                ord[i] = ord[j]
                ord[j] = tmp
                voisin.append(ord)
    return sorted(voisin, key= lambda x : retard_ordonnancement(x))
```

En utilisant la fonction de swapping (le premier voisin qui améliore la solution est positionné au début de la liste), nous obtenons le résultat suivant : **174085**

```
def all_swap_get_first (ordonnancement,retard) :
    voisin = []
    for i in range(len(ordonnancement)) :
        for j in range(len(ordonnancement)):
            if(i != j):
                ord = deepcopy(ordonnancement)
                tmp = ord[i]
                ord[i] = ord[j]
                ord[j] = tmp
                voisin.append(ord)
    k = 0
    while k < len(voisin) and retard_ordonnancement(voisin[k]) >= retard:
        k += 1
    if k < len(voisin):
        voisin.insert(0,voisin.pop(k))
    return voisin
```

En utilisant la fonction d'inversion (le voisin qui à la plus faible retard est positionné au début de la liste), nous obtenons le résultat suivant : **195787**

```
def all_inversion_get_better(ordonnancement,retard) : #195787
    voisin = []
    for i in range(len(ordonnancement)) :
        ord = deepcopy(ordonnancement)
        if i >= len(ord) -1 :
            tmp = ord[i]
            ord[i] = ordonnancement[0]
            ord[0] = tmp
```

```
            voisin.append(ord)
        else :
            tmp = ord[i]
            ord[i] = ordonnancement[i+1]
            ord[i+1] = tmp
            voisin.append(ord)
    return sorted(voisin, key= lambda x : retard_ordonnancement(x))
```

En utilisant la fonction d'insertion (le voisin qui à la plus faible retard est positionné au début de la liste), nous obtenons le résultat suivant : **176290**

```
def all_insertion_get_better(ordonnancement,retard): #176290 #sur le fichier 2 : 408650
    voisin = []
    for i in range(len(ordonnancement)) :
        for j in range(len(ordonnancement)) :
            if i != j :
                ord = deepcopy(ordonnancement)
                ord.insert(j,ord.pop(i))
                voisin.append(ord)
    return sorted(voisin, key= lambda x : retard_ordonnancement(x))
```

Dans le cadre des algorithmes discutés précédemment, le résultat le plus favorable obtenu est de 174 085 avec l'algorithme 'all-swap-get-first', initialisé avec l'ordonnancement obtenu à partir du premier fichier trié selon l'heuristique "ratio-heuristic-times-weight". Il est à noter que la valeur optimale pour cette instance est de 172 995, entraînant une erreur marginale de 0,63 %.

Pour le deuxième fichier, les algorithmes décrits ci-dessus produisent une latence de 408 650 avec l'algorithme 'all-insertion-get-better', en utilisant toujours l'heuristique précédente comme ordonnancement initial. La valeur optimale pour cette instance étant de 407 703, l'erreur associée est de 0,23 %.

Dans ces deux cas, nous nous approchons du résultat optimal. Cependant, lors de l'exécution de ces algorithmes, nous observons que les dernières étapes d'amélioration sont minimes. Cela indique que nous nous trouvons dans un optimum local, et que la poursuite de notre algorithme ne permettra probablement pas d'obtenir des améliorations significatives..

## Probleme d'optimisation

Dans le dessein d'améliorer nos résultats, nous avons conçu un algorithme exploitant deux voisinages distincts. Bien que l'algorithme semble opérationnel, aucune amélioration significative n'a été observée après plusieurs heures d'exécution, même lorsque la recherche s'étend à un voisinage plus vaste, en recourant à une exploration approfondie après l'épuisement du voisinage simple.

Ces deux algorithmes autorisent une exploration dans un voisinage élargi. Afin de réduire le temps d'exécution, une approche parallèle utilisant des threads a été mise en œuvre. Malgré cela, l'algorithme demeure excessivement chronophage, dépassant la capacité de traitement de l'ordinateur et conduisant à des crashs avant la complétion de l'exécution de l'algorithme.

```
def all_swap_get_better_with_append (ordonnancement,retard,append_list) :
    for i in range(len(ordonnancement)) :
        for j in range(len(ordonnancement)):
            if(i != j):
                ord = deepcopy(ordonnancement)
                tmp = ord[i]
                ord[i] = ord[j]
                ord[j] = tmp
                append_list.append(ord)
```

```
def all_double_swap_get_better(ordonnancement,retard) : # trop grande complexiter # ne
 ↪ fonctionne pas sur mon pc
    processe = []
    new_ord_list = []
    new_ord_list2 = []
    new_ord_list = all_swap_get_better(ordonnancement,retard_ordonnancement(ordonnancement))
    for elt in new_ord_list :
        p = multiprocessing.Process(target=all_swap_get_better_with_append,args=(elt,
 ↪ retard_ordonnancement(elt),new_ord_list2))
        processe.append(p)
```

```
        p.start()
    for process in processe :
        process.join()
    return sorted(new_ord_list2, key= lambda x : retard_ordonnancement(x))
```

Pour diminuer davantage le temps d'execution, nous avons remarqué que la premiére étape de l'algorithme était la même que la dernière étape du voisinage de base. Nous avons donc combiné ces deux algorithmes dans l'optique de gagner du temps de calcul :

```
def HC_double_voisinage_version_simple(ordonnancement,func1,func2) :
    continuer = True
    new_ord_list = []
    while continuer :
        new_ord_list = func1(ordonnancement,retard_ordonnancement(ordonnancement))
        if retard_ordonnancement(ordonnancement) > retard_ordonnancement(new_ord_list[0]):
            ordonnancement = new_ord_list[0]
        else:
            new_ord_list = func2(new_ord_list[0],retard_ordonnancement(new_ord_list[0]))
            if retard_ordonnancement(ordonnancement) > retard_ordonnancement(new_ord_list[0]):
                ordonnancement = new_ord_list[0]
            else :
                continuer = False
    return ordonnancement
```

Ici le double voisinnage est effectué seulement sur la meilleure liste (pour gagner en ressources et en temps de calcul). Il prends environ 5 minutes à converger vers un optimum local et permet d'atteindre un score de **176832** (éxécuté seul).

# ILS - Iterative Local Search

Dans la suite du TP il nous est demandé d'implémenter au moins une ILS. Nous proposons une recherche locale itérée avec les caractéristiques suivantes :

- Initialisation : nous initialisons notre ILS avec un ordonnancement issu de notre meilleure heuristique basique (cf ratio*weight 1)
- Recherche de base locale : notre recherche de base est une recherche type HillClimbing nous permettant de tomber dans un optimal local.
- Perturbation : la perturbation consiste à permuter deux tâches de manière aléatoire dans notre ordonnancement. Suite à cela, nous relançons une recherche locale type HillClimbing sur l'ordonnancement perturbé pour espérer tomber dans un optimal global, ou local proche du global.
- Critère d'acceptation : Nous décidons de garder l'ordonnancement seulement s'il est meilleur que le précédent.
- Critère d'arrêt : Nous fixons un maximum d'itérations pour que le programme se termine.

L'algorithme est le suivant :

**Algorithm 5** ILS Algorithm

```
function ILS(ordonnancement, MAX_ITER = 10)
    new_ordo ← ratio_heuristic_times_weight(ordonnancement)
    hc_new_ordo ← hill_climbing_get_better(new_ordo, all_swap_get_better)
    for i ← 0 to MAX_ITER do
        # Perturbation
        random_int ← random.randint(0, len(ordonnancement))
        random_int2 ← random.randint(0, len(ordonnancement))
        ordonnancement[random_int], ordonnancement[random_int2] ← ordonnancement[random_int2], ordonnanceme
        # HillClimbing
        new_ordo_bis ← hill_climbing_get_better(ordonnancement, all_swap_get_better)
        # Evaluation et mise à jour si besoin
        if retard_ordonnancement(new_ordo_bis) < retard_ordonnancement(hc_new_ordo) then
            hc_new_ordo ← new_ordo_bis
        end if
    end for
    return hc_new_ordo
end function
```

Le nombre d'itérations est fixé à 10 afin de pallier au coût et à la durée substantiels de la recherche en Hill Climbing. Ainsi, pour obtenir des résultats significatifs avant la présentation finale, nous avons orienté notre attention vers les temps d'exécution. En utilisant le Hill Climbing à double voisinage, un score de **176 832** a été atteint en **152 secondes (2 minutes)**.

Par la suite, nous avons décidé d'initialiser la recherche avec le résultat d'une de nos meilleures heuristiques constructives (voir Constructive ratio*weight 1). Cela a généré un score de **176 796** en **148 secondes (2 minutes)** d'exécution. La différence de score entre les deux exécutions n'est pas significative, alors que la divergence de score entre les deux heuristiques (constructive ratio*weight et basique ratio*weight) est notable. Ceci suggère que notre perturbation pourrait ne pas être optimale.

Une nouvelle perturbation a été envisagée : une insertion (voir méthode `all_insertion_get_better` 2), qui a considérablement amélioré notre méthode de Hill Climbing. Cela a abouti à un score de **173 472** en **276 secondes ($\tilde{5}$ min)** d'exécution, présentant une amélioration marquée vers la solution optimale de **172 995**. À noter que cette configuration sur le deuxième fichier `n_100_16.txt` a produit un score de **408 625**, avec un optimal de **407 703**.

Une autre option explorée est la combinaison de deux voisinages différents dans notre Hill Climbing à double voisinage. Précédemment, nous utilisions le même voisinage à deux endroits distincts dans le Hill Climbing, d'où le terme double voisinage. Actuellement, nous avons fusionné les voisinages d'insertion et de swap dans notre ILS, obtenant le score suivant : **173 375** en **336 secondes ($\tilde{5}$ min)**, pour un optimal de **172 995**. Bien que la différence soit modeste, cette combinaison de voisinages nous permet néanmoins de nous rapprocher de la solution.

Nous avons également testé l'ILS avec un Hill Climbing à voisinage simple. En initiant à partir de notre heuristique constructive et en appliquant une perturbation basée sur une insertion, nous obtenons un score de **174 335** pour un optimal de **172 995**, avec **228 secondes ($\tilde{4}$ min)** d'exécution. Cela met en évidence l'intérêt du Hill Climbing à double voisinage optimisé en termes de score et de temps.

# Etude Expérimentale

Voici le tableau qui récapitule les résultats que nous obtenons sur une partie des fichiers d'instances.

**Table 2:** Data Table

| File | Optimal | Score | Portion (%) | Temps (s) |
|---|---|---|---|---|
| n_100_16.txt | 407703 | 408283 | 99.86 | 295.586 |
| n_100_17.txt | 332804 | 333756 | 99.72 | 291.232 |
| n_100_18.txt | 544838 | 544838 | 100.00 | 300.470 |
| n_100_19.txt | 477684 | 477800 | 99.98 | 311.493 |
| n_100_35.txt | 19114 | 19626 | 97.41 | 176.253 |
| n_100_36.txt | 108293 | 109314 | 99.06 | 252.428 |
| n_100_37.txt | 181850 | 183523 | 99.08 | 284.315 |
| n_100_38.txt | 90440 | 92576 | 97.64 | 265.968 |
| n_100_39.txt | 151701 | 154439 | 98.21 | 300.291 |
| n_100_40.txt | 129728 | 130028 | 99.77 | 312.176 |
| n_100_41.txt | 462324 | 462804 | 99.90 | 375.435 |
| n_100_42.txt | 425875 | 426173 | 99.93 | 373.596 |
| n_100_43.txt | 320537 | 321156 | 99.81 | 313.207 |
| n_100_44.txt | 360193 | 360364 | 99.95 | 324.914 |
| n_100_85.txt | 284 | 341 | 83.28 | 266.278 |

Au fil de nos expériences, nous n'avons atteint l'optimal que une seule fois (fichier n_100_18.txt), cependant, l'ensemble des scores indique que nous convergeons vers une solution proche de l'optimal, une solution de bonne qualité, dans un temps d'éxécution raisonnable étant donné le nombre d'opérations à réaliser. Le principe des heurisitiques est donc respecté.