

UTBM IT45

# Problème d'affectation d'employés

## **Auteurs**

Thomas Sirvent  
Clément Delteil

# Sommaire

<b>I) Analyse du problème</b>	<b>2</b>
Entités et représentation	2
Règles du problème	2
<b>II) Proposition d'algorithme de résolution : Algorithme génétique</b>	<b>3</b>
Codage de l'élément de population (individu)	3
Génération population initiale	3
Fonction d'évaluation	4
Mécanisme de sélection des individus	4
Opérateur de croisement et de mutation	5
Paramètres de dimensionnement de déroulement	5
<b>III) Implémentation de la solution proposée en langage C++</b>	<b>6</b>
Travail préparatoire	6
Définition des classes	6
Remplissage population et liste de formations	6
Fonction is_free()	7
Fonction greedyFirstSolution()	7
Fonction balancingPopulation()	7
Opérateur de croisement.	8
Sélection par tournoi	8
Opérateur de voisinage	8
Expérimentation	8
Lecture des données	8
Jeux de données	9
Difficultés rencontrées	9
Optimisation et analyse critique	10
<b>IV) Conclusion</b>	<b>11</b>

## I) Analyse du problème

### A) Entités et représentation

Nous avons plusieurs entités à considérer dans ce problème. D'abord les interfaces, qui sont définies par leurs compétences et leurs spécialités s'il y a. Cela nous permettra de déterminer quelles missions elles peuvent réaliser et donc, leur agenda.

Pour les apprentis, nous n'avons que leur ID et leurs coordonnées, mais c'est assez pour pouvoir les assigner à nos interfaces. On remarquera que nous n'utilisons pas les coordonnées des apprentis. Ces coordonnées à notre disposition donnent la position du centre SESSAD, d'un centre de formation ou d'un apprenti en format cartésien  $(x, y)$ . Les formations, représentées par une liste de caractéristiques présentés ci-dessous.

ID	Spé	Comp	Jour	HDébut	HFin
Numéro identifiant l'apprenti	Spécialité ou centre de formation de l'apprenti	Compétence requise des Interfaces pour effectuer la formation. Correspond à la déficience de l'apprenti	Jour de la formation	Heure de début de la formation	Heure de fin de la formation

On remarque que cette entité comprend des informations propres aux apprentis (en rouge), aux interfaces (vert) et communes aux deux (orange).

### B) Règles du problème

Pour satisfaire notre problème, il faut que tous les apprentis aient un intervenant assigné, de plus, il faudra remplir 2 contraintes essentielles :

- Une contrainte **horaire** : L'interface devra travailler 8 h par jour maximum, dans une amplitude horaire de 12 h, et si possible 1 h de pause le midi.
- Une contrainte de **compétence** : L'interface doit effectuer des missions compatibles avec ses compétences.

Nous avons aussi des contraintes d'optimisation qui serviront à décider de l'optimalité de notre solution courante.

- Contrainte de **distance** : L'interface devrait parcourir le moins de distance possible pour ses missions.
- Contraintes de **spécialité** : Si possible, l'interface devrait effectuer des missions reliées à sa spécialité (si elle en a une).

## II) Proposition d'algorithme de résolution : Algorithme génétique

### A) Codage de l'élément de population (individu)

Suite à votre mail, nous avons modifié le codage d'un individu.

Le principe reste le même, nous représentons toujours de façon réelle les jours et les heures des formations.

Interface 1 :
1: 10 - 12   13 - 15   15 - 17
2: 10 - 12   14 - 19
3: 9 - 11   14 - 19
4: 10 - 12
5: 10 - 12
6: 10 - 12   14 - 19

*Ici notre individu travaille le vendredi de 10 h à 12 h puis de 14 h à 19 h*

Nous avons maintenant une clé dans un dictionnaire qui représente le jour de la semaine suivi de l'emploi du temps associé. Le nombre de formations par jour est maintenant dynamique. La seule contrainte reste le nombre d'heures journalières et hebdomadaires.

En parallèle de cet emploi du temps, nous gardons en mémoire plusieurs autres informations concernant nos interfaces. ([voir III → A](#))

Dans notre projet, les interfaces seront donc nos individus, présents dans une population qui sera notre solution à améliorer.

### B) Génération population initiale

---

**Algorithm 1:** Greedy\_Premiere\_Solution

---

**Input:** Population vide

**Output:** Population complète

```
for formation in formations do
  for interface in interfaces do
    if competence(formation) is competence(interface) then
      if is_free(interface, formation) then
        interface.append(formation)
      end
    end
  end
end
end
```

---

Ici, la fonction : competence() renvoie la compétence d'une formation ou d'une interface.

La fonction is\_free() renvoie vrai si l'interface est disponible au créneau de formation proposé.

## C) Fonction d'évaluation

Notre solution initiale est complète, nous n'avons donc plus qu'à l'améliorer. Pour cela, nous allons utiliser la fonction objective fournie.

Pour une solution  $s$ , on cherchera à minimiser  $z$  tel que :

$$z = 0.5 * (moy_d(s) + ecart_d(s)) + 0.5 * f_{corr} * pénalité(s)$$

avec :

- $moy_d(s)$  = distance moyenne parcourue par les employés pour  $s$
- $ecart_d(s)$  = écart type des distances des employés pour  $s$
- $f_{corr}$  = facteur de corrélation = moyenne de toutes les distances =  $\frac{\sum d_{ij}}{nbr\ missions}$
- $pénalité(s)$  = nombre de spécialités non satisfaites

## D) Mécanisme de sélection des individus

Nous avons décidé initialement de partir sur une sélection en roulette, représentative de la sélection naturelle darwiniste. Cependant, le problème du superhéros, comme vu dans le cours, nous pose problème, car on risque d'avoir une très mauvaise densité de population qui, à terme, risque de rendre impossible l'amélioration de notre solution

Ce score sera exprimé par une fonction d'adaptation (**fitness**) similaire à celle détaillée dans la section précédente. Elle prendra en compte la distance parcourue par l'interface, jugée par rapport à la moyenne de la population et le nombre de missions correspondant à ses spécialités.

$$f(i) = 0.7 * spécialité_i + 0.3 * \frac{1}{|moy_d(s) - d_i|}$$

Voici notre premier jet de cette fonction, où :

- $spécialité_i$  = nombre de missions de l'interface  $i$  en lien avec l'une de ses spécialités.
- $moy_d(s)$  = distance moyenne parcourue par les employés pour  $s$
- $d_i$  = distance parcourue par l'interface

Nous pondérons en faveur du critère de spécialité plus que celui de la distance, car nous voulons être sûr que nos apprentis puissent avoir le meilleur suivi possible. Après évaluation, nous répartissons les individus dans notre roulette : les individus avec les plus hauts scores auront les meilleures probabilités d'être choisis et inversement.

## E) Opérateur de croisement et de mutation

Suite à vos remarques par mail, nous avons modifié le codage de nos individus et de ce fait les opérateurs de croisement et de mutation ont, eux aussi, été vus modifiés.

Nos individus n'ont plus un seul créneau le matin et l'après-midi. Ils ont maintenant autant de créneaux que l'on veut tant qu'ils respectent les contraintes imposées à une interface. Ainsi, les croisements se feront toujours sur un créneau spécifique pour chacune des deux interfaces. Cependant, ils ne se feront plus forcément uniquement le matin avec le matin ou l'après-midi avec l'après-midi.

Par ailleurs, dans notre dossier d'analyse conception, nous avons écrit qu'il était difficile d'imaginer un opérateur de mutation étant donné que celui-ci pourrait entraîner la non-validité d'une solution. Or, nous nous sommes rendus compte que pour des nombres assez importants de formations à assigner, notre heuristique initiale pouvait ne pas toutes les affecter.

Par conséquent, nous entendrons maintenant par opérateur de mutation, un opérateur qui cherchera, à chaque génération à affecter les formations qui ne sont pas encore assignées. En effet, on peut imaginer qu'en fonction de l'ordre d'assignation initial de celles-ci, il se peut qu'après plusieurs croisements on puisse trouver une place dans l'emploi du temps d'une interface.

## F) Paramètres de dimensionnement de déroulement

La taille de notre population varie entre chaque instance du problème. Notre population sera de taille  $n$  où  $n$  représente le nombre d'interfaces. L'opérateur de croisement sera appliqué avec une probabilité  $P_c$  (autour de 0,6).

Concernant les critères d'arrêt, le nombre de générations variera en fonction du temps limite d'exécution et notre algorithme pourra être arrêté lorsque la population n'évoluera plus ou tout du moins plus suffisamment rapidement.

### III) Implémentation de la solution proposée en langage C++

#### A) Travail préparatoire

##### a) Définition des classes

Pour le besoin de notre solution, nous avons implémenté deux classes. La première représentant une formation, la seconde représentant une interface.

Formation	Interface
<ul style="list-style-type: none"><li>-id : Int</li><li>-day : Int</li><li>-startHour : Int</li><li>-endHour : Int</li><li>-comp : Int</li><li>-indexSpec : Int</li><li>-position : vector&lt;float&gt;</li></ul> <p>+displayFormation() : String</p>	<ul style="list-style-type: none"><li>-distance : Float</li><li>-fitness : Float</li><li>-hoursWorked : Int</li><li>-competence : Int *</li><li>-specialty : Int *</li><li>-currentPosition : vector&lt;float&gt;</li><li>-assignedMissions : vector&lt;float&gt;</li><li>-hoursWorkedPerDay : vector&lt;float&gt;</li><li>-time_table: map &lt; int, vector&lt;Formation*&gt; &gt;</li></ul> <p>+displayInterface() : String +displayTimeTable() : String +getPenalty() : Int +getInterfaceEvaluation() : Float</p>

##### b) Remplissage population et liste de formations

Avant la recherche d'une solution, nous remplissons systématiquement notre population et notre liste de formations en fonction de l'instance choisie.

Notre population est définie comme cela :

```
Interface *starting_population[NBR_INTERFACES];
```

Pour chaque Interface, nous lui assignons sa compétence, sa spécialité ainsi que sa position actuelle

Notre liste de formations est définie comme cela :

```
Formation *formations_list[NBR_FORMATIONS];
```

Pour chaque Formation, nous lui assignons son id, son index de spécialité, sa compétence, son jour, son heure début, son heure de fin et sa position.

#### c) Fonction `is_free()`

Cette fonction est l'une des plus importantes. En effet, elle conditionne la validité de notre solution.

Elle prend en paramètre l'index d'une formation, l'index d'une interface ainsi que la population actuelle et renvoie une paire `<bool,int>`.

Elle renvoie `<true, index>` si l'interface est disponible aux horaires de la formation où l'index correspond à l'emplacement de la formation dans l'emploi du temps de l'interface. En renvoyant l'index directement ici, on évite un nouveau parcours pour déterminer où l'on doit insérer la formation dans l'emploi du temps.

Cette fonction vérifie d'abord si l'amplitude horaire journalière ainsi que le quota horaire hebdomadaire ne sont pas dépassés avec la nouvelle formation. Auquel cas on regarde d'abord si on peut la placer au début de la journée ou à la fin. Si ce n'est pas le cas, alors on parcourt l'emploi du temps du temps pour trouver une place.

#### d) Fonction `greedyFirstSolution()`

L'algorithme proposé au sein du dossier d'analyse-conception est toujours le même.

Nous pouvons néanmoins faire quelques observations. Pour certaines générations d'instance, notre heuristique ne parvient pas à assigner toutes les formations. Notre solution n'est donc pas forcément complète. Autre observation, prévisible, mais qui orientera nos futurs choix. Les premières interfaces travaillent beaucoup plus que les dernières étant donné que ce sont toujours sur elles, dans un premier temps, que l'on tente d'assigner les formations.

#### e) Fonction `balancingPopulation()`

Pour répondre au problème de l'équilibrage de la population et réduire le nombre de générations de l'algorithme génétique nous avons mis en place une fonction d'équilibrage.

Celle-ci va déterminer l'interface qui travaille le plus, celle qui travaille le moins et va tenter d'envoyer un créneau vers la moins occupée. De cette manière, nous allons éviter d'avoir des interfaces qui travaillent 34 heures par semaine alors que d'autres travaillent seulement 2 heures par semaine.

Nous avons une fonction annexe qui nous renvoie le jour où l'interface travaille le plus. Ainsi nous allons parcourir cette liste de formations et tenter un croisement pour chacune d'elle avec l'interface la moins occupée.



#### f) Opérateur de croisement.

Étant donné que nous avons déjà la fonction `is_free` ([voir III->A->c](#)) qui conditionne le croisement, l'opérateur est seulement là pour modifier les éléments des interfaces.

Pour chaque croisement nous mettons donc à jour :

- la liste des missions assignées
- le nombre d'heures travaillées par semaine
- le nombre d'heures travaillées au jour du croisement
- l'emploi du temps au jour concerné

#### g) Sélection par tournoi

L'idée initiale de la sélection par roulette s'est finalement avérée incompatible avec notre projet. En effet, la sélection par roulette possède le défaut de mettre en avant un et un seul individu d'une population et donc, de faire beaucoup de croisement à partir de cet unique "super héros".

La sélection par tournoi nous donne l'avantage d'avoir plus de diversité dans notre population. D'autant plus que le tournoi est la seule solution technique applicable dans notre cas, car nous devons garder à l'esprit que le maximum de formations doivent être assignées afin d'avoir une solution convenable.

#### h) Opérateur de voisinage

Dans une optique d'amélioration de notre solution, nous procédons de la manière suivante :

Nous séparons d'abord notre population en 2 sous - ensembles (nommés "pool" dans notre projet), selon leurs compétences. Cela sert à faciliter la sélection et le croisement de 2 interfaces, car nous serons certains de leur compatibilité sur ce critère.

Dans un deuxième temps, nous parcourons une de ces pools et nous recherchons les 2 meilleures interfaces présentes . Cependant, comme dit plus tôt, il faut assigner le maximum de formations possible, ainsi, nous faisons en sorte que chaque sélection (sur la même génération) ne ressorte jamais la même paire. Cela nous permet de conserver les formations assignées préalablement, car nous serons certains qu'aucune interface sera "perdue" et ses formations avec.

N.B : Cela implique que les interfaces sont sélectionnées deux à deux. Dans les cas où la pool est de taille impaire, nous gardons juste la dernière interface dans la prochaine génération sans la modifier. Elle sera réévaluée la prochaine fois, et selon les croisements, pourrait être sélectionnée.

## B) Expérimentation

### a) Lecture des données

Pour la génération d'instances, nous utilisons la dernière version du générateur d'instance disponible sur Moodle.

Nous modifions dans notre main le fichier à inclure en fonction des besoins. Nous demandons cependant à l'utilisateur le temps de recherche souhaité pour optimiser la solution.

### b) Jeux de données

Voici quelques résultats obtenus avec des générations différentes :

Instance 96 :

```
Eval of starting pop is 4073.1  
Eval of balanced pop is 4317.64  
Final pop eval is : 4197.11
```

Instance 72 :

```
Eval of starting pop is 2547.92  
Eval of balanced pop is 2632.93  
Final pop eval is : 2620.24
```

Instance 60 :

```
Eval of starting pop is 1795.96  
Eval of balanced pop is 1747.21  
Final pop eval is : 1704.95
```

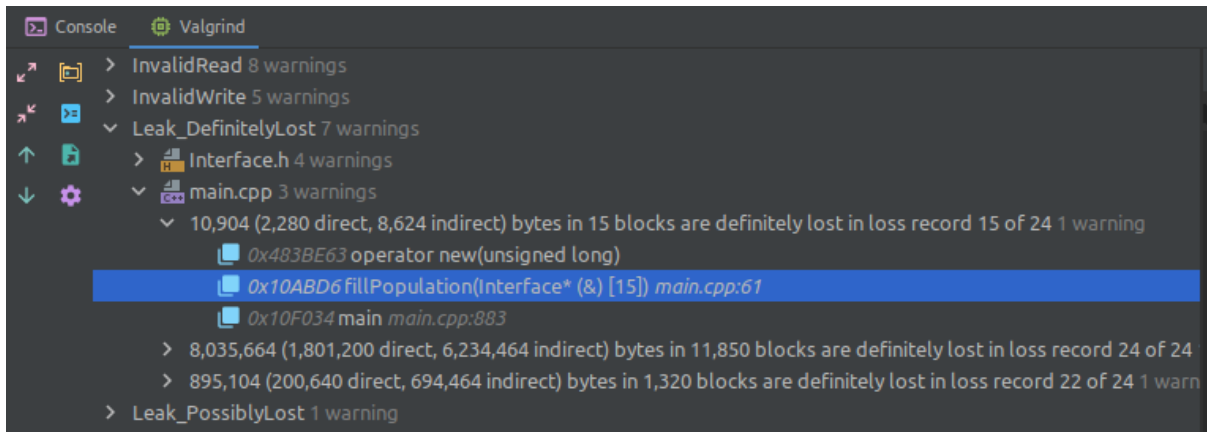
Vous pouvez remarquer que la population équilibrée n'a pas toujours une meilleure évaluation que la population initiale. En effet, malgré l'équilibrage des heures, nous ne prenons pas en compte, à ce moment-là, les spécialités des interfaces.

Pour ce qui est de l'évaluation finale avec l'algorithme génétique. Ici l'amélioration de la solution est systématique. Mais pour l'instance 96 par exemple, après l'équilibrage, nous ne retrouvons pas une meilleure solution que la population initiale.

### c) Difficultés rencontrées

Quelques fois, lorsque nous lançons notre programme en mode run, celui-ci se termine subitement à cause d'une segmentation fault. Tandis qu'en mode debug celui-ci fonctionnait toujours correctement. On appelle cela un [Heisenbug](#). Pour le résoudre, nous avons tenté d'utiliser Valgrind qui est un outil de programmation libre pour déboguer, effectuer du profilage de code et mettre en évidence des fuites mémoires.

Après exécution de celui-ci, plusieurs fuites ont été détectées :



```

> InvalidRead 8 warnings
> InvalidWrite 5 warnings
> Leak_DefinitelyLost 7 warnings
> Interface.h 4 warnings
> main.cpp 3 warnings
  > 10,904 (2,280 direct, 8,624 indirect) bytes in 15 blocks are definitely lost in loss record 15 of 24 1 warning
    0x483BE63 operator new(unsigned long)
    0x10ABD6 fillPopulation(Interface* (&) [15]) main.cpp:61
    0x10F034 main main.cpp:883
  > 8,035,664 (1,801,200 direct, 6,234,464 indirect) bytes in 11,850 blocks are definitely lost in loss record 24 of 24
  > 895,104 (200,640 direct, 694,464 indirect) bytes in 1,320 blocks are definitely lost in loss record 22 of 24 1 warn
> Leak_PossiblyLost 1 warning
```

Les plus importantes fuites ont été corrigées, mais certaines restent difficiles à corriger / interpréter à notre niveau.

Finalement le bug a été corrigé en retrouvant le cas qui n'était pas géré par nos fonctions.

### d) Optimisation et analyse critique

Notre opérateur de voisinage n'est sûrement pas le plus adapté. En effet, nous avons passé beaucoup de temps à corriger nos erreurs de mémoire dans notre programme et nous n'avons pas eu assez de temps pour essayer d'autres solutions. Dans l'état actuel des choses que l'on laisse 5 ou 40 secondes à l'utilisateur, la solution n'évoluera plus très rapidement.

En effet, comme notre opérateur de voisinage sélectionne deux à deux les meilleures interfaces d'une compétence donnée. Si au bout d'un moment on ne trouve pas de croisement à effectuer entre ces paires. Leur fitness ne va plus évoluer et ainsi nous allons toujours sélectionner les mêmes, etc. C'est un cercle sans fin.

C'est pour cela qu'avec plus de temps nous aurions pu tester d'autres idées pour la sélection des interfaces à croiser.

## IV) Conclusion

Concernant le projet, notre heuristique de solution initiale nous semble avoir répondu correctement au problème posé, mais l'optimisation de celle-ci via l'algorithme génétique s'est révélée plus difficile que prévu.

Si ce type de projet était à refaire, nous aurions peut-être choisi une autre métaheuristique pour répondre à notre problématique ou du moins, nous aurions plus approfondi la partie du voisinage. Nous aurions aussi adopté une approche plus rigoureuse envers les problématiques techniques de l'algorithme (complexité temporelle et spatiale).

Il a été dit en cours que l'opérateur du voisinage était bien souvent le plus difficile à trouver ou à régler. Nous avons réellement compris le sens de cette phrase dans la phase finale du projet, où nous avons réalisé que celui-ci n'était peut-être pas le plus adapté pour l'optimisation de notre solution.

Au niveau de notre développement personnel, ce projet fut très formateur. En effet, c'est la première fois que nous avons pu traiter un problème concret avec des outils vus en cours. De plus, il nous a permis de comprendre la construction de l'algorithme génétique, qui est quelque chose dont on entend parler de temps à autre.

De cette manière, nous ressortons de ce projet avec le sentiment de s'être rapproché une fois de plus vers le monde réel/professionnel actuel dans lequel nous allons nous lancer à la fin de nos études.