Livrable - Challenge IT45

BLANCKAERT Martin - DUPAYRAT Antoine

A. Conception initiale

Une fois les règles du challenge bien étudiées et appréhendées, nous avons conçu un plan de résolution. Nous avons privilégié la résolution par une façon **approchée**, car nous avons estimé que résoudre ce problème de façon exacte aurait une complexité trop élevée et serait trop long. Notre résolution fonctionne donc selon le principe suivant : l'obtention d'une **solution initiale**, puis dans un second temps **l'amélioration** de cette dernière.

I. Codage du problème

Afin de mener à bien la résolution et de faciliter l'accès aux informations et la compréhension du code, nous avons créé deux structures, une pour les interfaces et une pour les solutions.

Interface

La structure interface regroupe 6 éléments. Elle contient :

- la compétence
- les spécialités
- l'agenda. C'est un tableau à deux dimensions de taille [6][13], avec le premier index correspondant au jour, et le deuxième correspondant à l'heure du jour (allant de 0 à 13, représentant les heures de 6 à 19).
- l'agenda seul ne permettant pas d'identifier les apprenants suivis par l'interface, un tableau dynamique s'en charge.
- la distance totale parcourue
- le nombre de pénalités générées

Solution

La structure solution regroupe également 6 éléments. Elle contient :

- une liste de la même taille que le nombre d'interfaces, contenant toutes les structures d'interface
- la distance moyenne
- l'écart-type
- le facteur de corrélation
- le z
- le nombre de pénalités

De plus, on peut également citer la structure **Node**, permettant la création de l'arbre. Il contient simplement la structure solution, ainsi que son enfant gauche et son enfant droit.

II. Solution initiale

find_init_solution

L'algorithme permettant de trouver une solution initiale se concentre principalement sur l'affectation d'une interface à un maximum de créneaux.

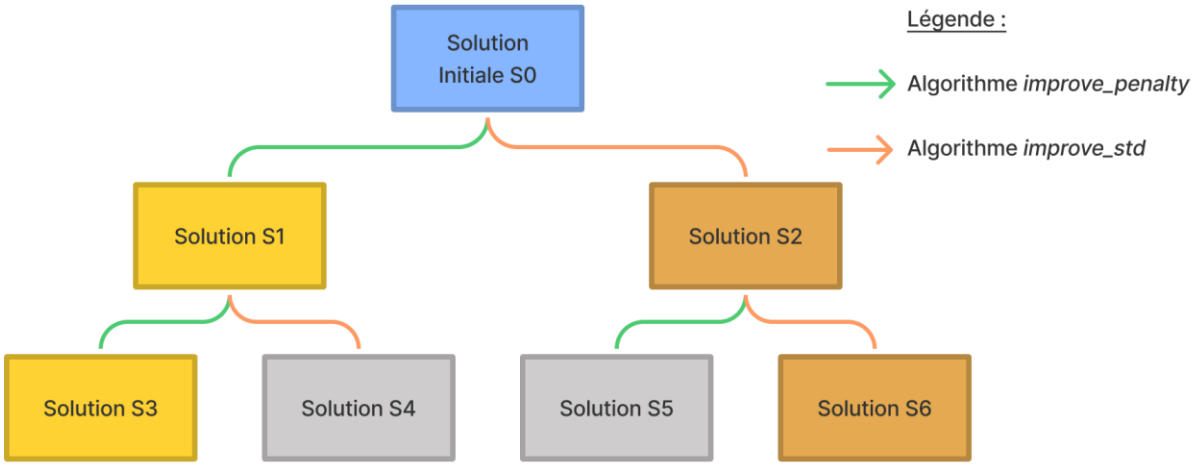
De plus, afin de garantir une complexité relativement faible, les deux tris sur les listes sont effectués à l'aide de l'algorithme *qsort* de la librairie standard de C. C'est un algorithme de tri de type QuickSort, soit de complexité moyenne **n.log(n)**, c'est-à-dire que c'est un des algorithmes de tri les plus rapides existant. Cependant, dans le pire scénario, sa complexité est quadratique.

III. Diversification et amélioration de la solution

improve_penalty
improve_std

Une fois cette solution initiale obtenue, nous essayons donc de l'améliorer. Les pistes d'améliorations sont nombreuses, notamment sur le plan de **l'homogénéisation**. En effet, *find_init_solution* maximisant le nombre de créneaux effectués par l'interface en tête de liste, cette dernière effectue beaucoup plus de trajets (et donc de distance) que les interfaces situées en fin de liste, potentiellement inutilisées. Cela résulte en un **écart-type très conséquent** entre employés, ce qui fait augmenter la fonction objectif.
Une seconde piste d'amélioration probante est le **travail sur les pénalités**. En effet, *find_init_solution* ne se penche pas au cas par cas sur les spécialités des interfaces, et donc par conséquent ajoute beaucoup de pénalités à la fonction objectif.

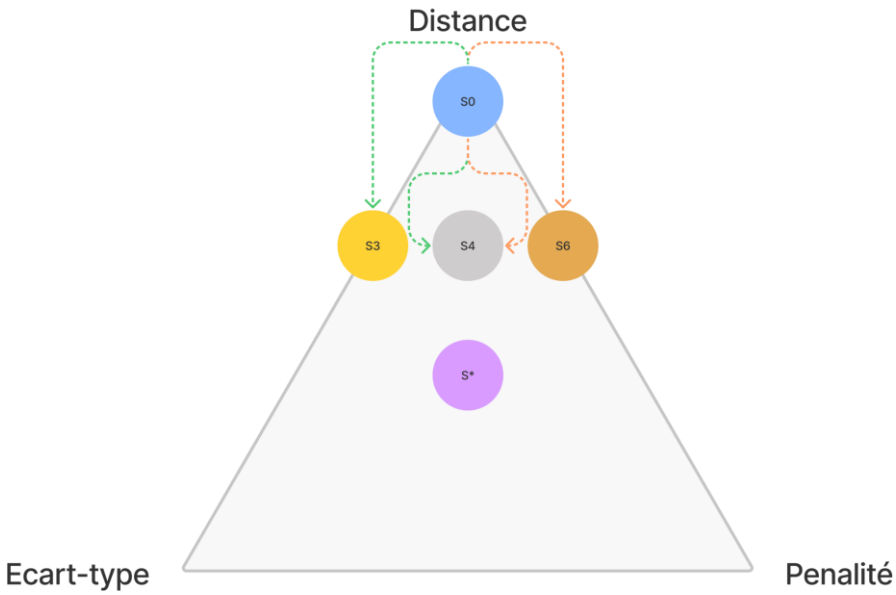
Nous avons donc créé deux algorithmes, le premier - *improve_std* - travaillant exclusivement sur l'écart-type, et le second - *improve_penalty* - sur les pénalités. Notre solution initiale ayant une distance totale optimisée (le minimum d'interfaces étant utilisées, on maximise les créneaux en une journée et limite donc les retours au centre SESSAD), nous allons ensuite appliquer ces deux algorithmes à la solution initiale, comme représenté ci-dessous.



Une fois un nombre fixé n d'itérations effectuées, on obtient une population de taille 2^n, qui contient des solutions diverses :

- celles tout à gauche (comme S3), qui ont une distance optimisée et des pénalités réduites.
- celles tout à droite (comme S6), qui ont une distance optimisée et un écart-type faible.
- celles qui ont subi les deux algorithmes (comme S4, S5), et qui ont été modifiées de façon à tout optimiser.

On a donc dans cette population des solutions, plus ou moins optimisées sur trois critères (cf. graphe ci-dessous).

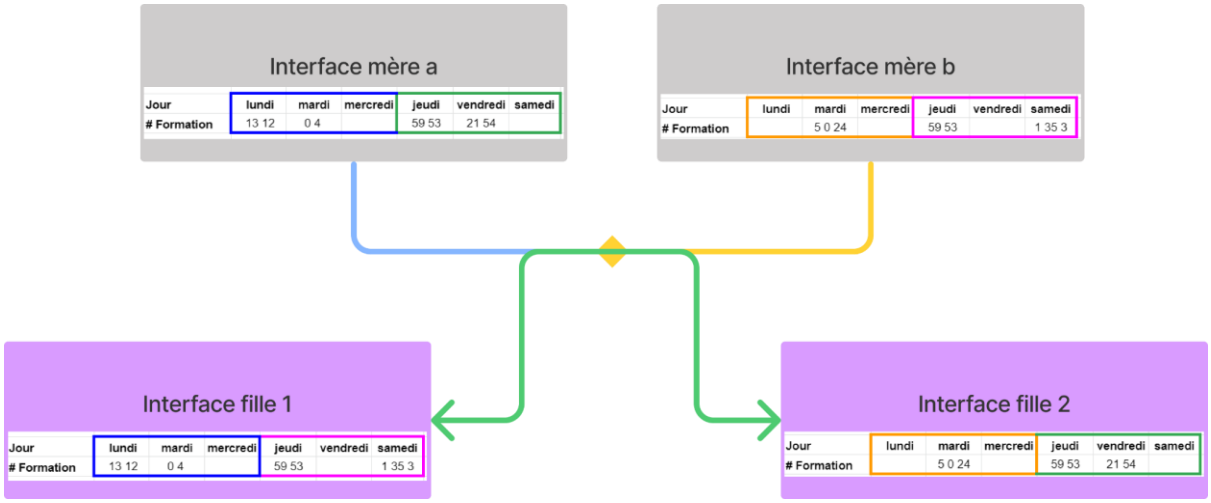


IV. Métaheuristiques

Au moment de la conception initiale de la méthode de résolution, nous souhaitions ensuite réutiliser la population de solutions et la coupler avec une métaheuristique afin d'améliorer la

solution. Nous nous sommes penchés sur deux façons de faire pour nous aider dans cette tâche : un algorithme 2-opt avec recherche tabou ou bien un algorithme génétique.

Une fois le projet plus avancé, nous avons conservé la piste de **l'algorithme génétique** afin d'améliorer notre solution initiale, donnée par l'heuristique énoncée précédemment. Cet algorithme est très basique : en puisant dans la population initiale, il croise les individus deux à deux selon des critères dépendants de la fonction objectif. Ici, nous effectuons les croisements sur les individus se ressemblant le moins dans la population initiale, selon l'opérateur de croisement suivant :
Chaque solution de la population étant désordonnée, nous choisissons un index aléatoire qui définit deux interfaces différentes chez les deux parents. Pour les deux interfaces choisies, nous allons prendre les formations de la première moitié de semaine chez la solution mère a, et la deuxième moitié de semaine chez la solution mère b. Pour chaque solution, nous avons donc un total de 3 journées qui changent pour une seule interface (cf. illustration ci-dessous).



Par conséquent, cela entraîne souvent des incompatibilités d'emplois du temps, et nous devons "réparer" la solution selon la méthode suivante :

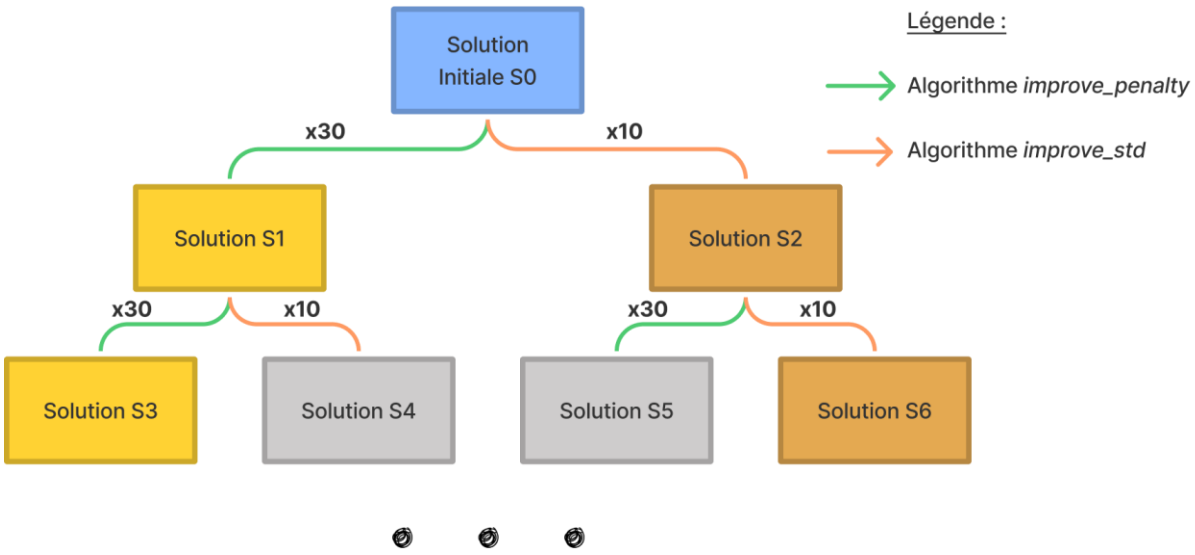
- Nous gardons en mémoire les formations impactées par le croisement, afin de vérifier qu'elles soient bien toutes présentes après le croisement
- Ensuite, nous vérifions que les formations ajoutées ne génèrent pas de doublons dans la solution.
- Si un doublon est détecté, il est supprimé et si la solution est irréparable, on garde la solution mère initiale comme solution fille.

B. Modifications

Outre des petites modifications sur des éléments de codage ou bien des façons d'accéder à des variables, notre conception a **relativement peu changé.**

Cependant, il y a une mécanique que nous n'avions pas anticipé et qui a beaucoup changé les choses, et qui intervient lors de l'application des heuristiques *improve_std* et

improve_penalty. En effet, une fois ces fonctions codées et fonctionnelles, nous nous sommes rendus compte que l'heuristique sur les pénalités réduisait beaucoup plus le *z* que celle sur l'écart-type. Cela nous a fait reconsidérer l'énoncé, et nous avons donc remarqué que la fonction objectif donnée dans l'énoncé pénalise beaucoup les pénalités, ce que nous n'avions pas vu dans notre première analyse du sujet. Afin de pallier cela, nous avons modifié le mode de fonctionnement de cette deuxième phase, afin de la rendre plus modulable et adaptée à n'importe quelle fonction objective. Au lieu de simplement créer deux enfants à chaque solution et d'effectuer une fois la fonction *improve_std* sur l'un et *improve_penalty* sur l'autre, nous **pondérons** le nombre de fois où l'on effectue la fonction. Par exemple, pour la fonction objectif donnée dans le sujet, nous avons déduit expérimentalement que, afin d'optimiser l'obtention d'une meilleure solution, il fallait effectuer à chaque profondeur 30 fois la fonction *improve_penalty*, et 10 fois *improve_std* (cf. illustration ci-dessous et annexe I). Cela permet d'obtenir des meilleures solutions **plus rapidement** dans l'arbre, et ainsi d'éviter de devoir faire un arbre de profondeur 60 avec énormément de solutions quasi-inchangées.



Il y a également un élément que nous avions sous-estimé : la performance de nos deux heuristiques d'amélioration selon les pénalités et l'écart-type, surtout avec les itérations modulables. En effet, une fois ces deux fonctions débuggées, sur notre jeu d'instances, le fait de faire passer notre solution initiale par ces deux algorithmes plusieurs fois divisait le z par **6**.
Bien que la surperformance de nos heuristiques soit plutôt un bon problème, cela a malheureusement eu des mauvaises conséquences sur la suite. En effet,
les solutions dans la population initiale ont des *z* assez éparpillés, avec des solutions excellentes comme nous l'avons vu dans le paragraphe précédent, et d'autres un peu moins bonnes. Cependant, cela endigue la performance de notre métaheuristique, qui peine à trouver une meilleure solution que celle générée à la fin de la deuxième phase pour des instances **petites ou moyennes**.

C. Résultats et performance

I. Les heuristiques brutes

Nos deux heuristiques ayant été conçues pour la fonction objectif donnée, elles sont très performantes pour cette dernière, mais peuvent vite être obsolètes si on modifie la fonction objectif. C'est pour cela que nous avons créé le système de pondération de l'arbre de la deuxième phase, afin de **généraliser** le fonctionnement de notre conception. Cependant, pour cette fonction objectif ci, notre algorithme est très performant car il donne un vaste panel comportant beaucoup de combinaisons possibles d'amélioration de l'écart-type, les pénalités, ainsi que la distance moyenne et le facteur de corrélation. Bien que nous ayons choisi de travailler uniquement avec le dernier étage de l'arbre pour générer notre population soumise à l'algorithme génétique, nous aurions également pu récolter *n* individus répartis dans l'arbre selon certains critères et selon la fonction objectif.

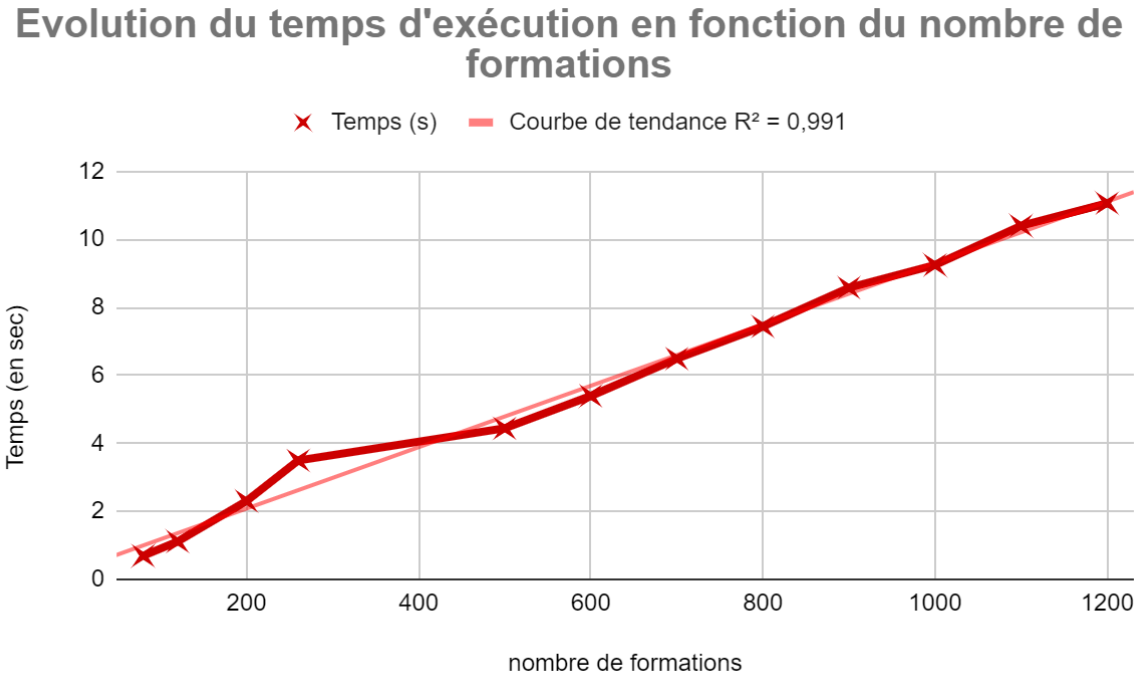
II. La métaheuristique - l'algorithme génétique

L'ajout de l'algorithme génétique a un impact relativement faible pour les problèmes à petits et moyens jeux de données, non seulement dû à la performance de nos heuristiques brutes mais également à la complexité de sa mise en place avec les problèmes de compatibilité qui amenuisent ses améliorations potentielles. De plus, il faudrait étudier la situation plus précisément afin de trouver un opérateur de croisement plus adapté, qui générerait peut être moins de problèmes de compatibilité. En effet, notre opérateur actuel ne change pas assez les solutions et entraîne très rapidement des stagnations de notre population d'une itération à l'autre. De plus, cet algorithme génétique coûte en temps d'exécution : la réparation des créneaux est gourmande en termes de calculs. Cependant, pour des plus gros problèmes, l'impact de l'algorithme génétique est **non négligeable** et apporte encore une optimisation supplémentaire.

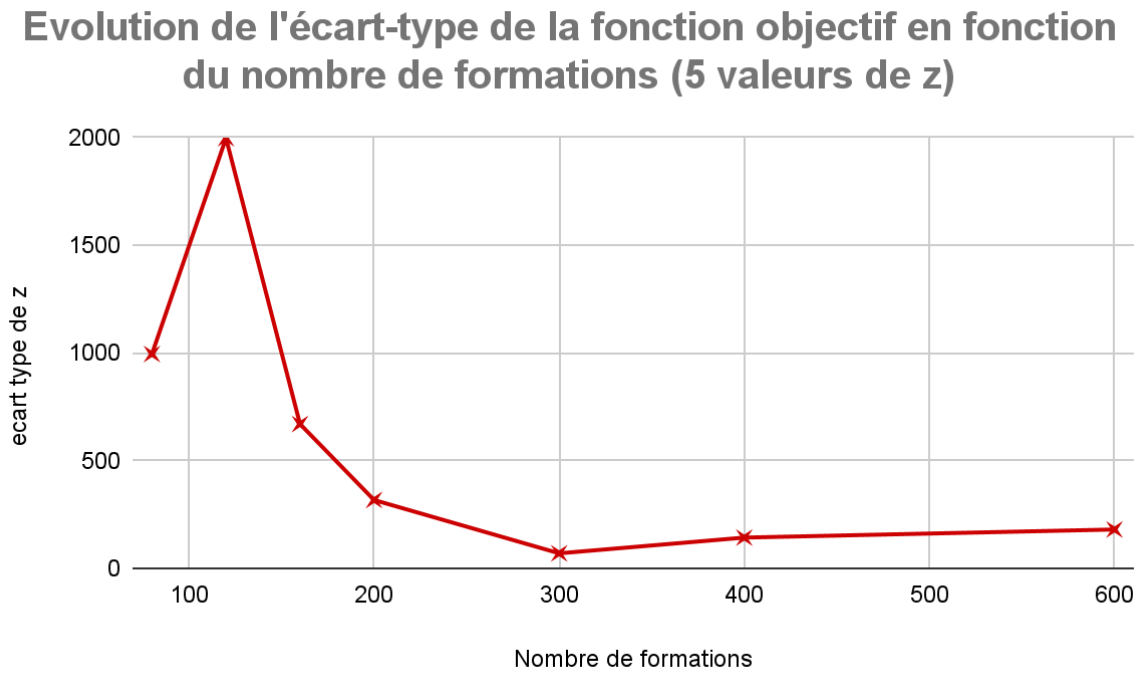
En somme, notre méthode est très optimisée pour le problème tel que défini dans l'énoncé, et peut l'être pour d'autres fonctions objectif à condition de paramétrer adéquatement à la fois la métaheuristique, mais surtout les heuristiques brutes. Notre méthode permet non seulement d'obtenir des solutions admissibles correctes en toutes circonstances, mais également d'obtenir un vaste panel de solutions, dans laquelle se situe probablement la solution optimale, ou bien une solution s'en rapprochant fort le cas échéant.

III. Statistiques et exécution

Nous avons effectué de nombreux tests avec des instances différentes, que ce soit en termes du nombre d'interfaces, de formations, etc. **Quel que soit le jeu de données fourni, notre algorithme a su trouver une solution initiale** (sauf dans le cas où il n'existait pas de solutions), et ensuite l'améliorer.
De plus, comme on peut le voir sur le graphique ci-dessous avec la courbe de tendance, même pour des gros jeux de données (les constantes sont disponibles en annexe 2), le temps d'exécution reste **linéaire**.



Comme on peut le voir avec le pic sur le graphe pour 100 formations, on remarque également que le résultat de la fonction objectif varie plus pour des petits jeux de données que pour des gros. Cela s'explique par le fait que, pour les gros jeux, il y a beaucoup d'interfaces ce qui facilite l'homogénéisation : on obtient presque tout le temps des solutions avec **aucune pénalité**, alors que ce n'est pas le cas pour les petits jeux. En effet, en fonction de l'aléatoire des formations, l'algorithme a moins de marge de manœuvre pour arranger les créneaux comme il le souhaite.



D. Problèmes rencontrés

Les premiers problèmes que nous avons rencontrés, assez tôt dans la conception de notre système de résolution, sont des problèmes relatifs à la représentation du problème, et surtout au **codage** d'une interface et d'une solution. Les choses imaginées dans la phase d'analyse du problème n'étaient pas forcément implémentables en C, et d'autres n'étaient pas optimisées pour ce langage et nous ont causé des problèmes d'implémentation. Nous avions également commencé à coder notre méthode de résolution sans les structures Interface et Solution dont nous avons parlé, et cela rendait le code brouillon et incompréhensible. Nous avons donc dû remanier ce dernier, causant quelques problèmes supplémentaires.

Le principal problème que nous avons rencontré au cours de ce projet sont cependant les **problèmes de mémoire**. Cela nous a cependant permis de découvrir un nouvel outil, Valgrind, qui est un outil de *memcheck* permettant de vérifier les accès en lecture et en écriture et de vérifier les fuites de mémoire. Cela nous a beaucoup aidé à débugger le code ainsi que pallier des problèmes que l'on ne voyait même pas à l'œil nu, et ainsi certifier que notre programme ne possède aucune fuite de mémoire.

E. Annexes

I. *Fonction improve_solution*

```
1   //Fonction d'amélioration de notre solution initiale par une heuristique classique
2   void improve_solution(Arbre *head, int depth) {
3
4       if(depth == 0)
5           return;
6
7       //Creer deux copies de la solution sol que l'on ajoute à l'arbre binaire
8       add_child(head, *((*head)->solution), 0);
9       add_child(head, *((*head)->solution), 1);
10
11      //Amélioration du fils gauche au niveau de l'écart type
12      for(int i = 0; i < INFLUENCE_STANDARD; i++)
13          improve_standard_error((*head)->leftchild->solution);
14
15      //Amélioration du fils droit au niveau du nombre de pénalités
16      for(int j = 0; j < INFLUENCE_PENALTY; j++)
17          improve_penalties((*head)->rightchild->solution);
18
19      //Amélioration des fils gauches et droits
20      improve_solution(&((*head)->leftchild), depth-1);
21      improve_solution(&((*head)->rightchild), depth-1);
22  }
```

On peut voir ici les paramètres *INFLUENCE_STANDARD* et *INFLUENCE_PENALTY*, qui pondèrent les heuristiques brutes et sont modulables relativement à la fonction objectif.

II. Paramètres utilisés pour les statistiques

NBR_APPRENANTS = x * 20
NBR_INTERFACES = (NBR_APPRENANTS/4 * 1.2);
NBR_FORMATIONS = NBR_APPRENANTS * NBR_FORMATIONS_PAR_SEMAINE;

NBR_FORMATIONS_PAR_SEMAINE = 1
DIMENSION_ZONE_GEOGRAPHIQUE = 200
NBR_CENTRES_FORMATIONS = NBR_SPECIALITES = 5
NBR_COMPETENCES = 2

III. Pseudo-code find_init_solution

- Remplir une liste avec tous les créneaux à pourvoir, en la triant dans l'ordre décroissant (créneaux dans la liste du plus long au moins long)
- Remplir une liste avec toutes les interfaces (en mettant celles ayant la double compétence en dernier et celles avec de multiples spécialités en premier)
- Tant que (liste de créneaux n'est pas vide **et** liste d'interfaces n'est pas vide)
    - Pour chaque créneau :
        - Si l'interface :
            - a la compétence requise (LPC ou signes)
            - n'a rien de prévu pendant le créneau à l'étude
            - a une journée de moins de 8h de travail
            - a une journée de moins de 12h d'amplitude
            - a une semaine de moins de 35h de travail
        - Alors on lui assigne le créneau étudié et on le retire de la liste
        - Sinon on passe créneau d'après
    - Fin Pour
    - On passe à l'interface d'après
- Fin tant que
- Si la liste de créneaux est vide alors on a une solution
- Sinon :
    - Tant que la solution de convient pas
        - Effectuer le 2-opt sur la solution
- Fin

IV. Pseudo-code improve_std

- Soient interface_min, interface_max, min = +∞, max = -∞
- Pour chaque interface :
    - Si la distance parcourue par l'interface est inférieure à min :
        - min ← la distance parcourue par l'interface

        - interface_min = i
    - Sinon, si la distance parcourue par l'interface est supérieure à max :
        - max ← la distance parcourue par l'interface
        - interface_max = i
    - Fin si
- Fin pour
- Tant que la distance parcourue par l'interface_max est supérieure à la distance parcourue par l'interface_min :
    - un des créneaux effectué par l'interface_max va à l'interface min (tout en vérifiant sa compatibilité)
- Fin

V. Pseudo-code improve_penalty

- Trier les interfaces par nombre de pénalités récoltées
- Sélectionner la première interface
- Pour chaque créneau c ne respectant pas la spécialité de la première interface :
    - Pour chaque interface i (à partir de la deuxième dans l'ordre trié) :
        - Si i a la spécialité de c :
            - Attribuer c à i
        - Fin si
    - Fin pour
- Fin