Université de technologie de
Belfort-Montbéliard

Challenge d'IT45 printemps 2022
Rapport

Problèmes d'Affectation, de Planification et de Routage des tournées des Employés

Thibault CHAUSSON
Léo VIGUIER

Responsable : Mira BOU SALEH
Professeur : Olivier GRUNDER

14 juin 2022

Résumé

Dans le but de réaliser ce projet, nous avons mis en œuvre les connaissances acquises en recherche opérationnelle tout au long de ce semestre. Nous allons essentiellement utiliser des algorithmes d'optimisation non exacte, plus précisément un algorithme génétique.

Tout au long de ce challenge, nous nous sommes efforcés à réaliser un programme permettant de générer des plannings pour un SESSAD, et ainsi attribuer un assistant spécialisé (une interface) pour chaque membre d'un de ces centres.

Fort de notre réflexion qui a été exposée sur le document d'analyse et conception, nous avons développé en Python un programme permettant d'afficher le planning de chaque intervenant en essayant de satisfaire le plus de contraintes possibles.

Table des matières

1 Présentation **1**

2 Génération de la population initiale **2**

2.1 Méthode de génération des solutions aléatoires 2

2.2 Méthode de génération semi-aléatoire 2

2.2.1 Temps d'exécution 2

3 Mise en place des différents objectifs **5**

3.1 Fitness employés 5

3.2 Fitness étudiants 6

3.3 Fitness SESSAD 7

4 Algorithme génétique **8**

4.1 Explication des différentes méthodes 9

4.1.1 Algorithme génétique à une fitness 12

4.1.2 Algorithme génétique en cascade 13

4.1.3 Algorithme génétique en moyenne 13

4.1.4 Algorithme génétique avec le front de Pareto 14

4.1.5 Éviter les blocages 15

4.2 Temps d'exécution, Instance « 45-4 » 16

4.2.1 Génétique classique 16

4.2.2 Génétique en cascade 16

4.2.3 Génétique en cascade par fitness 17

4.2.4 Génétique en moyenne normalisée 18

4.2.5 Génétique avec front de Pareto 19

4.3 Évolution de la fitness 20

4.3.1 Génétique classique 20

4.3.2 Génétique en cascade 22

4.3.3 Génétique en cascade par fitness 23

4.3.4 Génétique en moyenne normalisée 24

4.3.5 Génétique avec front de Pareto 25

5 Conclusion **27**

Annexes **i**

Table des images **ii**

Table des tableaux **iii**

Liste des codes sources **iv**

1 Présentation

Pour réaliser un programme pouvant résoudre ce problème de génération de planning, nous nous sommes dirigés vers le langage de programmation Python. Ce dernier permet de développer rapidement et de gérer de manière très intuitive les tableaux, les matrices, de plus il nous permet de visualiser assez facilement la performance de nos programmes en faisant des graphiques.

Vous pouvez exécuter facilement ce programme en suivant les quelques instructions suivantes.

L'installation des composants nécessaires au programme se fait de la manière suivante : L'installation requiert python version 3.0 ou supérieure d'installé sur la machine. Si vous ne l'avez pas, vous pouvez télécharger Python 3.8 en cliquant ce lien, ou à partir du Microsoft Store. Lors de l'installation à partir du site internet, veillez à bien cocher la case « Add Python 3.8 to PATH », sans quoi la prochaine étape n'aboutira pas. Sur Linux, Python 3 devrait déjà être installé mais voici la commande à exécuter dans le cas contraire :

```
sudo apt install python3 python3-pip
```

Sur Windows, exécutez le script install.bat, puis install_librairies.bat dans un nouveau terminal et sur Linux, le script install.sh. Sur Linux, si jamais pip3 n'est pas détecté sur votre machine, un prompt s'affichera pour vous demander si vous voulez l'installer vous-même, dans le cas d'un oui, la commande suivante sera exécutée :

```
sudo apt install python3-pip
```

Une fois finis, vous pouvez lancer le programme de la sorte :

— Windows :

```
py main.py -h
```

— Linux :

```
python3 main.py -h
```

2 Génération de la population initiale

2.1 Méthode de génération des solutions aléatoires

Le principe est le suivant : nous avons une matrice identité de la taille du nombre d'intervenants, car celle-ci fournira des listes avec $N_{intervenants} - 1$ 0 et un 1, qui permet donc d'assigner une mission à un et un seul intervenant, permettant donc de valider la contrainte 4. Aussi, en récupérant les compétences de chaque intervenant, on peut aussi attribuer une mission à un intervenant ayant la même compétence, validant ainsi 2ème contrainte. Enfin, l'algorithme assigne une mission après l'autre, validant finalement la 1ère contrainte. Cette méthode n'est pas 100% aléatoire, mais elle est bien plus efficace qu'un algorithme qui mettrait $N_{missions}$ 1 aléatoirement dans une matrice, car sa conception permet de valider 3 contraintes qui sont très restrictives. L'avantage de cette méthode est aussi qu'elle génère une population très diversifiée au niveau des gènes. Le sérieux problème que rencontre cet algorithme, c'est la vitesse. En effet, il a fallu 7 heures d'exécution pour générer 80 solutions uniques avec les instances du dossier 45-4, ce qui n'est pas du tout viable.

2.2 Méthode de génération semi-aléatoire

Le principe est le suivant : on choisit une mission aléatoirement qui sera attribuée à un intervenant aléatoire. Ce couple intervenant, mission est inchangeable tout au long de la création de la solution, c'est lui qui rend une solution unique. Ensuite, on parcourt chaque lignes et colonnes de notre matrice pour tester si en mettant un 1 à telle coordonnées, la solution serait toujours valide, sinon on laisse un 0. En cas de blocage (mission non attribuée par exemple), on abandonne cette solution et on recommence le processus. Cet algorithme est très efficace, car il permet de générer 100 solutions en une dizaine de secondes. Cependant, la population générée par cet algorithme risque de ne pas être très diversifiée car l'algorithme suis toujours le même ordre d'attribution.

2.2.1 Temps d'exécution

Après avoir expliqué les principes sous-jacents de ce programme, regardons son efficacité.

Nous allons réaliser quelque teste pour voir de façon graphique l'évolution du temps d'exécution pour générer une population initiale.

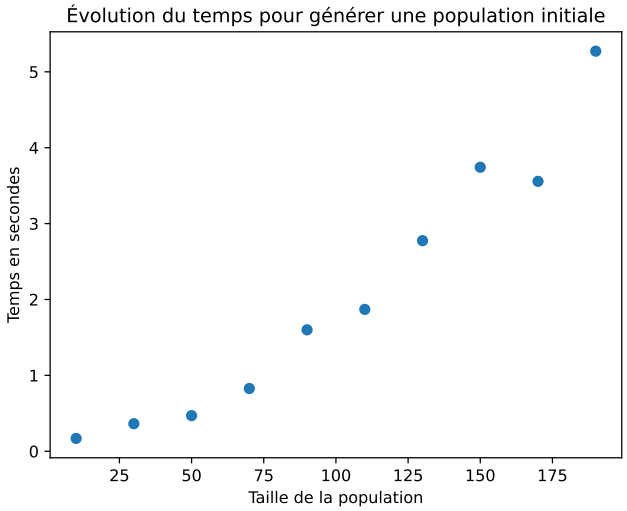


Image 1 – Temps pour générer une population initiale (instance « 45-4 »)

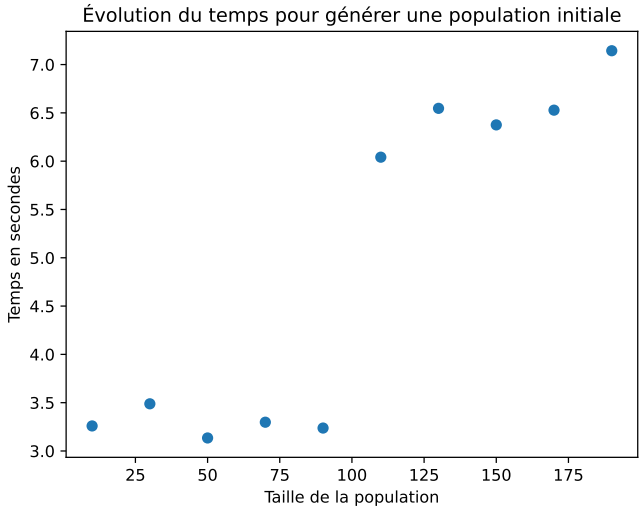


Image 2 – Temps pour générer une population initiale (instance « 96-6 »)

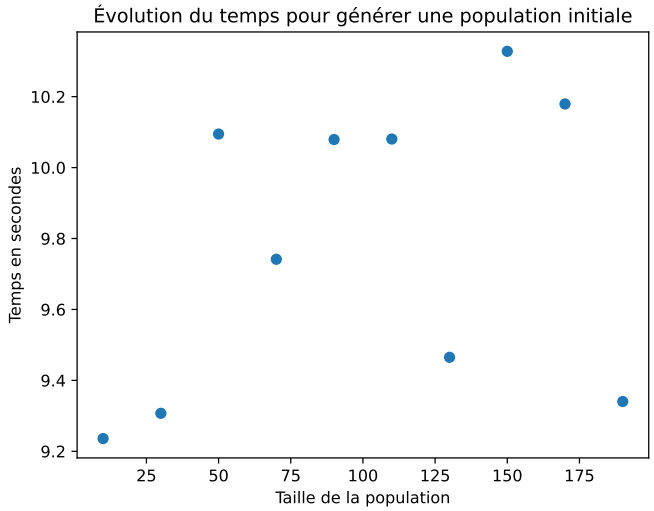


Image 3 – Temps pour générer une population initiale (instance « 100-10 »)

Grâce à cette technique de génération de population, nous pouvons générer plus de 100 plannings valides et différents en moins de 10 secondes. Alors que la génération totalement aléatoire met plus de 7 heures pour seulement 86 plannings.

La méthode de génération totalement aléatoire permettait d'avoir une population plus diversifiée, mais le temps d'exécution était inconcevable.

3 Mise en place des différents objectifs

Pour avoir la plus grande modularité, nous utiliserons des programmes capables de générer des fitness pour un planning à la fois. Nous les adapterons pour pouvoir générer des tableaux de fitness pour une population entière.

3.1 Fitness employés

Commençons par rappeler la formule :

$$f_{employés} = \frac{\zeta.\sigma_{WH}(s) + \gamma.\sigma_{OH}(s) + \kappa.\sigma_{D}(s)}{3}$$

```
1 def stats_heures(solution, intervenant, mission):
```

Avant de calculer la fitness des intervenants, il nous faut quelques statistiques les concernant, comme le nombre travaillé par chaque intervenant, le nombre d'heures non travaillés et le nombre d'heures supplémentaires. Pour ce faire, nous utilisons la liste des intervenants pour avoir leur contrat, le fichier contenant les missions pour le nombre d'heure de chaque mission et le planning pour connaitre à qui sont affecté chaque mission.

```
1 def activites_intervenants(solution, Int, Mis):
```

Cette fonction permet de retourner un dictionnaire des missions pour chaque intervenant en fonction du jour : edt = {1 :[], 2 :[], 3 :[], 4 :[], 5 :[]}, 1, 2, 3, 4 et 5 représente lundi, mardi, ..., vendredi, entre les crochets sera stocké les numéros de missions dans l'ordre chronologique.

```
1 def distance_employe(planning_1_mission, distance_matrice):
```

En réutilisant la fonction évoquée précédemment, nous pouvons facilement déterminer la distance parcouru chaque jour par employé, et ainsi la distance par semaine. S'il n'y a pas de mission durant une journée la distance est considérée comme nulle.

```
1 def fitnessEm(ecart_WH, ecart_OH, ecart_D, intervenants,
      distance):
```

Cette fonction permet de calculer la fitness des employés, comme demandé. De plus, on pourra l'initialiser grâce à la fonction :

```
1 def fitnessEmInitialisation(mission, inter, solution_1,
    distances):
```

3.2 Fitness étudiants

Il nous est suggéré la fonction suivante dans le but d'avoir la fitness des étudiants :

$$f_{étudiants} = \alpha.\pi(s)$$

```
1 def alpha(mission):
```

Pour calculer cette fitness, nous commençons par déterminer le coefficient α, pour ce faire nous récupérons le nombre de missions d'un planning.

```
1 def penalite_1(solution, MISSION, INTERVENANT):
```

Ensuite, nous cherchons les pénalités, dit autrement les missions pour lesquelles les employés n'ont pas la même spécialité que l'intervenant. Rien de plus simple il suffit de prendre le tableau du planning regarder à qui est affectée chaque mission et la comparer grâce aux fichiers « Missions.csv » et « Intervenants.csv » s'ils ont la même spécialité.

```
1 def fitnessEtudiants_1(solution, MISSIONS, INTERVENANTS):
```

Nous définissons la fitness d'un individu en récupérant la valeur de la fonction « alpha » et nous exécutons la fonction « penalite_1 » pour chaque individu.

```
1 def fitnessEtudiants_tout(SOLUTIONS, MISSIONS,INTERVENANTS):
```

Pour créer un tableau de toutes les fitness pour une population, nous exécutons la fonction précédente pour chaque individu et nous sauvegardons cette valeur dans un tableau.

3.3 Fitness SESSAD

Voici la fonction permettant de déterminer la fitness de la SESSAD :

$$f_{SESSAD} = \frac{\beta.\Sigma_{WOH}(s) + \kappa.moy(D(s)) + \kappa.max(D(s))}{3}$$

```
1  def sumWOH_1(mission, inter, solution):
```

Cette fonction permet de déterminer la somme du nombre d'heure non travaillée en utilisant la fonction « def activites_intervenants(solution) : » vu précédemment.

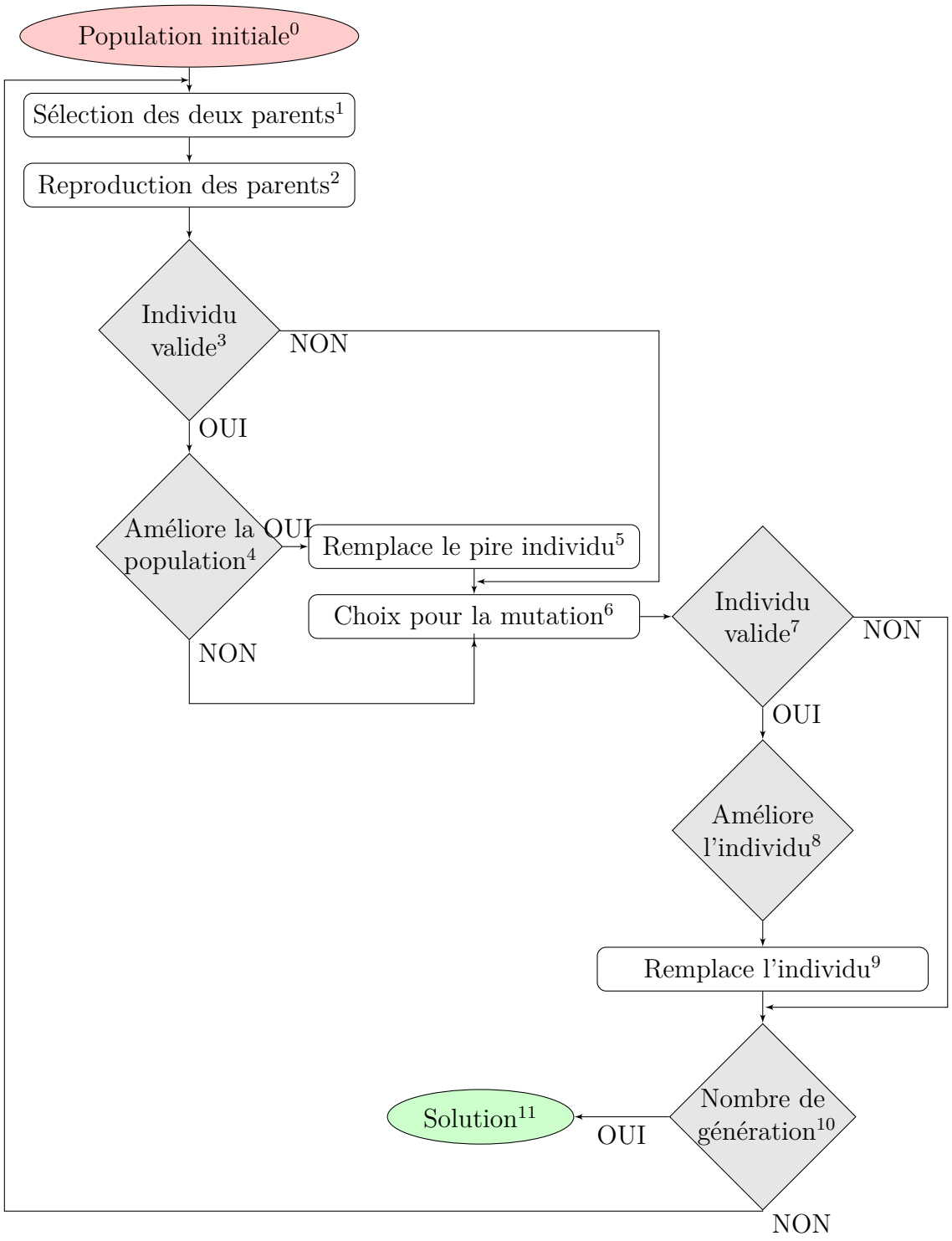
```
1  def beta():
2  def moyDist(disEmploy):
3  def maxDist(disEmploy):
```

Ces fonctions permettent de déterminer dans l'ordre : le coefficient β, la distance moyenne parcouru par les intervenants et la distance maximale réalisé par un intervenant.

```
1  def fitnessSESSAD(solution, dist_1_Semaine, inter, mis,
       matrice_distance):
```

Grâce à cette fonction nous calculons la fitness de la SESSAD pour un planning.

4 Algorithme génétique



Cet algorigramme permet de comprendre le fonctionnement de l'algorithme génétique dans les grandes lignes. Nous l'utiliserons tout au long de notre développement et nous le ferons évoluer en fonction de nos besoins.

4.1 Explication des différentes méthodes

Pour résoudre ce problème, nous allons commencer par coder un algorithme génétique qui permet de d'améliorer une fitness à la fois. Ensuite nous essayerons une amélioration en cascade, nous terminerons surement sur un choix par un front de Pareto (Pareto Front).

Dans le but de rendre notre programme le plus modulaire, nous avons écrit une fonction qui permet de calculer la fitness d'un individu, en lui précisant le type de fitness à exécuter : « étudiants », « employés » et « SESSAD ».

```
1 def choixFitness_1 (fit, mission, intervenants, fille,
    distances):
2 def choixFitness_tableau (fit, mission, intervenants,
    solutions, distances):
```

Il y aura une fonction permettant la reproduction de deux individus, nous avons décidé de faire un croisement en 1 point (dans le sens de la verticalité des plannings comme le montre le schéma).

Employés \ Missions	0	1	2	3	4
0	1	0	0	0	0
1	0	0	0	0	0
2	0	1	0	0	0
3	0	0	0	0	1
4	0	0	1	0	0
5	0	0	0	1	0

Tableau 1 – Parent 1

Employés \ Missions	0	1	2	3	4
0	0	0	0	1	0
1	0	0	0	0	0
2	1	0	1	0	1
3	0	0	0	0	0
4	0	1	0	0	0
5	0	0	0	0	0

Tableau 2 – Parent 2

Les parents se reproduisent.

Employés \ Missions	0	1	2	3	4
0	1	0	0	1	0
1	0	0	0	0	0
2	0	1	1	0	1
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0

Tableau 3 – Fille 1

Employés \ Missions	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	1	0	0	0	0
3	0	0	0	0	1
4	0	1	1	0	0
5	0	0	0	1	0

Tableau 4 – Fille 2

Cette opération est assurée par :

```
1 def reproduction (solution1, solution2, ligDebut, ligFin,
    colDebut, colFin):
```

Dernière fonctionnalité en commun la mutation, nous allons choisir un individu au hasard dans la population, nous réalisons une mutation verticale de la manière suivante, en choisissant une colonne aléatoirement et deux lignes :

Employés \ Missions	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	1	0	0	0	0
3	0	0	0	0	0
4	0	1	1	0	0
5	0	0	0	1	0

Tableau 5 – Individu

Nous réalisons la mutation :

Employés \ Missions	0	1	2	3	4
0	0	0	0	0	0
1	0	0	1	0	0
2	1	0	0	0	0
3	0	0	0	0	0
4	0	1	0	0	0
5	0	0	0	1	0

Tableau 6 – Muté

La mutation est réalisée par :

```
1 def mutation(solution):
```

4.1.1 Algorithme génétique à une fitness

Les arguments de chaque fonction des méthodes génétiques sont expliquées en annexe.

```
1 def genetique(solution, nbGeneration, probaMutation, distances
    , intervenants, mission, probaMissionEmpire, type_fit,
    colon):
```

Cette méthode permet d'avoir une amélioration assez importante de la fitness choisie, mais au détriment des autres.

Pour choisir les parents[1], nous utiliserons la fitness de chaque individu de notre population. Afin, de garder une part d'aléatoire, nous utiliserons la méthode de la roulette (Roulette Wheel Selection). Dans notre cas, nous voulons minimiser l'objectif, pour ce faire : notons n le nombre d'individu de notre population et $\forall i \in [\![1\ ;\ n]\!]$ f_i la fitness de l'individu i.

Pour avoir l'inverse de f_i :

$$f_i^{-1} = \frac{1}{f_i}$$

Pour avoir la pondération de f_i (sa probabilité d'être choisi) :

$$p_i = \frac{f_i^{-1}}{\sum_{j=1}^{n} f_j^{-1}}$$

Ainsi nous tirons de manière aléatoire avec pondération les deux parents, ce qui est possible grâce à la fonction :

```
1 def choixParents(fitness):
```

Après avoir choisi nos parents, ils se reproduisent et donnent deux filles[2], si leurs enfants respectent les contraintes[3], alors nous le regardons, s'ils améliorent la fitness[4] de l'individu à tuer (il a été choisi par une méthode de la roulette aussi). Si oui nous les intégrons dans la population en supprimant les pires individus[5], puis nous passons à la mutation et nous recommençons comme pour les étapes[3, 4], si l'individu muté est meilleur que le non muté nous remplaçons l'individu original[9]. Puis, nous recommençons à l'étape[1].

De plus, nous avons rajouter la possibilité de garder la mutation même si elle empire la fitness, pour créer de la diversité dans notre population.

4.1.2 Algorithme génétique en cascade

```
1  def genetiqueCascade(solutions, nbGeneration, probaMutation,
       distances, intervenants, mission, probaMissionEmpire,
       nbTour, colon):
```

L'algorithme en cascade reprend la fonction précédente en exécutant les algorithmes génétiques pour chaque fitness en transmettant la population entre les algorithmes. Nous pouvons aussi boucler sur cette procédure.

4.1.3 Algorithme génétique en moyenne

1. Moyenne classique : pour ce faire, nous prenons l'ensemble des fitness et nous en réalisons leur moyenne de la façon suivante :

$$moy = \frac{fitness_{employés} + fitness_{étudiants} + fitness_{SESSAD}}{3}$$

La fonction suivante permet de retourné le tableau des moyennes des fitness d'une population :

```
1  def moyenneFitness(fitness1, fitness2, fitness3):
```

2. Moyenne normalisée : nous normalisons les valeurs de nos fitness comme suit : Soit F le tableau de la fitness f_i pour une population P de n individus.
Notons :

$$min_F = min(F) = \min_{\forall x \in P} f_i(x)$$
$$max_F = max(F) = \max_{\forall x \in P} f_i(x)$$

De ce fait la norme de $f_i(x)$, avec $max_F - min_F \neq 0$ est :

$$\|f_i(x)\| = \frac{f_i(x) - min_F}{max_F - min_F}$$

La fonction suivante permet de retourné le tableau des moyennes normalisées des fitness d'une population :

```
1  def moyenneFitness_Norma(fitness1, fitness2, fitness3):
```

Ainsi nous avons :

```
def genetiqueMoyenneNorma(solutions, nbGeneration,
    probaMutation, distances, intervenants, mission,
    probaMissionEmpire, colon):
```

4.1.4 Algorithme génétique avec le front de Pareto

Pour optimiser les fitness des trois objectifs, nous allons utiliser le front de Pareto avec 3 dimensions.

Optimalité de Pareto à 2 dimensions dans une population de taille n, en notant Ω l'ensemble des allocations de ressources réalisables et P l'ensemble des allocations de ressources Pareto-optimales, alors l'allocation x fait partie de l'ensemble P si et seulement si :

- version faible : $\{y \in \Omega \mid y \succ_i x \ \forall i \in [\![1\ ;\ n]\!]\} = \varnothing$
- version forte : $\{y \in \Omega \mid y \succeq_i x \ \forall i \in [\![1\ ;\ n]\!] \ et \ \exists i \ tel \ que \ y \succ_i x\} = \varnothing$

Nous avons plus ou moins la même chose avec plus que 2 dimensions. Voici le code qui permet de vérifier si un point fait parti du front de Pareto :

```
for row in myArray[1:,:]:
    if sum([row[x] >= pareto_frontier[-1][x] for x in range(
    len(row))]) != len(row):
        # S'il est meilleur sur toutes les caractéristiques,
    ajoutez la ligne à pareto_frontier.
    pareto_frontier = np.concatenate((pareto_frontier, [row]))
```

Code source 1 – Front de Pareto

Ainsi la fonction permet de calculer le front de Pareto pour les 3 objectifs :

```
def pareto_frontier_multi(X,Y,Z):
```

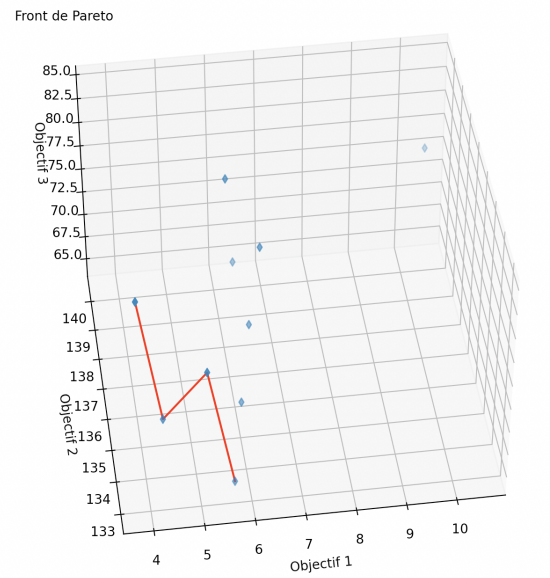


Image 4 – Exemple de front de Pareto

Nous choisirons les parents parmi ce front de Pareto et nous ajouterons les enfants seulement s'ils améliorent toutes les fitness.

Ainsi nous avons :

```
1  def genetiquePareto(solutions, nbGeneration, probaMutation,
       distances, intervenants, mission, probaMissionEmpire):
```

4.1.5 Éviter les blocages

Supposons qu'après un certain nombre de générations nous n'arrivons plus à trouver d'enfants valides par la reproduction de deux parents. Pour éviter ce problème, nous avons mis en place la procédure suivante : si nous ne trouvons plus d'enfants viable, alors nous remplaçons la moitié de la population par de nouveaux individus générés aléatoirement. Ceci est réalisé par la fonction suivante :

```
1  def remplacement(fitness_tab, sol, intervenants, missions,
       matrice_distance):
```

4.2 Temps d'exécution, Instance « 45-4 »

4.2.1 Génétique classique

Dans les conditions suivantes :

```
1 genetique(sol, i, 0.1, matrice_distance, intervenants,
      missions, 0.00, type, True)
```

Nous avons :

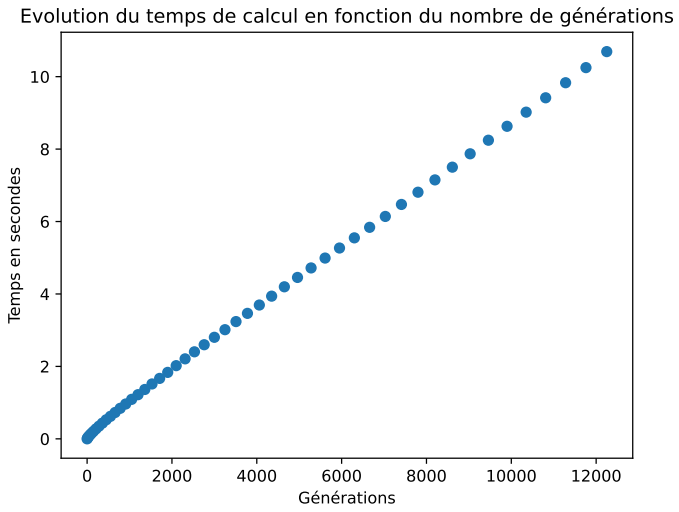


Image 5 – Temps d'exécution pour l'algorithme génétique (population initiale 100) (instance « 45-4 »)

4.2.2 Génétique en cascade

Dans les conditions suivantes :

```
1 genetiqueCascade(sol, i, 0.1, matrice_distance, intervenants,
      missions, 0.0, 3, True)
```

Nous avons :

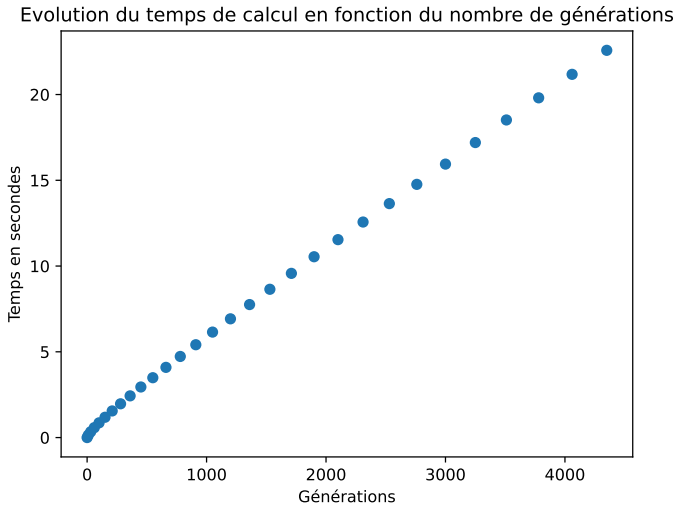


Image 6 – Temps d'exécution pour l'algorithme génétique en cascade (population initiale 100) (instance « 45-4 »)

Le nombre de générations correspond ici au nombre de génération moyenne par tours et par algorithme génétique.

4.2.3 Génétique en cascade par fitness

Dans les conditions suivantes :

```
1  genetiqueCascade_fit(sol, i, 0.1, matrice_distance,
       intervenants, missions, 0.0, 3, True)
```

Nous avons :

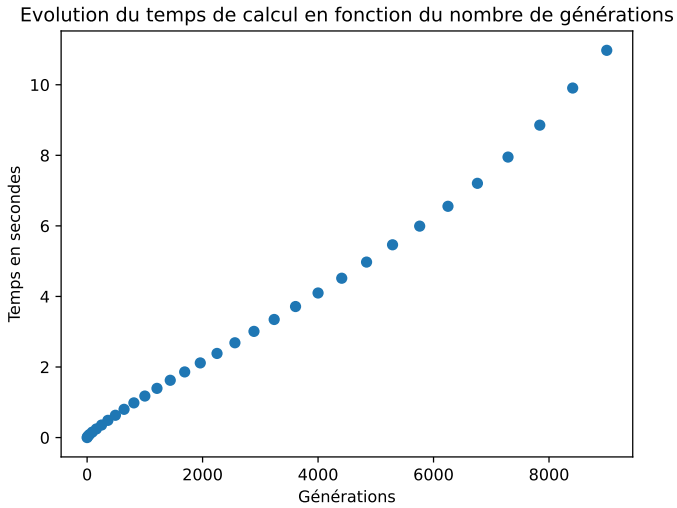


IMAGE 7 – Temps d'exécution pour l'algorithme génétique en cascade par fitness (population initiale 100) (instance « 45-4 »)

4.2.4 Génétique en moyenne normalisée

Dans les conditions suivantes :

```
1 genetiqueMoyenneNorma(sol, i, 0.1, matrice_distance,
      intervenants, missions, 0.0, True)
```

Nous avons :

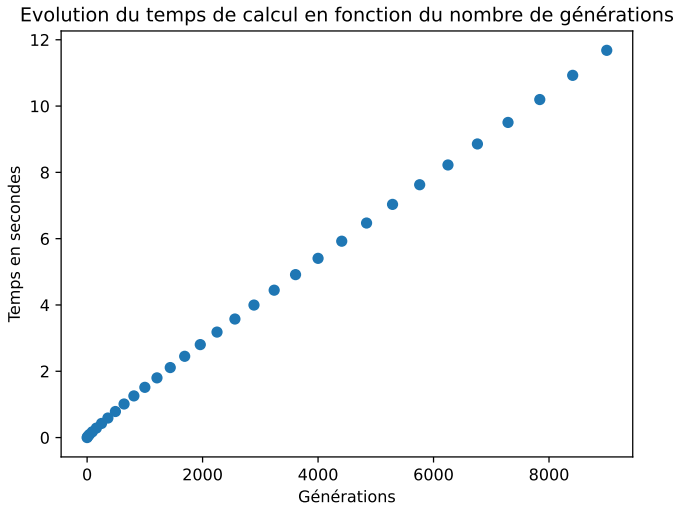


IMAGE 8 – Temps d'exécution pour l'algorithme génétique en moyenne normalisé (population initiale 100) (instance « 45-4 »)

4.2.5 Génétique avec front de Pareto

Dans les conditions suivantes :

```
1 gene.genetiquePareto(sol2, i, 0.1, matrice_distance,
      intervenants, missions, 0.0)
```

Nous avons :

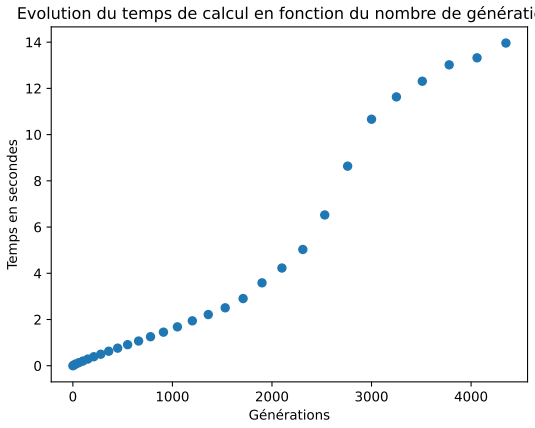


Image 9 – Temps d'exécution pour l'algorithme génétique avec front de Pareto (population initiale 100) (instance « 45-4 »)

De manière générale la durée d'exécution augment linéairement en fonction du nombre de génération à réaliser. Le plateau qui commence à apparaitre sur le graphique de Pareto peut s'expliqué par une impossibilité à trouver d'autres solutions valables.

4.3 Évolution de la fitness

4.3.1 Génétique classique

Dans les conditions suivantes :

```
1 gene.genetique(sol2, i, 0.1, matrice_distance, intervenants,
      missions, 0.0, "employe", True)
```

Nous avons :

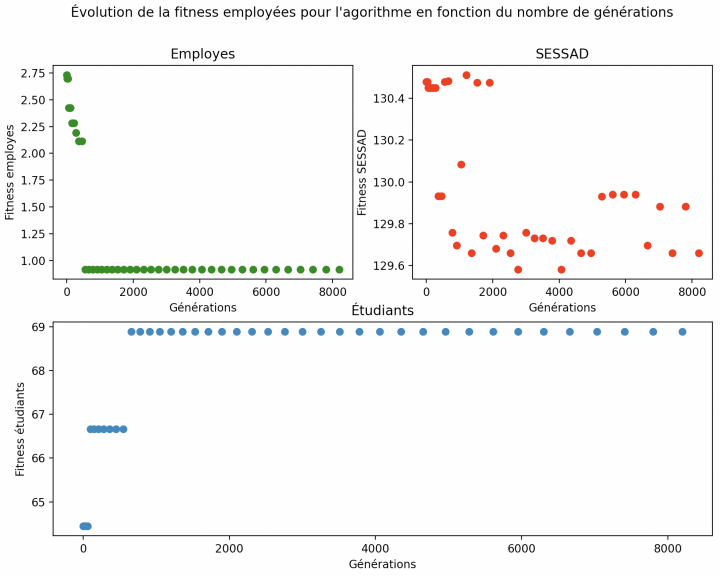


IMAGE 10 – Évolution de la fitness employées pour l'algorithme génétique (population initiale 100) (instance « 45-4 »)

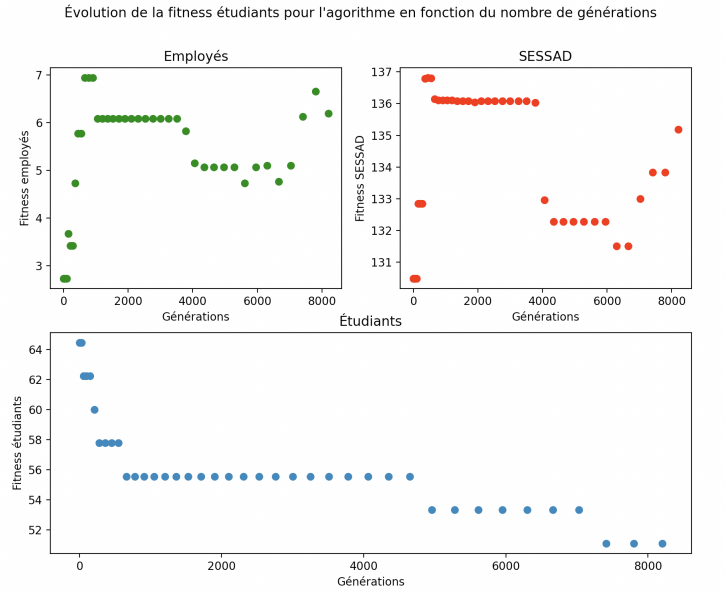


IMAGE 11 – Évolution de la fitness étudiant pour l'algorithme génétique (population initiale 100) (instance « 45-4 »)

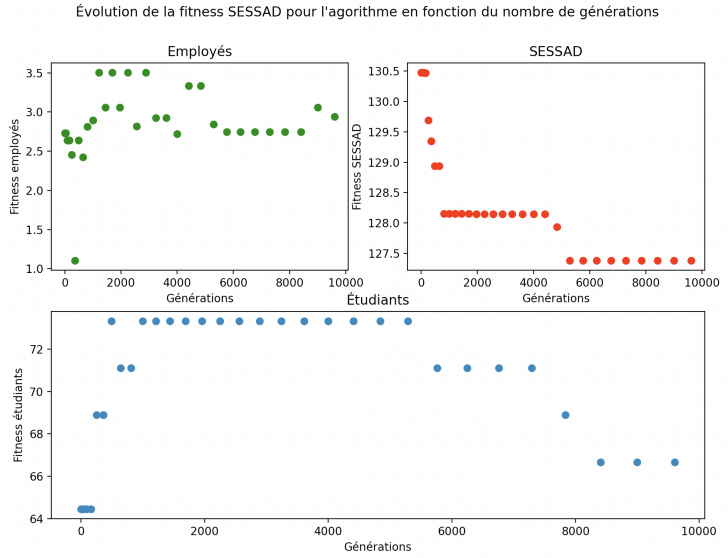


Image 12 – Évolution de la fitness SESSAD pour l'algorithme génétique (population initiale 100) (instance « 45-4 »)

Nous pouvons constater qu'après chaque exécution de notre algorithme génétique la fitness concernée est améliorée assez sensiblement, mais les autres fitness sont empirées.

4.3.2 Génétique en cascade

Dans les conditions suivantes :

```
1  gene.genetiqueCascade(sol2, i, 0.07, matrice_distance,
       intervenants, missions, 0.00, 3, True)
```

Nous avons :

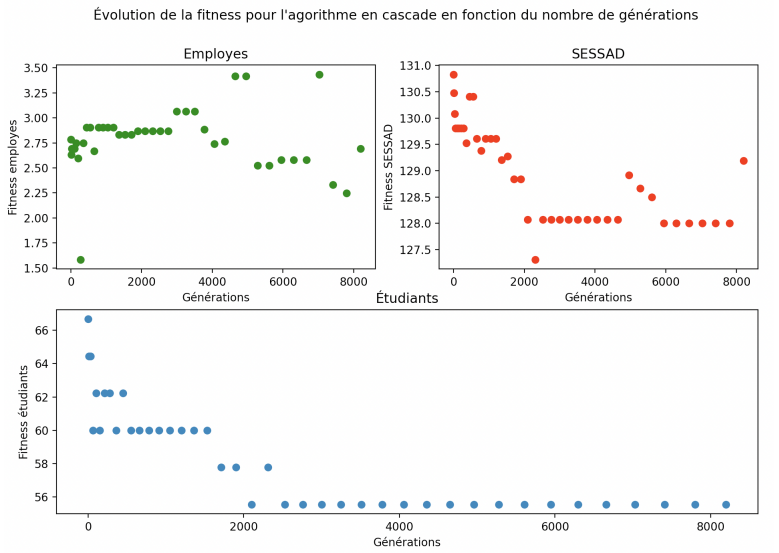


IMAGE 13 – Évolution de la fitness pour l'algorithme génétique en cascade (population initiale 100) (instance « 45-4 »)

L'évolution des fitness est très dépendante de l'ordre d'exécution des conditions à améliorer, c'est pour cela qu'il y a des fitness qui augmente, mais en moyenne il y a une amélioration de l'ensemble des fitness.

4.3.3 Génétique en cascade par fitness

Dans les conditions suivantes :

```
1 gene.genetiqueCascade_fit(sol2, i, 0.07, matrice_distance,
      intervenants, missions, 0.00, 3, True)
```

Nous avons :

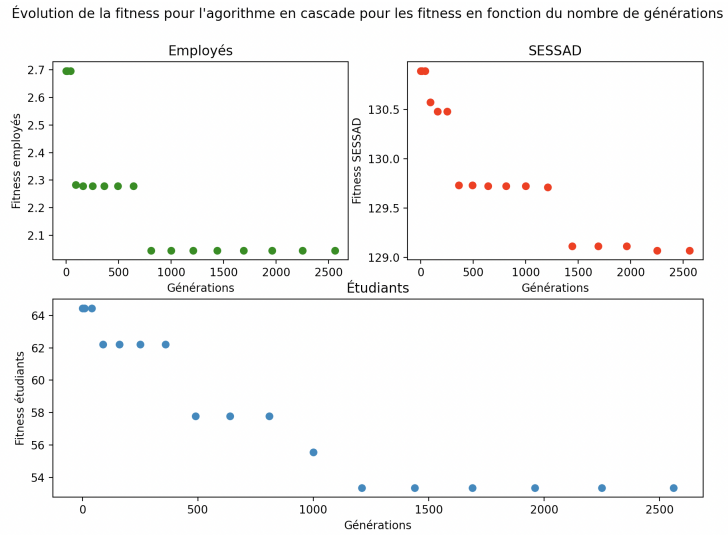


IMAGE 14 – Évolution de la fitness pour l'algorithme génétique en cascade par fitness (population initiale 100) (instance « 45-4 »)

Nous avons aussi essayer de faire l'amélioration en cascade, si la solution trouvée diminue la première fitness nous passons à la suivante.

4.3.4 Génétique en moyenne normalisée

Dans les conditions suivantes :

```
1 gene.genetiqueMoyenneNorma(sol2, i, 0.07, matrice_distance,
      intervenants, missions, 0.00, True)
```

Nous avons :

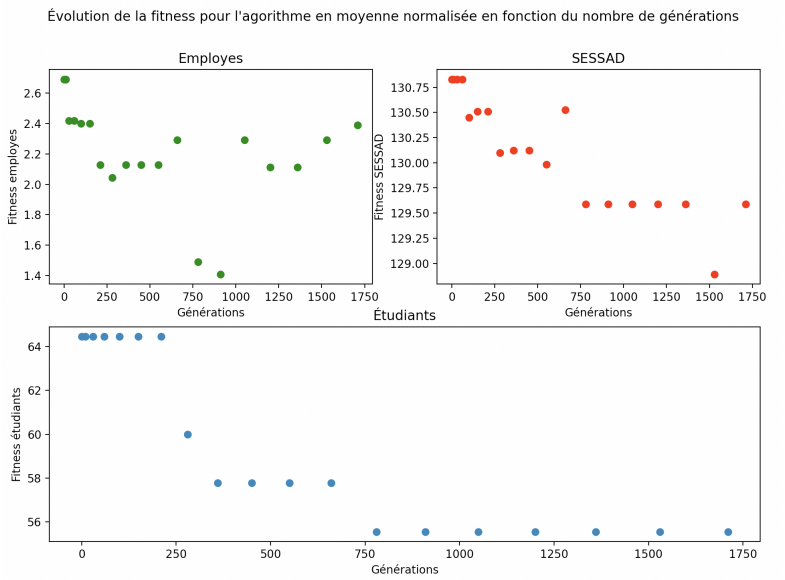


IMAGE 15 – Évolution de la fitness pour l'algorithme génétique en moyenne normalisée (population initiale 100) (instance « 45-4 »)

Les fitness sont améliorées pour optimiser en moyenne l'ensemble des contraintes, ceci peut en détériorer d'autres.

4.3.5 Génétique avec front de Pareto

Dans les conditions suivantes :

```
1 gene.genetiquePareto(sol2, i, 0.07, matrice_distance,
      intervenants, missions, 0.00)
```

Nous avons :

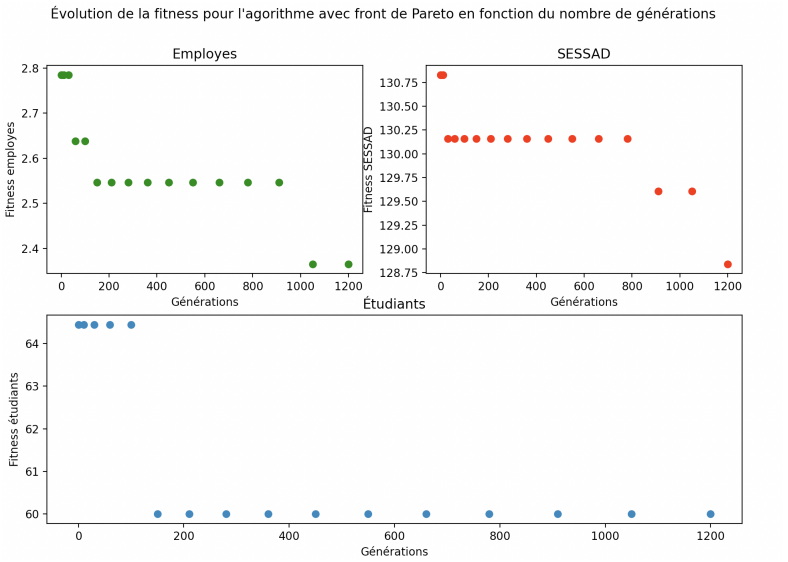


Image 16 – Évolution de la fitness pour l'algorithme génétique avec front de Pareto (population initiale 100) (instance « 45-4 »)

Le front de Pareto permet d'améliorer toutes les fitness sans en détériorer d'autre.

5 Conclusion

Pour conclure, les algorithmes génétiques permettent de trouver rapidement des plannings améliorants ceux généré aléatoirement.

De plus, ils sont très modulables et permettent de mettre en place des stratégies de résolution assez différente entre chaque programme.

Annexes

Nom des variables	Explications
solution	La population initiale du problème
nbGeneration	Le nombre de générations à réaliser
probaMutation	La probabilité d'avoir une mutation
distance	La matrice des distances de l'instance
intervenants	La matrice des intervenants de l'instance
mission	La matrice des missions de l'instance
probaMissionEmpire	La probabilité de garder une mutation qui empire la fitness
type_fit	Le type de fitness à améliorer
colon	Booléen qui autorise l'arrivé de nouveaux individus
nbTour	Le nombre de tour dans l'amélioration en cascade

Tableau 7 – Variables

Table des images

1 Temps pour générer une population initiale (instance « 45-4 ») 3

2 Temps pour générer une population initiale (instance « 96-6 ») 3

3 Temps pour générer une population initiale (instance « 100-10 ») 4

4 Exemple de front de Pareto 15

5 Temps d'exécution pour l'algorithme génétique (population initiale 100) (instance « 45-4 ») 16

6 Temps d'exécution pour l'algorithme génétique en cascade (population initiale 100) (instance « 45-4 ») 17

7 Temps d'exécution pour l'algorithme génétique en cascade par fitness (population initiale 100) (instance « 45-4) 18

8 Temps d'exécution pour l'algorithme génétique en moyenne normalisé (population initiale 100) (instance « 45-4 ») 19

9 Temps d'exécution pour l'algorithme génétique avec front de Pareto (population initiale 100) (instance « 45-4 ») 20

10 Évolution de la fitness employées pour l'algorithme génétique (population initiale 100) (instance « 45-4 ») 21

11 Évolution de la fitness étudiant pour l'algorithme génétique (population initiale 100) (instance « 45-4 ») 21

12 Évolution de la fitness SESSAD pour l'algorithme génétique (population initiale 100) (instance « 45-4 ») 22

13 Évolution de la fitness pour l'algorithme génétique en cascade (population initiale 100) (instance « 45-4 ») 23

14 Évolution de la fitness pour l'algorithme génétique en cascade par fitness (population initiale 100) (instance « 45-4 ») 24

15 Évolution de la fitness pour l'algorithme génétique en moyenne normalisée (population initiale 100) (instance « 45-4 ») 25

16 Évolution de la fitness pour l'algorithme génétique avec front de Pareto (population initiale 100) (instance « 45-4 ») 26

Table des tableaux

1 Parent 1 9
2 Parent 2 10
3 Fille 1 10
4 Fille 2 10
5 Individu 11
6 Muté 11
7 Variables i

Liste des codes sources

1 Front de Pareto 14