GLOESENER Maxime – SYED Tyemur

Optimisation multi-objectif : Pareto Local Search pour le problème d'affectation

Introduction

L'objectif de ce projet est de résoudre le problème d'affectation en multi-objectif. Nous devons attribuer m jobs à m machines, en sachant que chaque job à un coût défini par machine, l'objectif est de minimiser le coût total des jobs. Comme nous sommes en multi-objectif, nous n'avons pas une seule fonction objectif par machine mais bien plusieurs et il faut minimiser les coûts totaux.

On peut modéliser le problème sous la forme :

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} c1_{ij} x_{ij}$$

...

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} cm_{ij} x_{ij}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \; avec \; j = 1, \dots, n$$

$$\sum_{j=1}^{n} x_{ij} = 1 \; avec \; i = 1, \dots, n$$

$$x_{ij} \; appartient \; \text{à} \; \{0,1\} \quad i = 1, \dots, n \quad j = 1, \dots, n$$

La complexité de ce problème est dans le fait que les objectifs sont conflictuels.

Codage de la solution

Le codage de la solution correspond simplement à un vecteur de taille m, avec m qui est égal au nombre de jobs et au nombre de machines. On assigne à chaque machine d'indice i, le job qui se trouve à cet indice dans la liste.

Exemple de codage de solution = [0, 3, 2, 1, 4, 5, 6, 7]

Cela veut dire qu'on a assigné : le job 0 à la machine 0, le job 3 à la machine 1, le job 1 à la machine 3, le job 4 à la machine 4, le job 5 à la machine 5, le job 6 à la machine 6 et le job 7 à la machine 7.

Initialisation du problème

Pour initialiser le problème de la meilleure des manières, nous utilisons trois méthodes différentes.

Combinaisons linéaires

Comme vu en cours, résoudre le problème en mono-objectif en faisant des combinaisons linéaires des fonctions objectifs nous permet d'obtenir les solutions supportées du problème.

Pour réaliser les combinaisons linéaires des objectifs, nous générons simplement des coefficients en utilisant la fonction « product » de itertools. En utilisant cette fonction, nous pouvons générer rapidement les coefficients avec lesquels nous allons multiplier les fonctions objectifs pour réaliser les combinaisons linéaires.

Nous faisons donc simplement des combinaisons linéaires des objectifs et cela nous permet d'avoir une seule matrice de coûts. A partir de cette matrice de coûts qui est la somme pondérée des objectifs, nous pouvons résoudre le problème comme un problème mono-objectif.

Pour résoudre le problème en mono-objectif, nous utilisons la librairie scipy et plus particulièrement la fonction « linear_sum_assignment » qui nous permet d'obtenir une solution exacte pour le problème mono-objectif. Cette fonction utilise la méthode hongroise en $O(n^3)$ pour trouver la solution optimale du problème.

Pour choisir le nombre de solutions initialisées à partir de cette méthode, on modifie simplement la valeur du coefficient maximal utilisée lors de la génération des coefficients. Voici un exemple de génération de coefficients avec le coefficient maximal fixé à 2. On peut remarquer qu'on génère bien toutes les combinaisons possibles.

```
>>> max_coef = 2
>>> x = [int(i) +0.1 for i in range(max_coef)]
>>> list(product(x,repeat=4))
[(0.1, 0.1, 0.1, 0.1), (0.1, 0.1, 0.1, 1.1), (0.1, 0.1, 1.1, 0.1),
 (0.1, 0.1, 1.1, 1.1), (0.1, 1.1, 0.1, 0.1), (0.1, 1.1, 0.1, 1.1),
 (0.1, 1.1, 1.1, 0.1), (0.1, 1.1, 1.1, 1.1), (1.1, 0.1, 0.1, 0.1),
 (1.1, 0.1, 0.1, 1.1), (1.1, 0.1, 1.1, 0.1), (1.1, 0.1, 1.1, 1.1),
 (1.1, 1.1, 0.1, 0.1), (1.1, 1.1, 0.1, 1.1), (1.1, 1.1, 1.1, 0.1),
 (1.1, 1.1, 1.1, 1.1)]
```

Figure 1: Exemple de génération des coefficients

Pour les plus gros problèmes, nous avons utilisé un coefficient qui variait entre 50 et 80 en fonction du problème. Si on utilise un coefficient égal à 50, cela nous permet de générer 6250000 combinaisons linéaires et donc de résoudre 6250000 problèmes en mono-objectif. Cela se fait rapidement (quelques minutes) et il nous suffit de garder les solutions uniques par la suite. Nous pouvons déjà obtenir un nombre de solutions de l'ordre de quelques milliers pour les gros problèmes juste en résolvant des combinaisons linéaires des objectifs.

Nous avons remarqué que si on résolvait le problème en mono-objectif avec tous les coefficients égaux à 0 sauf 1, par exemple cette combinaison linéaire : fobj = 0*fobj1 + 0*fobj2 + 1*fobj3, alors on a un problème car si le solveur trouve plusieurs solutions optimales ayant la même valeur d'objectif pour fobj3, alors il va prendre une solution aléatoire. Cependant la solution prise de façon aléatoire n'est pas forcément la meilleure, en effet, si elle possède un coût supérieur pour les autres objectifs alors la solution ne fera pas partie du front de Pareto. C'est pour cela que nous ne faisons pas des coefficients

[0, 0, 0, 1] mais [0.1, 0.1, 0.1, 1]. Cela nous permet dans le cas expliqué juste avant, de donner un peu d'importance aux autres objectifs et donc de choisir parmi toutes les solutions ayant le même objectif, celle qui appartient au front de Pareto.

Exploration aléatoire

La deuxième méthode utilisée pour initialiser est simplement une exploration aléatoire du domaine. Lors de la première méthode d'initialisation, nous trouvons un grand nombre de solutions supportées et nous voulions aussi initialiser le problème avec des solutions non supportées.

Nous utilisons donc la fonction « permutation » de itertools qui nous permet de générer extrêmement efficacement toutes les solutions du domaine. Cela se fait avec un générateur python, ce qui est optimisé pour la mémoire. En effet, si on veut faire pareil avec une liste de taille n factorielle, on ne pourra jamais stocker toutes les permutations possibles en mémoire.

Pour initialiser le problème en utilisant cette méthode, on va fixer un nombre de solutions à trouver et tant qu'on aura pas trouvé ce nombre de solutions, on va explorer le domaine et chercher des solutions non dominées de façon aléatoire dans le domaine.

En pratique, nous avions implémenté cette exploration aléatoire en espérant trouver des solutions non supportées mais nous avons remarqué c'est une méthode assez naïve et sous efficace pour trouver des solutions non supportées.

En effet, si on initialise un grand nombre de solutions avec les combinaisons linéaires, il est compliqué de trouver des solutions non supportées qui ne sont pas dominées par les solutions initiales en explorant le domaine de façon aléatoire. Au contraire, si on initialise le problème avec pas ou peu de solutions issues des combinaisons linéaires, alors il est beaucoup plus simple de trouver des solutions non supportées mais nous avons remarqué par expérience, qu'il est plus efficace d'initialiser le problème avec les combinaisons linéaires. Cela s'explique principalement par le fait nous pouvons avoir des milliers de solutions supportées en quelques minutes de façon déterministe. Au contraire, si on explore de façon aléatoire, cela prend beaucoup plus de temps pour avoir quelques milliers de solutions et de plus, cela est stochastique.

En pratique, nous avons donc principalement initialisé les solutions avec des combinaisons linéaires et pour l'exploration aléatoire, on trouvait seulement quelques dizaines de solutions non supportées (entre 10 et 40). Lancer plus longtemps aurait permis de trouver plus de solutions mais cela n'est pas du tout efficace au niveau du temps.

Aléatoire

Lorsque nous avons testé notre algorithme en explorant les solutions initialisées avec les deux premières méthodes, nous avons remarqué que l'algorithme tournait assez rapidement et qu'on était bloqué à un certain nombre de solutions. L'algorithme avait tout exploré en assez peu de temps , il nous fallait donc augmenter le nombre de solutions initiales pour permettre à l'algorithme d'explorer plus de solutions.

Plusieurs solutions s'offrent à nous :

- Augmenter la valeur du coefficient des combinaisons linéaires : le problème c'est qu'à partir d'un certain seuil, augmenter le nombre de combinaisons linéaires va prendre beaucoup plus de temps. Si on passe d'un coefficient 80 à un coefficient 100, on passe de 40960000 combinaisons linéaires à résoudre à 100000000 mais au niveau du nombre de solutions différentes obtenues, cela ne change pas grand-chose car on trouve beaucoup de solutions semblables.

- Augmenter le seuil du nombre de solutions à trouver de façon aléatoire : le problème c'est que comme expliqué juste avant, lorsque le nombre de solutions initialisées avec les combinaisons linéaires est grand, cela prend beaucoup de temps de trouver des solutions non dominées. Cette méthode est donc peu efficace.
- Initialiser avec des solutions totalement aléatoires.

Nous avons choisi la dernière option. En effet, générer des solutions aléatoires est quelque chose de très rapide et cela nous permet de diversifier nos solutions initiales à explorer, ce qui est fort intéressant étant donné que lors de l'exploration, nous sommes plutôt dans une phase d'intensification.

Algorithme

Exploration

L'algorithme utilisé est un Pareto Local Search. Voici le pseudo-code :

```
# initialisation du problème
sols = initialisation()
# parcourir les solutions initialisées
for sol in sols:
    # calculer le voisinage des solutions
    voisins = voisinage(sol)
    # parcourir les voisins et chercher une meilleure valeur
    for voisin in voisins:
        # si voisin est déjà dans les solutions à explorer alors on passe au prochain voisin
        if voisin in sols:
            continue
        # calcul du score
        score_newsol = score(voisin)
        # si solution déjà présente alors on ne met pas à jour l'archive
        # mais on veut quand même explorer les voisins de cette solution
        if score_newsol in archive:
            sols.append(voisin)
            continue
        # si pas dominé et différent
        if score_newsol non dominé et différent des solutions de l'archive:
            # update de l'archive
            archive = update(archive, new_sol)
            sols.append(voisin)
```

Figure 2 : Pseudo code de l'algorithme

On commence par initialiser les solutions avec les 3 méthodes présentées avant. On parcourt les solutions initialisées.

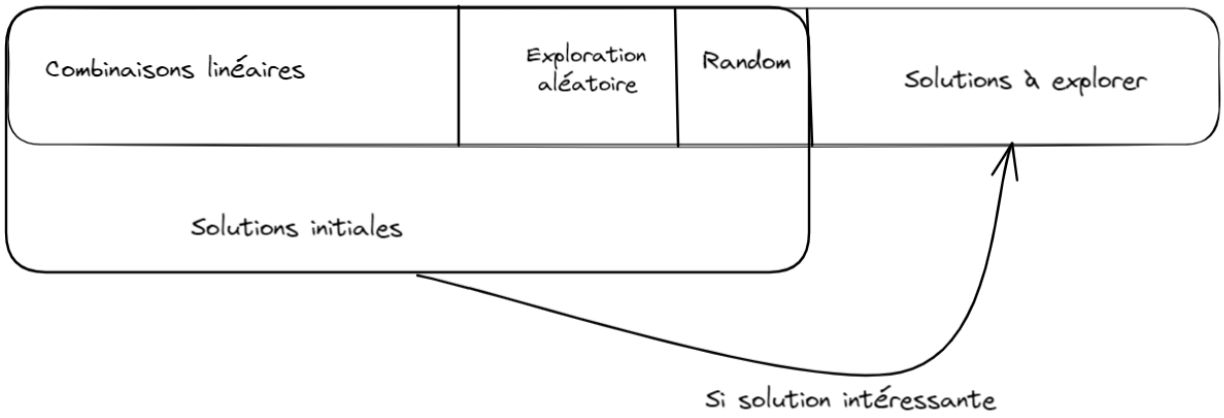
Pour une solution, on va calculer son voisinage. Cela va nous donner une liste de solutions à explorer. Si jamais un des voisins a déjà été exploré, alors on ne le prend pas en compte, cela nous permet d'éviter de réexplorer des solutions et donc de perdre du temps inutilement.

On calcule ensuite le score du voisin. Si ce score est déjà présent dans l'archive, alors on ne l'ajoutera pas à l'archive mais il est extrêmement important de quand même l'ajouter aux solutions à explorer. C'est un élément primordial que nous avons observé, si lors de l'exploration on trouve une solution qui a un coût similaire à un coût déjà présent dans l'archive de Pareto, alors il ne faut pas l'ajouter à

l'archive mais il faut absolument l'ajouter aux solutions à explorer car la plupart du temps, les voisins de bonnes solutions ont tendance à aussi être des bonnes solutions.

Pour finir, si on trouve une solution qui n'est pas dominée par les solutions actuelles de l'archive et qui est différente, alors on va mettre à jour l'archive. La mise à jour de l'archive consiste simplement à ajouter à l'archive la nouvelle solution trouvée qui n'est pas dominée et à supprimer de l'archive les solutions qui sont dominées par la nouvelle solution trouvée.

Voici un schéma qui représente les différentes phases d'exploration.



On peut identifier les 3 façons d'initialiser le problème. Cela forme un ensemble de solutions initiales à explorer. Lors de l'exploration on va trouver de nouvelles solutions intéressantes qu'on va explorer. Ces solutions intéressantes sont les solutions qui ne sont pas dominées par les solutions actuelles de l'archive de Pareto ou les solutions qui ont un score égal à une des solutions de l'archive de Pareto.

Cela permet aussi de mettre en avant le fait que bien initialiser est important. On pourrait théoriquement initialiser avec un grand nombre de solutions aléatoires, mais lors de nos expériences, on a remarqué qu'il était toujours plus intéressant d'explorer les voisins des solutions intéressantes, plutôt que d'explorer les voisins des solutions aléatoires. Lorsqu'on est limité en temps, il est donc important de mettre le nombre de solutions aléatoires à explorer assez bas pour être certain d'explorer les voisins des solutions intéressantes et pas les voisins des solutions aléatoires.

En pratique, pour les problèmes de taille 15, nous avons travaillé avec un nombre de solutions aléatoires de 10000, pour le problème de taille 20 de 1500 et pour le problème de taille 30 de 0. En effet, les problèmes de taille 15 et 20 tournent rapidement donc on est certain de tout explorer donc aucun problème.

Cependant, pour le problème de taille 30, nous avons arrêté l'algorithme après une quarantaine d'heures. Nous avons choisi de mettre le nombre de solutions aléatoires à 0 ainsi nous sommes certains de passer les 40h à explorer les voisins de solutions intéressantes au lieu d'explorer les voisins des solutions aléatoires qui sont souvent moins intéressants.

Voisinage

Pour obtenir le voisinage d'une solution, on va simplement faire toutes les permutations d'ordre 1 de la solution. On veut juste les permutations 1 à 1 pour intensifier un maximum et explorer autour des solutions déjà trouvées étant donné que la phase de diversification se fait surtout lors de l'initialisation. On a donc (n*(n-1)) / 2 voisins à explorer pour chaque solution avec n qui est la taille de la solution.

Exemple : solution = [2,4,3,1], on a donc 6 voisins : [4,2,3,1], [3,4,2,1], [1,4,3,2], [2,3,4,1], [2,1,3,4], [2,4,1,3]. Pour le plus gros problème de taille 435 on a un voisinage de taille 435, ce qui est tout à fait raisonnable.

Evaluation des résultats

Il est important de pouvoir évaluer la qualité de nos résultats. Comme nous sommes dans le cas d'un problème multi-objectif, il n'existe pas une seule solution optimale mais un ensemble de solutions qui forment le front de Pareto.

Calcul de l'hypervolume

L'hypervolume consiste à calculer le volume formé par un point de référence et le front de solutions trouvées. Dans un problème de minimisation, l'objectif est donc de maximiser l'hypervolume. Pour choisir un point de référence, on prend le maximum de chaque objectif dans les solutions trouvées qu'on multiplie par 1.1.

Voici une représentation de l'hypervolume pour un problème à 2 objectifs :

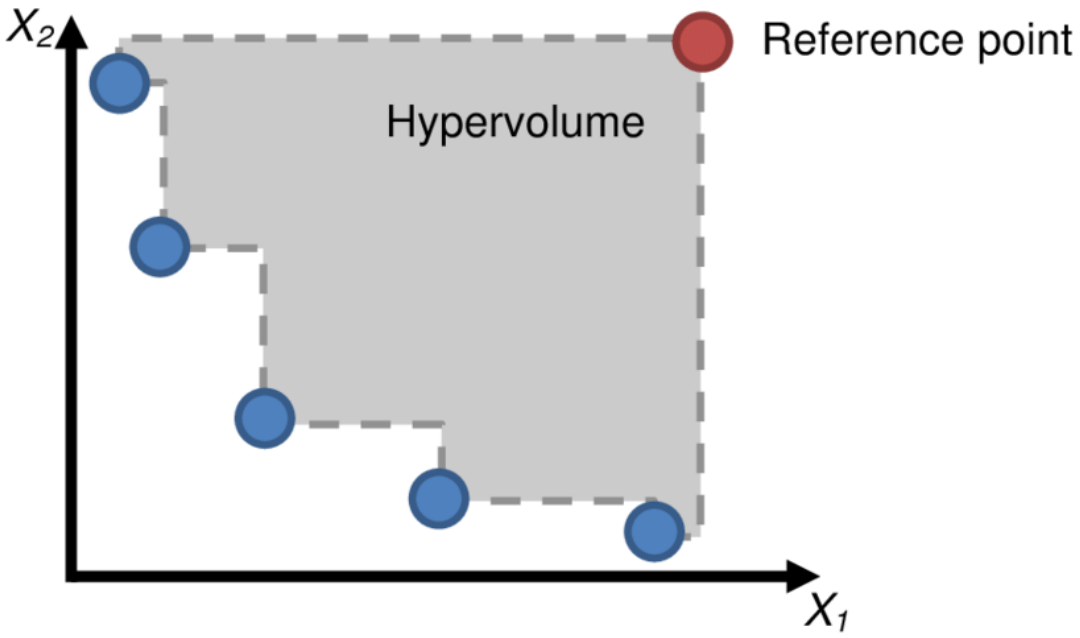


Figure 3 Hypervolume

Pour comparer les solutions, il suffit de trouver le point de référence commun et de calculer l'hypervolume. L'ensemble de solutions ayant le plus grand hypervolume sera considéré comme meilleur.

Pour calculer l'hypervolume, nous utilisons la librairie pymoo.

Comparaison des solutions dominées et dominantes

Pour comparer deux ensembles de solutions, on peut aussi regarder le nombre de solutions dominées, non dominées et communes entre les deux ensembles.

Comparaison de nos résultats avec les autres groupes

Nous avons pu comparer nos résultats avec ceux des autres groupes. Pour les problèmes de dimension 15 et 20, nous avons calculé un point de référence commun en prenant la valeur maximale pour chaque objectif de toutes les solutions, multipliée par 1.1.

Voici la comparaison pour le problème de taille 15 :

Equipe	Nombre total de solutions	Nombre de solutions identiques	Nombre de solutions dominées chez eux	Nombre de solutions dominées chez NOUS	Nombre de solutions non dominées chez eux	Nombre de solutions non dominées chez NOUS	Hypervolume Référence = [117.7, 233.2, 315.7, 442.2]
NOUS	3539						35.6609
Adrien Océane	3510	3440	14	13	3496	3526	35.6389
Antoine Isam	2833	405	2418	3	415	3536	33.6533
Antoine Selim	1678	67	1611	0	67	3539	29.3790
Chris Vivian	849	98	749	0	100	3539	32.3166
Valentin Tanguy	3476	3403	29	11	3447	3528	35.6318
Kevin Alan	4593	987	3606	0	987	3539	34.2401
Romain Florian	3536	3357	128	9	3408	3530	35.6325
Timo Bilal	3518	3445	18	13	3500	3526	35.6420
Tom Nicolas	839	1	838	0	1	3539	24.0899

Voici la comparaison pour le problème de taille 20 :

Equipe	Nombre total de solutions	Nombre de solutions identiques	Nombre de solutions dominées chez eux	Nombre de solutions dominées chez NOUS	Nombre de solutions non dominées chez eux	Nombre de solutions non dominées chez NOUS	Hypervolume Référence = [162.8, 341.0, 470.8, 600.6]
NOUS	9111						37.7882
Adrien Océane	7438	6106	1267	30	6171	9081	37.5291
Antoine Isam	3894	0	3894	0	0	9111	29.5407
Antoine Selim	1080	2	1078	0	0	9111	20.8525
Chris Vivian	1175	0	1175	0	0	9111	25.8446
Valentin Tanguy	8948	8393	267	103	8681	9008	37.7429
Kevin Alan	7701	271	7429	1	272	9110	34.6760
Romain Florian	4639	12	4623	0	16	9111	35.0780
Timo Bilal	8828	8396	213	98	8615	9013	37.7567
Tom Nicolas	661	0	661	0	0	9111	13.5771

Pour le 30, nous n'avons pas calculé de point de référence commun à toutes les solutions mais un point de référence commun aux 2 solutions comparées à chaque fois.

Voici la comparaison des problèmes de taille 30 :

Equipe	Nombre total de solutions	Nombre de solutions identiques	Nombre de solutions dominées chez eux	Nombre de solutions dominées chez NOUS	Nombre de solutions non dominées chez eux	Nombre de solutions non dominées chez NOUS	Hypervolume NOUS / EUX
NOUS	25510						
Adrien Océane	10170	981	8545	318	1625	25192	34.7 / 32.4
Antoine Isam	1528	0	1528	0	0	25510	35.3 / 6.7
Antoine Selim	1084	0	1083	0	1	25510	35.3 / 9.6
Chris Vivian	1199	1	1198	0	0	25510	36.2 / 14.6
Valentin Tanguy	17512	6318	8425	1289	9087	24221	34.87 / 34.2
Kevin Alan	9861	4	9855	0	6	25510	35.5 / 26
Romain Florian	9904	7	9897	0	7	25510	37.56 / 33.8
Timo Bilal	10991	8934	1340	346	9651	25164	34.8 / 34.3
Tom Nicolas	688	0	688	0	0	25510	35.3 / 3.9

Analyse des résultats

Pour les trois problèmes, nous avons à chaque fois le plus grand hypervolume de toutes les solutions proposées. Nous avons très peu de nos solutions qui sont dominées par celles des autres. Au contraire, nous dominons un nombre assez important de solutions chez les autres groupes pour chaque problème et cela est encore plus marqué pour les problèmes de taille 20 et 30.

Il est important de tenir compte du contexte de ces comparaisons. A première vue, nos solutions ont l'air meilleures que celles des autres mais il n'y avait pas de contraintes de temps sur l'algorithme. Nous avons laissé tourner l'algorithme environ 3/4 heures pour les problèmes de taille 15 et 20 et environ 40 heures pour le problème de taille 30. Selon nos informations, à l'exception de quelques groupes, la plupart des autres groupes ont laissé tourner leur algorithme en moyenne 30 minutes/1heure. Cela explique en partie les différences de résultats.

Pour ne pas comparer un algorithme qui a tourné 40 heures et un algorithme qui a tourné seulement une heure, nous allons maintenant comparer les solutions des autres groupes pour le problème de taille 30 avec nos solutions que nous allons générer en 20 minutes. Cela nous permettra de montrer que le temps est un facteur important pour la qualité des solutions mais qu'il n'est pas le seul.

Voici la comparaison des problèmes de taille 30 avec notre solution générée en 20 minutes :

Equipe	Nombre total de solutions	Nombre de solutions identiques	Nombre de solutions dominées chez eux	Nombre de solutions dominées chez NOUS	Nombre de solutions non dominées chez eux	Nombre de solutions non dominées chez NOUS	Hypervolume Référence = [225.5, 438.9, 636.9, 936.1]
NOUS	6813						40.48
Adrien Océane	10170	138	4832	474	5838	6339	38.5
Antoine Isam	1528	0	1528	0	0	6813	9.99
Antoine Selim	1084	0	1083	0	1	6813	15.1
Chris Vivian	1199	1	1198	0	0	6813	19.25

Valentin Tanguy	17512	1442	2977	1390	14535	5423	40.12
Kevin Alan	9861	0	9739	14	122	6799	31.26
Romain Florian	9904	8	9621	2	283	6811	37.2
Timo Bilal	10991	1592	418	1512	10573	5301	40.3716
Tom Nicolas	688	0	688	0	0	6813	5.016

Même avec un run de seulement 20 minutes, nous obtenons quand-même les meilleures solutions. Par rapport à un run de 40 heures, nous passons de 25510 solutions à seulement 6813 et d'un hypervolume de 40.82 à un hypervolume de 40.48 mais nos solutions restent de très bonne qualité et nous avons toujours le plus grand hypervolume parmi tous les groupes.

Cette comparaison permet de montrer que le facteur temps est un critère important pour la qualité des solutions trouvées, mais que ce n'est pas le seul critère à prendre en compte pour définir l'efficacité de l'algorithme. Il est important de :

- Bien comprendre comment initialiser les solutions en fonction de la taille du problème comme expliqué avant. Surtout lorsque le temps est restreint, il faut explorer les voisins des solutions intéressantes et non pas perdre trop de temps à explorer les voisins de solutions aléatoires.
- Intensifier l'exploration avec un voisinage simple et ne pas perdre du temps à vouloir faire un voisinage trop grand ou oublier des solutions avec un voisinage trop petit.
- Explorer intelligemment, c'est-à-dire ne pas réexplorer des solutions déjà explorées et surtout ne pas oublier d'explorer les voisins des solutions qui ont un score déjà présent dans l'archive. Il ne faut pas ajouter les solutions ayant un score déjà présent dans l'archive, mais il faut explorer les voisins de ses solutions.
- Utiliser des structures de données adéquates. Etant donné que c'est un algorithme qui doit explorer de nombreuses solutions, il est important d'avoir un codage adapté et de stocker les données dans des structures adaptées. Par exemple, nous savons que récupérer le coût d'une solution est une des opérations qui se fait le plus souvent dans l'algorithme (on doit le faire à chaque fois qu'on calcule un score). Pour optimiser cela, nous précalculons toutes les possibilités d'affectation et nous les stockons dans un dictionnaire.
  Par exemple, si on veut connaître la valeur des fonctions objectifs de l'attribution du job 1 à la machine 1, au lieu de devoir parcourir les matrices de coûts, nous avons précalculé les possibilités et tout stocké dans un dictionnaire.
  Le format est le suivant d = {(machine, job) : [f obj1, f obj2, f obj3, f obj4]}. Nous avons donc un dictionnaire de taille n*n avec n qui est la taille du problème. Si on a besoin des fonctions objectif de la machine 1 avec le job 1, on peut juste faire d[(1,1)] et récupérer la valeur de chaque objectif. La complexité est O(1). Cela nous évite de faire plusieurs opérations pour récupérer la valeur des objectifs dans chaque matrice de coûts.

Conclusion

En conclusion, nous sommes parvenus à implémenter un algorithme Pareto Local Search pour résoudre un problème multi-objectif. Nous avons montré lors de la comparaison que le facteur temps est un facteur important pour la qualité des solutions mais que ce n'est pas le seul facteur qui a de l'importance.

Les codes sont disponibles sur GitHub via le lien suivant :
https://github.com/MaximeGloesener/Multiobjective-optimization