Rapport : Algorithmes génétiques et problème du sac-à-dos avec plusieurs contraintes de coût.

Martin-Dipp Daryl
Neveu Pierre
M1 MIAGE

I- Implementation

Nous avons implémenté l'algorithme génétique en python et avons utilisé jupyter notebook pour mener les analyses. Vous retrouverez l'implémentation de l'algorithme dans le fichier "genetic_algorithm.py", des fonctions utiles aux analyses dans le fichier "functions.py" et les analyses dans le fichier "analyzes.ipynb".

II- Proposition d'amélioration de la fonction de réparation

Notre proposition d'amélioration de la fonction de réparation se base sur une simple réflexion du problème initial.
En effet, le but ici est de maximiser la somme des utilités des objets tout en respectant les contraintes de coût. De ce fait, nous avons eu l'intuition de vouloir garder les objets avec la plus grande utilité et les coûts les moins élevés. A l'inverse, nous voulons nous débarasser des objets avec une petite utilité et des coûts élevés.
C'est la raison pour laquelle nous avons décidé de baser la décision de notre fonction de réparation sur la comparaison des ratios "utilité / somme des coûts" de chaque objet. Plus précisément, notre fonction de réparation élimine les objets avec le ratio le plus petit.

Nous nous attendons au moins à trouver un exemple de paramètres spécifiques qui met en avant notre méthode, notamment en jouant sur les utilités et budgets. De plus, nous nous attendons principalement à avoir un meilleur temps de calcul avec notre fonction de réparation car elle a un caractère moins aléatoire sur la sélection des objets retirés et peut potentiellement avoir recours à moins d'opérations en moyenne.

NB: Ci-après, la fonction de réparation initiale proposée dans le sujet sera nommée "Algorithm 1 Repair" ou encore "subject repair method" et la fonction de réparation que nous proposons pourra être nommée "Ratio Repair" ou encore "utility/cost repair method"

III- Analyses et comparaison

Afin d'évaluer les performances de notre fonction de réparation par rapport à celle du sujet, nous étudierons d'abord la vitesse de convergence (i.e. le nombre de génération pour atteindre un maximum) et le niveau de maximisation de l'utilité sur un exemple général et aléatoire. Puis nous étudierons les mêmes indicateurs sur un exemple plus spécifique et choisis pour mettre en avant notre fonction de réparation. Le but de ce deuxième exemple est

de prouver l'efficacité de notre fonction dans au moins un cas particulier et pouvoir la combiner ultérieurement avec d'autres règles de réparation pour encore optimiser la fonction.

De plus, nous étudierons la diversité des deux fonctions et essayerons de jouer sur un paramètre pour l'améliorer.

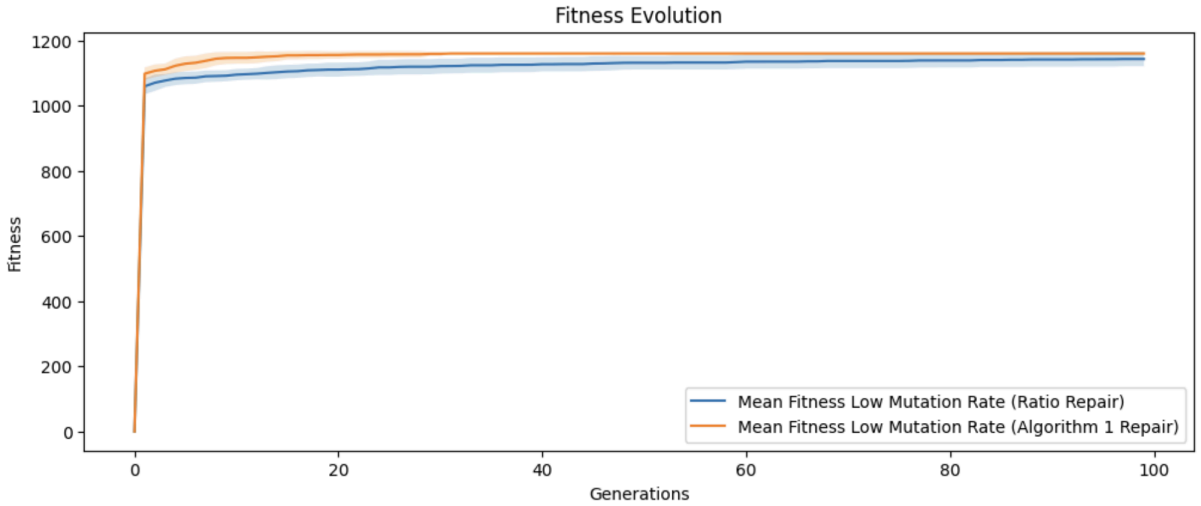
Enfin, nous comparerons les temps de calculs nécessaires avec chaque fonction pour différentes tailles de population.

3.1 Evolution de l'utilité

3.1.1 Paramètres 1: Exemple général

```
population_size = 100
crossover_rate = 0.7
num_generations = 100
budgets = [100, 200, 150, 120, 130]
elitism_rate = 0.1
num_instances = 30
mutation_rate = 0.01
```

3.1.2 Résultats 1



Ce premier résultat nous permet d'observer que les deux méthodes convergent à peu de choses près vers la même utilité. Cependant, on observe une convergence légèrement plus rapide avec la méthode du sujet.

3.1.3 Paramètres 2: Exemple spécifique

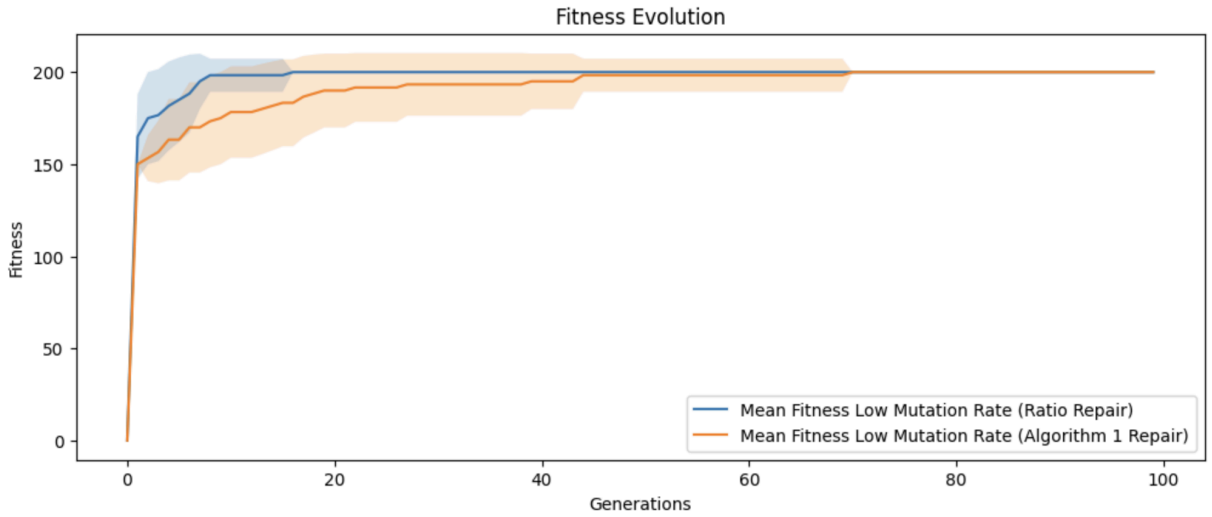
```
population_size = 100
crossover_rate = 0.7
num_generations = 100
budgets = [100, 200, 150, 120, 130]
elitism_rate = 0.1
num_instances = 30
mutation_rate = 0.01

# Generate utilities with the same value for all objects
utilities = np.ones(50) * 50

# Generate random costs for each object
costs = np.random.rand(50, 5) * 100
```

Dans cet exemple, on se propose de générer des paramètres de sorte à ce que tous les objets aient la même utilité et un coût différent. De ce fait, nous pourrons observer si notre méthode est efficace pour repérer les objets avec le meilleur ratio.

3.1.4 Résultats 2



Les résultats mettent en avant le principe de notre méthode de réparation comparée à celle du sujet.
Dans un premier temps, on observe une vitesse de convergence bien supérieure avec notre méthode (environ 10 génération contre environ 45 pour atteindre un maximum).
Ensuite, l'écart-type autour de la courbe montre d'autant plus une meilleure précision / convergence avec notre méthode.

3.2 Comparaison de la diversité en fonction du taux de mutation

3.2.1 Méthode de calcul de la diversité

Afin de pouvoir évaluer la diversité et son évolution au cours des différentes générations, nous avons utilisé la distance de Hamming (source: https://hal.science/hal-03595275/document).

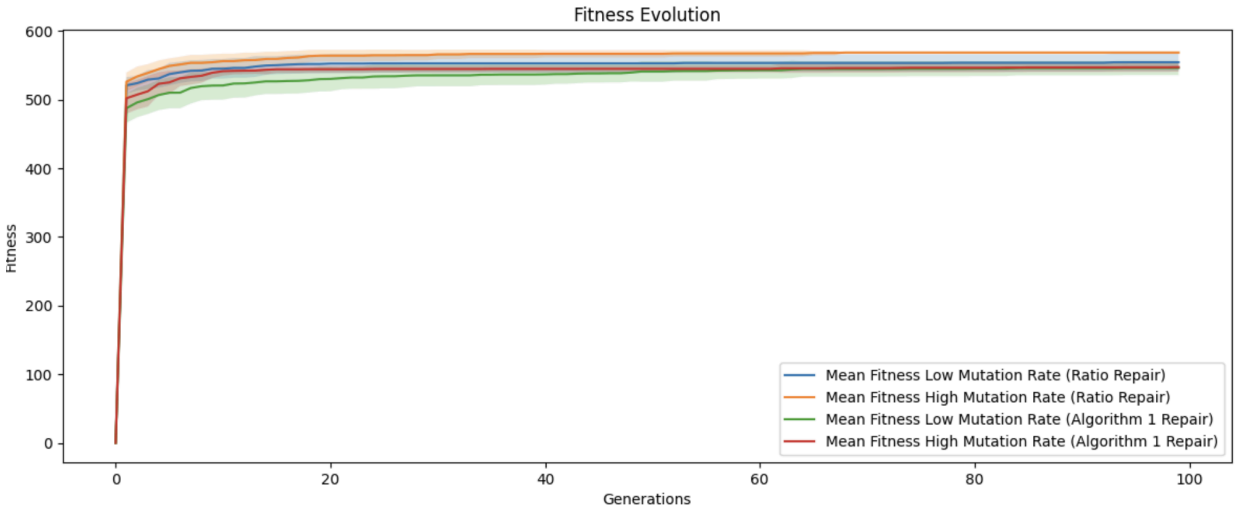
Cette méthode consiste simplement à évaluer le nombre de différences entre nos individus au sein d'une population. Dans notre problème, on peut mesurer le nombre de différences entre deux individus en comptant le nombre d'objets qu'ils ont en commun. Concrètement, on compte le nombre de fois où un objet est présent dans un sac et aussi l'autre puis on fait la moyenne pour chaque génération.

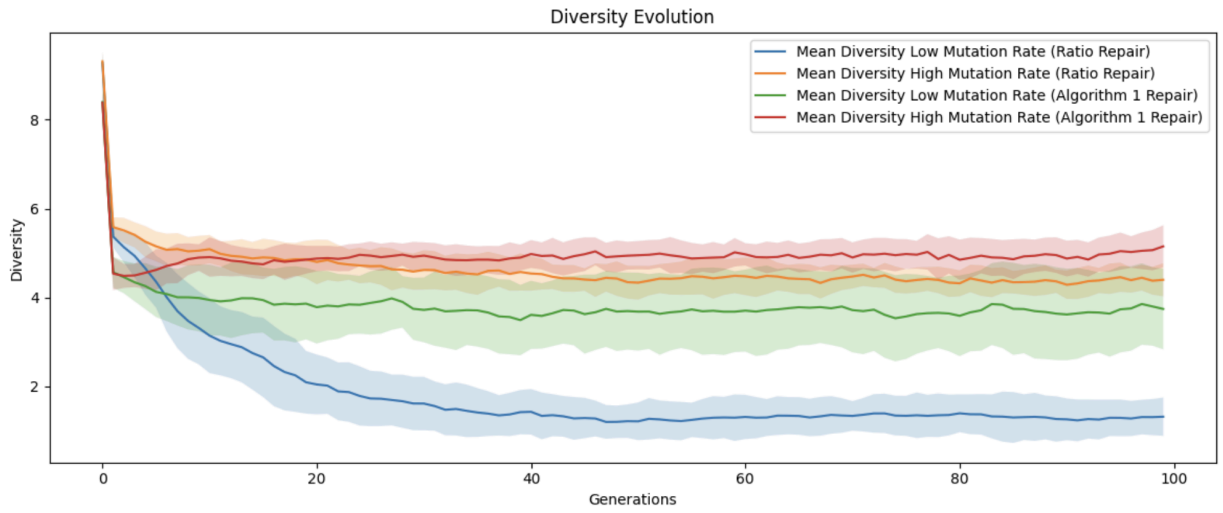
3.2.2 Diversité et taux de mutation

```
population_size = 100
crossover_rate = 0.7
num_generations = 100
budgets = [100, 200, 150, 120, 130]
utilities = np.random.uniform(50, 100, size=50)
costs = np.random.uniform(1, 50, size=(50, 5))
elitism_rate = 0.1
num_instances = 30
```

Avec ces paramètres et un taux de mutation de, nous avons observé un taux de diversité moyen plutôt faible avec les deux méthodes et en observant de plus près les paramètres que nous avons utilisés depuis le début, nous nous sommes rendu compte que le taux de mutation était très bas (0.01) et, aux vues de notre implémentation, pourrait affecter significativement la diversité. De ce fait, nous avons décidé d'étudier l'évolution de l'utilité et de la diversité en fonction du taux de mutation. On étudie avec 0.01 puis 0.1 de taux de mutation.

3.2.3 Résultats





low_ratio = 0.01
high_ratio = 0.1

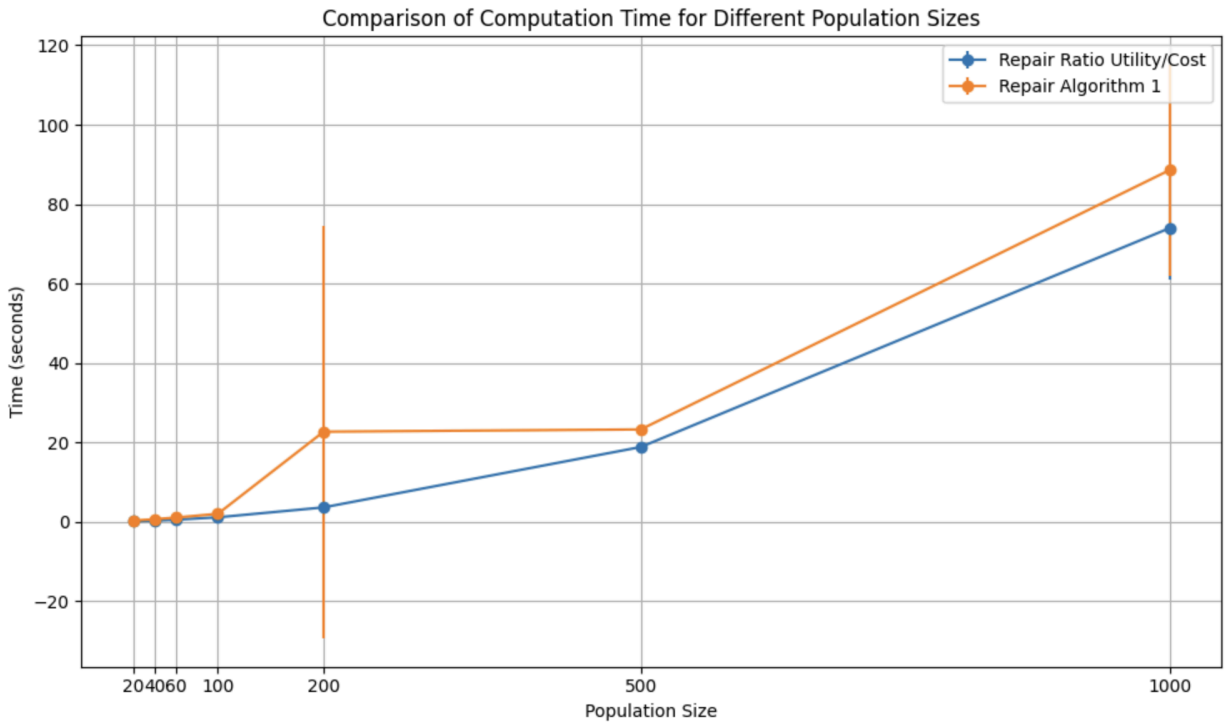
On peut observer une nette amélioration de la diversité en augmentant le taux de mutation ainsi qu'une utilité légèrement meilleure avec notre méthode de réparation. De plus, les écarts-types mettent en avant le caractère plus aléatoire de la méthode du sujet

3.2.3 Résultats

Evolution du temps de calcul

Enfin, nous avons comparé les deux méthodes sur le temps de calcul moyen en fonction de la taille de la population.

Ordinateur utilisé: MacBook Air 2024, Puce M3, 16Go RAM



Nous observons un temps de calcul inférieur avec notre méthode ainsi qu'un écart type significativement moins important (écart type = barres verticales).

Conclusion

Le caractère plus aléatoire de la méthode du sujet permet de ne pas s'étonner des résultats obtenus. En effet, la méthode, sans prendre en compte les caractéristiques des individus (utilités, coûts), se contente de supprimer les objets et d'en rajouter jusqu'à respecter la contrainte budgétaire. Il est alors normal d'observer des temps de calculs dispersés (écarts-types élevés) car un peu de malchance suffit à augmenter le nombre d'opérations de la méthode là où notre méthode prend en compte les coûts des objets dans sa règle de décision et réduit donc les opérations.

A contrario, on observe une diversité légèrement inférieure (mais reste tout de même satisfaisante) avec notre méthode. Ceci n'est pas non plus si étonnant puisque le caractère aléatoire de la méthode du sujet permet plus de diversité.

Enfin, nous pouvons estimer que l'efficacité de notre méthode de réparation n'est pas si spécifique que ça en comparaison avec la méthode du sujet. Le choix entre l'une et l'autre, au vu de nos résultats, peut se baser sur des préférences de diversité, temps de calcul ou encore temps de convergence.

Pour continuer, il serait intéressant de chercher de nouvelles méthodes afin de les combiner avec la nôtre pour continuer d'optimiser la règle de décision de réparation.