Rapport EI Jeux évolutionnaires

Arthur Hakimi, Florian Song, Farès Gati, Lucas Bello–Kern

January 2026

1 Préambule sur les algorithmes d'optimisation

Pour s'attaquer à des problèmes d'optimisation, plusieurs solutions algorithmiques s'offrent à nous. Par exemple faire une recherche exhaustive de l'ensemble des solutions, aussi appelée algorithme brute force, permet d'assurer d'avoir la réponse optimale mais est bien souvent impossible car bien trop coûteux en temps. On utilise aussi des algorithmes déterministes qui permettent d'explorer une partie de l'ensemble des solutions d'un problème en attribuant des heuristiques aux solutions partielles ce qui permet d'obtenir une solution approchée. Cependant, on obtient des complexités qui rendent impossibles l'exploitation de ces algorithmes sur des grands jeux de données.

Ici, on adopte une autre approche, en introduisant du hasard dans notre recherche de solution. L'algorithme stochastique que nous allons utiliser est appelé algorithme génétique car il s'inspire de la théorie de l'évolution : des individus (représentant chacun une solution) sont sélectionnés (sélection naturelle), se reproduisent puis certains individus mutent d'une génération à l'autre. Nous verrons que l'on peut définir différent critères pour définir la manière dont la sélection naturelle agit et différentes méthodes pour faire muter une partie de nos individus. L'avantage de ce type d'algorithme est que sa complexité temporelle dépend uniquement du nombre d'individus générés et du nombre de générations sur lesquelles on fait tourner l'algorithme.

2 Problème: Modélisation de la trajectoire 3D des brins d'ADN

On s'appuie sur un modèle déjà existant établi en 1993. Celui-ci associe à chaque dinucléotide une matrice de transformation dans l'espace (associant rotation et translation). Ainsi, on peut simuler la trajectoire d'un brin d'ADN à partir d'une séquence de nucléotides en multipliant les matrices de transformation correspondant à chaque 2-facteur.

Cependant, ce modèle ayant été développé pour de courts brins d'ADN, il ne prend pas en compte le phénomène de repliement des brins. Notamment, on observe chez les bactéries que les chromosomes et les plasmides (petite séquence d'ADN) sont circulaires, c'est-à-dire que le dernier nucléotide de la séquence est "collé" au premier.

Ainsi, nous avons cherché à adapter les tables de matrices de rotation associés aux dinucléotides à la séquence dont on souhaite modéliser la trajectoire. L'espace de recherche étant infini, nous avons utiliser un algorithme génétique dans lequel chaque individu sera une table de matrices de rotation et le but étant de sélectionner l'individu qui minimise l'erreur de repliement (obtenu en calculant à partir du dernier nucléotide l'hypothétique position du premier nucléotide d'un nouveau cycle puis en faisant la différence entre cette position et la position réelle du premier nucléotide) lors du tracé de la trajectoire.

3 Structure de l'algorithme et implémentation

On utilise un type d'algorithme génétique classique qui suit les étapes suivantes:

- Initialisation: On part d'une génération 0 de N individus générés aléatoirement
- Évaluation (classe `Fitness`): On évalue chaque individu avec la fonction d'évaluation que l'on cherche à minimiser.
- Sélection (classe `Selection`): On sélectionne, en introduisant ou non de l'aléatoire, une partie de nos individus selon certaines règles qui dépendent de leur évaluation.
- Reproduction (classe `Crossover`): On choisit deux parents parmi les individus sélectionnés et on construit un enfant avec les règles de construction qu'on a défini.
- Mutation (classe `Mutation`): On tire au sort certains individus (parfois uniquement les enfants) que l'on modifie légèrement.

On réitère les étapes après l'initialisation autant de fois que l'on veut de générations.

Pour chacune des étapes de l'algorithme (sauf l'évaluation), on implémente des classes abstraites. Ces classes donnent la structure de base de chacune des étapes, et permet d'avoir un algorithme génétique prenant en paramètres une méthode de sélection, une méthode de reproduction et une méthode de mutation. Ainsi le code est extrêmement modulaire car on peut changer très facilement les techniques utilisées.

3.1 Contraintes du problème

Les paramètres sur lesquelles on joue sont ceux de la matrice de rotation. Hors, on ne peut pas modifier ces paramètres à notre guise. On doit respecter certaines contraintes sur ces paramètres.

Premièrement, pour chaque valeur des matrices de rotation, on a un écart maximal à une valeur donnée dans `defaultRotTable` à respecter. Ainsi, dans le code de toute mutation (grâce aux classes abstraites), on fait bien en sorte de ne jamais sortir de cet intervalle permis. Deuxièmement, le fait que l'ADN est une double hélice impose des conditions supplémentaire sur les matrices de rotation. En effet, lorsqu'on parcourt l'ADN, le parcours fait par la branche de gauche et le parcours fait par la branche de droite sont censés amener à la même destination. Ainsi, puisque dans l'ADN, une nucléotide A est toujours associée à une nucléotide T, et une nucléotide C est toujours associée à une nucléotide G, les valeurs d'une matrice de rotation associées aux dinucléotudies "CC" et "GG", "AC" et "GT", etc doivent être égales.

3.2 Évaluation

Dans notre problème, on cherche à minimiser la distance entre le premier point de la trajectoire, qui est par convention en $(0, 0, 0)$ et l'hypothétique position du premier nucléotide si on la calculait à partir du dernier nucléotide en appliquant une matrice de rotation. On cherche donc à minimiser la norme de ce point. C'est pour cela que l'on prend comme fonction d'évaluation (appelé `Fitness` dans le code) la norme euclidienne de $\mathbb{R}^3$. De plus, on essaie de minimiser l'écart d'angle entre l'angle initial au niveau du premier point, et l'angle en sortie du dernier point. Ainsi, on ajoute à la norme euclidienne précédente la norme de l'écart entre la position du deuxième nucléotide et l'hypothétique position du deuxième nucléotide après avoir fait un tour complet. (En réalité on prend l'opposé de cette somme dans notre code pour avoir une fonction à maximiser et non à minimiser)

3.3 Sélections

On a implémenté 5 types de sélection pour les comparer et observer la convergence de la population de générations en générations.

- Sélection par élitisme `class Elitism`: On garde un pourcentage des individus qui ont les meilleures évaluations.
- Sélection par tournoi `class TournamentSelection`: On compare deux individus aléatoires de la population et on garde le meilleur des deux. On s'arrête quand on a sélectionné un nombre suffisant d'individus préalablement défini.
- Sélection par tournoi avec espoir `class TournamentWithHopeSelection`: C'est une sélection par tournoi mais il y a une chance qu'on choisisse le pire plutôt que le meilleur.
- Sélection par roulette `class RouletteSelection`: On effectue n tirages sans remise, chaque individu ayant une chance d'être pris qui correspond à la normalisation de l'exponentiel de son score.
- Sélection par rang `class RankSelection`: On choisit un pourcentage défini de la population et chaque individu a une chance d'être pris qui correspond à son rang sur la somme totale des rangs.

Par ailleurs on conserve toujours le meilleur sans qu'il passe de sélection.

3.4 Reproduction

Pour la reproduction (correspondant à la classe `Crossover` dans le code), on a implémenté plusieurs stratégies:

- Moyenne `MeanCrossover`: La valeur des coefficients de la matrice de rotation d'un enfant est la moyenne des coefficients des matrices de rotation des parents. Intuitivement, on obtient un enfant qui mixe le comportement des deux parents, en faisant un entre-deux des parents.
- Moyenne pondérée par l'évaluation `FitnessWeightedMeanCrossover`: On utilise le même procédé que dans moyenne, mais les coefficients de la matrice de rotation de l'enfant sont plus proches de l'individu ayant une meilleure évaluation.
- Choix entre les valeurs des parents `ChooseBetweenParentsCrossover`: Pour chaque coefficient de la matrice de rotation de l'enfant, on prend soit la valeur du premier parent, ou celle du deuxième parent. Ainsi on calque la reproduction humaine qui transmet les gènes soit de la mère, soit du père.

3.5 Mutations

Pour les mutations, on a implémenté de nombreuses stratégies:

- `Mutation`: classe abstraite qui permet d'avoir une structure commune à toutes les mutations:
  - `mutateValue` (abstraite): modifie la valeur d'un seul coefficient, prend en arguments `e` la valeur actuelle de ce coefficient, `delta` l'écart type associé à ce coefficient (donné dans `table.json`) ainsi que `logFit`, le logarithme en base 10 de 1 plus l'opposé de la fitness
  - `mutate`: modifie un individu, prend en arguments `individu` l'individu à muter, et `fitness` le score de cet individu

  - `mutatePopulation`: modifie une population entière, prend en arguments `population` la liste d'individus à muter, et `fitnesses` le score associé à chacun de ces individus
  - `__str__` (abstraite): permet d'afficher sous forme de chaîne de caractères le nom de la mutation (utile pour les benchmarks)

- `GaussianMutation`: classe abstraite fille de `Mutation` qui permet d'avoir une structure commune à toutes les mutations gaussiennes pour en faciliter l'implémentation
  - `__init__`: prend en argument `sigma` qui représente l'écart-type de la gaussienne et stocke cette valeur
  - `name` (abstraite): représente le type de mutation gaussienne sous forme de chaîne de caractères
  - `__str__` (abstraite): utilise `name` et `sigma` pour afficher le nom de la mutation sous forme de chaîne de caractères
  - `gaussian`: retourne une valeur aléatoire suivant une loi gaussienne centrée d'écart-type `sigma` (ces valeurs aléatoire sont bien évidemment indépendantes pour chaque individu et pour chaque valeur de la table de rotations pour un individu)

- `GaussianAdditiveMutation` (représentée par `G+`): classe fille de `GaussianMutation` qui ajoute une valeur de `gaussian` à tous les coefficients de l'individu. L'intérêt de cette mutation est qu'elle reste en général proche de la valeur d'entrée

- `GaussianAdditiveDeltaMutation` (représentée par `G+`Δ): classe fille de `GaussianMutation` qui ajoute une valeur de `gaussian` multiplié par Δ l'écart-type associé à ce coefficient à tous les coefficients de l'individu. Cette méthode à les

- `GaussianMultiplicativeMutation` (représentée par `G*`): classe fille de `GaussianMutation` qui multiplie par `exp(gaussian)` tous les coefficients de l'individu. Ceci permet de prendre en compte directement l'échelle de chaque valeur (cette approche s'est révélée moins utile, peut être qu'elle serait meilleure avec des valeurs "non bornées")

- `GaussianAdditiveDeltaLog10FitnessAnnealedMutation` (représentée par `G+`ΔF): classe fille de `GaussianMutation` qui ajoute une valeur proportionnelle à la valeur de `gaussian`, à l'écart-type de la variable considérée ainsi qu'à `logFit`, qui correspond au logarithme en base 10 de 1 plus l'opposé du score de l'individu considéré (la distance entre le premier et le dernier nucléotide de la chaîne, qualifiant donc la qualité d'un repliement)

- `ThresholdMutation` (représentée par `T`): classe fille de `Mutation` est une "métamutation", elle prend en entrée `mutation_probability` et une mutation et n'applique la mutation donnée en arguments qu'avec une probabilité `mutation_probability`. L'intérêt de cette mutation est qu'elle permet de modifier moins souvent les individus, indépendemment des mutations en elles-mêmes

- `SimulatedAnnealingMutation` (représentée par `SA`): classe fille de `Mutation` est elle aussi une "métamutation", elle prend en entrée une mutation, une clé `key` et un coefficient α. Cette multiplication applique la mutation sous-jacente, en multipliant l'attribut `key` de cette mutation par α après l'évaluation de chaque population. Par exemple pour une mutation gaussienne, avec `key`=σ, l'écart-type de la gaussienne pour la génération t sera $\sigma = \sigma_0 * \alpha^t$. C'est la méthode dite du recuit simulé.

3.6 Paramètres supplémentaires

Pour affiner nos méthodes de sélection et de mutation, nous avons introduit des paramètres généraux:

- `keepRate`: c'est la proportion de la population sera sélectionné par l'algorithme de sélection.
- `duplicateRate`: c'est le taux de la population finale qui est directement muté à partir des individus obtenus après `keepRate`, sans se reproduire (ils peuvent aussi se reproduire ensuite)
- `saltRate`: on ajoute un certain pourcentage de population totalement aléatoire pour rajoute de la diversité dans les codes génétiques (peut s'apparenter biologiquement à de l'immigration)

3.7 Tests

Des tests unitaires ont été réalisés sur chaque classe afin de s'assurer du bon fonctionnement de chaque méthode implémentée. On se retrouve avec une couverture du code supérieur à 92% pour chaque classe. Les lignes non recouvertes correspondent majoritairement à des méthodes abstraites dont la méthode précise se trouve dans les sous-classes vérifiées. Par ailleurs, random.seed a été utilisé dans les tests pour les rendre reproductibles.

- Test des sélections `test_selection.py`:
  - Tests généraux: On teste la taille de l'échantillon sélectionné par chaque méthode de sélection, ainsi que la validité des individus choisis. On vérifie également si le meilleur individu est systématiquement sélectionné (`test_size_of_population`, `test_best_individual_is_always_selected`, `test_indices_are_valid`).
  - Pour chaque type de sélection, on vérifie si les meilleurs individus ont plus de chances d'être choisis que les moins performants (`test_tournament_favors_best`, `test_roulette_favors_best`, `test_rank_favors_best`).
  - Dans le cas de l'élitisme, on cherche à confirmer que les meilleurs individus sont bien sélectionnés (`test_elitism_select_exact_best`), tandis que pour le tournoi avec espoir, on cherche à confirmer la possibilité qu'un individu moins performant puisse être choisi avec une certaine probabilité (`test_tournament_with_hope_allows_worst`).
- Test des mutations `test_mutation.py`:
  - On vérifie d'abord que les valeurs d'angles supérieures ou égales à 2 ne sont pas modifiées et qu'on crée un nouveau individu (`test_above_threshold_are_not_mutated` et `test_mutation_does_not_modify_original_individual`).
  - On s'assure que la population mutée conserve bien sa taille (`test_mutate_population_preserves_size`.
  - Les différents types de mutation sont testés afin de vérifier que les individus mutés sont valides autrement dit les valeurs des angles restent dans un intervalle considéré (`test_mutations_GAM`, `test_mutations_GADM`, `test_mutations_GADLogM`, `test_mutations_GMM`).
  - Dans le cadre du recuit simulé, on contrôle également que le paramètre sigma diminue d'un facteur alpha à chaque génération (moins de probabilité de mutation au cours du temps) (`test_alpha_decay`).

  - Enfin, on vérifie que lorsque la probabilité de mutation est nulle (`test_threshold_mutator_zero_probability`), l'individu n'est pas muté, et que lorsqu'elle vaut un, l'individu est systématiquement muté (`test_threshold_mutator_full_probability`).

- Test de crossover `test_crossover.py`:
  - On a d'abord effectué des tests généraux, en vérifiant notamment le nombre d'enfants créé après croisement et en s'assurant que lorsqu'il n'y a pas de population initiale, aucun descendant n'est généré (`test_population_size`, `test_empty_population`).
  - On vérifie également que les enfants possèdent exactement les mêmes clés que les parents et que les longueurs des listes de la table de rotation sont bien conservées (`test_rot_table_keys_preserved` et `test_rot_table_lengths_preserved`).
  - Pour la méthode par la moyenne, on vérifie que les valeurs des angles de l'enfant sont toujours comprises entre celles des deux parents (`test_mean_values_are_between_parents`).
  - Pour une méthode où l'individu est davantage influencé par le parent ayant la meilleure fitness, on s'assure que les valeurs des angles de l'enfant sont bien plus proche de ceux du parent le mieux évalué (`test_best_fitness_parent_influences_more`).
  - Enfin, pour la méthode consistant à choisir les valeurs des angles parmi les deux parents, on contrôle que chaque valeur d'angle de l'enfant correspond bien à l'une des valeurs des deux parents (`test_child_values_come_from_parents`).

- Test RotTable `test_RotTable.py`: On a gardé le test proposé dans le github duquel on est parti sauf qu'au lieu de construire une instance à partir d'un fichier JSON, on convertit le fichier JSON dans un dictionnaire et on construit une instance à partir d'un dictinnaire.

Remarque: la propriété de la symétrie et la complémentarité des brins d'ADN sont respectés sans tester car les modifications ont été directement réalisées dans `RotTable.py` et `Traj3D.py`

4 Résultats et améliorations

Pour visualiser nos résultats et ainsi avoir des pistes d'amélioration, nous utilisons des fonctions benchmark qui testent différentes combinaisons de mutations et de sélections pour différentes tailles de population.

4.1 Premiers résultats

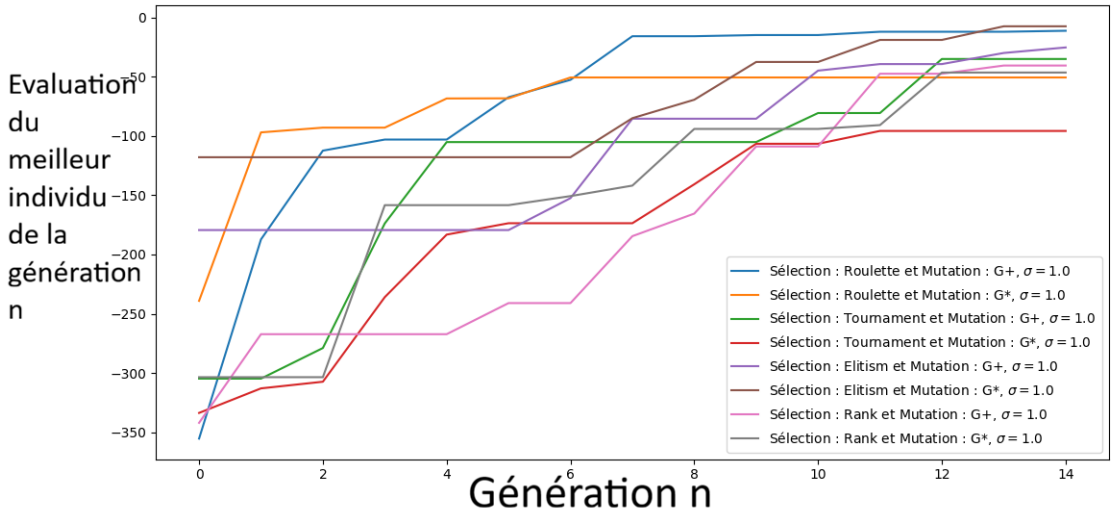

Figure 1: Tracé de l'évaluation du meilleur individu en fonction de la génération pour chaque couple (sélection,mutation)

Pour comparer nos 4 premières méthodes de sélection et nos 2 premières méthodes, nous avons tracé l'évaluation du meilleur individu en fonction de la génération pour chaque association de sélection-mutation avec une simulation de 1024 individus sur 16 générations sur le plasmide de 8000 nucléotides. Nous avons alors constaté que les associations (roulette, gaussienne additive) et (élitisme, gaussienne multiplicative) avait pour l'instant les meilleurs performances. Nous avons alors obtenu 7 Angstrom pour l'erreur de repliement en utilisant la meilleure table de rotation. C'est un résultat prometteur (un écart de 7 Angstrom correspond à l'écart entre deux nucléotides dans un brin d'ADN) mais nous avons exploité nos résultats pour améliorer notre algorithme génétique.

4.2 Optimisation

La première optimisation effectuée a été faite sur le fichier `Traj3D.py`. On remarque que initialement, lors du calcul d'une trajectoire, on effectuait 6 multiplications de matrices pour appliquer la transformation d'une dinucléotide sur la position actuelle. Or, sur ces 5 matrices, deux sont des constantes `__MATRIX_T`, et trois dépendent seulement de la dinucléotide en question `Rz` (apparaissant 2 fois) et `Q`. Ainsi, on peut précalculer pour chaque dinucléotide le produit matriciel entre ces 5 matrices. Lorsqu'on applique la transformation liée à une dinucléotide, on ne fait plus qu'une multiplication matricielle au lieu de 5, réduisant grandement les calculs.

Ensuite pour améliorer notre algorithme génétique, on a implémenté de nouveaux types de mutation et de sélection. Ces types ont été évoqués précédemment et donc leur définition a déjà

été donné.

Enfin, on a ajouté les paramètres `keepRate`, `duplicateRate` et `saltRate` pour pouvoir plus facilement changer ces paramètres généraux.

4.3 Résultats finaux

Pour comparer les différentes méthodes, on a utilisé une fonction de tests nommée `benchmark` qui permet de tester de nombreuses combinaisons de méthodes de sélection, reproduction, mutation en modifiant les différents paramètres de ces méthodes.

4.3.1 Benchmark Sélection

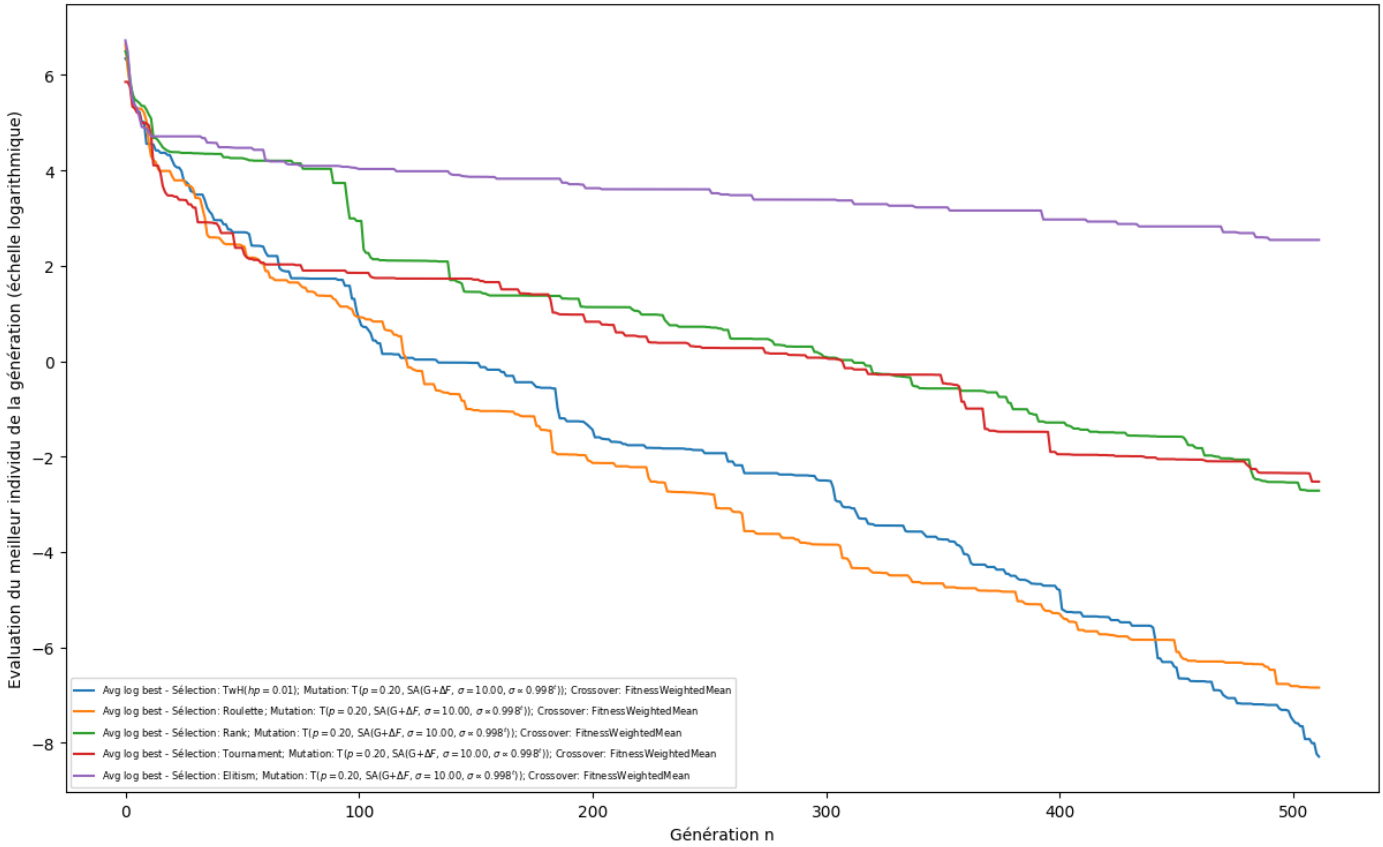

Figure 2: Benchmark des différents types de sélection

Ci-dessus est figuré le résultat des tests sur les différentes méthodes de sélection pour une population de 64 individus sur 512 générations, un `keepRate` de $\frac{1}{8}$, un `duplicateRate` de $\frac{1}{8}$ et un `saltRate` de $\frac{1}{64}$. La méthode de mutation utilisée est `GaussianAdditiveDeltaLog10FitnessAnnealedMutation` avec un paramètre de recuit simulé de 0.998 agrémentée de la métamutation `ThresholdMutation`

qui impose que 20% de la population soit mutée. La méthode de reproduction utilisée est `FitnessWeightedMean`.

On remarque que les méthodes de sélection les plus efficaces sur ce test sont **Roulette** et **Tournoi avec espoir** avec `p = 0.01`, avec des performances significativement meilleurs que les autres méthodes de sélection. Cela se comprend par le fait que ces sélections permettent de conserver un minimum d'aléatoire et ainsi de ne pas tomber dans des mauvais minimas locaux en continuant à explorer jusqu'à tomber dans l'un des meilleurs minimas locaux.

4.3.2 Benchmark Crossover

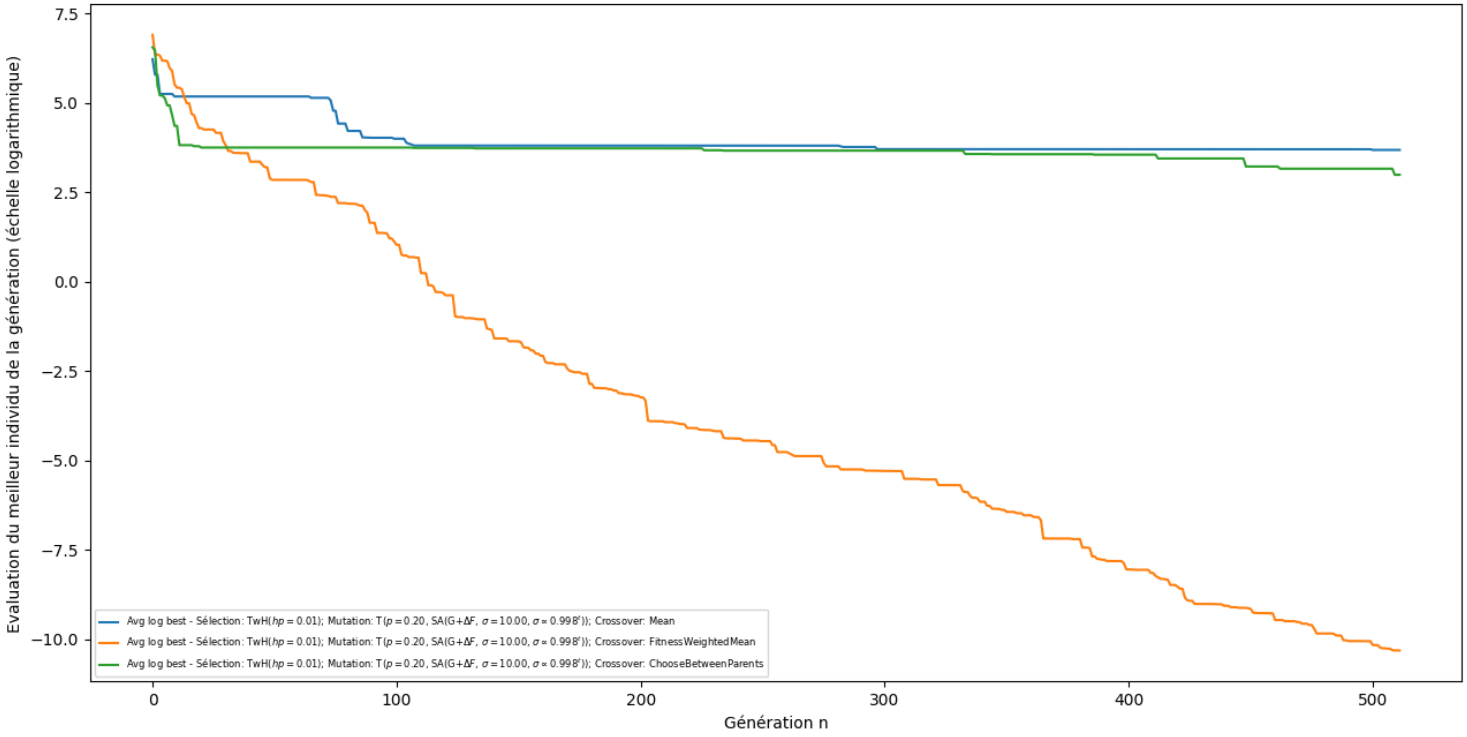

Figure 3: Benchmark des différents types de Crossover

Ci-dessus est figuré le résultat des tests sur les différentes méthodes de sélection pour une population de 64 individus sur 512 générations, un `keepRate` de $\frac{1}{8}$, un `duplicateRate` de $\frac{1}{8}$ et un `saltRate` de $\frac{1}{64}$. La méthode de mutation utilisée est `GaussianAdditive` `DeltaLog10FitnessAnnealedMutation` avec un paramètre de recuit simulé de 0.998 agrémentée de la métamutation `ThresholdMutation` qui impose que 20% de la population soit mutée. La méthode de sélection utilisée est Tournoi avec Espoir avec $p = 0.01$.

On constate que la méthode de reproduction `FitnessWeightedMean` est bien plus performante que les 2 autres.

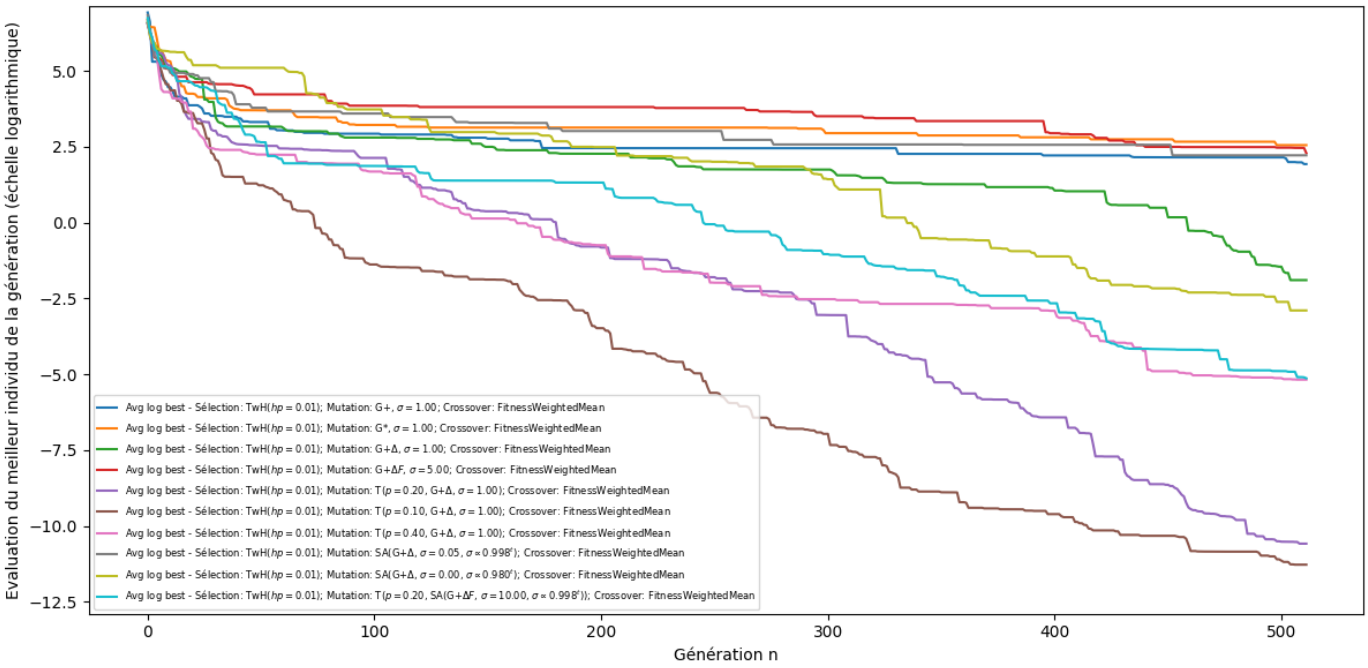

Figure 4: Benchmark des différents types de mutation

4.3.3 Benchmark Mutation

Ci-dessous est figuré le résultats des tests sur les différentes méthodes de mutation un `keepRate` de $\frac{1}{8}$, un `duplicateRate` de $\frac{1}{8}$ et un `saltRate` de $\frac{1}{64}$. La méthode de sélection utilisée est Tournoi avec espoir avec `p = 0.01`. La méthode de reproduction utilisée est `FitnessWeightedMean`.

On remarque que la meilleure méthode de mutation sur ce test est `GaussianAdditiveDelta` avec une proportion mutée de la population de 10% par génération. Cependant dans des expérimentations sur plus de générations, nous avions constaté de meilleurs résultats en y ajoutant la méthode du recuit simulé avec $\alpha = 0,998$.

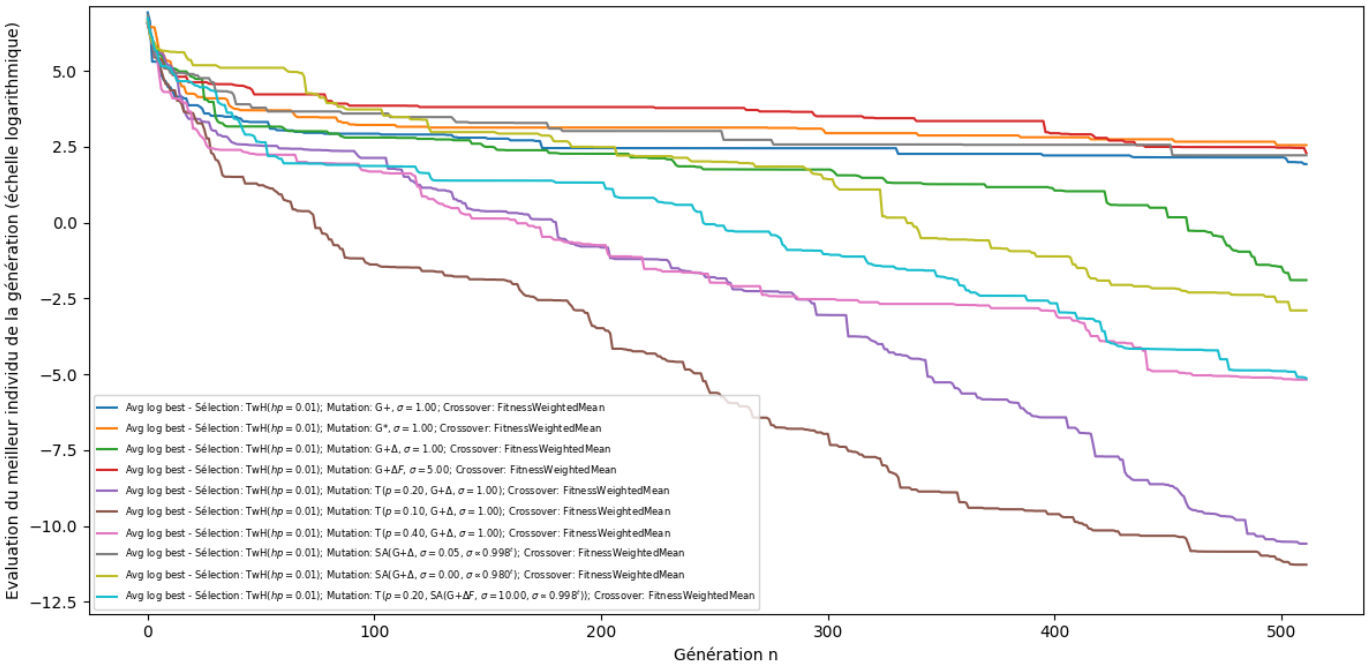

Figure 5: Benchmark des différents types de mutation

5 Annexes

5.1 Couverture des tests

```
_____ coverage: platform linux, python 3.13.5-final-0 _____

Name              Stmts   Miss  Cover   Missing
-----------------------------------------------
gen/selection.py     86      7    92%   9, 13, 20, 36, 61, 81, 116
-----------------------------------------------
TOTAL                86      7    92%
========== 7 passed in 0.71s ==========
```

Figure 6: Couverture de Sélection

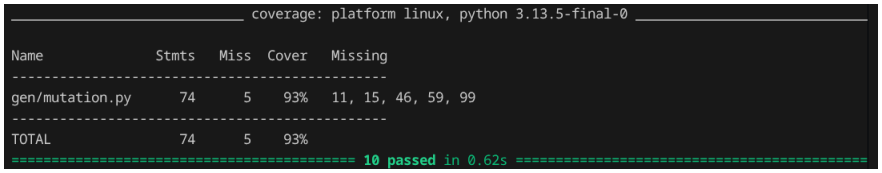

Figure 7: Couverture de Mutation

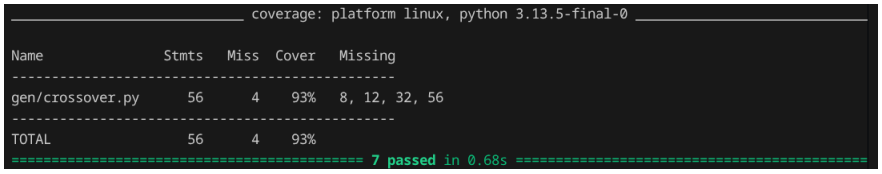

Figure 8: Couverture de Crossover

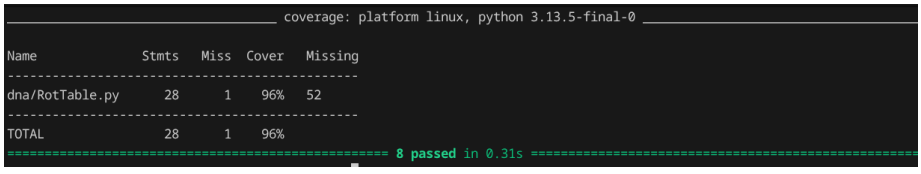

Figure 9: Couverture de RotTable