

**Rapport – Enseignement d’intégration**

**ST2 (Théorie des Jeux) - EI Algorithmique Génétique**

**Réalisé par le groupe 2 :**

Arthur Leene

Ilyess Doragh

Lucas Béziers

Ameur Echaabi



Janvier 2022

I – Algorithme génétique

Introduction

Nous avons utilisé la programmation orientée objet lors de la réalisation de notre algorithme afin qu’il reste le plus générique possible. Ainsi, avant de décrire son fonctionnement, il nous semble important de préciser les classes utilisées par le programme. Les classes utilisées sont :

* RotTable : Donnée par l’énoncé, nécessite de créer des méthodes pour écrire/accéder aux valeurs de la matrice associée
* Traj3D : Donnée par l’énoncé, représente une trajectoire 3D
* Individu : Correspond à un individu qu’on va pouvoir sélectionner, croiser avec les autres et muter grâce aux méthodes associées
* Population : C’est l’ensemble des individus, avec notamment l’attribut population qui est une liste des individus
* Evaluation : Permet d’affecter un score à chaque individu (méthode décrite plus bas)
* Algo\_gen : Algorithme complet qui est exécuté dans Main.py

Les variables d’entrées de l’algorithme sont :

* N : Le nombre d’individus dans la population (reste constant au cours du temps)
* T : Le nombre de passages dans le cycle sélection/croisement/mutation

Tout le long de l’algorithme, il est nécessaire de conserver les symétries de la matrice de rotation. En effet, certains dinucléotides sont appariés (même Twist, même Wedge, Direction opposées), ce sont les paires :

(AA,TT);(AC;GT);(AG,CT);(CA,TG);(CC,GG);(GA,TC)

Évaluation

L’objectif de l’algorithme est de trouver modifier les valeurs de la table de rotation originale afin de rendre le plasmide circulaire, i.e. les deux extrémités sont “collées” l’une à l’autre. Il est donc important de pouvoir évaluer à l’aide de la classe Evaluation à chaque étape chaque individu afin de savoir s’il se rapproche de cet objet ou non. Ainsi, notre fonction d’évaluation utilise trois critères :

* D : La distance entre le premier et le dernier nucléotide de la chaîne car elle décrit si la chaîne s’approche d’une boucle
* Ps : Le produit scalaire entre le vecteur correspondant au premier dinucléotide et le vecteur du dernier nucléotide de la chaîne car il rend compte de la qualité du raccordement de la chaîne

L’algorithme va donc chercher à faire diminuer le score des individus noté S = f(D,Ps) tel que:

Note :

Nous avons essayé d’utiliser Nc le nombre d’angles critiques, i.e. le nombre d’angles étant plus aigus (donc plus petits) qu’un angle choisi (dans notre cas) car il permet de juger si la courbe semble régulière mais il est trop dépendant de la chaîne choisie : si une séquence de 3 nucléotides forme un angle critique, le compteur augmentera à chaque fois que cette séquence apparaîtra.

Fonctionnement détaillé

Initialisation

On commence par générer une population de N individus avec des paramètres aléatoires (dans les intervalles autorisés), ce qui est permis par la méthode generate() de la classe RotTable : ce sera notre population initiale.

Sélection

Nous avons opté pour une sélection par tournoi : chaque individu va affronter un adversaire tiré aléatoirement dans la population et seul le gagnant sera sélectionné pour la suite. Cependant, si la taille de la population est impaire, il y aura un individu seul qui n’affrontera personne. Pour pallier ce problème, on utilise la méthode get\_max() : celle-ci parcourt la population et renvoie le couple (indice dans la liste,S,D) de l’individu ayant le pire score. On supprime donc le pire individu de la population pour avoir une taille paire.

On crée ensuite une nouvelle population vide appelée new\_population. Tant que population n’est pas vide : on sélectionne le premier individu de la liste et un autre aléatoirement (afin de rendre le tournoi utile pour conserver de la diversité génétique) et on les fait s’affronter. On les évalue, on sélectionne celui qui a le plus petit score, on l’ajoute à new\_population et on supprime ensuite les deux individus de population. Lorsque new\_population est totalement générée (i.e. quand population est vide), on recopie new\_population dans population.

Croisement

Le croisement entre deux individus est une méthode de la classe Individu qui prend en entrée le deuxième individu du couple et renvoie les deux enfants. Afin de conserver les symétries de la matrice de rotation, nous avons opté pour un croisement en 3 points pour les valeurs associées tel que Enfant1 (Enfant2 possédant les gènes “opposés”) :

* partage avec Parent1 : (AA,TT);(AC,GT);(AG,CT) et CG;GC
* partage avec Parent2 : (CA,TG);(CC,GG);(GA,TC) et TA;AT

Le croisement sur toute la population correspond à la méthode croisement\_global(). On parcourt la moitié de la population (ou partie entière de la moitié si la taille est impaire) et on réalise un croisement entre un individu de la population et un individu généré aléatoirement puis on ajoute à population les deux enfants. Si la population après sélection avait une taille impaire, on rajoute un individu généré aléatoirement pour obtenir une population de taille paire. Enfin, si la population initiale (avant sélection) avait une taille impaire, on rajoute un individu généré aléatoirement pour retrouver la taille d’origine.

Note:

Initialement nous souhaitions réaliser le croisement entre deux individus de la population mais nous avons remarqué des cas de “consanguinité”, c’est-à-dire qu’à force de reproduire les individus entre eux et entre leurs enfants, la population tendait vers une uniformisation des individus et donc empêchait la minimisation du score.

Mutation

La mutation d’un individu est une méthode de la classe Individu. Chaque gène parmi les 32 (16 nucléotides contenant 2 valeurs modifiables) a une chance Pm distribuée uniformément entre alpha et beta (ici 0.001 et 0.01), sachant que si un gène mute, son gène apparié mute aussi.

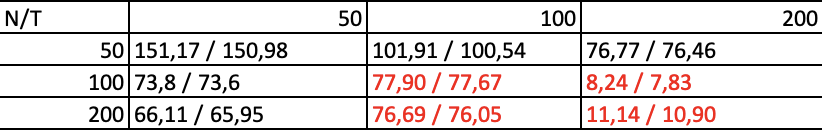
Pour cela, on crée une liste de taille 32 qui contient des valeurs lp[i] uniformément distribuées entre 0.001 et 0.01. Ensuite, on parcourt la liste et on réalise une épreuve de Bernoulli de probabilité lp[i] : si elle réussit, on attribue au gène i une valeur alétoire distribuée uniformément dans l’intervalle possible. Enfin, on vérifie que chaque gène est correctement apparié en attribuant à son gène apparié sa valeur.

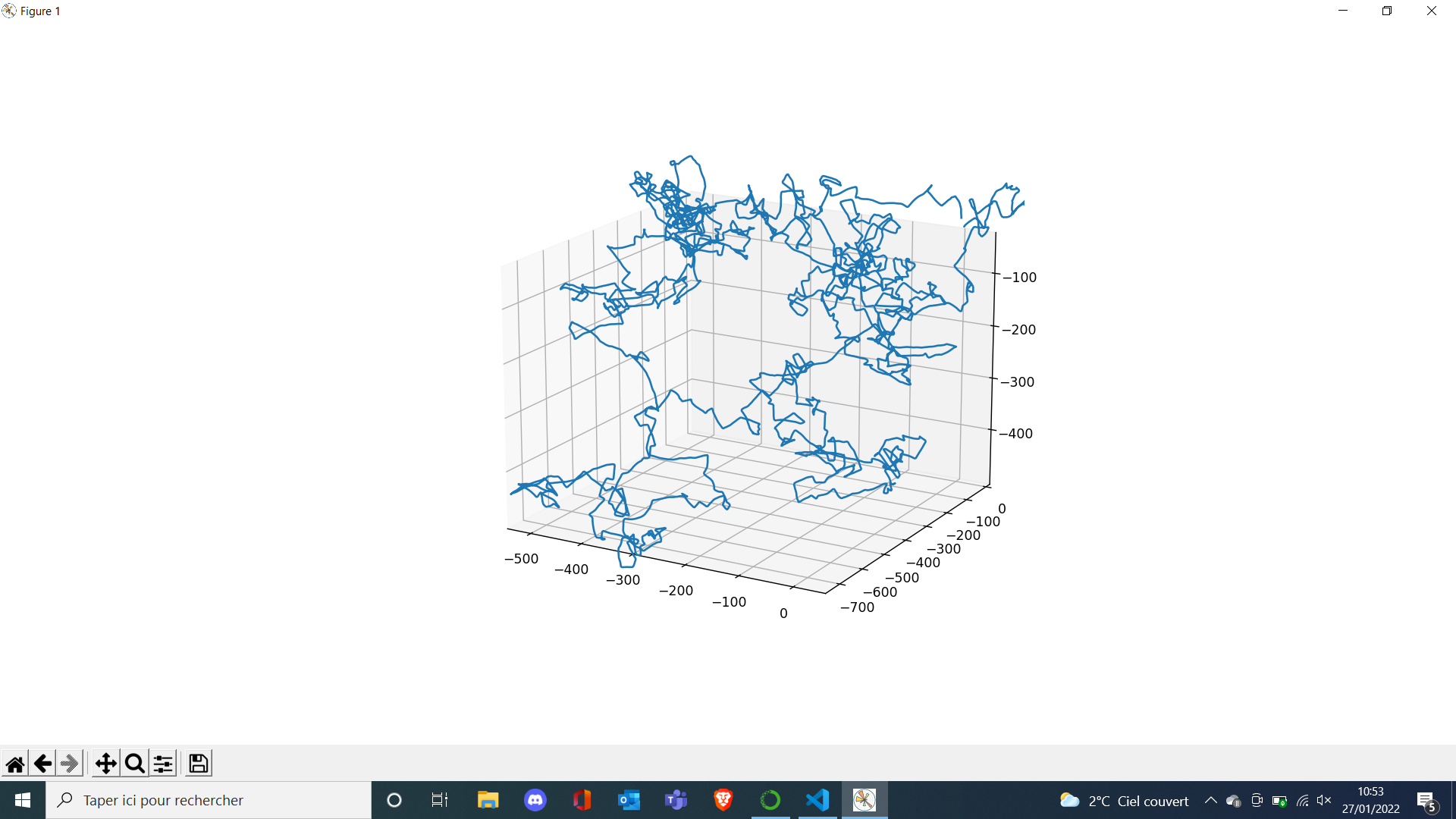
La mutation réalisée sur l’ensemble de la population est la méthode mutation\_Afin de toujours

conserver le meilleur individu, on décide d’appliquer la méthode mutation() à chaque individu de la population sauf à celui qui a le meilleur score (grâce à la méthode get\_min()). Cette méthode mutation\_globale().

Résultats

Tests pour N (nombre d’individus dans une population) et T (nombre de cycle), les valeurs en rouge n’ont été réalisées qu’une seule fois, celles en noir sont des moyennes de 3 simulations.





*Résultat avec une distance D=15.265 (N=30,T=300) pour le plasmid\_8k*

L’algorithme permet de trouver une solution beaucoup plus proche qu’avec la table de rotation originale. Sachant que ce problème n’est pas à traiter en urgence, on peut se permettre d’attendre plusieurs minutes avant d’avoir une solution.

II – Algorithme de type Monte-Carlo Tree Search

**Principe de l’algorithme**

L’algorithme de Monte Carlo est un algorithme d’optimisation qui permet, depuis une configuration initiale, de créer les configurations suivantes ainsi que de les évaluer. Il va ensuite effectuer un parcours de graphe afin de chercher les solutions les plus prometteuses et en générer les configurations suivantes.

L’algorithme de parcours de graphe de Monte Carlo se base initialement sur le ratio de partie réussie / partie simulée, ici, l’indicateur permettant d’évaluer quel sommet est le meilleur sera la distance entre les deux extrémités de la chaîne de nucléotides.

**Fonctionnement détaillé**

**Initialisation**

* L’algorithme MCTS crée la racine du graphe à l’aide de la configuration initiale qui lui est donnée. La structure adoptée ici pour un sommet est la suivante :
  + node = {‘table’: rot\_table, ‘children’:{‘node1’:{..}, ’node2’:{..},..}, ‘parent’: parent\_node, ‘score’: score }
* L’étape suivante est la création de la première génération de fils
  + Fonction gen()
* Puis l’évaluation de tous les fils
  + Fonction eval\_and\_update()

**Récursivité**

* On procède à la sélection : On part de la racine et on descend en profondeur dans l’arbre en choisissant le nœud avec le score le plus faible à chaque étage.
  + Fonction selection()
* Une fois arrivé à la profondeur de nœuds terminaux :
  + S'il y a un nœud terminal non évalué, on l’évalue (même évaluation que pour l’algo gen)
  + S'ils sont tous évalués, on choisit celui avec le score le plus faible
* On procède à la génération depuis ce nœud :
  + Ici la génération se fait par distribution uniforme sur domaine de définition des coefficients en sachant que chaque profondeur correspond à la modification de 1 coefficient
  + Une autre approche aurait été de modifier chaque gène à chaque sommet fils, ceci aurait permis de mieux cerner l’influence d’un gène.
* On évalue un nœud de cette nouvelle génération et on met à jour son score ainsi que celui de tous ses ascendants. Pour évaluer les nœuds :
  + Pour le nœud terminal on fait N simulations (N étant une entrée pour MCTS) aléatoires en faisant varier les coefficients encore non fixés, on en prend ensuite la moyenne.
  + Pour un nœud qui a des enfants, on fait la moyenne des scores des enfants évalués.
* On reprend le procédé.

**Structure du code**

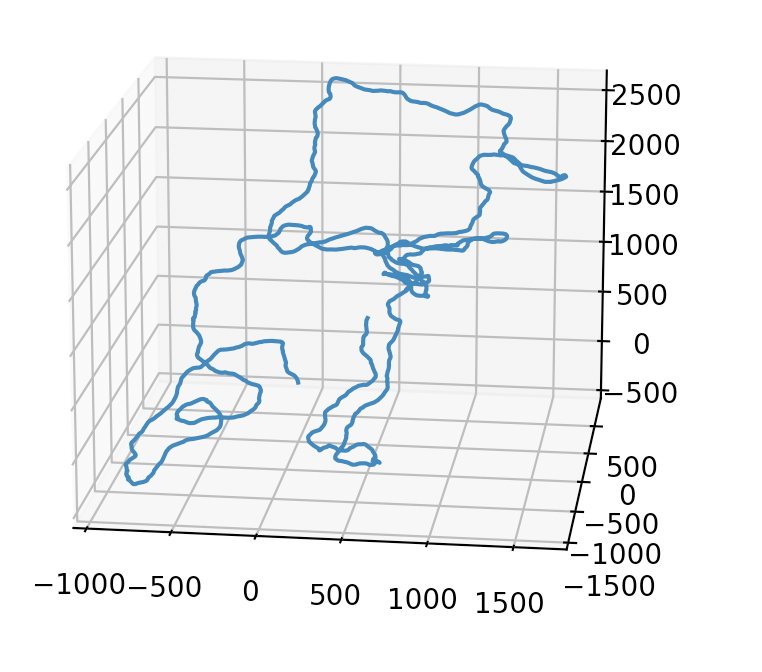
On a donc dans MCTS les fonctions :

* gen(node)
* Eval\_and\_update(node)
* Selection()
* Process() (fonction qui exécute tout l’algorithme et renvoie le meilleur sommet trouvé)

Puisque cet algorithme est entièrement générique, il lui est nécessaire de faire appel à une autre classe qui agira sur les objets traités dans les sommets (ici ce sont des rot\_table sous forme de dictionnaire mais cela aurait pu être une matrice par exemple). En l’occurrence, on utilisera la classe Table. Cette dernière dispose des méthodes suivantes :

* Get\_depth() : permet de connaitre la profondeur dans l’arbre d’un sommet
* Children() : crée une table fils en fonction de la profondeur de la table (Modifie seulement les coefficients encore non traités)
* Gen\_random() : génère une table aléatoire sans modifier les coefficients déjà choisis.

Par souci de temps, l’algorithme présente quelques défauts qui font que le meilleur score renvoyé obtenu est d’environ 400 :



*Meilleur résultat pour l’algorithme de MonteCarlo*