Project Report

Algorithmes Génétiques pour la biologie

Enseignement d'intégration Théorie des Jeux

Projet développé et présenté par

Ouadjou Tarik
Toro Javier
Santos Da Rosa Alexsandro
Fretault Quentin

Sous la supervision de

Joan Hérisson
Paolo Ballarini

Motivation

Nous disposons de données nous permettant de représenter des séquences textuelles d'ADN en une trajectoire 3D. Cependant les observations faites en laboratoire ne correspondent pas au modèle donné : la trajectoire de notre plasmide doit être circulaire. Nous devons donc modifier ce modèle informatiquement pour qu'il puisse répondre à ces contraintes.

Observations et approches

En étudiant les trajectoires des chaînes obtenues grâce à la table initiale établie par les biophysiciens nous pouvons constater que la distance moyenne entre deux points consécutifs de la trajectoire est environ de 3,3.

Nous cherchons à minimiser une fonction à déterminer permettant de joindre le début et la fin de la trajectoire de notre séquence d'ADN.

Nous proposerons deux approches pour résoudre ce problème d'optimisation. Dans un premier temps nous étudierons un algorithme basé sur la méthode du recuit simulé, puis nous nous intéresserons à un algorithme génétique.

Table des matières

1 Recuit Simulé

Algorithme du recuit simulé :

Il y a deux grandes étapes lors de la conception d'un algorithme du recuit simulé :

— La fonction d'évaluation : comment comparer les solutions ? Qu'est-ce qu'une bonne solution ?

— La sélection des voisins : étant donné un état (une table de rotation dans notre cas), comment la modifier pour obtenir un état voisin ? En d'autres termes, comment explorons-nous l'espace des solutions ?

Initialement, nous avions choisi comme fonction de score la norme euclidienne entre le dernier et le premier point. Ainsi notre chaîne d'ADN sera bien circulaire. Cependant cela n'est pas possible physiquement : les raccordements doivent être "continus". Pour cela il faut étudier les angles formés par des points consécutifs et vérifier que la chaîne n'est pas brisée. Mais vérifier cette condition pour tous les points de la trajectoire est extrêmement coûteux sachant que l'on va calculer notre fonction de score très régulièrement, nous ne pouvons pas nous permettre de faire cela. Nous nous contenterons de vérifier que l'angle formé au niveau du raccordement est convenable.

L'espace des solutions est soumis à des contraintes : la biologie nous apprend qu'il existe des symétries à respecter entre les dinucléotides. En effet A et T sont complémentaires, de même que G et C. Ainsi les dinucléotides AA et TT doivent conserver les mêmes propriétés par exemple. Certains dinucléotides sont leurs propres symétriques, c'est le cas de AT. Finalement, il n'y a que 20 paramètres indépendants à modifier. La fonction `voisins` permet d'explorer l'espace des solutions. Elle prend en entrée une table de rotation et renvoie 40 nouvelles tables. Ces tables sont une copie de la table en entrée, où l'on a ajouté ou diminué l'un des coefficients (et donc potentiellement son complémentaire) par $c \cdot \sigma$, où c est un coefficient bien choisi et σ est l'écart type de ce coefficient. Dans l'algorithme nous appellerons `voisins` avec $c = \text{coeff} \cdot X, \quad X \sim \mathcal{U}[0,1]$ et coeff qui décroît à chaque itération. Ce choix a été fait après différents essais, il permet d'explorer rapidement l'espace au début, puis de ne plus avoir de sauts à la fin.

L'initialisation se fait avec la table de rotation établie par les biophysiciens. On introduit des mesures de temps pour définir une condition d'arrêt. La notion de température introduit l'aspect probabiliste de cet algorithme. A chaque itération, on calcule les voisins d'un état, si son score est meilleur alors on conserve cet état. Sinon, il est possible que cet état soit conservé dépendamment de la température et de sa distance à la meilleure solution actuelle.

Difficultés pour la fonction objectif

Dans un premier temps nous avons considéré la distance euclidienne entre le point initial et le point final. Néanmoins, cette approche est très rapidement limitée car les trajectoires obtenues sont souvent chaotiques. Ainsi on a eu l'idée de rajouter en plus de la distance une valeur qui pénalisera les individus dont l'angle entre l'avant dernier et le premier point par rapport au dernier point n'est pas proche de 180 degrès. Nous nous sommes confrontés à plusieurs problématiques :

— Doit-on pondérer notre fonction d'évaluation avec des coefficients impactant plus ou moins l'angle et la distance ? Il est complexe de déterminer des coefficients efficaces

sachant que nous travaillons avec des réels qui varient beaucoup (distance entre 1 et 10000 par exemple)

— Nous pouvons fixer un angle "raisonnable" et ajouter un poids infini aux tables de rotation qui ne respectent pas cette condition. Cela réduit beaucoup notre progression dans la recherche de la distance minimale.

— Dans tous les cas nous n'étudions que l'angle au raccordement entre le premier et dernier point, cela n'assure pas d'avoir une trajectoire "lisse" au global. Evaluer les angles entre tous les points consécutifs de la trajectoire est bien trop coûteux en ressources.

Nous avons testé notre algorithme avec et sans prendre en compte les angles.

Résultats obtenus

Avec les paramètres suivants :

— Température initiale : 1 et $T' = 0.96 \cdot T$

— Coeff initial : 3 et $C' = 0.95 \cdot C$

— Temps maximum : 60 secondes

Pour la chaîne avec 8000 nucléotides, nous obtenons une distance finale entre les deux points : 3.43. Nous n'avons pas considéré les angles ici. (voir la table JSON en annexe et la figure en annexe ?)

En considérant les angles on a obtenu une distance de 8.04 et l'angle formé vaut 150 degrés.

2 Algorithme génétique

L'algorithme génétique est plus complexe que le recuit simulé et demande différent choix et réfléxions que nous allons détailler ici.

Nous utilisons la programmation orienté objet en python. Nous avons crée une classe `GeneticAlgorithm` contenant :

- Un entier `population_size` représentant le nombre d'individus manipulés.
- Une liste `population` contenant les individus qui sont des `RotTable`
- Une liste `score` contenant les scores d'évaluation pour chaque individu de notre population
- Un flottant `mutation_prob` compris entre 0 et 1 représentant la probabilité de mutation d'un individu

Initialisation de la population

Pour initialiser notre population, nous avons écrit une fonction `uniform_randomize` qui renvoie une table où chaque coefficient est distribué selon une loi uniforme dans l'intervalle

$$[v_{\text{init}} - 2\sigma, v_{\text{init}} + 2\sigma]$$

où v_{init} est la valeur dans la table de rotation initial et σ son écart type.

Fonction d'évaluation

Nous avons utilisé les mêmes fonctions de score que dans la partie du recuit simulé et ce pour les mêmes raisons.

Séléction des individus

Une fois notre population évaluée, il faut choisir les meilleurs individus pour converger vers un minimum. Nous avons choisi d'implémenter une méthode de sélection par tournoi : on prend deux individus au hasard dans notre population, puis on compare leur score. Celui qui a le score le plus faible reste en vie, celui qui a le score le plus élevé est tué. Ce processus permet de conserver une certaine hétérogénéité dans notre population puisqu'il est possible que deux individus avec un score élevé s'affrontent.

Croisement et génération des enfants

Nous souhaitons garder une taille de population constante, or après le tournoi notre population est divisée par 2. Il faut donc créer des nouveaux individus (enfants) en croisant des individus déjà existants (parents). Nous avons implémenté deux méthodes : `simple_crossover` et `double_crossover`. La première choisit un entier n aléatoirement entre 1 et 9 et fait un échange des gènes des deux parents. L'enfant reçoit les n premiers gènes du premier parent et le reste du second. La deuxième fonction fait pareil mais avec 2 points de coupure aléatoirement choisis. Grâce à la fonction `symetrizeTable` nous nous assurons que les contraintes physiques concernant la symétrie des dinucléotides sont conservées.

Mutation des gènes

Pour le moment notre algorithme dépend uniquement de la population initale. Or pour espérer atteindre un minimum (même local) nous devons explorer une plus grande partie de l'espace des solutions. Ici rentre en jeu la mutation. Pour chaque nouvel enfant de notre population, on a une probabilité `mutation_prob` d'appeler la fonction `mutate`. Cette fonction modifie aléatoirement un élément de la table de rotation. La nouvelle valeur suit une loi $\mathcal{N}(\mu, \sigma^2)$ où μ est la valeur de la table initale et σ son écart-type.

Résultats obtenus

Suite à de nombreux test nous avons pu obtenir plusieurs résultats très satisfaisants. En appliquant l'algorithme nous avons pu obtenir des scores entre 1 et 12 selon la fonction de score qui prend la distance euclidienne entre le point initial et le point final.

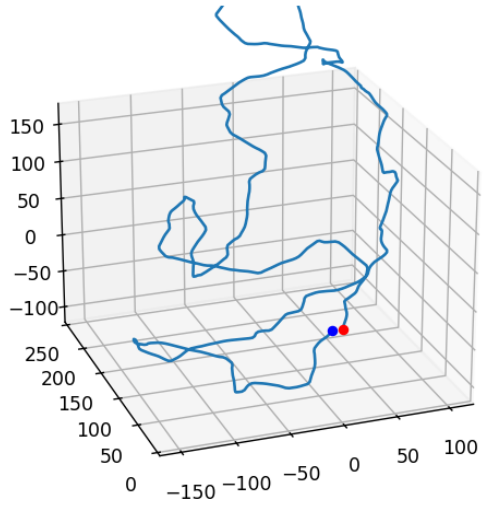

FIGURE 1 – Plot zoomé d'un score de 1.6 obtenu en 2 minutes et 100 itérations. Les paramètres sont : La séquence plasmid_8k, Une population de 40, une probabilité de mutation de 0.1

On peut voir que notre courbe semble discontinue, en effet notre fonction de score prend uniquement en compte la distance entre le point final et le point intial sans considérer si la courbe est lisse. Ainsi notre algorithme cherchant à optimiser ce score nous renvoie une courbe assez chaotique mais ayant un bon score.

3 Optimisations et tests

Optimisations

Les algorithmes présentés ici sont très dépendants de la performance de calcul : plus notre algorithme est rapide, plus il peut explorer des solutions ce qui est bénéfique. Nous avons donc procédé à quelques optimisations.

Dans le cas ou ne nous considérons que la distance comme fonction d'évaluation, il n'est pas nécessaire de calculer tous les points de la trajectoire mais uniquement le dernier. Cela nous permet de diminuer le nombre de produits matriciels qui sont en général coûteux.

Nous avons utilisé des outils adaptés au calcul pour Python, comme les listes par compréhension et les bibliothèques `Numpy`. Nous avons utilisé `deepcopy` pour copier efficacement les tables de rotations. C'est une opération que l'on fait très souvent donc c'est important qu'elle soit optimisée.

Tests

Nous avons utilisé une approche Test Driven Development, des tests unitaires ont été réalisés pour un maximum de fonctions. Comme c'est normalement le cas, tous les tests unitaires ont été effectués, dans la mesure du possible, en tenant compte des cas limites. Cependant, le caractère aléatoire des fonctions a posé quelques problèmes. C'est pourquoi, dans la mesure du possible, une graine a été définie pour rendre la fonction déterministe et permettre les tests. Dans certains cas, il n'a pas été possible de rendre la fonction déterministe, de sorte que l'on a seulement vérifié si la fonction présentait le résultat attendu. Tous les tests ont été créés et exécutés à l'aide du package `unittest`.

4 Conclusion et comparaison

L'objectif initial est rempli, nous avons réussi à implémenter les deux méthodes proposées, avec différents paramétrages possibles (2 fonctions d'évaluations, simple ou double cross-over...). Toutefois, les résultats sont mitigés.

D'une part, certains résultats sont bons : pour la chaîne de 8000 nucléotides, lorsque l'on prend des paramètres bien choisis et assez de temps nous arrivons à des résultats concluants. Nous rappellons qu'une distance de 3.3 est acceptable, ce qui est obtenu en moins de 2 minutes avec l'algorithme génétique et nous en sommes parfois proches en 60 secondes avec le recuit simulé. Dans la théorie, nos algorithmes sont fonctionnels, on a bien une fonction d'évaluation que l'on minimise au fil des itérations.

D'autre part nous avons constaté la forte instabilité de notre système. Puisque l'on utilise 8000 fois la table de rotation, un faible changement de quelques paramètres entraîne parfois des gros écarts de distance. Nous avons beaucoup de mal à contrôler ces instabilités.

Au delà des questions biologiques, le temps de calcul devient un problème pour la chaîne de 180 000 nucléotides, nous n'explorons clairement pas assez l'espace des solutions. Les résultats obtenus sont nettement moins saisfaisants.

Les deux approches présentent leurs avantages et leurs inconvénients.

Tout d'abord concernant le temps de calcul, l'algorithme génétique ainsi que le récuit simulé permettent d'être très malléable. En effet dans le cas du récuit simulé le choix des paramètres (en particulier de la température) ainsi qu'une contrainte de temps permet d'obtenir un résultat dans le temps voulu. De même pour l'algorithme génétique, la présence de nombreux individus en simultanés permet d'assurer en permanence une solution minimale correcte.

La principale distinction se joue sur le temps de convergence et sur la qualité du resultat, en effet l'algorithme génétique est un algorithme plus couteux, ainsi il converge généralement plus lentement. Néanmoins la solution qu'il propose est généralement plus intéressante

5 Annexe

Di-nucléotide	**Twist** (Ω)	**Wedge** (σ)	**Direction** (δ)
AA	35.67	7.2	-154
AC	34.4	-1.03	143
AG	27.7	33.89	2
AT	33.30	2.6	0
CA	34.57	76.48	-64
CC	2.18	2.1	-57
CG	29.8	6.7	0
CT	27.7	33.89	-2
GA	36.9	5.3	120
GC	40	5	180
GG	2.18	2.1	57
GT	34.4	-1.03	-143
TA	37.47	0.9	0
TC	36.9	5.3	-120
TG	34.57	76.48	64
TT	35.67	7.2	154

FIGURE 2 – Table obtenue sans considérer les angles, donnant une distance finale de 3.3

Di-nucléotide	**Twist** (Ω)	**Wedge** (σ)	**Direction** (δ)
AA	35.54	7.41	-154
AC	32.92	8.42	143
AG	25.25	6.18	2
AT	29.37	3.07	0
CA	32.98	-122.43	-64
CC	33.47	3.83	-57
CG	31.11	5.00	0
CT	25.25	6.18	-2
GA	38.60	3.92	120
GC	37.73	3.75	180
GG	33.47	3.83	57
GT	32.92	8.42	-143
TA	35.49	-2.19	0
TC	38.60	3.92	-120
TG	32.98	-122.43	64
TT	35.54	7.41	154

FIGURE 3 – Tableau obtenu pour un score de 1,76, un temps d'exécution de 2 minutes et 100 itérations. Les paramètres sont : La séquence plasmid_8k, Une population de 40 individus et une probabilité de mutation de 0.3.