BeeBots
Version 0.0-1
Alain D.

BeeVolve
Algorithme génétique stigmergique

Ce document vise à présenter dans les grandes lignes le prototype d'algorithme génétique développé pour assister *Stigmee* dans la gestion des recherches et la création assistée de brins pertiviraux. Je commencerai ci-dessous par poser quelques bases concernant les algorithmes génétiques avant de parler un peu plus de l'implémentation de celui-ci et du reste à faire.

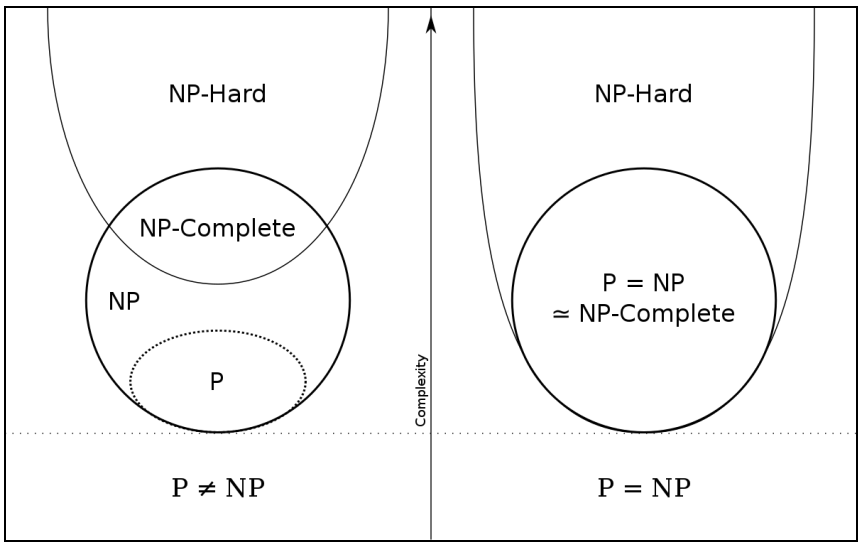
La v0 de mes scripts sera postée très prochainement sur GitHub, dès que j'aurai eu le temps de faire un peu de nettoyage et de correction.

Un algorithme génétique, c'est quoi ?

Définition

Les algorithmes génétiques sont des algorithmes évolutionnistes visant notamment à trouver des solutions « les plus proche possible d'une solution optimale » dans un temps acceptable. Ok. Mais encore ?

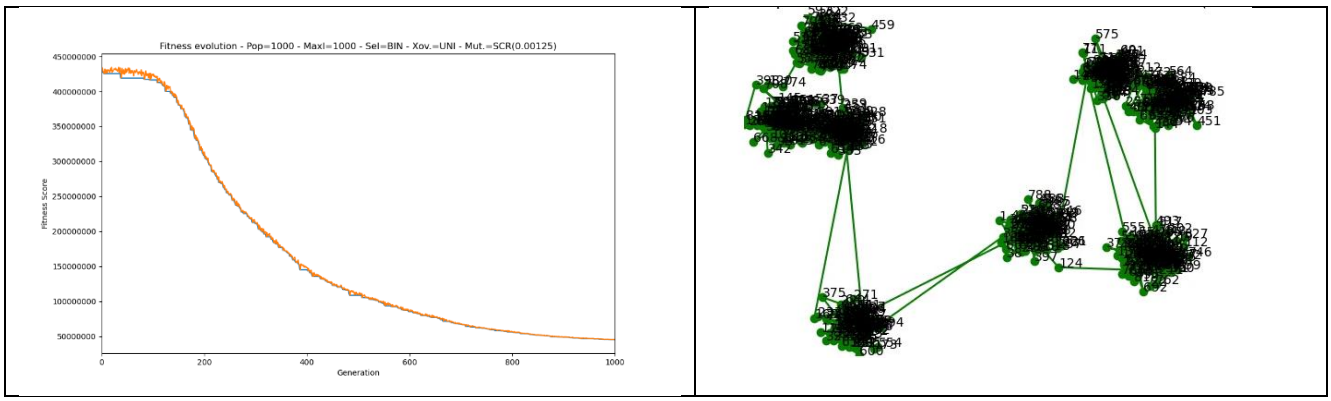
On classe généralement les problèmes d'optimisation en plusieurs catégories selon leur niveau de complexité (et *time-complexity*), définit comme suit :



Sans trop rentrer dans le détail de ce genre de choses qui ont tendance à me donner mal au crane (et en particulier sur l'apocalypse qui résulterait de la validité du côté droit de ce schéma, Homer Simpson pouvant en témoigner), la complexité d'un problème est mesurable en fonction de la capacité à le résoudre et ou à vérifier son résultat dans un temps polynomial. Un certain nombre de problèmes d'optimisation n'ayant pas de solution vérifiables en un temps raisonnable bénéficient donc grandement d'algorithmes tels que les algorithmes génétiques, compromis d'optimisation permettant de fournir une solution « acceptable » (mais non optimale, sauf à résoudre en effet le problème en question) en un temps raisonnable.

Classes de problèmes

Une classe de problèmes particulièrement intéressants à traiter avec des algorithmes génétiques sont les problèmes d'optimisation combinatoire, notamment NP-Hard. Par exemple, l'utilisation d'algorithmes de ce type sur TSP (*Travelling Saleman Problem*) donne d'assez bon résultats, et ces bons résultats sont dépendants d'un certain nombre de choix que je détaillerai un peu plus loin (par exemple, il a été montré que l'utilisation d'une initialisation Nearest-Neighbour permettait de fournir des solutions initiales se trouvant constamment dans les 25% meilleurs solutions (en fait plutôt à environs 25% au-dessus du coût fourni par l'algo Held-Karp Lower Bound)



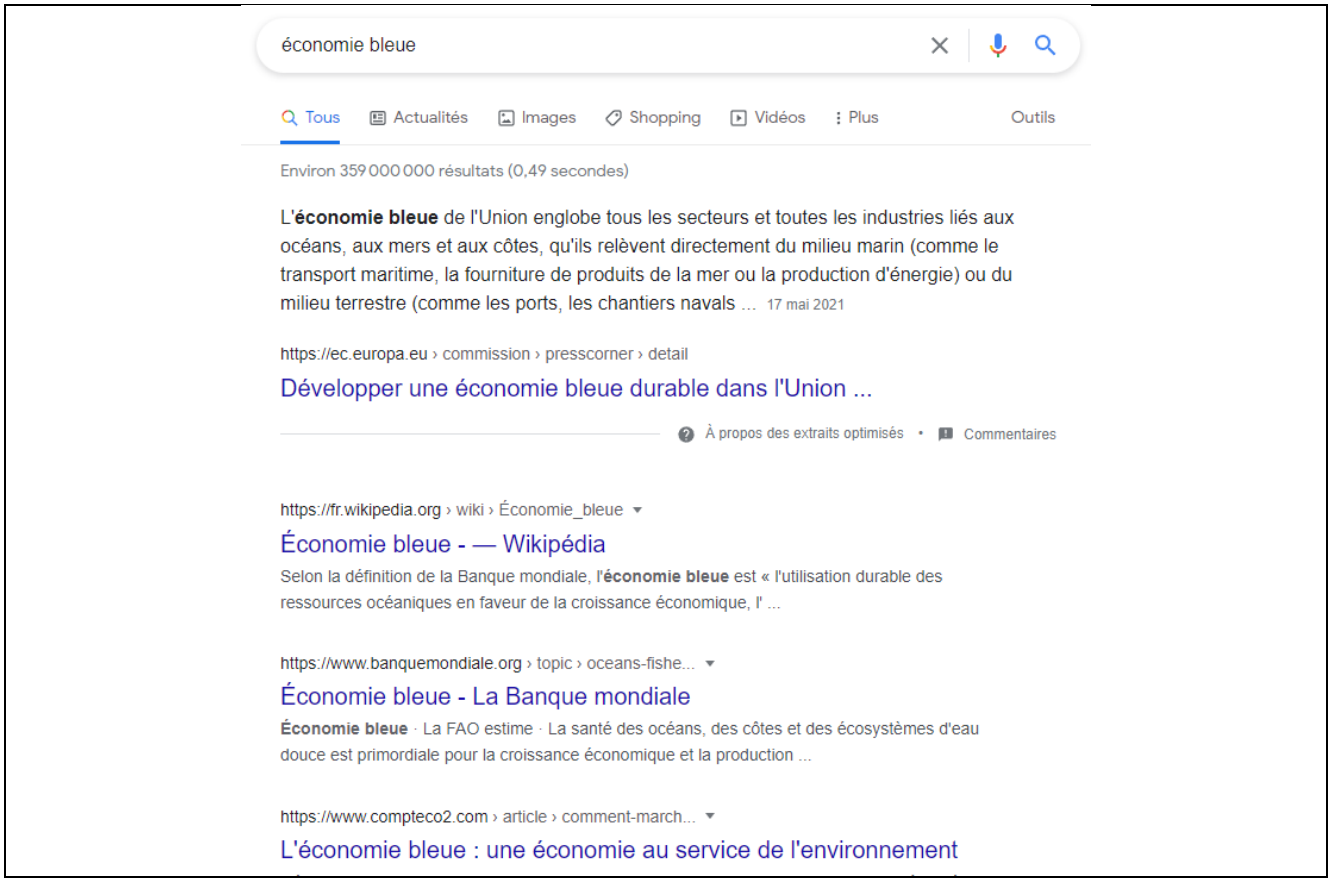
Un exemple de test d'algorithme génétique sur TSP

Quel rapport avec Stigmee ?

En soi, la constitution en un temps raisonnable de collections d'URL pertinentes concernant un sujet donné s'apparente à un problème d'optimisation combinatoire en réalité assez proche des problèmes évoqués précédemment. L'objectif est donc ici de minimiser une fonction d'évaluation qui détermine (maximise) la pertinence des résultats (score de pertiviralité comme score de fitness) tout en conservant un temps d'exécution raisonnable par rapport à une sélection aléatoire des URL

Un exemple de fonction d'évaluation permettant de tester le bon fonctionnement de l'algorithme consiste à déterminer si la solution fournie s'approche de celle trouvée lors de la recherche. Bien que ce ne soit pas l'objectif de Stigmee de répliquer purement et simplement

les résultats de recherche des moteurs existants, faire cette comparaison permet de valider le bon fonctionnement du processus évolutionniste (l'évolution mène alors à la minimisation du score de fitness)



Par exemple, pour mesurer la pertinence d'un tri, par exemple le tri de notre liste d'URL, il est possible de mesurer son nombre d'inversions. Une fonction d'évaluation possible *pour tester la pertinence* de notre algorithme peut donc se baser sur ce calcul, je reviens sur ce sujet dans la partie Evaluation.

Et la stigmergie dans tout ça ?

Evidemment, si notre algorithme a pour seul résultat de nous renvoyer un résultat de recherche proche-mais-pas-totalement de celui de Google, autant arrêter de se presser la cervelle et juste utiliser pour notre brin… le résultat trié fourni par Google. Dit comme ça, ça fait un peu captain'obvious, mais cela revient à dire qu'en dehors de la phase de mise au point algorithmique, la fonction d'évaluation sera remplacée par une fonction stigmergique notant la pertiviralité du brin, et que c'est donc ce critère qui sera le critère d'évolution / de convergence.

Dans un algorithme génétique, la compétition entre individus est matérialisée par un processus de croisement (**crossover**) produisant un / des enfants à partir de parents issus d'une fonction de **sélection.** A cela s'ajoutent de manière stochastique des **mutations** dont le but est généralement de perturber le score de fitness et de maintenir la diversité de la population, ce qui évite la convergence anticipée vers une solution non satisfaisante (les évènements

aléatoires sont souvent introduits en algorithmique et en optimisation pour pallier aux phénomènes de « local optimum »)

Or ce qui nous intéresse dans la stigmergie, c'est la possibilité pour les utilisateurs d'interagir dans la conception des brins d'Url en vue de favoriser certains contenus plutôt que d'autres. Donc pour transformer notre algorithme génétique standard (possédant une fonction d'évaluation classique basée sur la distance entre 2 url ou le nombre de permutations) en un algorithme stigmergique, il « suffit » de conserver les propriétés évolutionnistes tout en modifiant les propriétés suivantes :

- **Crossover :** Le choix des URLs à conserver au sein d'un brin est laissé aux utilisateurs (le moteur de recherche propose, et les Stigmers disposent)
- **Evaluation :** L'évaluation n'est plus basée sur le critère du « meilleur tri vs. Google » mais sur d'autres critères (maximisation d'un score de *pertinence / viralité* d'une playlist au sein du réseau Stigmee).
- **Mutation :** Plusieurs possibilités pour cela (soit de conserver un mode de fonctionnement complètement stochastique, soit interagir avec d'autres stigmers en leur demandant, sans connaissance du sujet, laquelle parmi N url ils préfèrent...

Note: Les choix de design qui en découleront pour l'indexation des playlist sur le réseau seront donc très importants, j'y reviens un peu dans la partie TODO

Etapes d'un algorithme génétique traditionnel

0 - Initialisation

L'initialisation consiste comme son nom l'indique à instancier la population d'origine. Une population est un ensemble de n individus (ici des brins ou « strands ») évaluables les uns par rapport aux autres. Bien que ces individus soient comparables, ils ne sont pas « identique » lors de l'initialisation. Un exemple ci-dessous montre 2 brins différents :

[1, 5, 3, 8, 4, 0, 2, 7, 11, 13, 10, 15, 9, 17, 12, 6, 14, 20, 16, 18, 19] : Strand Fitness: 317
[3, 7, 0, 1, 4, 2, 9, 5, 12, 14, 11, 8, 6, 13, 15, 17, 16, 10, 18, 20, 19] : Strand Fitness: 299

Bien que basés sur les mêmes 21 **gènes** élémentaires (la collection d'url), l'ordre est ici différent, ce qui influe sur le score de fitness.

Pour créer une diversité à partir d'un nombre restreint de gènes, plusieurs solutions existent

- Shuffle : génération aléatoire de l'ordre des gènes
- NNI (Nearest-neighbour insertion) : on choisit un point de départ et on insère en séquence les gènes voisins

1 – Mating pool & définition d'une génération

Le mating pool est formé par les candidats (parents) au processus de sélection (génération). Ici aussi plusieurs solutions existent en vue de favoriser ou ralentir le processus d'évolution / de sélection. Parmi lesquelles on peut citer

- La recopie complète de la pop (même chance pour tous)
- Proportion du pool en fonction du score de fitness de chacun (plus ou moins de chance d'être sélectionné)
- Sélection par tournois
- Mise à jour partielle ou complète du *mating pool*…

La solution la plus simple consiste d'implémentation consiste en une mise à jour complète du *mating pool* à chaque tour à partir de la dernière génération de population (incluant donc un / des enfants remplaçant ayant remplacé un / des parents lors du précédent round.

2 – Crossover

Le **crossover** est le processus d'appariement de plusieurs individus (brins) en vue de générer des brins enfants. Là encore plusieurs techniques existent en fonction généralement de la catégorie de problème et des besoins en performance de l'algorithme. J'en mets ici les liens vers la description de deux d'entre eux ci-dessous, parmi les plus utilisées, à savoir *Order1* et *Uniform* crossover.

Note : Mon implémentation actuelle d'Uniform Crossover sera amenée à évoluer si les brins doivent devenir évolutifs ou si on a besoin de croiser des brins comprenant des url totalement différentes

3 – Mutation

Comme expliqué précédemment, le processus de mutation est stochastique et vise à introduit un surcroit de diversité en vue d'échapper à un phénomène de local minima. Une façon simple de se figurer ce phénomène consiste à se rappeler que lors d'une opération de crossover, la totalité des gènes ne sont pas impactés, et que la population est mise à jour potentiellement dans sa totalité à chaque round. Un minimum local peut donc résulter du fait que parents et enfants possèdent tous un grand nombre de similarités avantageuses après quelques tours, ce qui ralentis la découverte de nouvelles améliorations et donc le processus d'évolution (ce phénomène est également appelé convergence prématurée)

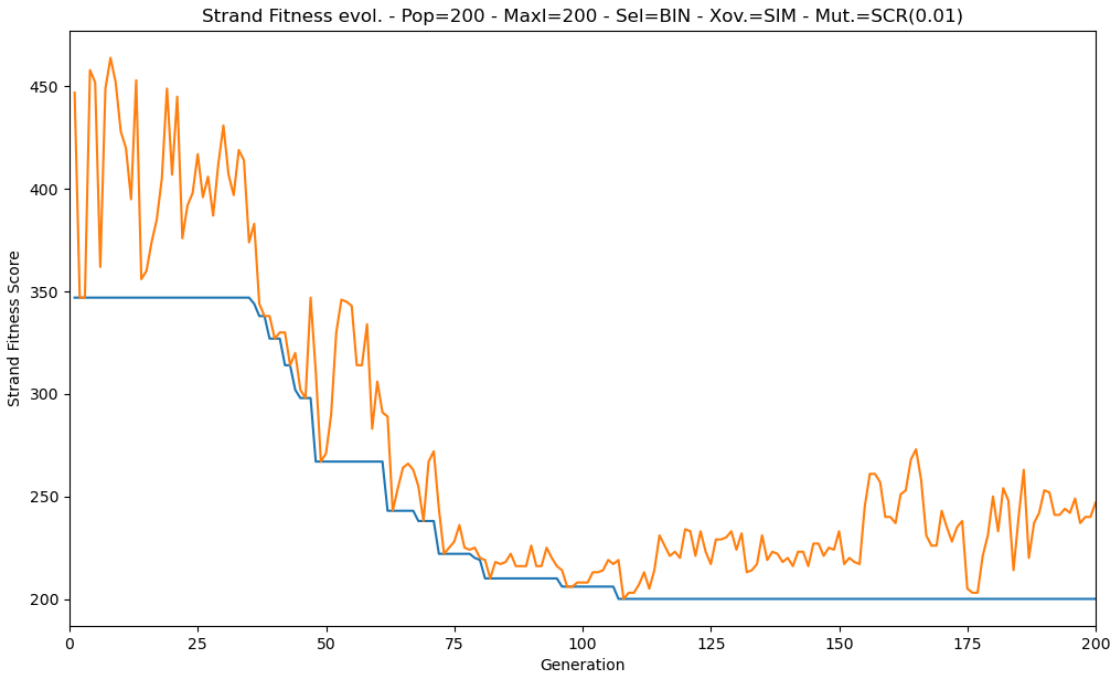
Là encore plusieurs techniques sont existantes pour caractériser une mutation, les 2 plus simple étant Scramble et Inversion

4 – Evaluation

La fonction d'évaluation peut prendre diverses formes selon le type de problème à traiter. Par exemple, la fonction d'évaluation d'un problème de type TSP consiste généralement à définir

la *fitness* d'une solution comme la somme des distances entre les différents points reliés par cette solution (incluant le retour au point de départ).

Dans le cas d'un problème de pertinence d'une recherche, la mesure de fitness peut consister à établir si le tri de la solution proposée est correct, par exemple en calculant le nombre d'inversions. Lors de la création d'une nouvelle solution enfant, on mesure également son score et on le compare au score des autres solutions présentes dans la population. On peut alors estimer les améliorations apportées d'une génération (itération) à une autre.



Implémentation de BeeVolve

(bzzZzz)

BeeSearch - Module de recherche

Dans une certaine mesure et pour amorcer le processus de création de brins, il est envisagé un minimum de scrapping assisté par utilisateur. Cela signifie qu'on a donc besoin d'une API permettant ce type de collecte. Ce n'est pas le cas de DuckDuckGo, qui limite volontairement ce type de procédés, que nous devons contourner d'une autre manière ultérieurement.

A minima, le module *BeeSearch* prend en charge Google pour l'instant, mais devrait supporter à terme toute autre API qu'on souhaite (qu'on peut ?) y ajouter. A terme, si le projet décide de

fork son propre moteur de recherche, celui-ci devra aussi être intégrable sous forme d'API dans ce module.

```
from BeeSearch import GoogleSearch

# Search instanciation test

b = GoogleSearch("Economie bleue")
results = b.getResults()
for result in results:
    print(result)

"""
(0, 'https://ec.europa.eu/commission/presscorner/detail/fr/ip_21_2341')
(1, 'https://www.un.org/africarenewal/fr/magazine/d%C3%A9cembre-2018-mars-
2019/economie-bleue-une-opportunit%C3%A9-pour-l%E2%80%99afrique')
(2, 'https://fr.wikipedia.org/wiki/%C3%89conomie_bleue')
(3, 'https://www.banquemondiale.org/fr/topic/oceans-fisheries-and-coastal-
economies')
[...]
(122, 'https://www.challenges.fr/')
(123, 'https://particulier.edf.fr/fr/accueil/gestion-contrat/options/ejp.html')
(124, 'https://www.linkedin.com/company/expertise-france')
"""
```

BeeStrand – Module de gestion d'un brin

Ce module définit pour l'instant les caractéristiques minimales d'un brin (dans le contexte d'un algorithme génétique) à l'aide de la classe *Strand* mais pourra servir de support / héritage aux autres méthodes d'AI du projet Stigmee. L'appel à cette classe se fait de la manière suivante (exemple pour construire la population initiale de mon algo :

```
        # 1. initialize the population up to popSize
        for j in range(0, self.popSize):

            strand = Strand(_strandSize=self.strandSize,
                            _origGenes=self.data,
                            _childGenes=None, _nni=False)
            strand.computeStrandFitness()
            self.population.append(strand)
```

Les propriétés sont :

- strandSize (taille d'un brin)
- origGenes (contenu d'un gène indexé - url)
- nni (type d'initialisation. False = random, True = nearest-neighbour insertion)

Les méthodes sous-jacentes sont pour l'instant les suivantes :

- getStrandFitness (retourne le score de fitness d'un Strand)
- computeStrandFitness (calcule le score de fitness)

Les éléments manquants sont essentiellement les caractéristiques annexes d'un brin / des url sous-jacentes, des métadonnées et méthodes éventuelles de stockage / d'indexation (tags, indexes physiques), la gestion de la pertiviralité. Difficile d'en savoir plus tant qu'on n'aura pas commencé à étudier en profondeur la génération et le partage de certains brins, ce qui nous amènera à modifier cette classe en profondeur certainement

BeeVolve – Algo. Génétique

Ce module jette les bases d'un algorithme génétique, et repose sur les méthodes suivantes (je passe pour l'instant volontairement le détail des différents paramètres sui devraient de toute façon être amenés à évoluer dans les prochaines semaines, au fur et à mesure que de nouvelles méthodes seront implémentées)

1. Initialisation

Une fois la classe crée avec ses paramètres d'appels, la méthode suivante s'occupe de la génération initiale de la population. Le mode actuel utilise des instances de strand générées aléatoirement à partir des solutions fournies par la recherche

```
self.initPopulation()
```

2. Run

Ceci est la boucle principale de l'algorithme et se répète jusqu'à complétion du nombre maximum d'itérations :

```
    def run(self):
        """
        General execution template.
        Iterates for a given number of steps
        """
        self.iteration = 0
        while self.iteration < self.maxIter:
            self.GeneticStigmergicStep()
            self.iteration += 1
```

3. GeneticStigmergicStep

Cette methode est responsable de mettre à jour le *mating pool* au regard du round précédent, puis ensuite d'appeler la méthode newGeneration, qui peut être vue comme la sous boucle principale responsable d'un round complet d'évolution, à savoir Sélection + Crossover + Mutations + évaluations.

```
    def GeneticStigmergicStep(self):
        """
        One step in the Genetic Stigmergic main algorithm
        1. Updating the mating pool with current population
        2. Creating a new Generation using selection / crossover / mutation
        """
        self.updateMatingPool()
        self.newGeneration()
```

4. Evolution d'une génération

En fonction des paramètres fournis, les possibilités sont les suivantes pour chaque domaine Sélection / Crossover / mutation / évaluation :

Sélection :

```
parent1, parent2 = self.randomSelection()
```

ou

```
parent1, parent2 = self.binaryTournamentSelection()
```

Crossover :

```
child = self.similarityBasedCrossover(parent1, parent2)
```

Il s'agit en réalisé simplement d'une implémentation d'Uniform Crossover qui était prévue pour gérer les cas de non-recouvrement (différence partielle d'allèles entre deux brins) mais cette solution est probablement mauvaise. Le crossover peut tout à fait se baser sur la totalité des URLs disponibles même si toutes ne sont finalement pas affichées par un brin. Ceci uniformise la gestion des différentes versions et facilite le curating d'une recherche

Mutation :

```
self.scrambleMutation(child)
```

Mutation simple par scrambling d'un pourcentage de gêne, la survenance étant déterminée par un paramètre d'entrée de la classe (`-a|--alterpct`)

Evaluation :

Actuellement l'évaluation se fait via la seule méthode de la classe Strand disponible à des fins de mise au point de l'algorithme, à savoir *computeStrandFitness()*. Il est souhaitable de définir très rapidement une méthode complémentaire en function des choix faits par les utilisateurs. Cette méthode devra répondre aux besoins d'interfaçage avec les choix réalisés, si possible de manière asynchrone (opération d'évolution réalisée directement sur les brins en base de données et ainsi propagée sur le réseau)

TODO

Au 20/11/2021, en fonction de tous les éléments déjà évoqués, le reste à faire minimum dans le cadre de cette v0 est le suivant :

- Possibilité d'appel de la classe de recherche directement depuis *BeeVolve* en lieu et place d'un fichier d'url, ce qui n'est pas encore fait même si c'est trivial.
- Possibilité de checkpoint régulier en base de l'état d'un « processus évolutionniste complet » (nécessaires pour pallier aux déconnections et autre, on ne peut nécessairement pas demander à un utilisateur de conserver sa session synchro en permanence et les évolutions de brins seront de toute manière amenées à être un processus quasi continu d'amélioration, les brins devant être en plus partageable rapidement. Et on ne peut pas non plus demander à l'algo de repartir de 0 chaque fois que les X utilisateurs forkant ce brin se déconnectent. Coté base, ça veut donc dire que l'API proposée doit au moins prendre en compte des notions de shared-locks et timestamp d'avancement (du type de ce que fait un SCN coté Oracle)
- Besoin donc d'une première API d'appel base de données et donc de définir plus correctement le modèle adapté (tant pour la création que l'évolutivité ET le partage des Strands et leurs métadonnées, et aussi pour la matérialisation des choix utilisateurs sur ces brins, le tagging…) Bien entendu pour des tests, on peut avoir une v0 qui fait le stockage en centralisé ou en local
- Implémentation d'une fonction d'évaluation stigmergique prenant également en compte les choix d'archi évoqués ci-dessus.
- Autres implémentations souhaitables notamment du crossover (Order-1 étant de mémoire moins consommateur / time-complex que Uniform crossover, mais je n'ai pas testé)