10 March 2019

Alexandre ALLANI
Adrien CLOTTEAU
François GUIBOURT
Vincent HIAULT
Yann BOUENAN

Projet Aide à la décision

Algorithme génétique

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Contents

1 Introduction

Le problème qui va être traité concerne l'optimisation de la location de hub au sein d'un réseau. Nous avons à disposition une liste de lieu devant être connectés les uns aux autres par un réseau. Certains lieux (nœuds au sein d'un graphe), peuvent être des hubs, c'est à dire connectés à plusieurs autres lieux. Tandis que d'autres lieux vont être des spokes, c'est à dire qu'ils ne communiquent qu'avec un seul hub. Pour résumer, un Hub est connecté à des hubs/spoke et un spoke est connecté au maximum à un hub (Single allocaiton hub problem). Il est supposé par la suite, que les lieux sont fixes et que chaque lieux doit être connecté au réseau.

Le problème va être d'optimiser les contraintes qui vont régir le réseau. Pour chaque hub nous avons les contraintes suivantes :

- Coût fixe (propre au noeud)
- Coût Variable (dépendant du flux de marchandise passant par le noeud).
- Le transfert de flux entre deux hub est soumis à une réduction de coût
- Chaque hub a une capacité de marchandise limitée

Il est possible de résoudre de manière optimale ce problème sur un nombre limité de ville en utilisant les méthodes de résolutions linéaires (avec GLPK par exemple). Cependant ici nous nous concentrons sur un problème rassemblant plus de 30 locations différentes. Nous avons donc choisi d'utiliser un algorithme génétique pour trouver une solution optimale à ce problème.

Le but de ce rapport est de décrire le design de notre algorithme génétique, ainsi que son implémentation. Nous présenterons ensuite les résultats que nous avons eu, ainsi que l'influence des différents paramètres sur la résolution du problème.

Vous pouvez retrouver le code sur le lien suivant.La manière de lancer l'algorithme y est aussi expliqué. `https://github.com/Syndorik/Genetic-Algorithm`.

2 Design de l'algorithme génétique

Dans cette partie nous allons voir comment a été designé l'algorithme génétique pour résoudre le problème de location de hub.

2.1 Représentation des solutions

Avant de travailler sur l'algorithme génétique en lui même il est important de trouver une représentation conforme des solutions. Le but est de trouver une structure permettant de décrire une solution à ce problème. En particulier, il faut qu'elle respecte la contrainte de *single allocation*, qu'il soit possible de distinguer les hubs des spokes et qu'il n'y ait pas de spokes non connectés au réseau.

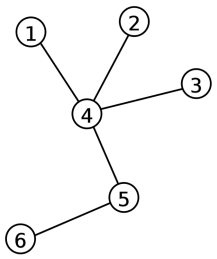


Figure 1: Graphe obtenu à partir du code Prüfer : [4,4,4,5]

Pour cela nous avons choisi de travailler avec un codage de Prüfer. Le code de Prüfer une fois décodé nous donne accès à un graphe qui représentera la solution proposée. De plus les nœuds contenus dans le code de Prüfer correspondent aux hubs (car cela implique la connexion avec au moins 2 nœuds), et, inversement, ceux qui ne sont pas dedans sont les spokes.

L'exemple de la **Figure 1** montre bien que les nœuds 4 et 5, présents dans le code de Prüfer, sont effectivement des hubs , tandis que les noeuds 1, 2, 3 et 6 sont des spokes. Cette représentation convient donc parfaitement comme base de l'algorithme génétique.

Fitness function

La fitness function est ce que l'on cherche à optimiser dans la création du réseau. Elle est accessible via ce **lien**. Cette fonction résume simplement les différents coûts à minimiser pour l'ensemble du réseau, les coûts fixes pour les hubs et les coûts variables dépendant du flux traversant le réseau.

2.2 Opérateur de Crossover

Afin de créer une nouvelle génération, nous devons établir un moyen de mélanger les différents individus. L'idéal serait que cette opération de *crossover* ne donne pas de solution irréalisable. Dans notre cas, la capacité de chaque hub doit être respectée, sinon le réseau ne peut pas être crée.

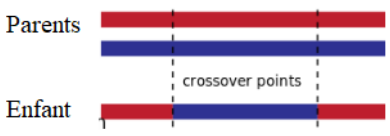


Figure 2: Two-point crossover

Il était difficile de trouver un opérateur ne retournant pas de solution irréalisable. Nous avons choisi l'opérateur *k-point crossover*. Le principe est de diviser le code de Prüfer des deux parents en k points différents (k étant fixé en tant que paramètre), et ensuite de réaliser un crossover normal, c'est à dire échanger les différentes parties du code de Prüfer. La **Figure 2** illustre ce procédé pour un $k = 2$.
Afin de pallier au problème possible de non conformité des solutions, nous avons choisi d'affecter une valeur très grande à la fitness function. Ainsi, lors de l'étape de la sélection **Section 2.3**, ces solutions auront peu de chance d'être sélectionnées. De plus, ceci va rajouter un paramètre à l'algorithme génétique en plus de ceux déjà existant (nombre d'individus dans la population, nombre de génération maximale, probabilité de mutation).

2.3 Opérateur de Sélection

Le but de cet opérateur est de sélectionner les solutions parmi une population d'individus qui vont être les parents utilisés dans l'opérateur de *crossover*. Ici nous utilisons un *k-way torunament*. Le principe est de choisir k individus dans la population de base. Ensuite un tournois est fait entre ces k individus, et celui qui a la meilleure fitness est choisi (ie, l'individu qui a la plus faible fitness parmi les k individus).
Dans l'algorithme, cela permet de choisir les deux parents auxquels vont être appliqué l'opérateur de crossover. Ce processus (Sélection par tournois + crossover) répété assez de fois permet de créer la population descendante. Cependant, il est à noter que s'il y a élitisme, le meilleur individu de la population initial sera conservé dans la population descendante.
Cet opérateur permet de privilégier les individus ayant une bonne fitness (vu qu'ils vont gagner les tournois s'ils sont sélectionnés) tout en gardant des individus moins bons permettant de faire un peu plus d'exploration (et donc de retourner moins souvent un minimum local). Ceci rajoute donc un autre paramètre à notre algorithme : le nombre k d'individus à choisir pour un tournois.

2.4 Opérateur de mutation

Cet opérateur permet d'avoir plus de variété au sein de la population et d'effectuer des opérations impossibles par crossover. Ici nous avons créé plusieurs opérateurs de mutation différents que nous avons testé.

> Les fonctions de mutations sont disponnibles dans le fichier nommé GA.py disponnible sur `https://github.com/Syndorik/Genetic-Algorithm/blob/master/lib/GA.py`.

Dans les paragraphes qui suivent nous explicitons chaque opérateur de mutation créé, et si nous l'avons finalement gardé. Nous partons du principe que la condition de probabilité est remplie dans chacun des cas. Les mutations effectuées se basent soit sur la fonction **swap** soit la fonction **swap_2opt**

Opérateur swap

A partir d'un code de Prüfer , deux indices sélectionnés au préalable sont échangés. Exemple pour un code de Prüfer $= [4, 4, 5, 6, 2, 2, 6, 8]$ et les indices 2 et 5 :

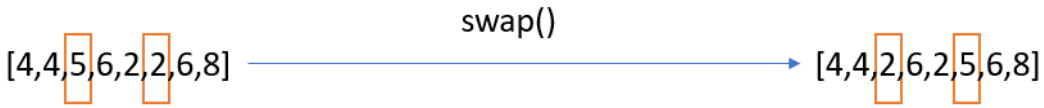


Figure 3: swap operator

Opérateur swap_2opt

A partir d'un code de Prüfer, et deux indices sélectionnés au préalables, une inversion de la route créé par l'ensemble des éléments compris entre les indices (indices exclus) est réalisé. Exemple pour un code de Prüfer $= [1, 5, 4, 3, 2, 6]$ et les indices 0 et 5 :

Figure 4: swap_2opt operator

2.4.1 Fonction : mutate

Cet opérateur est l'opérateur le plus basique. Il se base sur la fonction swap. Le principe est simple, deux indices du code de Prüfer de l'individu sont tirés, la fonction swap est appliquée et l'individu modifié est retourné. Cette fonction est très rapide à exécuter mais ne donne pas des résultats assez satisfaisant.

2.4.2 Fonction : mutate_swap

Cet opérateur ce base sur la fonction swap. Ici, au lieu de faire des permutations aléatoires, nous allons tester les permutations possibles une par une jusqu'à obtenir une permutation donnant un individu dont la fitness est meilleure que celle de l'individu original. Une fois obtenue, l'algorithme ne cherche plus de meilleurs permutation et l'individu modifié est retourné. Dans le cas où aucunes permutations ne donne un meilleur individu, l'individu originel est retourné.
Cette fonction est assez bonne en terme d'exploitation. Le local search appliqué permet d'avoir des individus qui sont toujours meilleurs (ou du moins tout aussi bien) que l'individu original. Cependant, le temps d'exécution est assez long, et il est assez fréquent de tomber dans un minimum local et ne plus pouvoir en sortir.
Cela s'explique par le fait que les individus une fois muté ne laissent plus assez de place pour l'exploration car il y a moins d'individus ayant de moins bonnes fitness.

2.4.3 Fonction : mutate_swap_nerfed

Ici,le même principe de fonctionnement de la fonction *mutate_swap* est utilisé. Cependant cette fois-ci, il n'y a pas de permutations sur l'ensemble des permutations, mais uniquement sur un échantillon d'indices pris aléatoirement (correspondant à 1/3 des indices du code de Prüfer). De plus si aucune mutation améliore la fitness, ce n'est pas l'individu original qui est renvoyé, mais l'individu ayant eu la meilleure fitness parmi tout les individus créés.

Cette alternative se retrouve être la meilleure. Le temps d'exécution est assez court et les résultats convaincants. Cela s'explique par le fait que le local search est bien plus limité que pour la fonction *mutate_swap* au vu des modifications apportées. Il y a alors un bon compromis entre **exploration et exploitation**

2.4.4 Fonction : mutate_2opt

De même, le principe de fonctionnement de *mutate_swap* est repris, la fonction de permutation utilisée n'est pas *swap* mais *2opt*.
Les résultats obtenus étaient assez décevant et le temps d'exécution très long.

2.5 Critères d'arrêts

Deux critères ont été retenus pour arrêter l'algorithme génétique :

- **Nombre d'itération** : ou nombre de génération. Après un certain nombre d'itération, l'algorithme s'arrête. Ce critère permet de ne pas rester coincé dans une boucle infinie. En général il n'est pas atteint car le second critère est atteint avant. Le nombre de génération est un paramètre mais pour l'ensemble des tests effectués, il a été fixé à 100. Parmi les 68 tests réalisés, il n'a jamais été atteint.
- **Evolution de fitness nulle** : Si au bout d'un certain nombre d'itération, la fitness n'évolue plus l'algorithme s'arrête. Ce critère d'arrêt a été fixé à 8. Ainsi, s'il n'y a plus d'amélioration au bout de 8 itérations l'algorithme est arrêté. Ce critère d'arrêt est souvent atteint. 8 générations sont laissées car il est possible de sortir du minimum local par exploration (crossover)

2.6 Déroulement de l'algorithme génétique

Le code correspondant à cette étape est disponnible dans le fichier App.py `https://github.com/Syndorik/Genetic-Algorithm/blob/master/lib/App.py`

Nous avons désormais l'intégralité des éléments pour créer l'algorithme génétique. Dans le cas standard, les paramètres suivants sont utilisés:

- Nombre de génération : 100
- Taille de la population : 100
- Probabilité de mutation : 0.4
- K point crossover : 3
- Taille du tournois : 7
- Élitisme : Il y en a
- Méthode de permutation : swap

La première génération est composé de 100 individus dont le code de Prüfer est généré aléatoirement parmi les nœuds disponibles. Ensuite, à chaque itération (qui peut aller jusqu'à un nombre maximal de 100 dans notre cas), on va appliquer le processus suivant:

- Evolution :
  - Création de la population descendante: avec l'élitisme, le meilleur individu de la population initial est inclus dans la population descendante. Par la suite, la population descendante est complétée comme expliqué **section : 2.3**
  - La population étant créé, on applique l'opérateur de mutation. La population descendante est alors créée.
- Local search : Si au bout de 3 itérations, la fitness n'évolue plus, le processus suivant est appliqué :
  - Sélection des 5 meilleurs individus (en terme de fitness)
  - Pour chaque individu, l'ensemble des 2 permutations possibles va être testé, et le meilleur individu est retourné.
  - Si parmi les 5 meilleurs individus, il y a des doublons (des individus ayant le même code de Prüfer), le processus se répète jusqu'à obtenir 5 individus différents.
- Critère d'arrêt: si un des critères d'arrêt est atteint, l'algorithme s'arrête et retourne le meilleurs résultat.

L'étape de Local Search est très importante dans l'algorithme, car cela permet souvent de sortir d'un minimum local en renouvelant la population. Cependant, l'appliquer sur un grand pourcentage de la population pourrait limiter les capacités d'exploration de l'algorithme. C'est pour cela que nous nous limitons à 5 individus différents.

3 Résultats et tests

> Vous pouvez retrouver les résultats des tests dans le dossier **log**. Le script lancé pour obtenir ces résultats est disponnible dans le dossier **script**. `https://github.com/Syndorik/Genetic-Algorithm`.

Dans cette partie nous allons montrer le meilleur résultat obtenu (paramètres inclus) ainsi que les différents tests réalisés pour tester notre algorithme.

3.1 Meilleur Résultat

Le meilleur résultat, sur l'ensemble des exécutions est obtenu avec les paramètres par défaut.
La fitness est de 1.579×10^8
Le code de Prüfer associé est :
[22, 20, 0, 24, 18, 26, 19, 1, 11, 15, 17, 14, 11, 20, 11, 6, 11, 16, 17, 11, 6, 6, 11, 11, 25, 11, 11, 23]

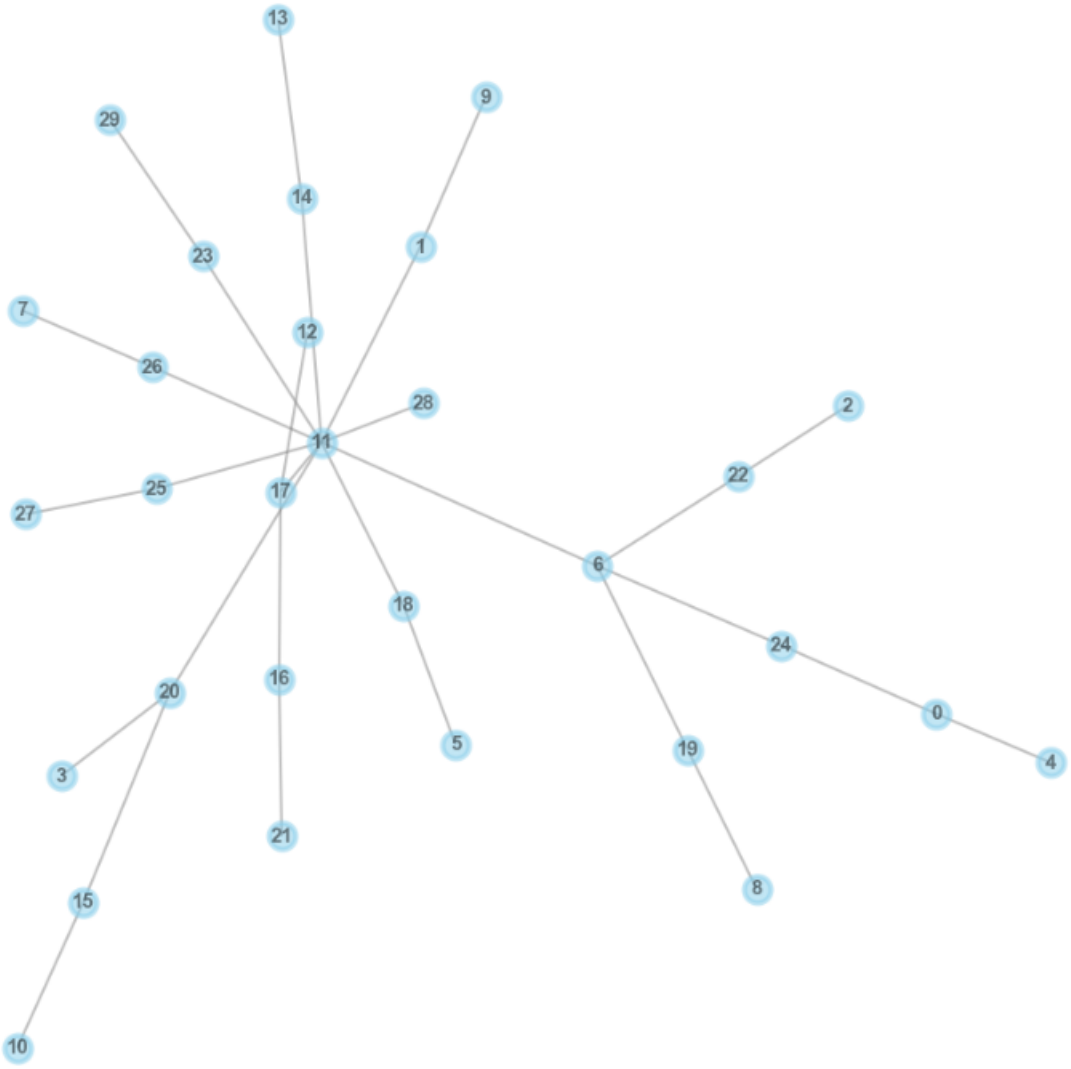


Figure 5: Meilleure solution trouvé par l'algorithme

3.2 Tests

L'algorithme étant assez long à s'exécuter, il est assez difficile de faire une vraie étude statistique avec une centaine d'échantillon pour prouver les tendances que nous allons expliciter ci-dessous.
Ils faut donc prendre avec précaution les analyses faites ci-dessous.

3.2.1 Test sur la taille de la population et le nombre d'individus dans un tournois

Ce test est réalisé sur deux paramètres car lors des tests préliminaires (tests lancés lors de la création de l'algorithme), les deux éléments semblaient être étroitement liés.
Les tests ont été effectué avec les paramètres par défaut excepté pour les paramètres suivants :

- Nombre d'individus dans la population : [50,100,150,200,250,300]
- Nombre d'individus dans le tournois : [3,5,7,10]

Pour chaque combinaisons possibles de paramètre l'algorithme a été lancé et les valeurs de fitness ont été enregistré ainsi que le code de Prüfer associé.

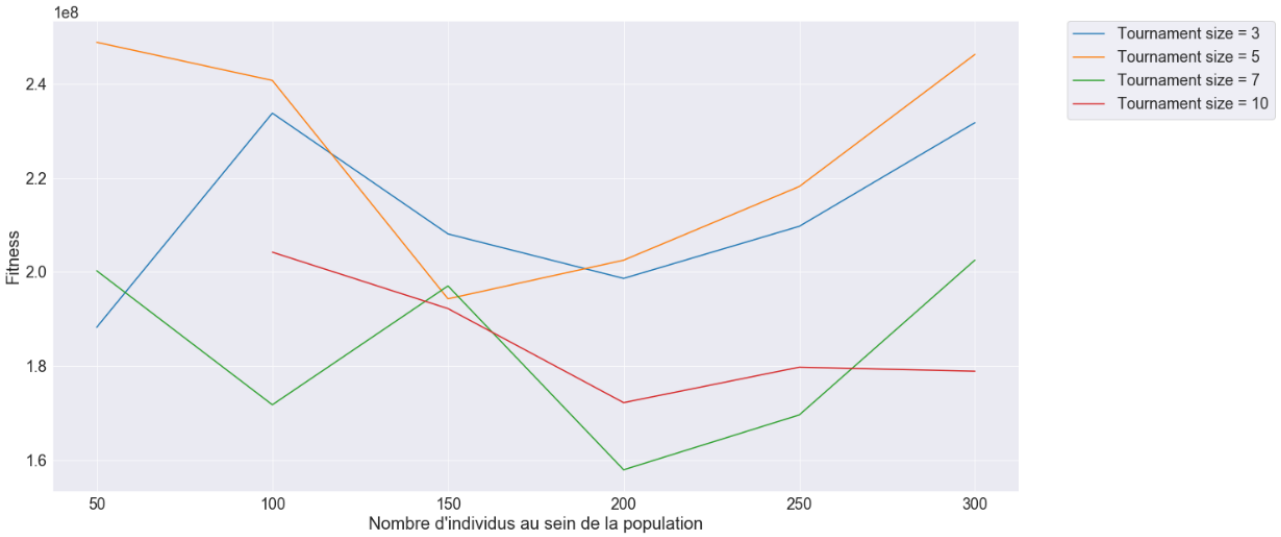


Figure 6: Influence des paramètres taille de la population et taille du tournois

Une certaine tendance entre les deux paramètres peut être observé: plus la population est grande, plus la taille du tournois doit être grande aussi afin d'avoir une fitness correct. Inversement, si la taille de la population est petite, il est intéressant d'avoir un petit tournois.
Cela s'explique par le fait qu'un tournois petit pour une grande population va être semblable à sélectionner au hasard les individus. Donc il y a très peu de bons individus qui vont être dans la population descendante. Tandis que pour un tournois grand au sein d'une petite population, le tournois va en général toujours donner les mêmes résultats (les mêmes individus). Donc il y aura très peu de diversification au sein de la population menant assez rapidement à un minimum local.

Pour des questions de temps d'exécution, une solution avec une population moyenne et une taille de tournois moyenne peut être privilégié. Dans les cas de nos tests, il s'avère que les meilleurs performances sont obtenus pour une taille de population égale à 200 et une taille de tournois égale à 7. Il est possible qu'avec une taille de tournois plus grande, et une population tout aussi grande les résultats soient meilleurs. Cependant nous allons rester sur les paramètres par défaut de l'algorithme (ie taille de la population = 200 et taille du tournois = 7)

3.2.2 Test sur le k point crossover

Le but de ce test est de voir l'influence du nombre de "coupe" dans le crossover sur la fitness finale. Les tests ont été effectué avec les paramètres par défaut excepté pour les paramètres suivants :

- Le nombre de "coupe" k : [1,2,3,4]

Le temps d'exécution étant long, nous avons lancé les tests 4 fois pour chaque probabilité de mutation. Le graphique ci-dessous est basé sur la médiane des résultats obtenus. Deux autres graphiques sont disponibles en **Annexe : 5.1**. Le premier est le même graphique mais basé sur la moyenne, le second représente la répartition des différents tests. Que ce soit avec la médiane ou la moyenne, nous pouvons supposer que

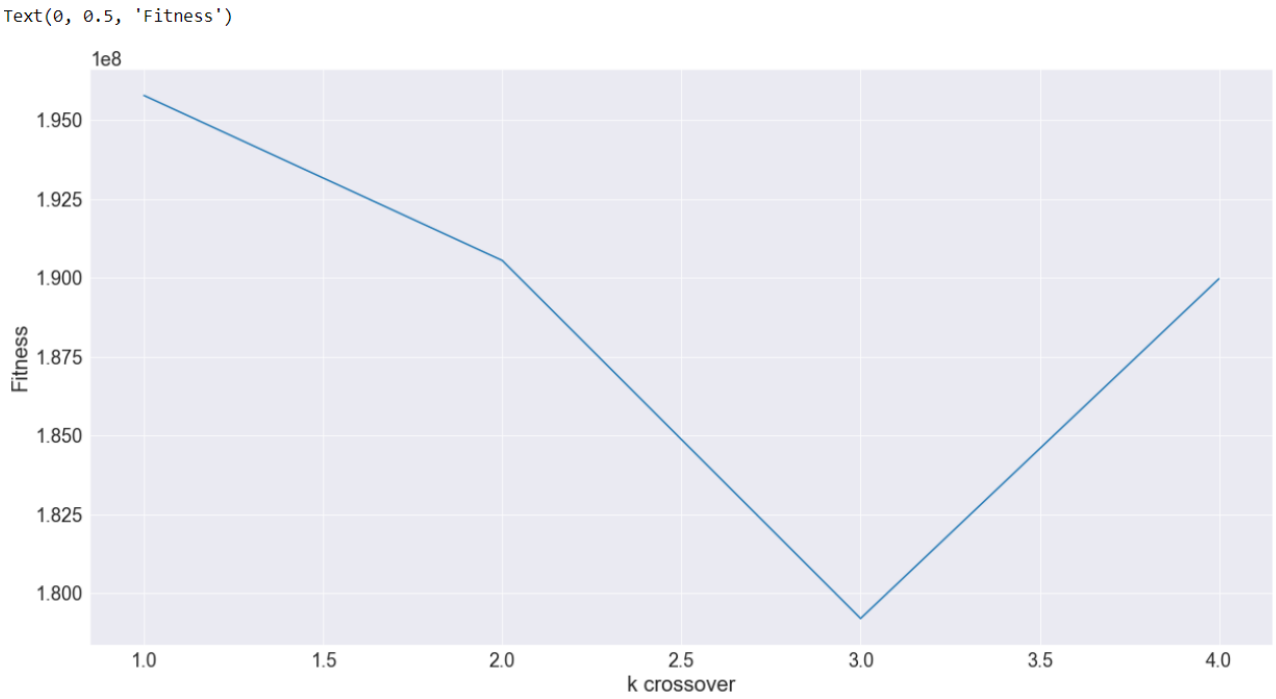


Figure 7: Médiane des probabilités de mutations

l'algorithme se comporte de manière optimale pour k =3, ce qui est le paramètre par défaut choisi.
Il est à noter que pour confirmer cette hypothèse, nous devrions faire des tests pour des valeurs de k supérieurs à 4. Cependant par manque de temps, nous ne pouvons pas exécuter les tests.

3.2.3 Test sur la probabilité de mutation

Le but de ce test est de voir l'influence de la probabilité de mutation sur la fitness finale. Les tests ont été effectué avec les paramètres par défaut excepté pour le paramètre suivant :

- Probabilité de mutation m : [0.1,0.3,0.4,0.6,0.8]

De même que pour le k-point crossover, le graphique ci-dessous est basé sur la médiane des 4 essaies, d'autres graphiques sont disponnibe en **Annexe : 5.2** Selon le graphique, plus la probabilité de mutation est grande, meilleur sont les résultats (que ce soit en médiane ou en moyenne). Cela paraît assez logique, car une grande partie de notre local search est effectué dans la fonction de mutation. Ainsi, lorsqu'il y a peu de mutation, il y a peu de local search, donc des solutions moins optimisés.
Il est cependant étonnant de voir qu'une forte probabilité de mutation entraîne de meilleurs performances qu'une probabilité moyenne car l'exploitation est privilégié à l'exploration. Donc le risque d'être dans un minimum local est plus grand.

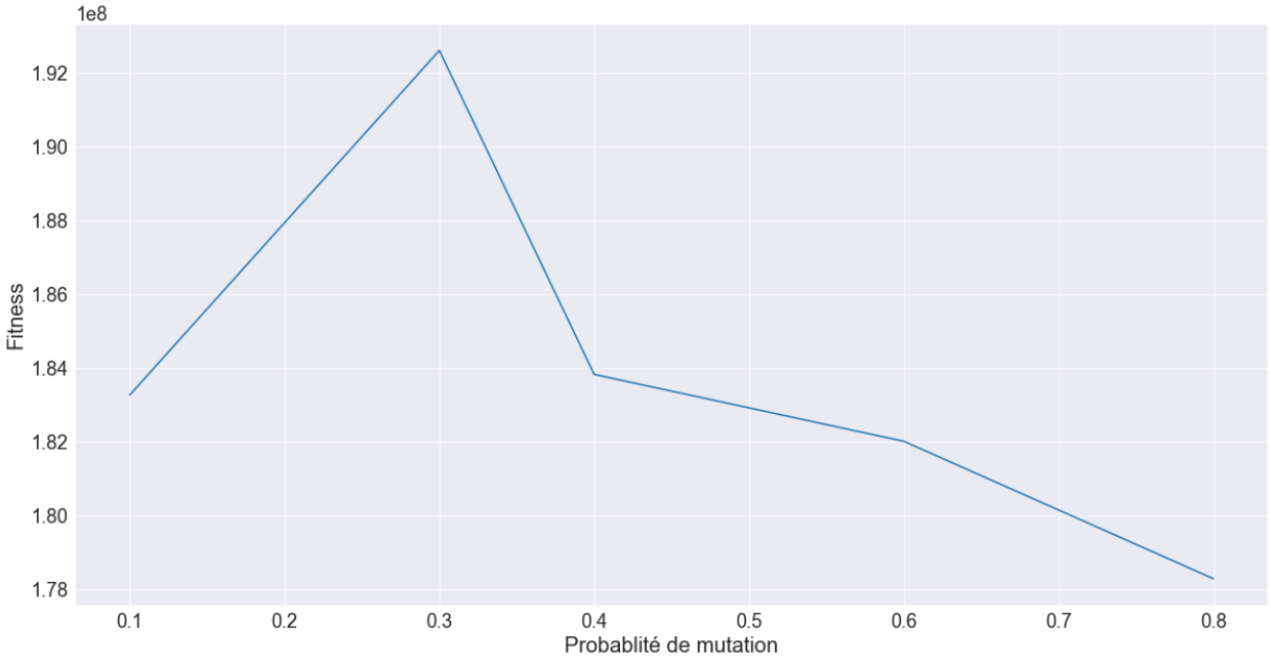


Figure 8: Médiane des probabilités de mutations

3.2.4 Test sur la méthode

Le but de ce test est de voir quelle méthode de permutation est la meilleure pour notre problème. Les tests ont été effectué avec les paramètres par défaut excepté pour lz paramètre suivant :

- méthode : ["swap","2opt"]

Pour chaque combinaisons possibles de paramètre l'algorithme a été lancé et les valeurs de fitness ont été enregistré ainsi que le code de Prüfer associé.

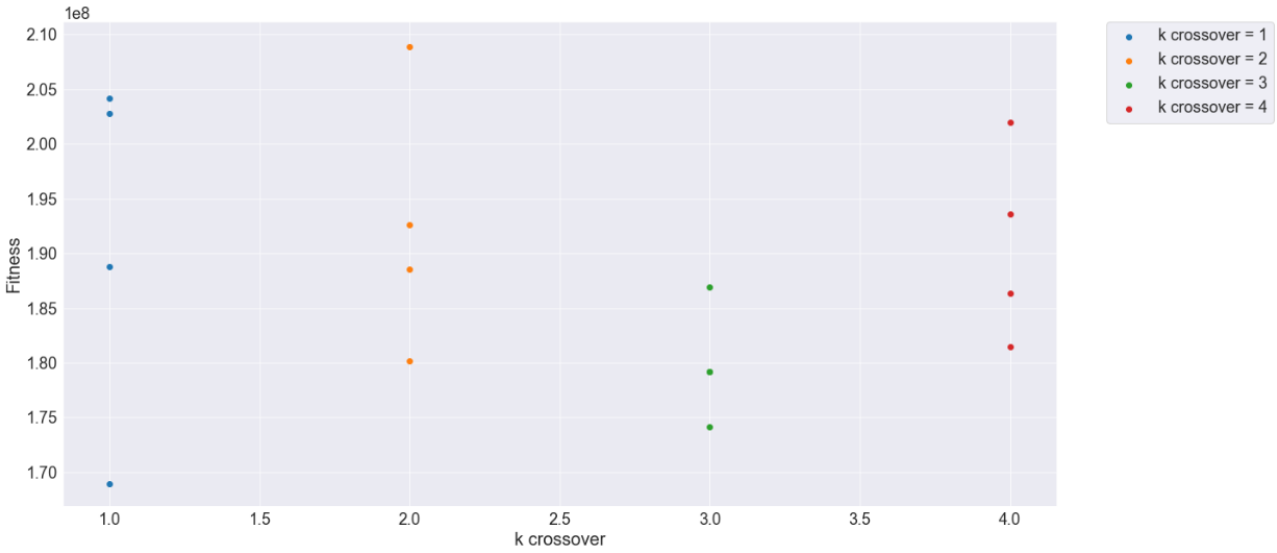
4 Conclusion

En conclusion, l'algorithme développé a donné avec l'ensemble des tests une fitness de $1.579x10^8$. En moyenne il retourne une fitness de $1.7x10^8$. Le temps d'exécution dépend beaucoup des paramètres comme discuté dans ce rapport. En moyenne il prend environ 10 minutes à s'exécuter, cela peut aller jusqu'à 30 minutes dans le pire des cas.

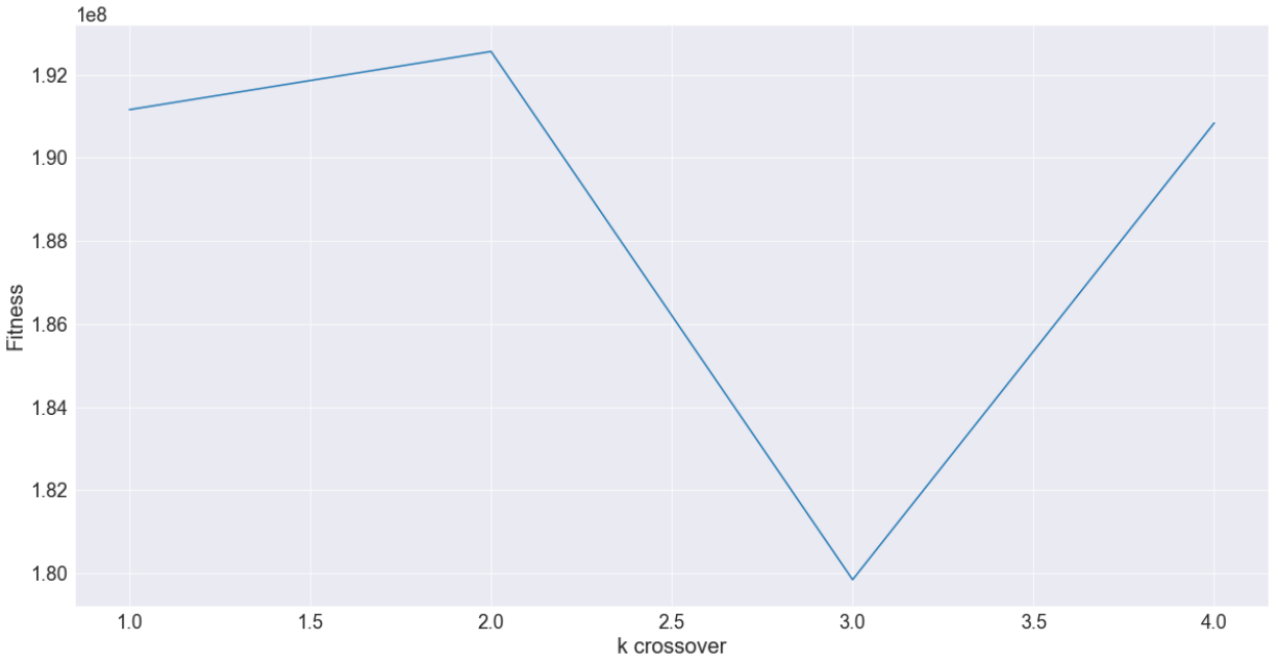
5 Annexe

5.1 Graphique sur les k point crossover

5.1.1 Répartitions selon les tests

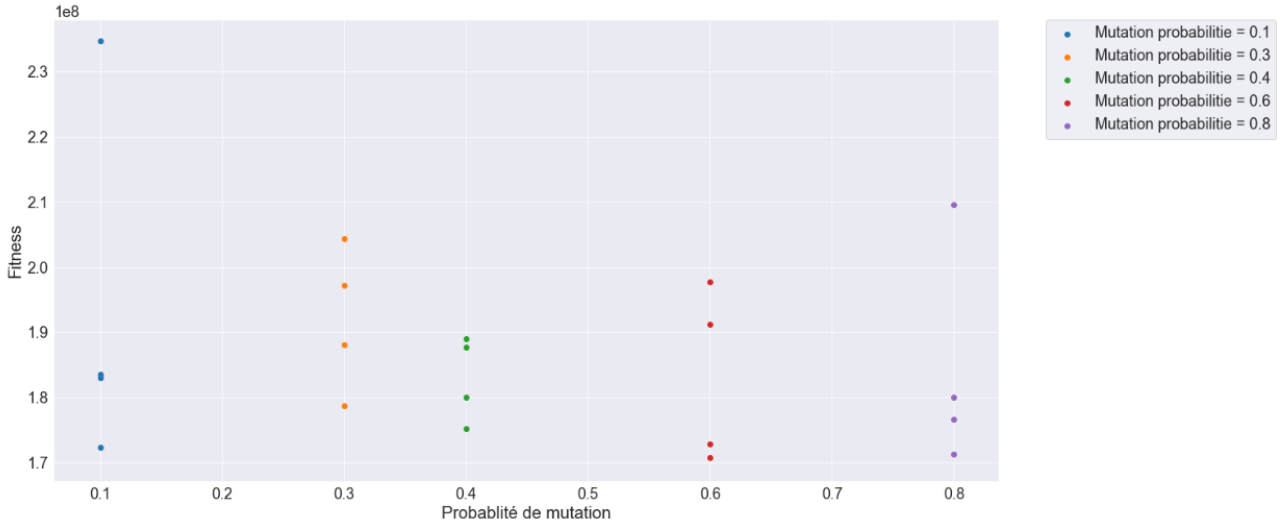


5.1.2 Moyenne de la fitness en fonction de k

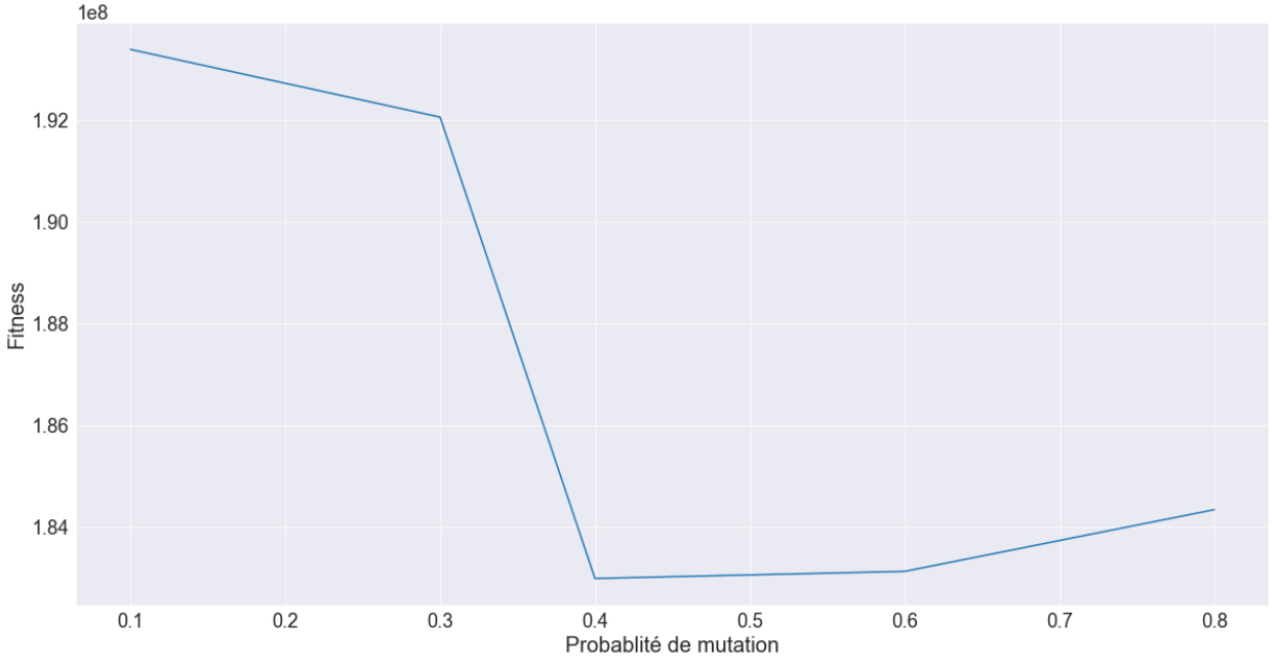


5.2 Graphique sur les mutations

5.2.1 Répartitions selon les tests



5.2.2 Moyenne de la fitness en fonction de la probabilité de mutation



IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

IMT Atlantique Bretagne - Pays de la Loire - www.imt-atlantique.fr

Campus de Brest
Technopôle Brest-Iroise
CS 83818
29238 Brest Cedex 03
T +33 (0)2 29 00 11 11
F +33 (0)2 29 00 10 00

Campus de Nantes
4, rue Alfred Kastler - La Chantrerie
CS 20722
44307 Nantes Cedex 03
T +33 (0)2 51 85 81 00
F +33 (0)2 51 85 81 99

Campus de Rennes
2, rue de la Châtaigneraie
CS 17607
35576 Cesson Sévigné Cedex
T +33 (0)2 99 12 70 00
F +33 (0)2 99 12 70 08