Rapport

February 22, 2024

Université bretagne occidentale

Faculté des Sciences et Technologies

Spécialité : Informatique

Module : Méthode computationnelle

Projet : Résolution du Problème du Voyageur de Commerce avec l'Algorithme Immunitaire

Année Universitaire : 2023-2024

Présenté par :

- Boulmaali Linda Imene
- Miszczuk Ivan

Date de Soumission : 23 février 2024

1 Introduction

Curieux par sa nature, l'être humain essaye de comprendre le fonctionnement de l'univers, en s'inspirant de la nature et en utilisant son intelligence pour apprendre et apporter une touche de création dans chaque génération.

Notre génération coïncide avec une ère d'explosion technologique, telles que l'Internet des objets (IoT), l'intelligence artificielle (IA)..etc, la taille et le volume des données avec lesquelles les entreprises doivent travailler connaissent une croissance exponentielle, qui nécessitent un traitement avec des méthodes de résolution intelligentes pour optimiser la complexité des problèmes traitant cette grande masse de données, en matière de coût en temps et en espace.

D'où la naissances des méthode d'optimisation, qui vise à explorer l'espace de recherche à travers des stratégies permettant de guider à trouver une solution optimale, cette dernière étant généralement non déterministe, ne donne pas une garantie absolue d'optimalité, mais en contrepartie permet de trouver très rapidement de bonne solutions à un problemes donné.

Dans ce projet, nous allons voir l'algorithme immunitaire pour résoudre le problème du voyageur de commerce. Nous evaluerons nos solutions plusieurs paramètres et méthodes. Avant, nous donnons un aperçu sur la définition des algorithmes d'optimisation et les problèmes combinatoires.

2 Les problèmes combinatoires et les algorithmes d'optimisation

Dans la mathématique, en particulier, dans la théorie des graphes, on recense un grand nombre de problèmes combinatoires, qu'on retrouve parfois dans la vie quotidienne, et très souvent dans le monde industriel et dans le monde de la recherche.

D'où la naissance de l'optimisation combinatoire, une branche liée à la recherche opérationnelle qui consiste à trouver dans un ensemble discret de solutions réalisable, la meilleure solution à un problème donné, comme par exemple, trouver le plus court chemin entre deux sommets, ou trouver le meilleur coup d'une position comme dans le jeu d'échecs.

Trouver une solution optimale dans un ensemble discret et fini est un problème facile en théorie, il suffit d'essayer toutes les solutions, et de comparer leurs qualités pour en tirer la meilleure solution. Cependant, en pratique, l'énumération de toutes les solutions est très coûteuse en temps et en espace; deux facteurs qui déterminent la difficulté et la classe d'un problème donné.

3 L'algorithme immunitaire

L'algorithme immunitaire, s'inspirant du système immunitaire, se déroule à travers plusieurs étapes, comprenant la création d'une population, le clonage des individus les plus performants, la sélection et la mutation. Souvent, ces étapes font l'objet de plusieurs itérations avant d'aboutir à une solution satisfaisante, mais qui n'est pas forcément la plus optimial,

4 Problème du voyageur de commerce

Le problème du voyageur de commerce (TSP) est un problème d'optimisation combinatoire. Il consiste à trouver le parcours le plus court qui visite chaque ville d'une liste donnée exactement une fois et retourne à la ville d'origine.

5 Implementaion

- Nous avons implémenté les méthodes muteAC, mutationClones, selectionMeilleursEtClones-Mutes, mutationMoinsBons, et remplacementMauvaisParNouveaux.
- Pour la méthode mutationMoinsBon, nous effectuons de moins en moins de mutations sur les anticorps qui se rapprochent des meilleurs individus.
- Dans le code C, nous avons ajouté des arguments pour permettre le changement des types de méthodes. L'exécution se fait via une commande prenant en compte un type pour la méthode muteAC (muteAc_echange: 0, muteAC_inversion : 1, muteAc_translation : 2) et un type pour la sélection des meilleurs individus (selectionMeilleur_deux_à_deux : 0, selectionMeilleur_trie : 1).
- Nous avons également développé un script Python pour exécuter plusieurs fois le programme.c, en calculant au passage le temps d'exécution.

Caractéristiques de la machine de test :

AMD Ryzen 7 5800X 8-Core Processor

6 Evaluation et Analyse des résultats

Pour 8 et 16 villes, le programme a été exécuté 500 fois chacun. Pour 30 villes, il a été exécuté 2000 fois, et pour 100 villes, le programme a été exécuté 834 fois, totalisant ainsi 3834 essais.

Les paramètres ont été générés de manière aléatoire dans les plages suivantes :

- nbIndividus : un nombre aléatoire entre (100, 2000)
- pourcentageClone : un nombre aléatoire entre (2, 100)
- pourcentageNouveauxIndividus : un nombre aléatoire entre (2, 100)
- nbGenerations: un nombre aléatoire entre (100, 5000)
- nbGenerationInjection: un nombre aléatoire entre (5, nbGenerations - nbGenerations * 0.5)
- typeMuteAc: entre (muteAc_echange:0, muteAC_inversion : 1, muteAc_translation : 2)
- typeMeilleur: égal à (selectionMeilleur_deux_à_deux : 0, selectionMeilleur_trie : 1)

Pour la méthode mutationClones, nous avons maintenu le nombre de mutations à 1, car sur plusieurs essais, nous avons remarqué qu'augmenter le nombre à plus de 2 réduisait la performance de la solution.

6.1 Analyse des données

[ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

[ ]:
```
# Chargement des résultats
file_path_8 = "resultat8.csv" # résultat pour 8 villes
file_path_16 = "resultat16.csv" # résultat pour 16 villes
file_path_30 = "resultat30.csv" # résultat pour 30 villes
file_path_100 = "resultat100.csv" # résultat pour 100 villes

# Lecture des fichiers :
data_8 = pd.read_csv(file_path_8)
data_16 = pd.read_csv(file_path_16)
data_30 = pd.read_csv(file_path_30)
data_100 = pd.read_csv(file_path_100)

# Creation des dataframes
df_8 = pd.DataFrame(data_8)
df_16 = pd.DataFrame(data_16)
df_30 = pd.DataFrame(data_30)
df_100 = pd.DataFrame(data_100)
```

[ ]:
```
df_30.head(10) # permet d'afficher les 10 première valeurs dans df_100
```

[ ]:

	nbIndividus	nbColnes	nbNouveaux	nbGeneration	nbGenerationInjection	\
0	240	54	94	1239	26	
1	1380	97	21	931	108	

```
2          925        82           7          765                   216
3          408        14          93         1045                   328
4          834        49          64         1374                    62
5          263        19          99         1744                   728
6         1681        82           4          777                   205
7          649         7          20          867                   347
8         1738       100          17         2403                   856
9          440        87          49         2078                   927
```

```
   methodeMutAc  methodeSelection       cout  tempExecution
0             2                  1  52.551971          5.784
1             0                  0  62.812521          5.915
2             2                  0  54.779443          6.009
3             1                  0  56.919269          7.670
4             0                  0  60.795506         10.434
5             0                  0  60.981239          6.765
6             1                  1  47.598404          8.441
7             0                  1  65.078007         15.021
8             0                  0  52.951005          7.739
9             0                  0  56.039492          5.745
```

[ ]:
```
# Suppression des lignes dupliquées

df_8.drop_duplicates(inplace=True)
df_16.drop_duplicates(inplace=True)
df_30.drop_duplicates(inplace=True)
df_100.drop_duplicates(inplace=True)
```

[ ]:
```
df_100.info() # affiche les informations concernant le dataframe df_100
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 834 entries, 0 to 833
Data columns (total 9 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   nbIndividus            834 non-null    int64
 1   nbColnes               834 non-null    int64
 2   nbNouveaux             834 non-null    int64
 3   nbGeneration           834 non-null    int64
 4   nbGenerationInjection  834 non-null    int64
 5   methodeMutAc           834 non-null    int64
 6   methodeSelection       834 non-null    int64
 7   cout                   834 non-null    float64
 8   tempExecution          834 non-null    float64
dtypes: float64(2), int64(7)
memory usage: 65.2 KB
```

[ ]:
```
# Lets check the dimension of the data set
print(df_8.shape) # affiche la dimension de df_30, nous avons tester 500␣
 ↪combinaisons de paramètre differents.
print(df_16.shape) # affiche la dimension de df_100, nous avons tester 500␣
 ↪combinaisons de paramètre differents.
print(df_30.shape) # affiche la dimension de df_30, nous avons tester 2000␣
 ↪combinaisons de paramètre differents.
print(df_100.shape) # affiche la dimension de df_100, nous avons tester 834␣
 ↪combinaisons de paramètre differents.
```

```
(500, 9)
(500, 9)
(2000, 9)
(834, 9)
```

[ ]:
```
print(df_100.columns) # affiche les colonnes de notre dataframe
```

```
Index(['nbIndividus', 'nbColnes', 'nbNouveaux', 'nbGeneration',
       'nbGenerationInjection', 'methodeMutAc', 'methodeSelection', 'cout',
       'tempExecution'],
      dtype='object')
```

[ ]:
```
# Attributs qualitatifs
attributs_qualitatifs = ['methodeMutAc', 'methodeSelection']

# Sélectionner toutes les colonnes sauf 'methodeMutAc' et 'methodeSelection'
df_subset_8 = df_8.drop(columns=attributs_qualitatifs)
df_subset_16 = df_16.drop(columns=attributs_qualitatifs)
df_subset_30 = df_30.drop(columns=attributs_qualitatifs)
df_subset_100 = df_100.drop(columns=attributs_qualitatifs)
```

Pour 8 villes

[ ]:
```
# cette commande génère un résumé statistique des données contenues dans le␣
 ↪DataFrame df_subset_8.
df_subset_8.describe()
```

[ ]:

	nbIndividus	nbColnes	nbNouveaux	nbGeneration	\
count	500.00000	500.000000	500.0000	500.00000	
mean	15.97400	50.540000	51.2580	19.76000	
std	8.79181	30.121271	28.1866	6.17695	
min	1.00000	2.000000	2.0000	10.00000	
25%	9.00000	25.000000	27.0000	14.00000	
50%	16.00000	50.000000	52.5000	20.00000	
75%	24.00000	77.250000	74.0000	25.00000	
max	30.00000	100.000000	100.0000	30.00000	

```
       nbGenerationInjection        cout  tempExecution
count             500.000000  500.000000     500.000000
mean                7.246000   20.771967       5.020190
std                 2.290847    3.050242       0.019378
min                 5.000000   17.888544       5.005000
25%                 5.000000   17.888544       5.008000
50%                 7.000000   21.659049       5.012000
75%                 9.000000   21.659049       5.025000
max                15.000000   37.404533       5.124000
```

Nous pouvons remarquer que le coût minimum est : 17.888544

Pour 16 villes

[ ]:
```
# cette commande génère un résumé statistique des données contenues dans le DataFrame df_subset_16.
df_subset_16.describe()
```

[ ]:
```
       nbIndividus    nbColnes  nbNouveaux  nbGeneration  \
count   500.000000  500.000000  500.000000    500.000000
mean    147.458000   51.658000   50.766000    158.292000
std      87.995356   28.364876   28.939977     82.763896
min       1.000000    2.000000    2.000000     10.000000
25%      70.750000   28.000000   24.000000     88.750000
50%     148.000000   53.000000   50.000000    159.500000
75%     224.000000   75.000000   75.000000    228.250000
max     300.000000  100.000000  100.000000    300.000000

       nbGenerationInjection        cout  tempExecution
count              500.00000  500.000000     500.000000
mean                41.81200   26.563164       5.050992
std                 33.44215    6.225723       0.061107
min                  5.00000   19.313708       5.005000
25%                 14.00000   21.135563       5.016000
50%                 31.00000   25.489178       5.029500
75%                 62.25000   30.891820       5.059250
max                141.00000   50.734736       5.492000
```

Nous pouvons remarquer que le coût minimum est : 19.313708

30 villes

[ ]:
```
df_subset_30.describe()
```

[ ]:
```
       nbIndividus     nbColnes   nbNouveaux  nbGeneration  \
count  2000.000000  2000.000000  2000.000000   2000.000000
mean   1054.465000    51.972000    50.639500   1279.456000
std     547.986576    28.483677    28.453899    693.577647
min     100.000000     2.000000     2.000000    100.000000
```

```
25%     581.000000    27.000000    26.000000    669.000000
50%    1047.500000    53.000000    51.000000   1262.000000
75%    1524.000000    77.000000    75.000000   1876.750000
max    2000.000000   100.000000   100.000000   2499.000000

       nbGenerationInjection          cout  tempExecution
count             2000.00000  2000.000000    2000.000000
mean               315.11700    54.903703      20.248301
std                268.52591     7.303956      25.843780
min                  5.00000    46.371631       5.027000
25%                 94.00000    48.973273       6.310250
50%                244.00000    53.197382       9.916500
75%                471.00000    58.968346      21.208750
max               1220.00000    85.342069     243.020000
```

Nous pouvons remarquer que le coût minimum est : 46.371631

100 villes

[ ]:
```
df_subset_100.describe()
```

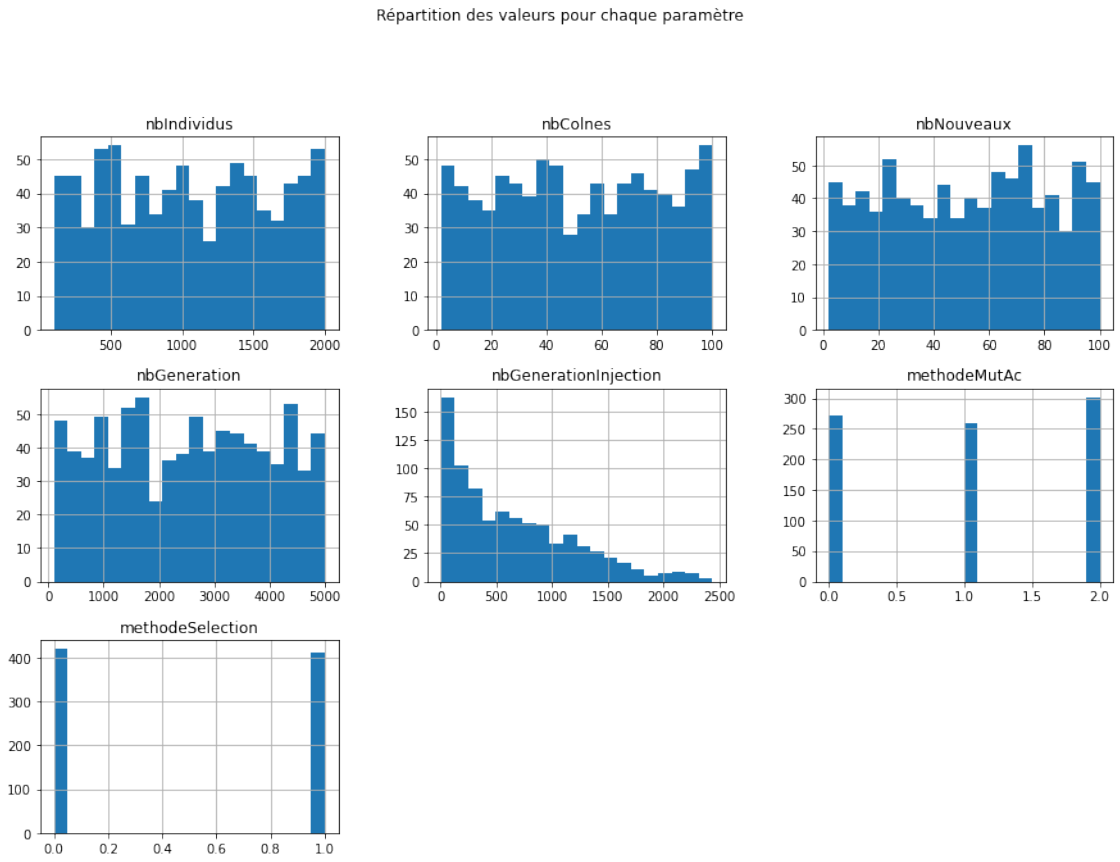
[ ]:
```
       nbIndividus    nbColnes  nbNouveaux  nbGeneration  \
count    834.00000  834.000000  834.000000    834.000000
mean    1050.20024   51.571942   51.571942   2535.357314
std      561.42850   29.166084   28.692604   1421.402567
min      101.00000    2.000000    2.000000    103.000000
25%      547.00000   27.000000   26.000000   1331.250000
50%     1031.50000   51.000000   52.000000   2572.500000
75%     1522.75000   77.000000   75.000000   3754.500000
max     1999.00000  100.000000  100.000000   4999.000000

       nbGenerationInjection         cout  tempExecution
count             834.000000  834.000000     834.000000
mean              645.526379  204.475814      45.508017
std               548.138005   37.648080      70.178670
min                 5.000000  145.170299       5.103000
25%               182.500000  172.954055       9.717750
50%               515.500000  200.025568      19.484500
75%               985.000000  224.498748      47.015750
max              2439.000000  341.860032     753.186000
```

Nous pouvons remarquer que le coût minimum est : 145.170299

[ ]:
```
# Sélectionner toutes les colonnes sauf 'cout' et 'tempExecution'
df_100_dist = df_100.drop(columns=['cout', 'tempExecution'])

# Dessiner des histogrammes pour les variables numériques de df_100
plt.figure(figsize=(15, 10))
```

```
df_100_dist.hist(bins=20, figsize=(15, 10))
plt.suptitle('Répartition des valeurs pour chaque paramètre', y=1.02)
plt.show()
```

```
<Figure size 1080x720 with 0 Axes>
```



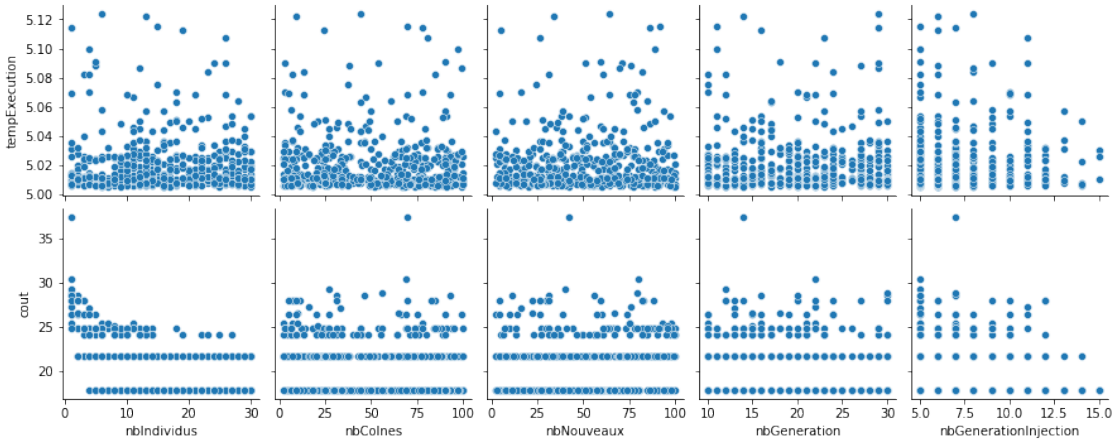
Comme illustré par les graphiques, nous avons systématiquement varié les valeurs de nos paramètres de manière aléatoire afin d'évaluer leur impact sur le coût. Cette approche nous a permis d'explorer de manière exhaustive l'influence de chaque paramètre sur les résultats obtenus.

6.2 Analyse du coût et du temps d'exécution par rapport aux paramètres : 'nbIndividus', 'nbClones', 'nbNouveaux', 'nbGeneration', 'nbGenerationInjection'.

Un scatter plot est un graphique qui représente la distribution de deux variables en affichant les points de données selon leurs coordonnées, permettant ainsi de visualiser les relations et tendances entre ces variables.

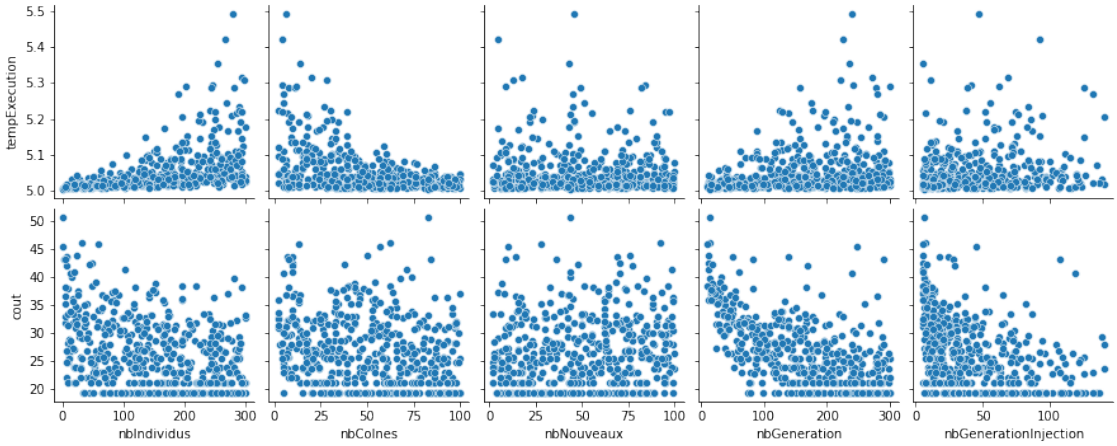
8 villes

[ ]:
```
# Pairwise Scatter Plots with tempExecution on the y-axis
sns.pairplot(df_8, x_vars=['nbIndividus', 'nbColnes', 'nbNouveaux',␣
 ↪'nbGeneration', 'nbGenerationInjection'],
             y_vars=['tempExecution', 'cout'], kind='scatter')
plt.show()
```



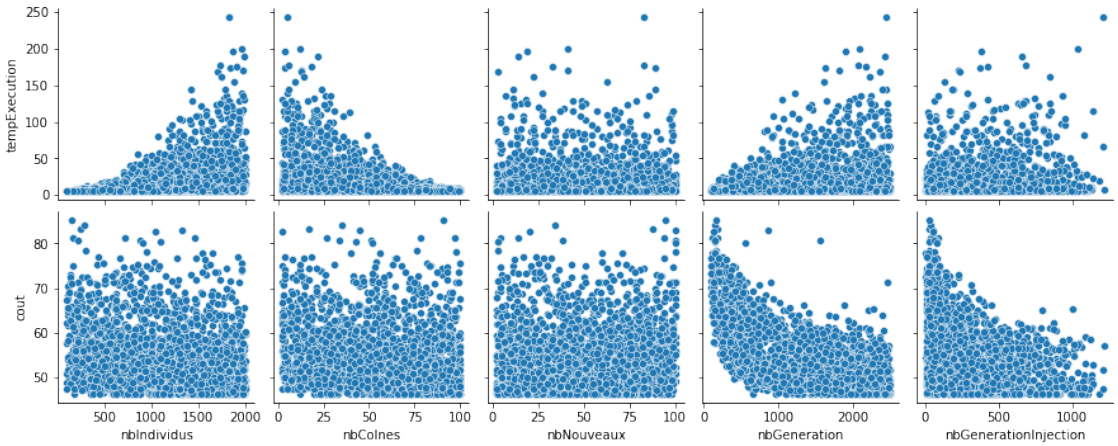
16 villes

[ ]:
```
sns.pairplot(df_16, x_vars=['nbIndividus', 'nbColnes', 'nbNouveaux',␣
 ↪'nbGeneration', 'nbGenerationInjection'],
             y_vars=['tempExecution', 'cout'], kind='scatter')
plt.show()
```



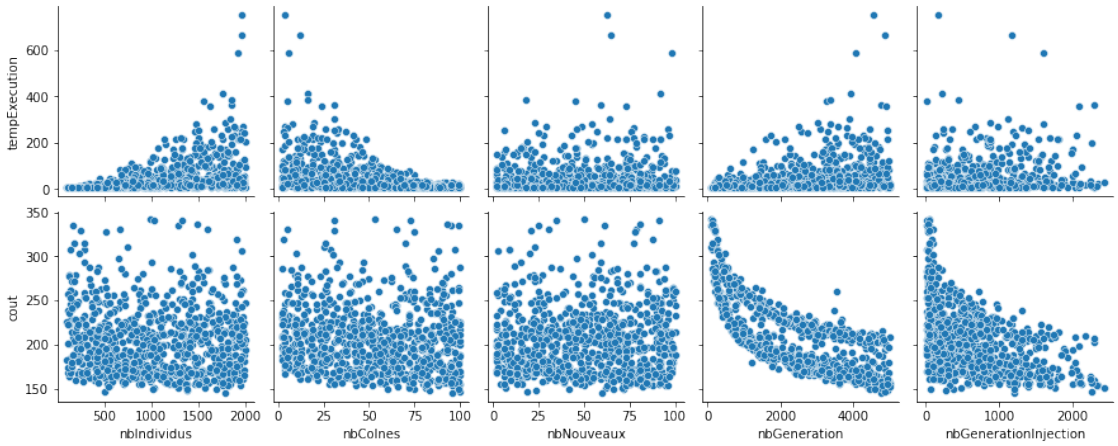
30 villes

[ ]:
```
sns.pairplot(df_30, x_vars=['nbIndividus', 'nbColnes', 'nbNouveaux',␣
 ↪'nbGeneration', 'nbGenerationInjection'],
             y_vars=['tempExecution', 'cout'], kind='scatter')
plt.show()
```



100 villes

[ ]:
```
import seaborn as sns
import matplotlib.pyplot as plt
sns.pairplot(df_100, x_vars=['nbIndividus', 'nbColnes', 'nbNouveaux',␣
 ↪'nbGeneration', 'nbGenerationInjection'],
             y_vars=['tempExecution', 'cout'], kind='scatter')
plt.show()
```



Nous observons :

- Avec 8 villes, nous ne pouvons pas vraiment voir l'impact des changements des paramètres.
- À partir de 16 villes, nous pouvons observer certaines tendances et l'impact de certains paramètres sur le coût et le temps d'exécution. Parmi les tendances que nous pouvons remarquer :
  - Le nombre d'individus influence légèrement le coût ; plus le nombre d'individus augmente, plus le coût diminue, cependant, le temps d'exécution augmente.
  - Nous remarquons aussi que le nombre de clones a une influence assez aléatoire sur le coût, avec une légère amélioration à partir de 20% de clones. Cependant, on observe que plus on augmente le nombre de clones, plus le temps d'exécution diminue, montrant ainsi qu'une proportion entre 75% et 100% donne de bons résultats en termes de temps d'execution.
  - Le nombre de nouveaux individus influence légèrement les performances en termes de coût et de temps d'exécution. Une légère préférence est observée entre 20% et 50%.
  - Plus on augmente le nombre de générations, plus le coût diminue, mais le temps d'exécution augmente.
  - Plus on augmente le nombre de générations d'injection, plus le coût diminue avec une légère influence sur le temps d'exécution qui augmente légèrement.
- Avec 30 et 100 villes, nous confirmons l'influence des paramètres et les résultats obtenus avec 16 villes. En effet, nous obtenons des résultats assez similaires avec 30 et 100 villes, et nous observons :
  - Qu'à partir de plus de 400 individus, l'impact de ce paramètre sur la performance du coût se stabilise.
  - Le nombre d'itérations influe fortement sur la performance de la solution en termes de coût et dépend particulièrement du nombre de villes ; plus on augmente le nombre de villes, plus il est nécessaire d'augmenter le nombre de génération.

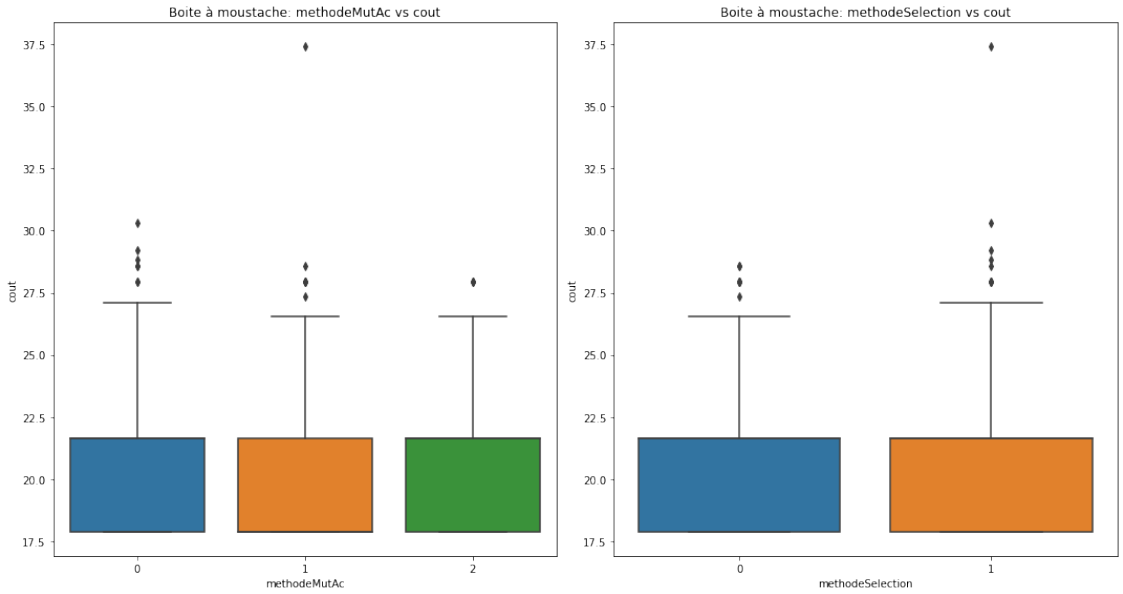
6.2.1 Analyse du coût et du temps d'exécution par rapport aux méthodes Selection-Meilleur et MuteAc en utilisant les boîtes à moustaches

Les boîtes à moustaches (boxplots) fournissent une visualisation synthétique de la distribution et de la dispersion des données, mettant en évidence les quartiles, les médianes et les valeurs aberrantes dans un ensemble de données.

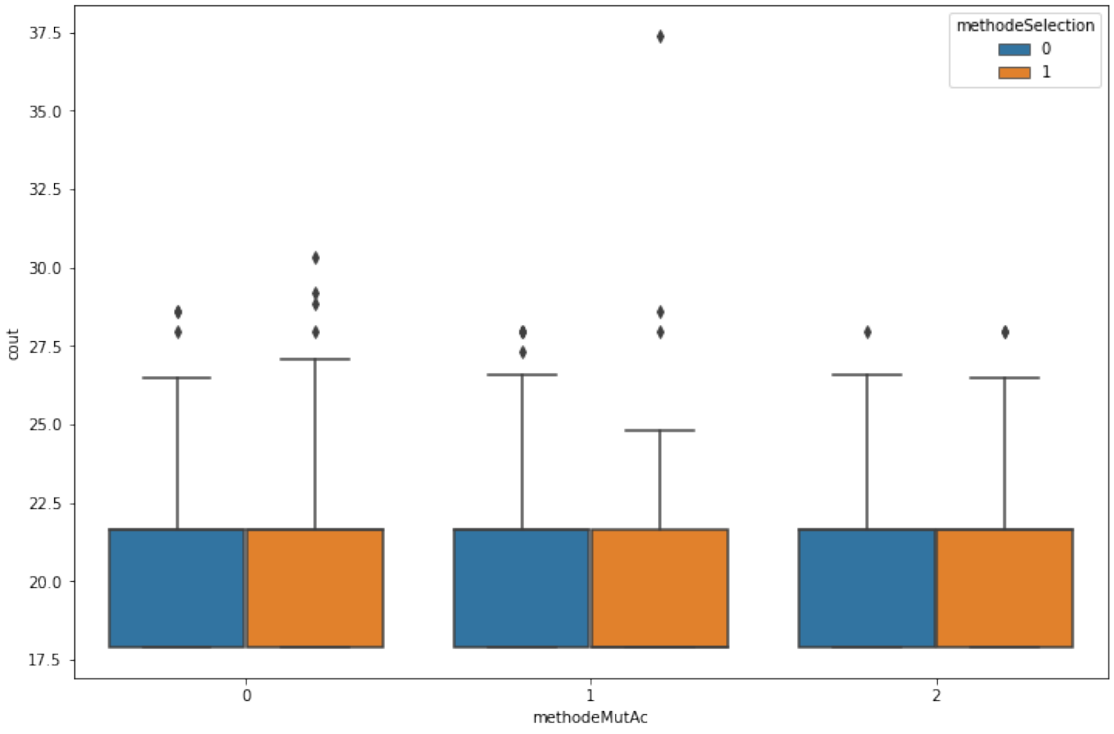
8 villes :

Cout :

[ ]:
```
plt.figure(figsize=(15, 8))
for i, column in enumerate(['methodeMutAc', 'methodeSelection']):
    plt.subplot(1, 2, i + 1)
    sns.boxplot(x=column, y='cout', data=df_8)
    plt.title(f'Boite à moustache: {column} vs cout')

plt.tight_layout()
plt.show()
```

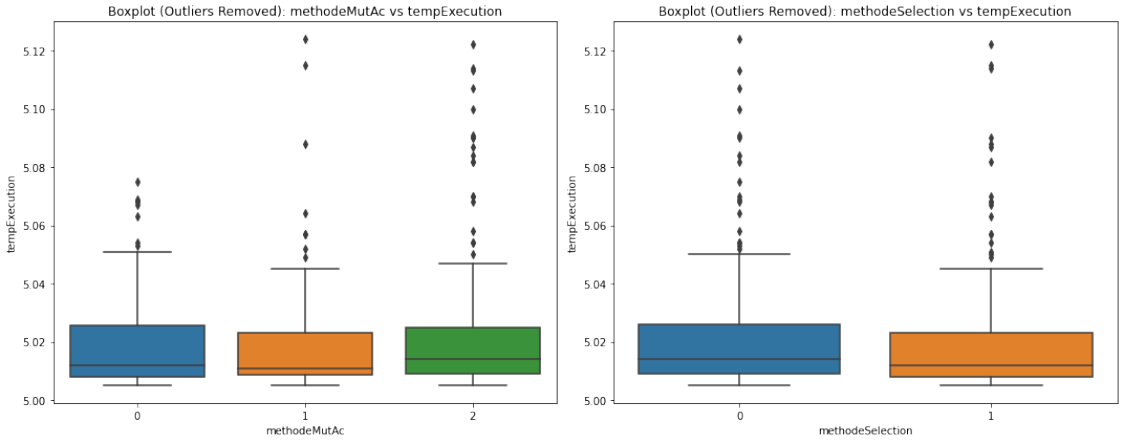


[ ]:
```
plt.figure(figsize=(12, 8))
sns.boxplot(x='methodeMutAc', y='cout', hue='methodeSelection', data=df_8)
plt.show()
```

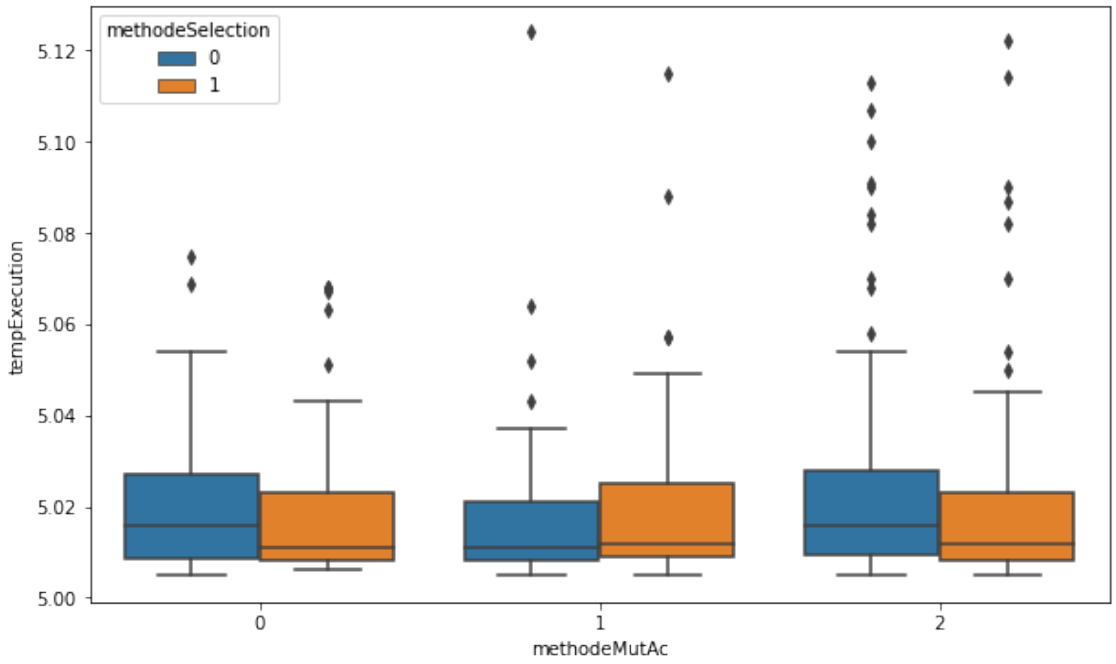


Temps d'execution :

[ ]:
```
# Display boxplots after removing outliers
plt.figure(figsize=(15, 6))
for i, column in enumerate(['methodeMutAc', 'methodeSelection']):
    plt.subplot(1, 2, i + 1)
    sns.boxplot(x=column, y='tempExecution', data=df_8)
    plt.title(f'Boxplot (Outliers Removed): {column} vs tempExecution')

plt.tight_layout()
plt.show()
```



[ ]:
```
plt.figure(figsize=(10, 6))
sns.boxplot(x='methodeMutAc', y='tempExecution', hue='methodeSelection',␣
↪data=df_8)
plt.show()
```

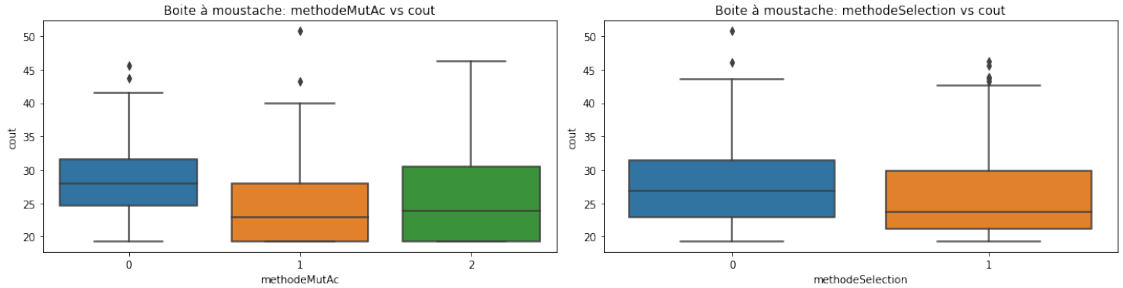


Avec seulement 8 villes, il est difficile de déterminer la méthode qui offre les meilleurs résultats en terme de cout et de temps d'execution.

16 villes :

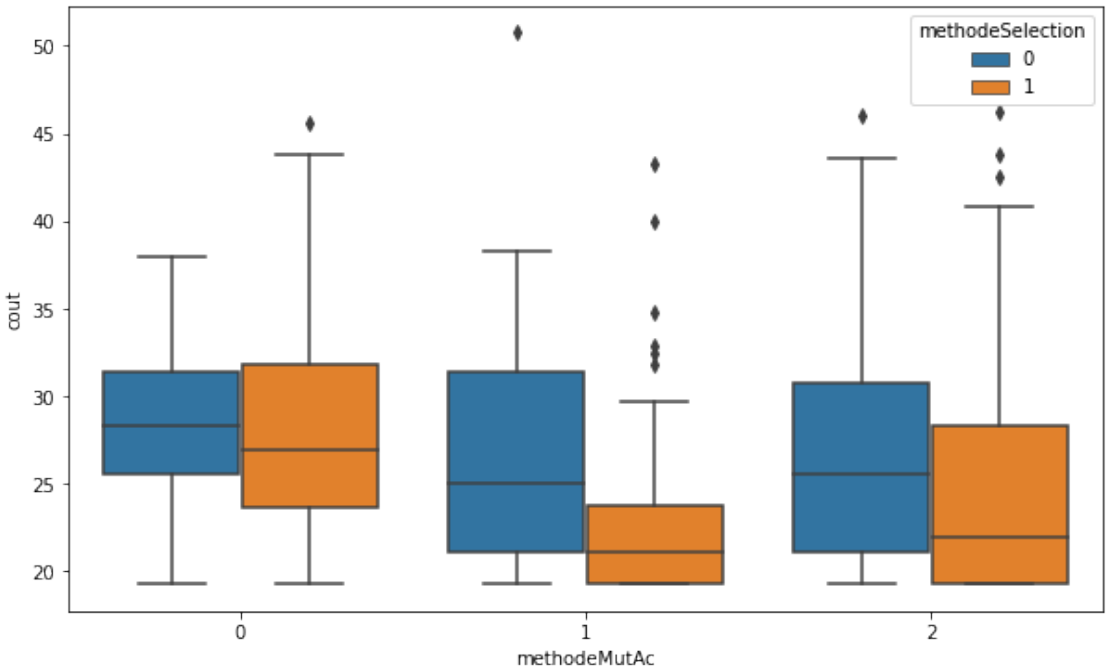
Couts :

```
[ ]: plt.figure(figsize=(15, 4))
for i, column in enumerate(['methodeMutAc', 'methodeSelection']):
    plt.subplot(1, 2, i + 1)
    sns.boxplot(x=column, y='cout', data=df_16)
    plt.title(f'Boite à moustache: {column} vs cout')

plt.tight_layout()
plt.show()
```



On remarque que pour ce cas d'étude, la méthode muteAc inversion donne généralement de meilleurs résultats par rapport aux deux autres méthodes, échange et translation, de même pour la méthode de selectionMeilleur Trie.

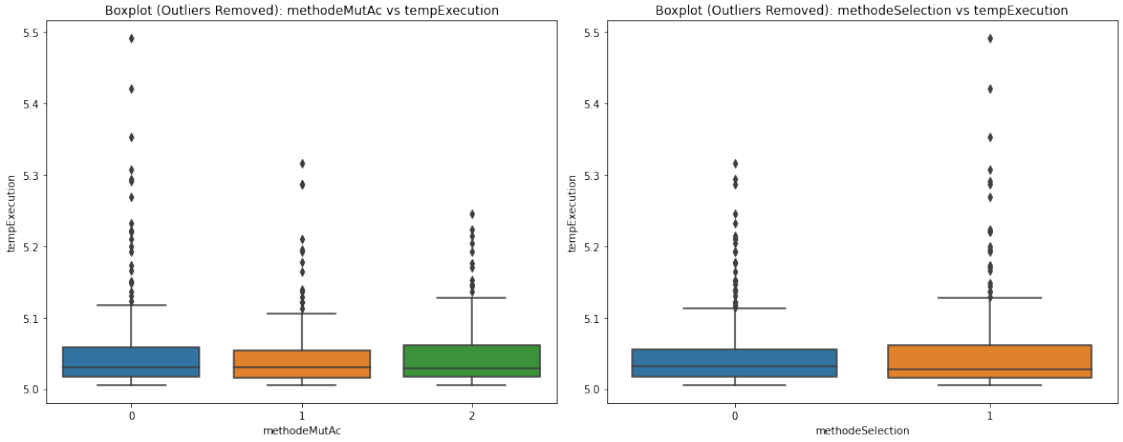
[ ]:
```
plt.figure(figsize=(10, 6))
sns.boxplot(x='methodeMutAc', y='cout', hue='methodeSelection', data=df_16)
plt.show()
```



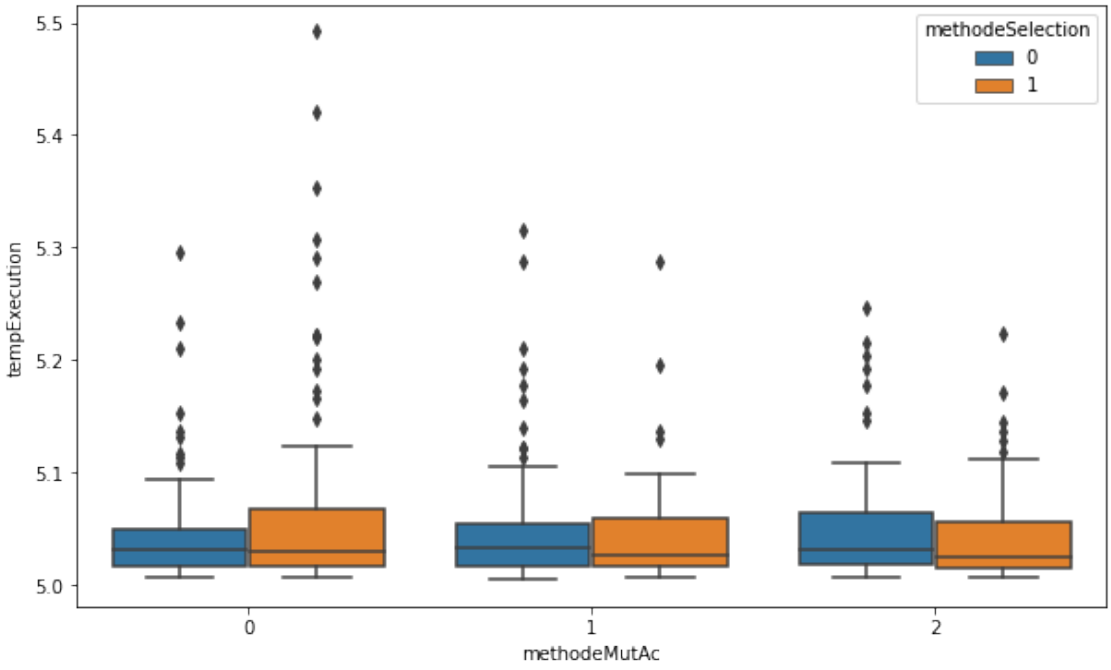
Ce schéma nous permet d'observer la combinaison des différentes méthodes. Comme vu précédemment, la combinaison de muteAc en inversion avec la méthode selectionMeilleur en tri donne les meilleurs résultats.

Temps d'execution :

[ ]:
```
plt.figure(figsize=(15, 6))
for i, column in enumerate(['methodeMutAc', 'methodeSelection']):
    plt.subplot(1, 2, i + 1)
    sns.boxplot(x=column, y='tempExecution', data=df_16)
    plt.title(f'Boxplot (Outliers Removed): {column} vs tempExecution')

plt.tight_layout()
plt.show()
```



[ ]:
```
plt.figure(figsize=(10, 6))
sns.boxplot(x='methodeMutAc', y='tempExecution', hue='methodeSelection',␣
↪data=df_16)
plt.show()
```

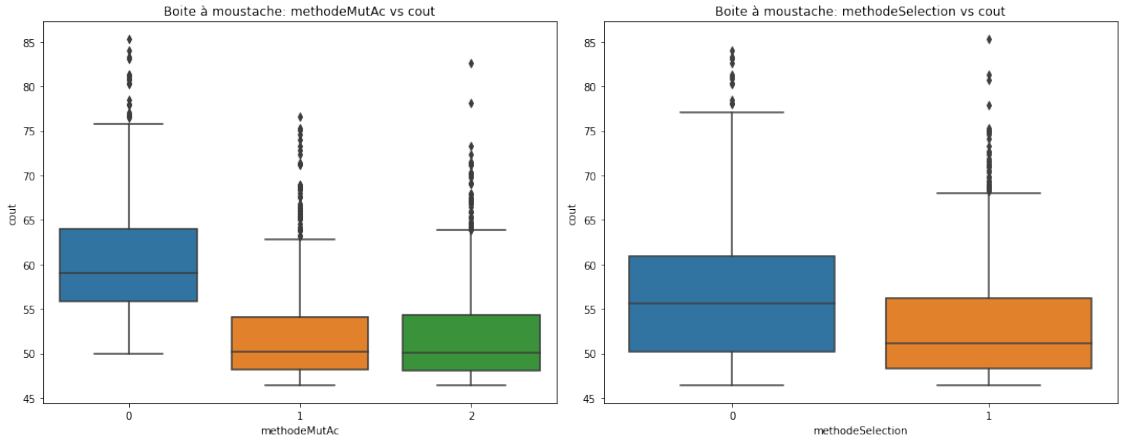


Nous observons que la méthode muteAc inversion est légèrement plus rapide que les deux autres méthodes. Cependant, la méthode selectionMeilleur tri est plus lente que celle effectuée deux à deux. Nous remarquons également que la combinaison de muteAc échange avec la méthode selectionMeilleur deux à deux consomme le moins de temps.

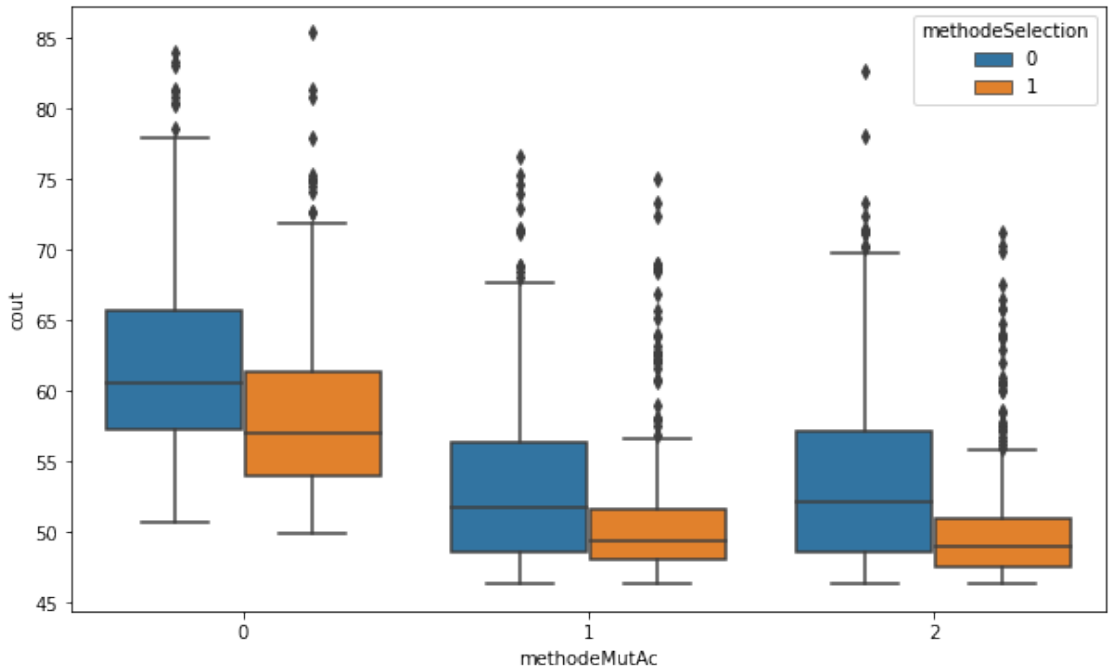
30 villes :

Couts :

[ ]:
```
plt.figure(figsize=(15, 6))
for i, column in enumerate(['methodeMutAc', 'methodeSelection']):
    plt.subplot(1, 2, i + 1)
    sns.boxplot(x=column, y='cout', data=df_30)
    plt.title(f'Boite à moustache: {column} vs cout')

plt.tight_layout()
plt.show()
```

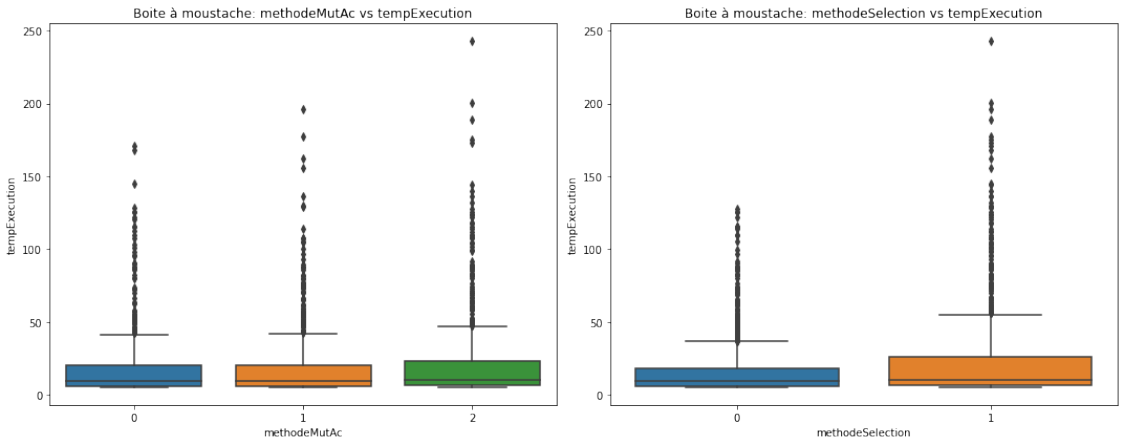


[ ]:
```
plt.figure(figsize=(10, 6))
sns.boxplot(x='methodeMutAc', y='cout', hue='methodeSelection', data=df_30)
plt.show()
```

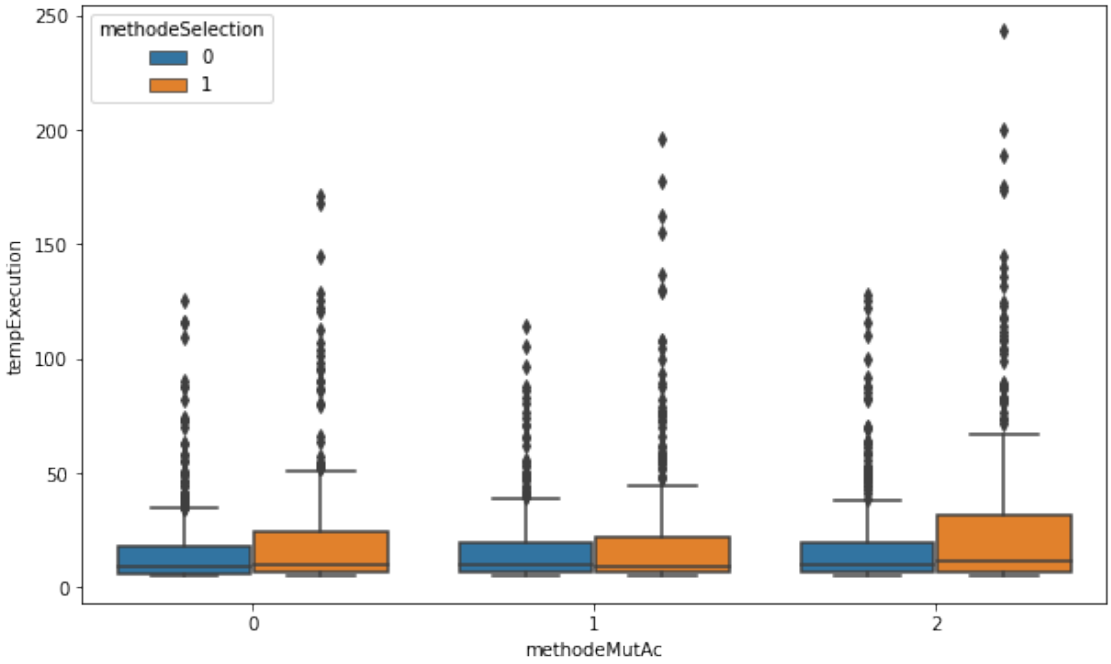


Temps d'execution :

```
[ ]: plt.figure(figsize=(15, 6))
     for i, column in enumerate(['methodeMutAc', 'methodeSelection']):
         plt.subplot(1, 2, i + 1)
         sns.boxplot(x=column, y='tempExecution', data=df_30)
         plt.title(f'Boite à moustache: {column} vs tempExecution')

     plt.tight_layout()
     plt.show()
```



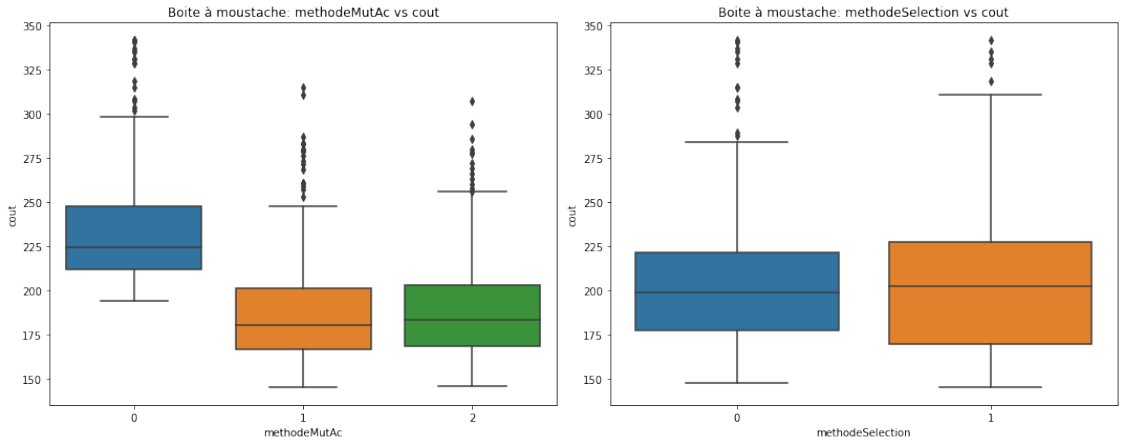
[ ]:
```
plt.figure(figsize=(10, 6))
sns.boxplot(x='methodeMutAc', y='tempExecution', hue='methodeSelection',␣
 ↪data=df_30)
plt.show()
```



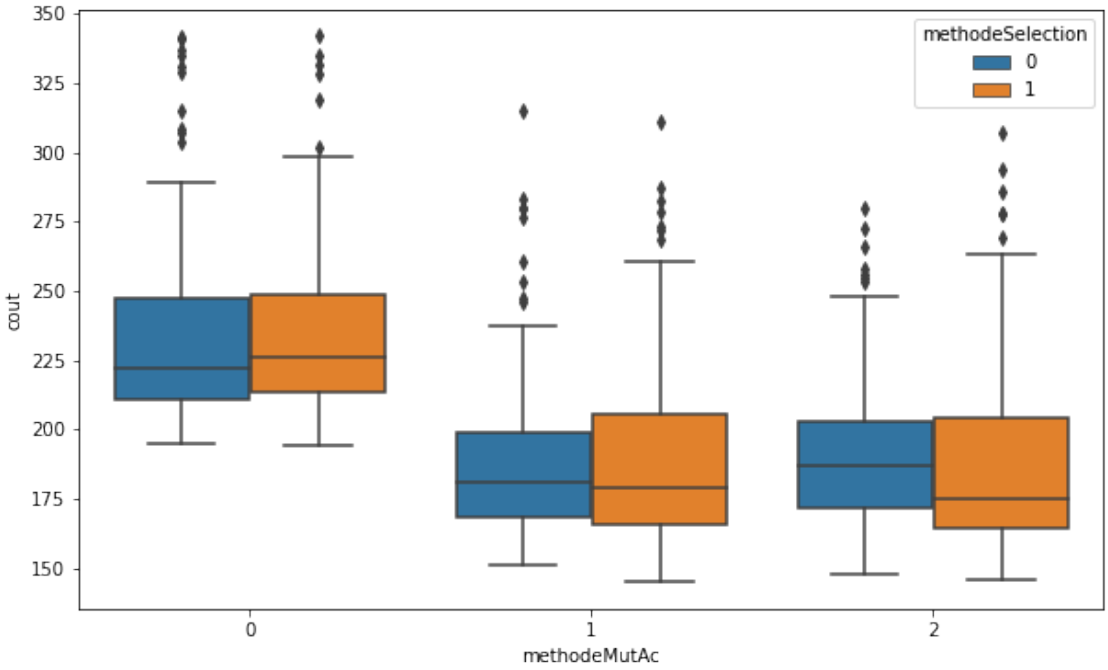
100 villes :

6.2.2 cout :

[ ]:
```
# Display boxplots after removing outliers
plt.figure(figsize=(15, 6))
for i, column in enumerate(['methodeMutAc', 'methodeSelection']):
    plt.subplot(1, 2, i + 1)
    sns.boxplot(x=column, y='cout', data=df_100)
    plt.title(f'Boite à moustache: {column} vs cout')

plt.tight_layout()
plt.show()
```



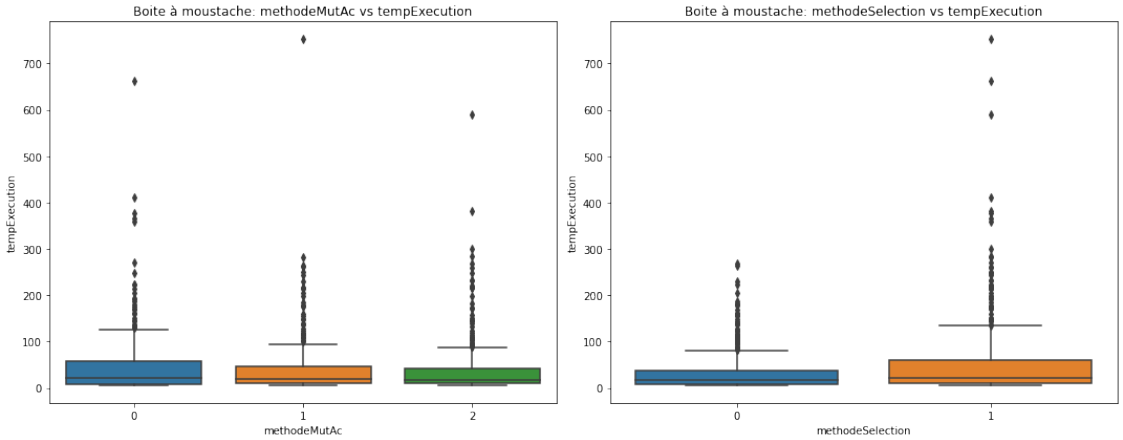
[ ]:
```
plt.figure(figsize=(10, 6))
sns.boxplot(x='methodeMutAc', y='cout', hue='methodeSelection', data=df_100)
plt.show()
```



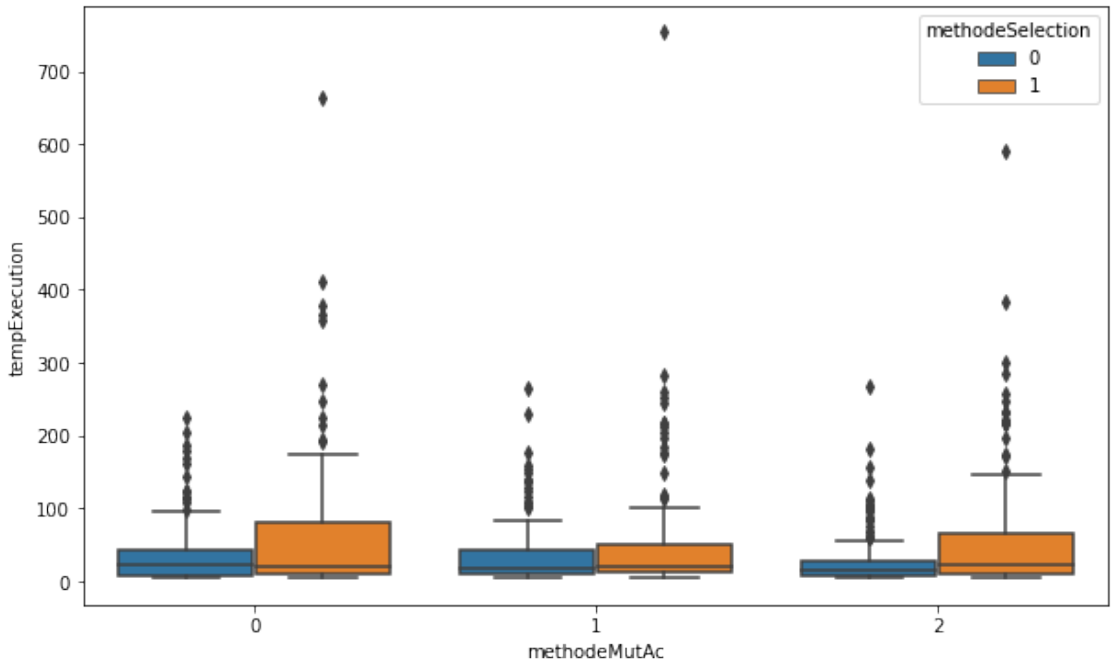
Comme nous avons pu le constater à partir de 16 villes, nous confirmons avec 30 et 100 villes que la méthode d'inversion donne de meilleurs résultats par rapport à la méthode d'échange, avec une légère performance par rapport à la méthode de translation. De plus, la méthode de tri selection-Meilleur montre des meilleurs performance en termes de coût, ce qui implique, d'après le dernier graphique généré, que généralement la combinaison de muteAc en inversion avec selectionMeilleur en tri donne les meilleures performances en termes de coût.

Temps d'execution :

[ ]:
```
# Display boxplots after removing outliers
plt.figure(figsize=(15, 6))
for i, column in enumerate(['methodeMutAc', 'methodeSelection']):
    plt.subplot(1, 2, i + 1)
    sns.boxplot(x=column, y='tempExecution', data=df_100)
    plt.title(f'Boite à moustache: {column} vs tempExecution')

plt.tight_layout()
plt.show()
```



[ ]:
```
plt.figure(figsize=(10, 6))
sns.boxplot(x='methodeMutAc', y='tempExecution', hue='methodeSelection',␣
↪data=df_100)
plt.show()
```



En termes de temps d'exécution, la méthode deux à deux est moins coûteuse que le tri, et généralement, les méthodes d'inversion et de translation présentent les meilleures performances en termes de temps d'exécution. Le programme dépend fortement du nombre de générations établi en paramètres.

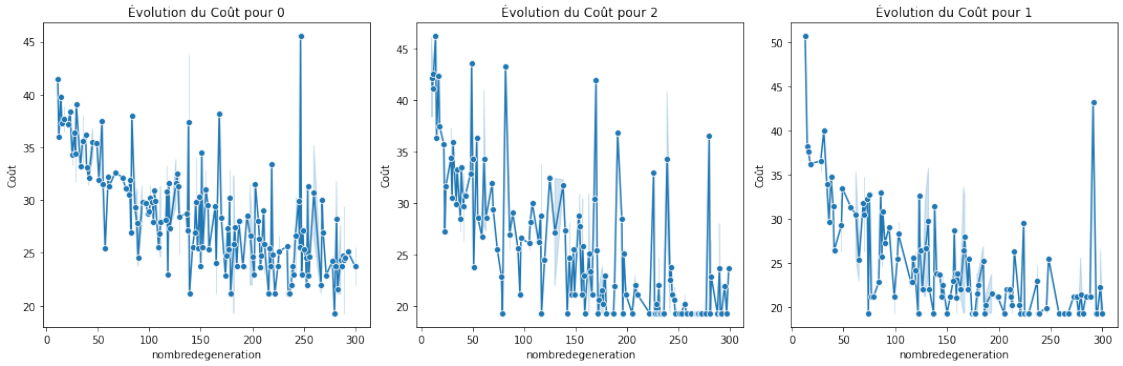
6.3 Analyse du coût et du temps d'exécution par rapport au nombre de générations pour chaque méthode SelectionMeilleur et MuteAc

16 villes :

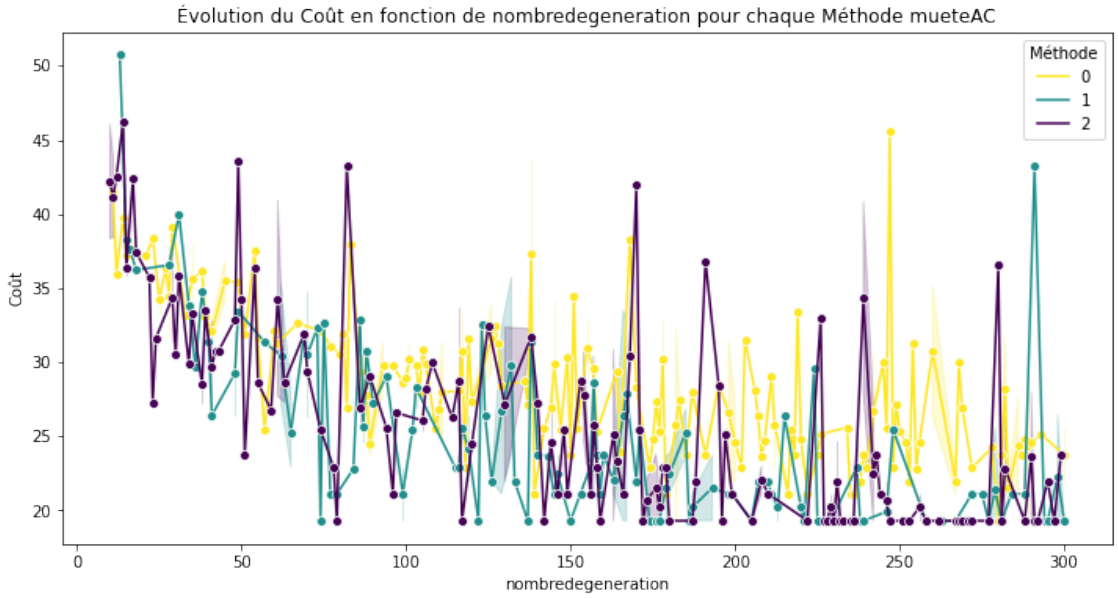
Cout :

[ ]:
```
import seaborn as sns
import matplotlib.pyplot as plt

# Créer un diagramme de lignes pour chaque méthode
methods = df_16['methodeMutAc'].unique()

plt.figure(figsize=(15, 5))

for i, method in enumerate(methods):
    plt.subplot(1, len(methods), i + 1)

    # Filter data for the current method
    method_data = df_16[df_16['methodeMutAc'] == method]
```

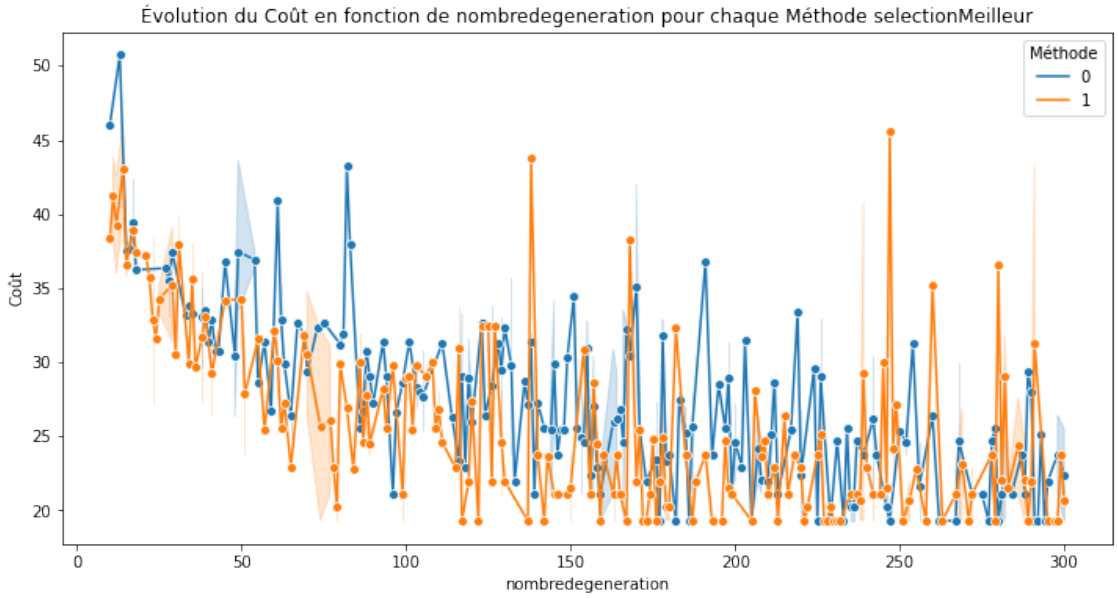
```
    # Create a line plot
    sns.lineplot(x='nbGeneration', y='cout', data=method_data, marker='o')

    # Set titles and labels
    plt.title(f'Évolution du Coût pour {method}')
    plt.xlabel('nombredegeneration')
    plt.ylabel('Coût')

plt.tight_layout()
plt.show()
```



[ ]:
```
# Créer un diagramme de lignes
plt.figure(figsize=(12, 6))
sns.lineplot(x='nbGeneration', y='cout', hue='methodeMutAc', data=df_16,␣
 ↪marker='o', palette='viridis_r')

# Ajouter des titres et des légendes
plt.title('Évolution du Coût en fonction de nombredegeneration pour chaque␣
 ↪Méthode mueteAC')
plt.xlabel('nombredegeneration')
plt.ylabel('Coût')
plt.legend(title='Méthode')

# Afficher le diagramme de lignes
plt.show()
```

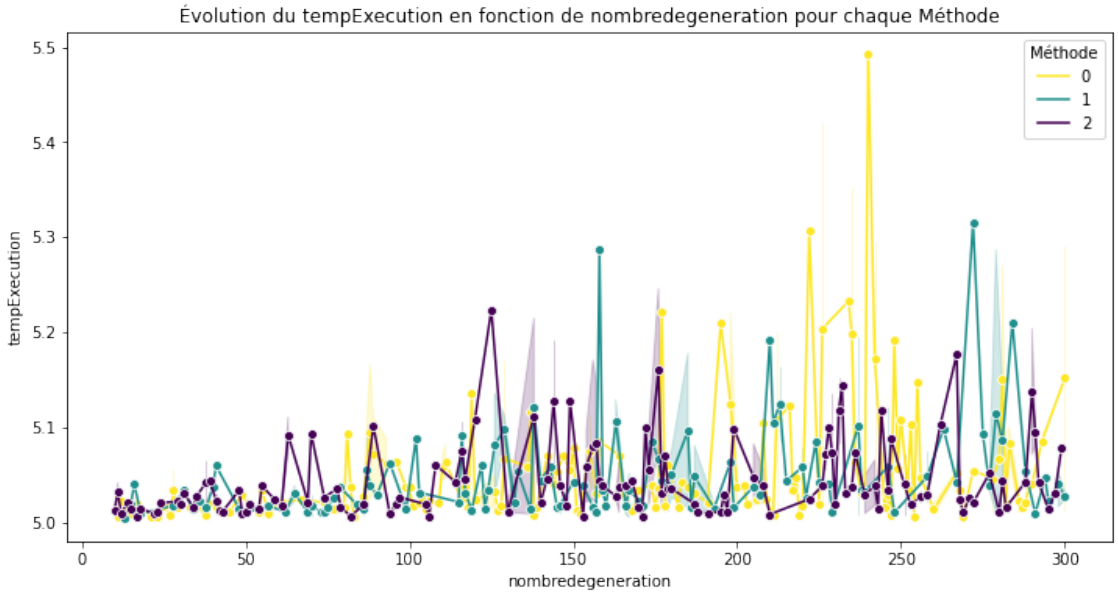


[ ]:
```
# Créer un diagramme de lignes
plt.figure(figsize=(12, 6))
sns.lineplot(x='nbGeneration', y='cout', hue='methodeSelection', data=df_16,␣
 ↪marker='o')

# Ajouter des titres et des légendes
plt.title('Évolution du Coût en fonction de nombredegeneration pour chaque␣
 ↪Méthode selectionMeilleur')
plt.xlabel('nombredegeneration')
plt.ylabel('Coût')
plt.legend(title='Méthode')

# Afficher le diagramme de lignes
plt.show()
```

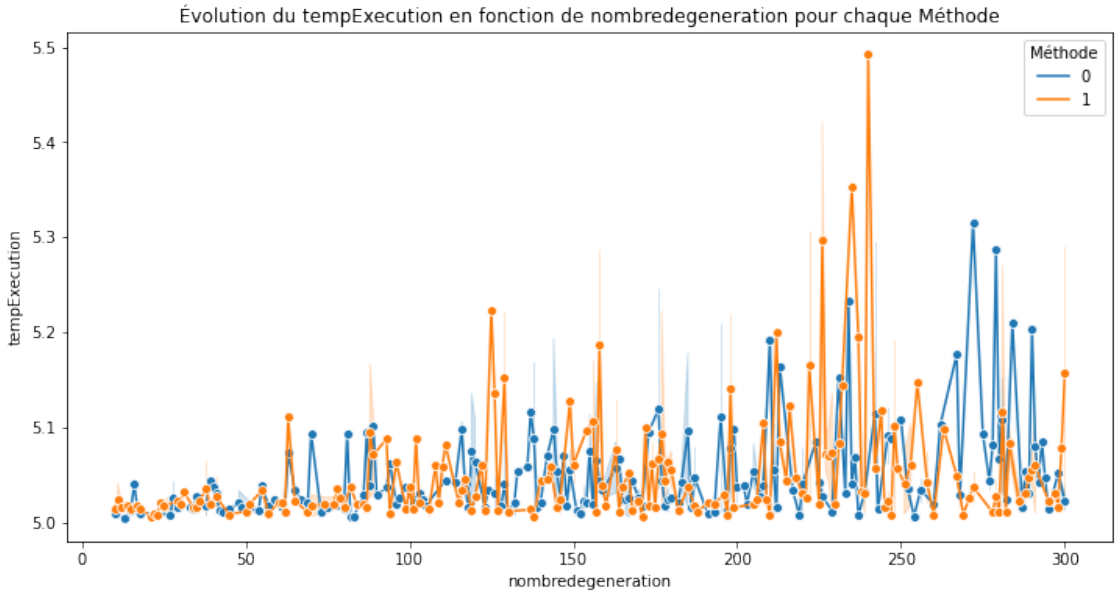


Temps d'execution

[ ]:
```
# Créer un diagramme de lignes
plt.figure(figsize=(12, 6))
sns.lineplot(x='nbGeneration', y='tempExecution', hue='methodeMutAc',␣
 ↪data=df_16, marker='o', palette='viridis_r')

# Ajouter des titres et des légendes
plt.title('Évolution du tempExecution en fonction de nombredegeneration pour␣
 ↪chaque Méthode')
plt.xlabel('nombredegeneration')
plt.ylabel('tempExecution')
plt.legend(title='Méthode')

# Afficher le diagramme de lignes
plt.show()
```



[ ]:
```
# Créer un diagramme de lignes
plt.figure(figsize=(12, 6))
sns.lineplot(x='nbGeneration', y='tempExecution', hue='methodeSelection',␣
 ↪data=df_16, marker='o')

# Ajouter des titres et des légendes
plt.title('Évolution du tempExecution en fonction de nombredegeneration pour␣
 ↪chaque Méthode')
plt.xlabel('nombredegeneration')
plt.ylabel('tempExecution')
plt.legend(title='Méthode')

# Afficher le diagramme de lignes
plt.show()
```

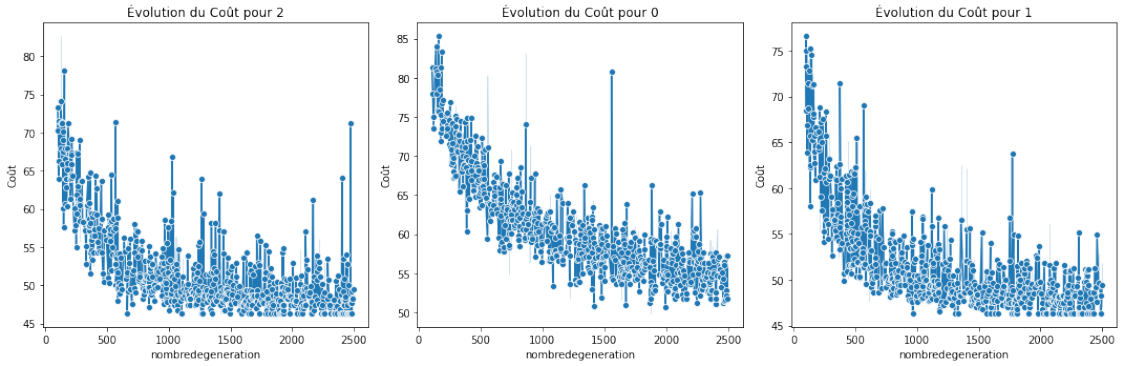


30 villes :

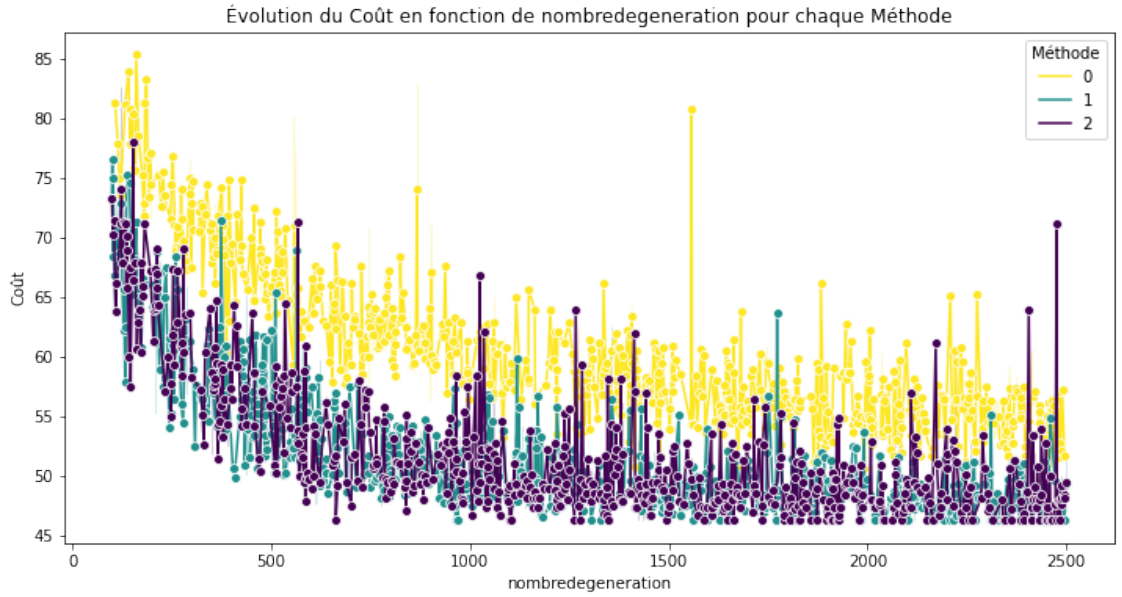
Cout :

```
[ ]: import seaborn as sns
     import matplotlib.pyplot as plt

     # Créer un diagramme de lignes pour chaque méthode
     methods = df_30['methodeMutAc'].unique()

     plt.figure(figsize=(15, 5))

     for i, method in enumerate(methods):
         plt.subplot(1, len(methods), i + 1)

         # Filter data for the current method
         method_data = df_30[df_30['methodeMutAc'] == method]

         # Create a line plot
         sns.lineplot(x='nbGeneration', y='cout', data=method_data, marker='o')

         # Set titles and labels
         plt.title(f'Évolution du Coût pour {method}')
         plt.xlabel('nombredegeneration')
         plt.ylabel('Coût')

     plt.tight_layout()
```

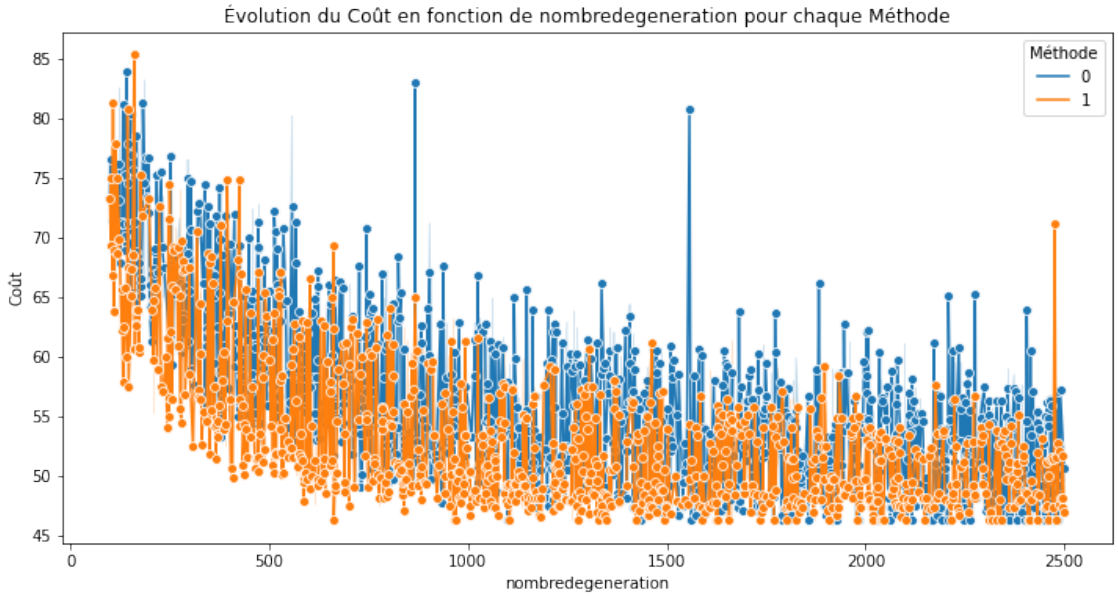
```
plt.show()
```



[ ]:
```
# Créer un diagramme de lignes
plt.figure(figsize=(12, 6))
sns.lineplot(x='nbGeneration', y='cout', hue='methodeMutAc', data=df_30,␣
 ↪marker='o', palette='viridis_r')

# Ajouter des titres et des légendes
plt.title('Évolution du Coût en fonction de nombredegeneration pour chaque␣
 ↪Méthode')
plt.xlabel('nombredegeneration')
plt.ylabel('Coût')
plt.legend(title='Méthode')

# Afficher le diagramme de lignes
plt.show()
```



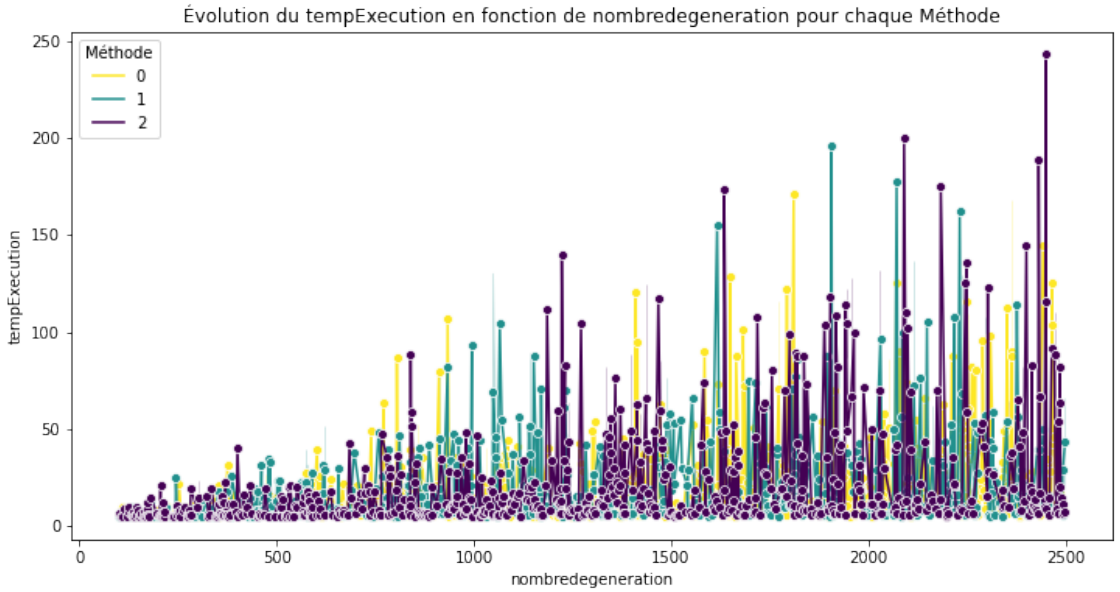
[ ]:
```
# Créer un diagramme de lignes
plt.figure(figsize=(12, 6))
sns.lineplot(x='nbGeneration', y='cout', hue='methodeSelection', data=df_30,␣
 ↪marker='o')

# Ajouter des titres et des légendes
plt.title('Évolution du Coût en fonction de nombredegeneration pour chaque␣
 ↪Méthode')
plt.xlabel('nombredegeneration')
plt.ylabel('Coût')
plt.legend(title='Méthode')

# Afficher le diagramme de lignes
plt.show()
```



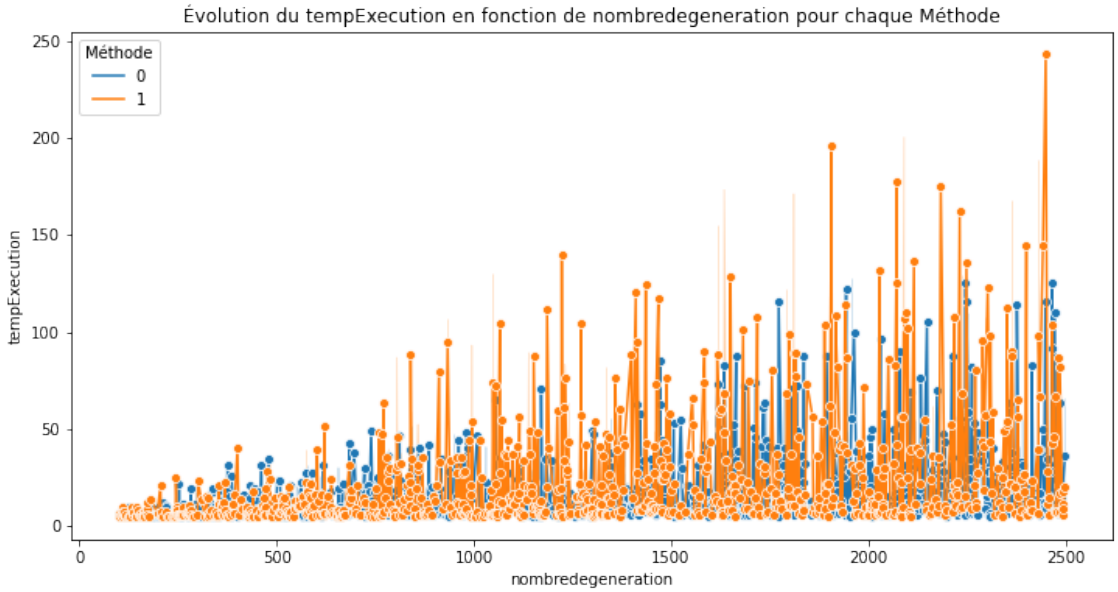
Temps d'execution

[ ]:
```
# Créer un diagramme de lignes
plt.figure(figsize=(12, 6))
sns.lineplot(x='nbGeneration', y='tempExecution', hue='methodeMutAc',␣
 ↪data=df_30, marker='o', palette='viridis_r')

# Ajouter des titres et des légendes
plt.title('Évolution du tempExecution en fonction de nombredegeneration pour␣
 ↪chaque Méthode')
```

```
plt.xlabel('nombredegeneration')
plt.ylabel('tempExecution')
plt.legend(title='Méthode')

# Afficher le diagramme de lignes
plt.show()
```



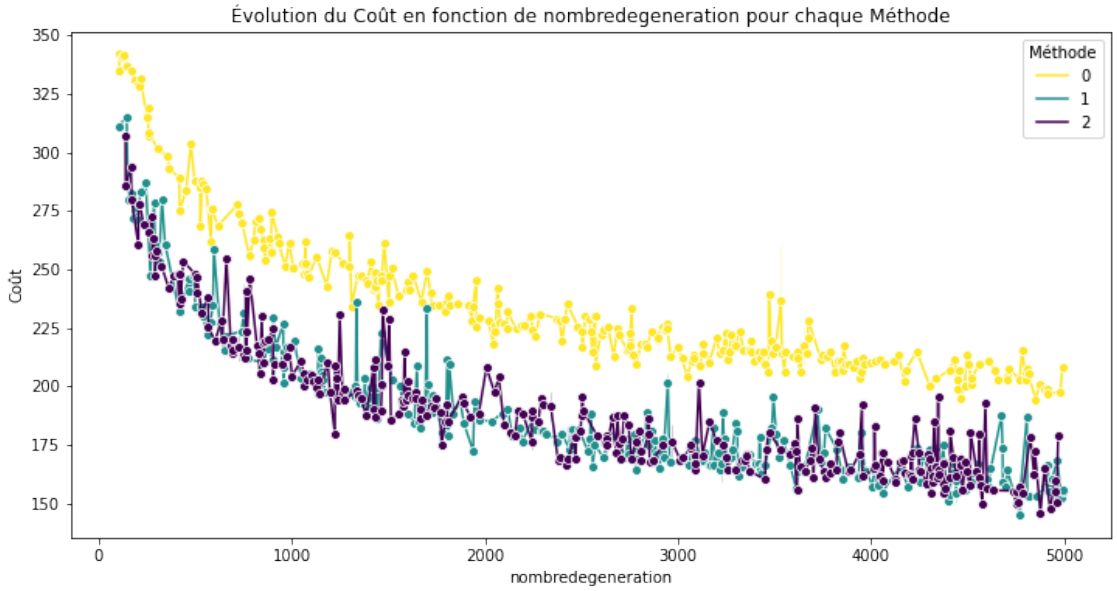
[ ]:
```
# Créer un diagramme de lignes
plt.figure(figsize=(12, 6))
sns.lineplot(x='nbGeneration', y='tempExecution', hue='methodeSelection',␣
 ↪data=df_30, marker='o')

# Ajouter des titres et des légendes
plt.title('Évolution du tempExecution en fonction de nombredegeneration pour␣
 ↪chaque Méthode')
plt.xlabel('nombredegeneration')
plt.ylabel('tempExecution')
plt.legend(title='Méthode')

# Afficher le diagramme de lignes
plt.show()
```



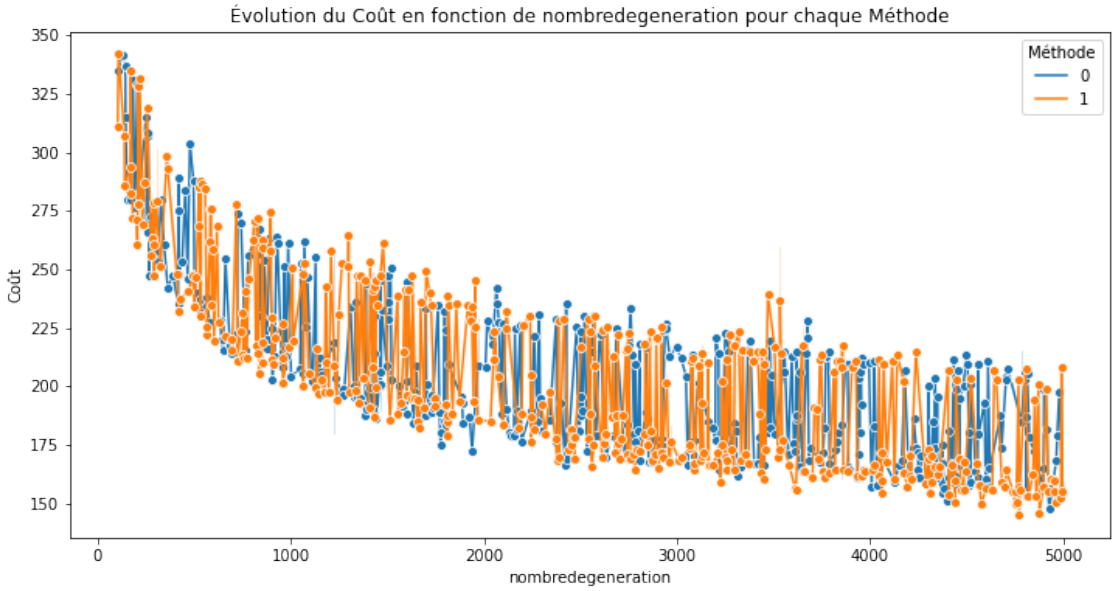
100 villes :

cout :

[ ]:
```
# Créer un diagramme de lignes
plt.figure(figsize=(12, 6))
sns.lineplot(x='nbGeneration', y='cout', hue='methodeMutAc', data=df_100,␣
 ↪marker='o', palette='viridis_r')

# Ajouter des titres et des légendes
plt.title('Évolution du Coût en fonction de nombredegeneration pour chaque␣
 ↪Méthode')
plt.xlabel('nombredegeneration')
plt.ylabel('Coût')
plt.legend(title='Méthode')

# Afficher le diagramme de lignes
plt.show()
```

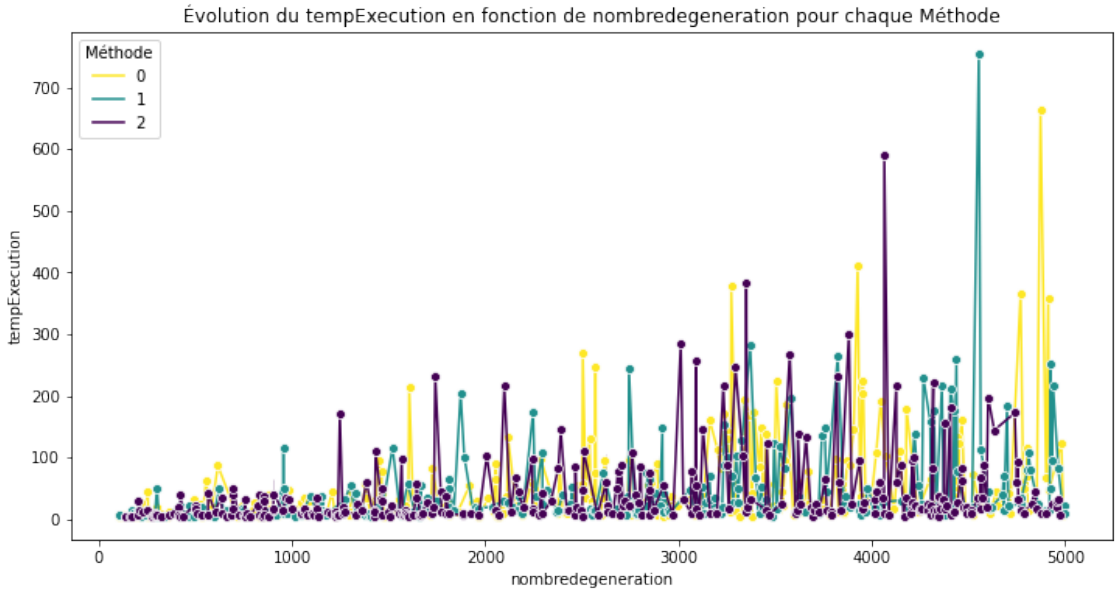


[ ]:
```
# Créer un diagramme de lignes
plt.figure(figsize=(12, 6))
sns.lineplot(x='nbGeneration', y='cout', hue='methodeSelection', data=df_100,␣
 ↪marker='o')

# Ajouter des titres et des légendes
plt.title('Évolution du Coût en fonction de nombredegeneration pour chaque␣
 ↪Méthode')
plt.xlabel('nombredegeneration')
plt.ylabel('Coût')
plt.legend(title='Méthode')

# Afficher le diagramme de lignes
plt.show()
```

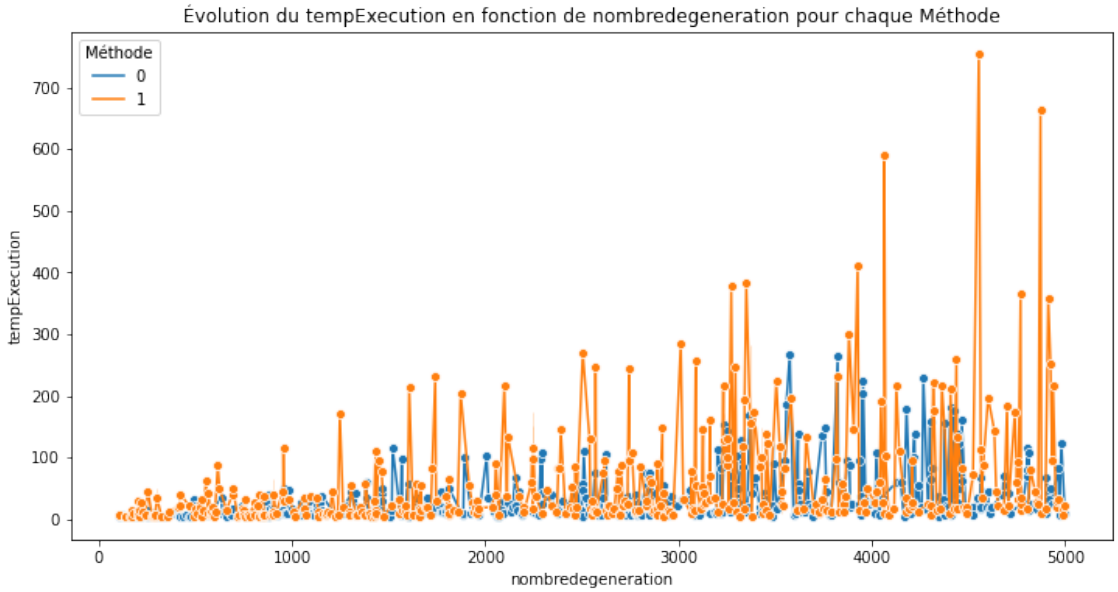


Temps d'execution

[ ]:
```
# Créer un diagramme de lignes
plt.figure(figsize=(12, 6))
sns.lineplot(x='nbGeneration', y='tempExecution', hue='methodeMutAc',␣
 ↪data=df_100, marker='o', palette='viridis_r')

# Ajouter des titres et des légendes
plt.title('Évolution du tempExecution en fonction de nombredegeneration pour␣
 ↪chaque Méthode')
plt.xlabel('nombredegeneration')
plt.ylabel('tempExecution')
plt.legend(title='Méthode')

# Afficher le diagramme de lignes
plt.show()
```



[ ]:
```
# Créer un diagramme de lignes
plt.figure(figsize=(12, 6))
sns.lineplot(x='nbGeneration', y='tempExecution', hue='methodeSelection',␣
 ↪data=df_100, marker='o')

# Ajouter des titres et des légendes
plt.title('Évolution du tempExecution en fonction de nombredegeneration pour␣
 ↪chaque Méthode')
plt.xlabel('nombredegeneration')
plt.ylabel('tempExecution')
plt.legend(title='Méthode')

# Afficher le diagramme de lignes
plt.show()
```



Dans ces lignes pour 16, 30 et 100 villes, plusieurs observations peuvent être faites :

Performance en coût :

- On observe plusieurs variations pour la valeur des coûts au fil de plusieurs itérations dans chaque méthode choisie.
- Avec un nombre de générations inférieur à 1000, la méthode MutAc Échange donne des résultats assez médiocres par rapport aux méthodes Inversion et Translation.
- À mesure que le nombre d'itérations augmente, il devient assez difficile de distinguer la méthode la plus optimale, car toutes les méthodes commencent à converger vers la meilleure solution.
- Cependant, nous pouvons constater que les méthodes Inversion et Translation convergent plus rapidement vers la meilleure solution que la méthode Échange.
- La méthode de meilleure sélection deux à deux converge moins rapidement que la méthode de tri qui est plus performante.

Performance en temps d'exécution :

- Le temps d'exécution des méthodes de muteAC varie souvent et a généralement un temps d'exécution assez proche.
- Le temps d'exécution de la méthode de sélection meilleure triée est plus lent que la méthode deux à deux.

7 Synthèse Générale de l'Évaluation

8 villes

[ ]:
```
filtered_rows = df_8[df_8['cout'] == 17.888544]

# Trouver la ligne avec le temps d'exécution le plus bas
```

```
ligne_temps_min = filtered_rows[filtered_rows['tempExecution'] ==␣
 ↪filtered_rows['tempExecution'].min()]

# Display the filtered DataFrame
ligne_temps_min.head(5)
```

[ ]:

	nbIndividus	nbColnes	nbNouveaux	nbGeneration	nbGenerationInjection	\
96	7	74	99	12	6	
224	16	67	20	12	6	
225	17	85	9	17	6	
226	30	52	60	23	5	
298	16	68	89	14	7	

	methodeMutAc	methodeSelection	cout	tempExecution
96	2	1	17.888544	5.005
224	2	1	17.888544	5.005
225	2	1	17.888544	5.005
226	2	0	17.888544	5.005
298	0	0	17.888544	5.005

16 villes

[ ]:
```
filtered_rows = df_16[df_16['cout'] == 19.313708]

# Trouver la ligne avec le temps d'exécution le plus bas
ligne_temps_min = filtered_rows[filtered_rows['tempExecution'] ==␣
 ↪filtered_rows['tempExecution'].min()]

# Display the filtered DataFrame
ligne_temps_min.head(5)
```

[ ]:

	nbIndividus	nbColnes	nbNouveaux	nbGeneration	nbGenerationInjection	\
195	34	90	6	251	114	

	methodeMutAc	methodeSelection	cout	tempExecution
195	2	1	19.313708	5.009

30 villes

[ ]:
```
filtered_rows = df_30[df_30['cout'] == 46.371631]

# Trouver la ligne avec le temps d'exécution le plus bas
ligne_temps_min = filtered_rows[filtered_rows['tempExecution'] ==␣
 ↪filtered_rows['tempExecution'].min()]

# Display the filtered DataFrame
ligne_temps_min.head(5)
```

[ ]:
```
    nbIndividus  nbColnes  nbNouveaux  nbGeneration  nbGenerationInjection  \
53          363        88          13          2339                    843

    methodeMutAc  methodeSelection       cout  tempExecution
53             1                 1  46.371631          5.271
```

100 villes

[ ]:
```
filtered_rows = df_100[df_100['cout'] ==145.170299]

# Trouver la ligne avec le temps d'exécution le plus bas
ligne_temps_min = filtered_rows[filtered_rows['tempExecution'] ==␣
 ↪filtered_rows['tempExecution'].min()]

# Display the filtered DataFrame
ligne_temps_min.head(5)
```

[ ]:
```
     nbIndividus  nbColnes  nbNouveaux  nbGeneration  nbGenerationInjection  \
245         1781        96          60          4773                   1203

     methodeMutAc  methodeSelection        cout  tempExecution
245             1                 1  145.170299         25.751
```

En analysant les résultats obtenus, nous avons observer que :

- La combinaison des méthodes `selectionMeilleur_trie` et `muteAc_inversion` donne généralement les meilleurs résultats.
- Une proportion entre 75% et 100% de clones donne de bons résultats en termes de temps d'exécution. Au-delà de 20%, le nombre de clones a une influence constante sur la performance du coût.
- Le nombre de nouveaux individus est légèrement mieux observé avec des valeurs entre 20% et 50%.
- Plus on augmente le nombre de générations, plus le coût diminue, mais le temps d'exécution augmente.
- Augmenter le nombre de générations d'injection diminue le coût avec une légère influence sur le temps d'exécution.
- Au-delà d'un nombre d'individus supérieur à 400, l'impact de ce paramètre sur la performance du coût se stabilise.
- Le nombre d'itérations influence fortement la performance de la solution en termes de coût et dépend particulièrement du nombre de villes ; plus on augmente le nombre de villes, plus il est nécessaire d'augmenter le nombre d'itérations.

8 Conclusion

En conclusion, notre projet a mis en œuvre l'algorithme immunitaire pour résoudre le Problème du voyageur de commerce, réalisant 3834 essais avec diverses configurations. L'analyse approfondie des résultats a dévoilé que la combinaison des méthodes "selectionMeilleur_trie" et "muteAc_inversion" a généralement conduit aux performances optimales. Notamment, le maintien

d'une proportion de clones entre 75% et 100% s'est avéré bénéfique en termes de temps d'exécution, tandis qu'au-delà de 20%, le nombre de clones a manifesté une influence constante sur la performance du coût.

Par ailleurs, l'expérimentation a révélé que l'introduction d'un pourcentage de nouveaux individus compris entre 20% et 50% a généré des résultats prometteurs. Des investigations sur les variations du nombre de générations et du nombre d'itérations ont indiqué que l'augmentation de ces paramètres se traduit par une amélioration du coût au détriment du temps d'exécution. Enfin, des seuils critiques ont été identifiés, notamment un nombre d'individus supérieur à 400, où l'impact sur la performance du coût semble se stabiliser.

Cette étude souligne l'importance cruciale de l'ajustement précis des paramètres pour une optimisation réussie du Problème du voyageur de commerce à travers l'algorithme immunitaire.

Parmi les améliorations potentielles à apporter dans le choix des paramètres, nous envisageons :

- Développer un algorithme prédictif capable d'estimer le nombre d'itérations nécessaires pour un nombre donné de villes, offrant ainsi une optimisation plus efficace.
- Explorer d'autres combinaisons de méthodes et de paramètres pour évaluer de nouvelles perspectives d'optimisation et élargir la portée de notre analyse.