Rapport de projet : Application de l'algorithme génétique sur le problème du Voyageur de Commerce

Hugo FERNANDES TELES (20211672) & Akshey MOHAMMED KHOKAN (20224297)

1. Introduction

1.1. Présentation du Problème du Voyageur de Commerce

Le problème du voyageur de commerce est un classique en optimisation combinatoire et c'est l'un des problèmes les plus étudiés en informatique. Il consiste à trouver l'itinéraire le plus court pour un voyageur qui doit visiter un ensemble de villes, une seule fois, puis revenir à son point de départ. L'objectif est donc de minimiser la distance totale parcourue.

Pour résoudre ce problème, on connaît les distances entre chaque ville et les autres, ce qui le rend utile dans des domaines comme la logistique, où optimiser les trajets est crucial. Ce qui le rend particulièrement complexe, c'est qu'il est NP-complet, ce qui signifie qu'il n'existe pas d'algorithme capable de donner une solution optimale en temps polynomial. Concrètement, cela veut dire qu'en augmentant le nombre de villes, le temps de calcul explose de manière exponentielle, rendant la recherche de la solution optimale de plus en plus difficile.

1.2. Introduction aux Algorithmes Génétiques

Pour résoudre notre problème, on a utilisé un algorithme génétique. C'est une méthode qui s'inspire de la façon dont les espèces évoluent dans la nature. On simule la sélection naturelle, la reproduction et les mutations pour trouver les meilleures solutions possibles.

Un algorithme génétique comprend les étapes suivantes :

1. **Population initiale :** Une population initiale de solutions générés aléatoirement.
2. **Évaluation de la fitness :** Chaque individu est évalué à l'aide d'une fonction de fitness qui mesure sa qualité en tant que solution au problème.
3. **Sélection :** Les individus les plus aptes sont sélectionnés pour la reproduction.
4. **Croisement :** Les individus sélectionnés sont combinés pour créer de nouveaux individus en échangeant des parties de leurs chromosomes.
5. **Mutation :** Des modifications aléatoires sont apportées aux chromosomes des enfants pour introduire de la diversité dans la population.
6. **Remplacement :** Les enfants remplacent une partie de la population existante pour former la nouvelle génération.
7. **Itération :** Les étapes 2 à 6 sont répétées jusqu'au nombre de générations demandées ou un autre critère souhaité.

1.3. Objectifs du Projet

Ce projet a pour objectif d'explorer l'utilisation d'un algorithme génétique pour résoudre le problème du voyageur de commerce. Nous avons pour but de :

1. Développer un algorithme génétique capable de fournir des solutions efficaces pour ce problème.
2. Concevoir et implémenter une interface graphique simple pour visualiser les résultats de l'algorithme de manière intuitive.
3. Évaluer les performances de l'algorithme génétique en testant différentes configurations de jeux de données et de paramètres.

Ce rapport présentera les choix de conception, l'implémentation de l'algorithme et de l'interface graphique, ainsi que l'analyse des résultats obtenus.

2. Méthodologie

2.1. Représentation du Problème

Dans notre implémentation du problème du voyageur de commerce, nous avons utilisé une approche orientée objet, c'est-à-dire que nous avons créé des éléments de code (appelés objets) qui représentent directement les villes et les chemins. Chaque ville et chaque chemin sont ainsi des objets avec leurs propres caractéristiques et actions, ce qui simplifie la gestion et la manipulation des données. On a certes placé les villes selon leurs coordonnées exactes sur la carte, mais le calcul de distance entre les villes s'est fait en fonction de la taille du graphique et non avec des vraies distances, pour pouvoir optimiser nos mesures et travailler plus simplement sur l'algorithme.

- **Classe** `Ville` **:**
    - Chaque ville est représentée par un objet de la classe `Ville`, contenant son nom (`name`), sa longitude (`lng`) et sa latitude (`lat`).
    - La méthode `get_distance(ville)` calcule la distance euclidienne entre deux villes. On a choisi cette méthode pour simplifier cette partie du code puisqu' on calcule des distances qui sont dans un même pays, c'est-à- dire que le taux d'erreur va être minimal.
    - La méthode `get_next_ville(liste_villes)` choisit une ville aléatoire dans une liste de villes, cette méthode est utilisée pour former la population initiale.
- **Classe** `Route` **:**
    - Un chemin est une liste ordonnée d'objets `Ville`, représentant l'ordre de visite.
    - La classe `Route` contient la distance totale (`distance`), la fitness (`fitness`), la ville de départ (`depart`) et la liste de toutes les villes (`liste_villes`).
    - La méthode `trace_ma_route()` génère un chemin qui visite toutes les villes une seule fois, en évitant les doublons.
    - La méthode `stack_la_distance()` calcule la distance totale du chemin.
    - La méthode `evaluer()` calcule la fitness comme l'inverse de la distance.
      Cela permet de plus facilement évaluer les meilleurs chemins de chaque génération.

    - La méthode `muter()` effectue une mutation par échange de deux villes aléatoires. On est conscients qu'il existe d'autres méthodes plus complexes mais on ne savait pas comment les implémenter sans aide externe. On a décidé de rester avec cette méthode plus simple.
    - La méthode `get_villes()` convertit le chemin de villes en une simple liste de leurs noms. Cela nous permet d'afficher le chemin sous forme de texte, ce qui est utile pour vérifier que le programme fonctionne correctement et pour avoir une version lisible du meilleur chemin à chaque génération.
- **Classe `Population` :**
    - La classe `Population` gère une liste de chemins (`chemins`), la ville de départ (`depart`), et la liste de toutes les villes (`villes`).
    - La méthode `election()` retourne le chemin avec la meilleure fitness. On l'utilise pour avoir le meilleur chemin à chaque génération.
    - La sélection des parents a été effectuée par la méthode du `tournoi()`. Ce choix s'explique par la volonté de maintenir une diversité significative au sein de la population. Une telle diversité favorise une convergence plus efficace vers une solution optimale. Si nous avions opté pour une sélection de seulement les individus les plus performants à chaque génération, l'évolution de la population aurait été principalement dû aux mutations, limitant ainsi les possibilités d'exploration de solutions alternatives.
    - La méthode `reproduction()` effectue un croisement par découpage et remplissage en utilisant deux parents.
    - La méthode `generer_nouvelle_population()` génère une nouvelle population en utilisant la sélection par tournoi, les croisements et des possibles mutations.

2.2. Conception de l'Algorithme Génétique

Notre algorithme génétique est conçu en suivant les étapes clés :

- **Création de la population initiale :**
    - Nous générons une population initiale de 200 chemins aléatoires en utilisant la méthode `trace_ma_route()` de la classe `Route`.
- **Fonction d'évaluation de la fitness :**
    - La fitness de chaque chemin est calculée comme l'inverse de sa distance totale.
- **Sélection des parents :**
    - Nous utilisons la sélection par tournoi avec une taille de tournoi égale à 1/20 de la taille de la population.
- **Croisement :**
    - Nous utilisons un croisement par découpage et remplissage des parties manquantes.
- **Mutation :**
    - Nous utilisons une mutation par échange de deux villes aléatoires avec un taux de mutation de 0.1.
- **Critère d'arrêt :**
    - L'algorithme s'arrête après un nombre de générations spécifié par l'utilisateur.

- **Elitisme**
    - La méthode `generer_nouvelle_population` garde le meilleurs des deux parents et de l'enfant pour la prochaine génération.

2.3. Implémentation de l'Interface Graphique

Nous avons développé une interface graphique simple à l'aide de la bibliothèque Tkinter de Python. L'interface permet aux utilisateurs de :

- Sélectionner un pays et un nombre de villes à générer aléatoirement.
- Visualiser la carte du pays avec les villes générées.
- Sélectionner la ville de départ.
- Lancer l'algorithme génétique et visualiser le chemin optimal trouvé.
- Visualiser l'évolution des résultats de chaque génération dans une nouvelle fenêtre.
- Effacer la carte et les résultats.

Pour obtenir les informations sur les villes (noms, latitudes, longitudes) et les pays, nous avons utilisé l'API GeoNames. Nous faisons des requêtes HTTP en spécifiant le code pays et le nombre de villes souhaité. Les données renvoyées au format JSON sont traitées pour extraire les informations nécessaires.

La ligne de code #choix_random.append(random.sample(villes["geonames"], min(N,len(villes["geonames"])))) permet de randomiser les villes obtenues de l'API. Cette randomisation est essentielle pour générer des jeux de données variés

3. Résultats

3.1. Présentation des Résultats Obtenus

Nous avons testé notre algorithme génétique sur différentes données aléatoires, en variant le pays, le nombre de villes et la ville de départ. Les résultats montrent que l'algorithme trouve des solutions en un temps raisonnable.
L'interface graphique permet de visualiser le chemin optimal, avec des villes représentées par des points et le chemin tracé par des lignes. Le chemin devient progressivement plus court au fil des générations, et le meilleur chemin est mis en évidence à la fin.
Une fenêtre affiche la diminution de la distance totale à chaque génération, avec des variations rares où la distance augmente temporairement avant de se stabiliser sur un chemin plus optimal.
Augmenter le nombre de générations améliore la solution mais augmente aussi le temps de calcul, d'où la nécessité de trouver un équilibre. Le meilleur chemin est illustré en jaune.

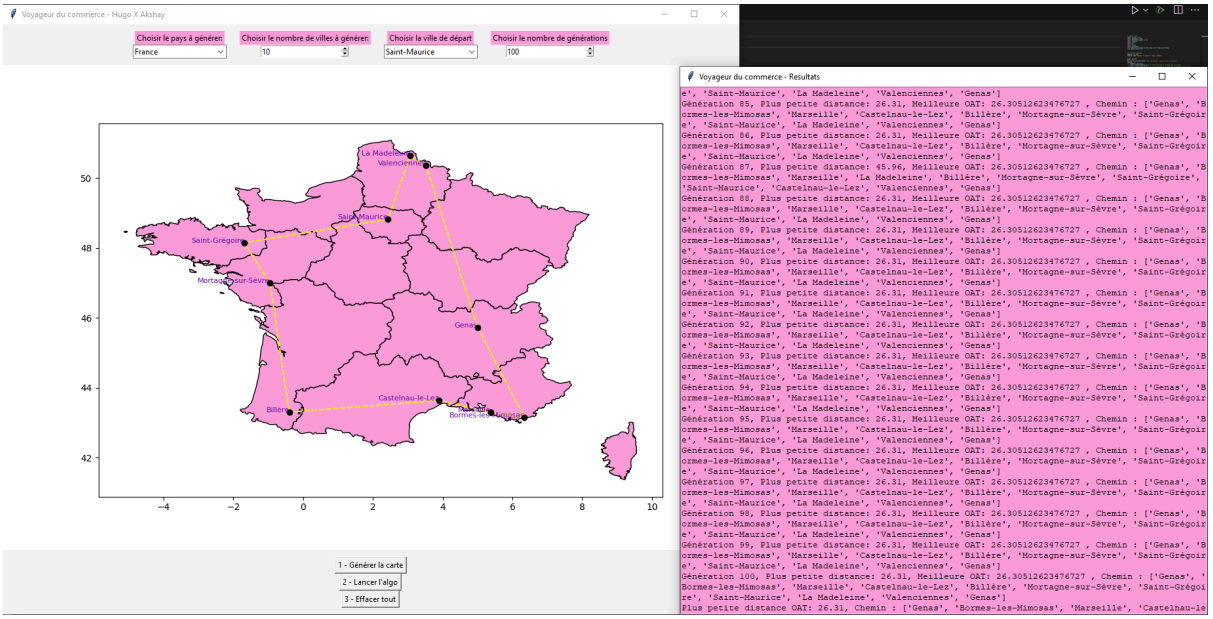

Figure 1A - Exemple de résultats obtenus pour des paramètres lambdas.

Le temps de calcul augmente avec le nombre de villes, et pour un grand nombre de villes, l'algorithme peut prendre plusieurs minutes sans garantir une solution optimale.

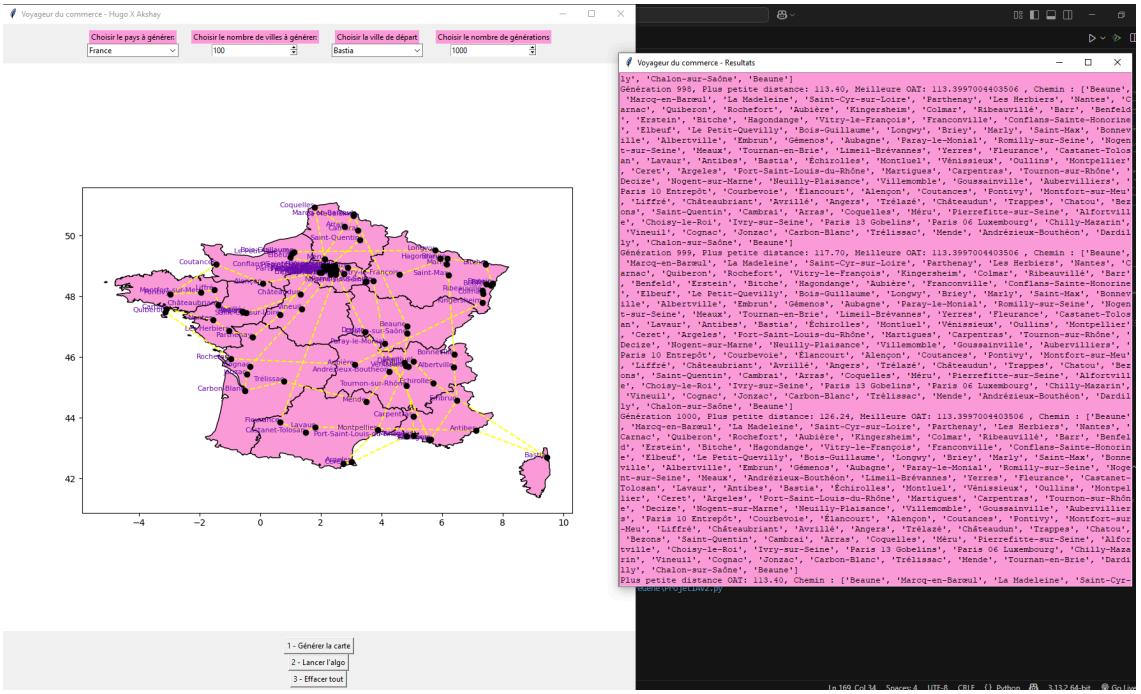

Figure 1B - Exemple de résultats obtenus pour un taux de mutation élevé.

On constate une variation significative dans le choix du meilleur chemin à chaque génération dû au nombre important du taux de mutation.

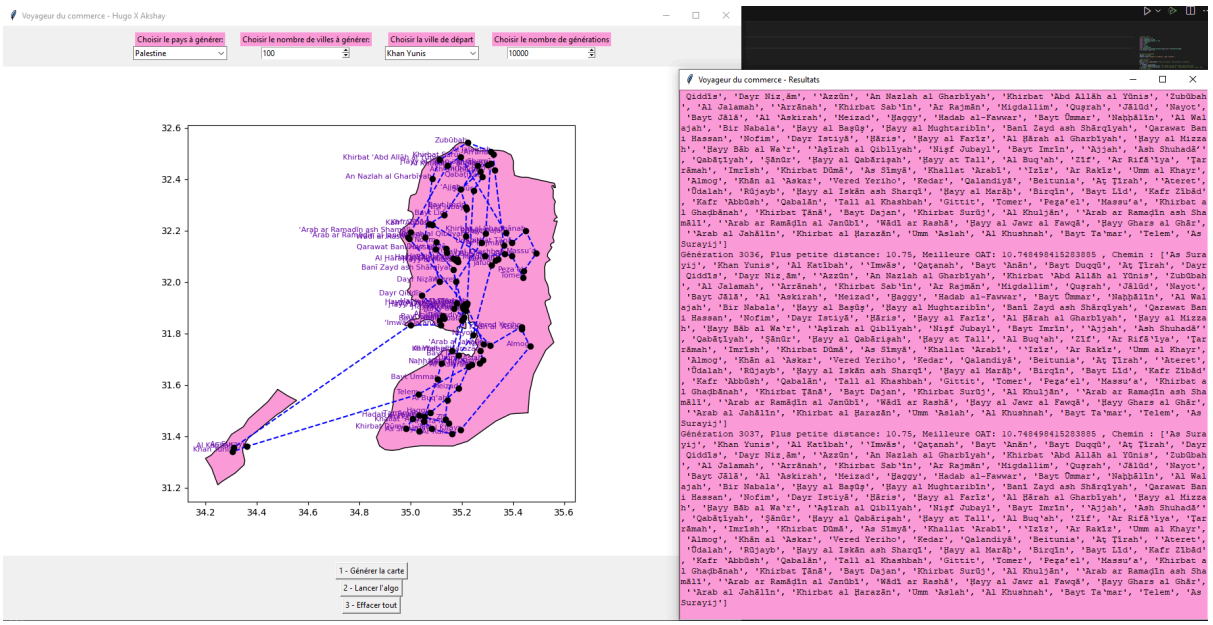

Figure 2 - Exemple de résultats obtenus pour des paramètres poussés à l'extrême.

100 Villes et 10000 générations en paramètres ->
42 minutes pour 3186 générations ! Donc ~ 2h10 pour tout explorer…

3.2. Analyse des Résultats

Les résultats montrent que l'algorithme génétique fonctionne bien pour résoudre le problème du voyageur de commerce. En général, il converge vers une solution assez bonne après un nombre raisonnable de générations. La vitesse à laquelle il converge dépend des paramètres de l'algorithme (comme la taille de la population, le taux de mutation, etc.) et du nombre de villes. L'algorithme est assez robuste et donne de bons résultats, même pour des jeux de données assez grands.

Cependant, l'algorithme ne trouve pas toujours la solution optimale, et le temps de calcul peut devenir long pour des instances avec beaucoup de villes. La qualité des solutions dépend aussi des paramètres choisis, qui doivent être ajustés manuellement. Enfin, la méthode de croisement n'est pas toujours optimale et peut parfois donner des résultats un peu moins bons.

3.3. Améliorations Possibles

Pour améliorer l'algorithme, on peut optimiser les paramètres avec des techniques adaptées et des heuristiques pour une meilleure population initiale. La parallélisation réduira le temps de calcul, tandis qu'une visualisation améliorée permettra de suivre son évolution. Ajouter une barre de progression offrira un suivi visuel pendant le calcul.

4. Conclusion

En conclusion, ce projet a démontré l'efficacité des algorithmes génétiques pour résoudre le problème du voyageur de commerce. Bien que l'algorithme développé ne garantisse pas de trouver la solution optimale, il est capable de trouver des solutions sous-optimales en un temps raisonnable pour des instances de taille modérée. Les améliorations proposées pourraient permettre d'optimiser les performances de l'algorithme et d'étendre son applicabilité à d'autres problèmes d'optimisation. Les temps d'exécution de l'algorithme s'expliquent cependant par l'utilisation de Python qui ne mobilise suffisamment pas les ressources de l'ordinateur. L'algorithme génétique est un outil puissant pour l'optimisation, et ce projet à permis d'en explorer une des nombreuses applications.

5. Annexe

5.1. Fichier principal avec interface

```
# coding: utf-8
import tkinter as tk
import geopandas as gpd
import matplotlib.pyplot as plt
import requests
import json
import os
import random
import EntitiesV2
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from tkinter import ttk


## Interface graphique
fenetre = tk.Tk()
fenetre.title("Voyageur du commerce - Hugo X Akshay")

# Pays au choix
path = "Pays"
dir_list = os.listdir(path) # Tous les fichiers .shp sont dans le répertoire
liste_p = []
for element in dir_list:
    liste_p.append(element[:2]) # On rajoute les noms de fichiers dans notre liste
liste_pays = list(set(liste_p)) # Il y a des doublons dû à la présence de fichiers .shx et .shp
# ON BOYCOTT FORT
liste_pays.remove("il")
liste_pays.remove("ru")


def get_country(pais: str, n: int) -> list: # Oui "pais" c'est "pays" en espagnol (je crois)
    """
        Prend en compte un pays et renvoie
        les informations concernant ses villes
    """
    # Obtention des villes et de leurs informations
    user = "crakshay" # hugot_s
    api_url = 'http://api.geonames.org/searchJSON?country={}&featureClass=P&maxRows={}&username={}'.format(pais, n, user)
    response = requests.get(api_url, headers=None)
    cities = json.loads(response.text)
    # Le résultat étant un str, utiliser le module le json nous permet de le transformer en dict
    if len(cities["geonames"]) > 0:
        return cities
    else:
        return -1

# Pour la sélection du pays
new_dico = {}
for element in liste_pays:
    pays = get_country(element,1)
    # Si on a les fichiers du pays et que l'API fournit des infos sur celui-ci
    if pays != -1:
```

```
        new_dico.update({pays["geonames"][0]["countryName"]:pays["geonames"][0]["countryCode"] })
    # Si l'API n'a aucune info sur le pays, on supprime les fichiers shp et shx
    if pays == -1:
        os.remove("Pays/"+element+".shp")
        os.remove("Pays/"+element+".shx")

select_frame = tk.Frame(fenetre)
select_frame.pack(pady=10)
liste = ttk.Combobox(select_frame, values=list(sorted(new_dico.keys())))
liste.current(0)

# Nombre de villes générées aléatoirement
nombre = tk.Label(select_frame, text="Choisir le nombre de villes à générer:", bg="#fa9cdb")
pays = tk.Label(select_frame, text="Choisir le pays à générer:", bg="#fa9cdb")
scroll = tk.Spinbox(select_frame, from_=0, to=1000)

def generer_carte():
    """
        Prend en compte le nombre N de villes générées aléatoirement, et le pays retenu
        Et génère la carte avec les villes
    """
    # On obtient les informations retenues dans la fenêtre
    N = int(scroll.get())
    country = new_dico[liste.get()].lower()
    pays = "Pays/"+country+".shp"

    # On crée la carte
    carte = gpd.read_file(pays)
    carte.plot(ax=ax, color="#fa9cdb", edgecolor="black")

    # Liste des villes aléatoirement générées
    villes = get_country(country, 1000)
    choix_random = []
    choix_random.append(random.sample(villes["geonames"], min(N,len(villes["geonames"]))))
    # Même si on dépasse le nombre de villes voulu avec N, on prendra le nombre max de villes du pays

    # On crée les villes comme des objets
    global liste_v, nom_vers_ville
    liste_v = []
    for ville in choix_random[0]:
        liste_v.append(EntitiesV2.Ville(ville['name'], float(ville['lng']), float(ville['lat'])))

    # On place les villes sur la carte
    liste_n = []
    for v in liste_v:
        ax.scatter(v.lng, v.lat, color="black", marker="o", zorder=5)
        ax.text(v.lng, v.lat, v.name, fontsize=8, ha="right", color="#6A0DAD")
        liste_n.append(v.name)
    nom_vers_ville = {v.name: v for v in liste_v}

    # Sélection de la ville de départ
    global depart, listeVilles
    depart = tk.Label(select_frame, text="Choisir la ville de départ", bg="#fa9cdb")
    depart.grid(row=0, column=2, padx=10)
    listeVilles = ttk.Combobox(select_frame, values=liste_n)
    listeVilles.current(0)
    listeVilles.grid(row=1, column=2, padx=10)

    # Sélection nombre générations
    global gen, select_gen
    select_gen = tk.Label(select_frame, text="Choisir le nombre de générations", bg="#fa9cdb")
    select_gen.grid(row=0, column=3, padx=10)
    gen = tk.Spinbox(select_frame, from_=1, to=10000)
    gen.grid(row = 1, column=3, padx=10)


def algo():
    """
        Execute notre algorithme de parcours
    """
    depart_v = listeVilles.get()
    nb_gen = int(gen.get())
    start = nom_vers_ville[depart_v]

    # Création population
    liste_c = []
    for i in range(200): # Taille de la population
        route = EntitiesV2.Route(start,liste_v)
```

```
        route.trace_ma_route()
        liste_c.append(route)
    populasse = EntitiesV2.Population(liste_c, start, liste_v)

    # Sélection tah la Champion's league
    fenetre2 = tk.Toplevel(fenetre)
    fenetre2.title("Voyageur du commerce - Resultats")
    resultats = tk.Text(fenetre2, width=100, height=50, bg="#fa9cdb")
    resultats.pack()

    best_dist = float('inf')
    best_chemin = None
    for generation in range(nb_gen):
        populasse.generer_nouvelle_population()
        meilleure_route = populasse.election()
        meilleure_route.get_villes()
        if meilleure_route.distance < best_dist:
            best_dist = meilleure_route.distance
            best_chemin = meilleure_route.chemin
        # Tracer le chemin
        liste_x = []
        liste_y = []
        for ville in meilleure_route.chemin:
            liste_x.append(ville.lng)
            liste_y.append(ville.lat)
        for line in ax.lines:
            line.remove()
        ax.plot(liste_x,liste_y,color='blue',linestyle='dashed')
        canvas.draw()
        # Fenêtre avec informations
        resultats.insert(tk.END, f"Génération {generation + 1}, Plus petite distance: {meilleure_route.distance:.2f},
Meilleure OAT: {best_dist} , Chemin : {meilleure_route.chemin_trad}" + "\n")
        resultats.see(tk.END)
        fenetre2.update()

    # Le meilleur pour la fin
    liste_x = []
    liste_y = []
    for ville in best_chemin:
        liste_x.append(ville.lng)
        liste_y.append(ville.lat)
    # Retour à la ville de départ
    liste_x.append(best_chemin[0].lng)
    liste_y.append(best_chemin[0].lat)

    for line in ax.lines:
        line.remove()
    ax.plot(liste_x, liste_y, color='yellow', linestyle='dashed')
    canvas.draw()

    # Affichage du chemin final correct
    chemin_names = [ville.name for ville in best_chemin] + [best_chemin[0].name]  # cycle complet
    resultats.insert(tk.END, f"Plus petite distance OAT: {best_dist:.2f}, Chemin : {chemin_names}" + "\n")


def rafraichir():
    """
        Cette méthode permet d'effacer tout le contenu de la fenêtre
    """
    ax.clear()
    depart.destroy()
    listeVilles.destroy()
    gen.destroy()
    select_gen.destroy()
    canvas.draw()

fig, ax = plt.subplots(figsize=(6, 5))
canvas = FigureCanvasTkAgg(fig, master=fenetre)
canvas_widget = canvas.get_tk_widget()
canvas_widget.pack(fill=tk.BOTH, expand=True)

bouton_frame = tk.Frame(fenetre)
bouton_frame.pack(pady=10)
bouton = tk.Button(bouton_frame, text="1 - Générer la carte", command=lambda : generer_carte())
# Bouton lançant la génération de villes
bouton2 = tk.Button(bouton_frame, text="3 - Effacer tout", command = lambda : rafraichir())
# Bouton lançant le rafraichissement de la carte
bouton3 = tk.Button(bouton_frame, text="2 - Lancer l'algo", command = lambda : algo())
```

```
# Bouton lançant l'algorithme de parcours


nombre.grid(row=0, column=1, padx=10)
scroll.grid(row=1, column=1, padx=10)
pays.grid(row=0, column = 0, padx=10)
liste.grid(row=1, column = 0, padx=10)
bouton.grid(row=0, column=0, padx=10)
bouton2.grid(row=2, column=0, padx=10)
bouton3.grid(row=1, column=0, padx=10)


fenetre.mainloop()
```

5.2. Fichier secondaire avec les classes pour l'Algorithme

```
import math
import random

class Ville():
    def __init__(self, name:str, lng: float, lat: float):
        self.name = name
        self.lng = lng
        self.lat = lat

    def get_distance(self, ville):
        """
            Obtient la distance entre deux villes données.
        """
        return math.sqrt((ville.lat-self.lat)**2 + (ville.lng-self.lng)**2)

    def get_next_ville(self, liste_villes: list):
        """
            Choisit aléatoirement la prochaine ville à atteindre.
        """
        return random.choice(liste_villes)


class Route():
    def __init__(self, depart, liste_villes: list):
        self.distance = 0
        self.fitness = 0
        self.depart = depart
        self.mutation = 0.1
        self.liste_villes = liste_villes
        self.chemin = [depart]
        self.chemin_trad = []
        self.next = depart.get_next_ville(liste_villes)

    def trace_ma_route(self):
        """
            Initie la création d'un individu.
        """
        while len(self.chemin) < len(self.liste_villes):
            if self.next in self.chemin:
                self.next = self.depart.get_next_ville(self.liste_villes)
                continue # On évite de rajouter la ville comme doublon
            self.chemin.append(self.next)
            self.depart = self.next
            self.next = self.depart.get_next_ville(self.liste_villes)
        self.chemin.append(self.chemin[0])  # Ferme le cycle

    def get_villes(self):
        """
            Obtient les noms des villes d'une route.
        """
        self.chemin_trad = [ville.name for ville in self.chemin]

    def stack_la_distance(self):
        """
            Permet d'accumuler toutes les distances intervilles.
```

```
        """
        self.distance = 0
        for i in range(len(self.chemin)-1):
            villeA = self.chemin[i]
            villeB = self.chemin[i+1]
            self.distance += villeA.get_distance(villeB)

    def evaluer(self):
        """
            L'inverse de la distance est la fitness
            (donc plus facilement interprétable).
        """
        if self.distance == 0:
            self.fitness = 0
        else:
            self.fitness = 1/self.distance

    def muter(self) -> list:
        """
            On permute ici deux villes de manière random.
        """
        if random.random() < self.mutation:
            i, j = random.sample(range(len(self.chemin) - 1), 2)  # on exclut le dernier
            self.chemin[i], self.chemin[j] = self.chemin[j], self.chemin[i]
            self.chemin[-1] = self.chemin[0]  # refermer proprement le cycle
        return self.chemin


class Population():
    def __init__(self, liste_chemins: list, depart, liste_villes: list):
        self.chemins = liste_chemins
        self.depart = depart
        self.villes = liste_villes

    def election(self) -> list:
        """
            Elit l'individu au meilleur fitness
            de chaque génération.
        """
        max = self.chemins[0]
        for element in self.chemins:
            element.stack_la_distance()
            element.evaluer()
            if element.fitness > max.fitness:
                max = element
        return element

    def tournoi(self) -> list:
        """
            Permet de choisir des parents via un tournoi.
        """
        taille_tournoi = len(self.chemins)//20
        participants = random.sample(self.chemins, taille_tournoi)
        gagnant = participants[0]
        for chemin in participants:
            chemin.stack_la_distance()
            chemin.evaluer()
            if chemin.fitness > gagnant.fitness :
                gagnant = chemin
        return gagnant

    def reproduction(self, parent1, parent2) -> list:
        """
            Effectue un croisement entre deux routes parents
            passés en paramètres.
        """
        taille = len(parent1.chemin) - 1  # exclut la dernière ville (cycle)
        debut = random.randint(0, taille - 2)
        fin = random.randint(debut + 1, taille - 1)

        enfant = [None] * taille
        enfant[debut:fin+1] = parent1.chemin[debut:fin+1]

        index_parent2 = 0
        for i in range(taille):
            if enfant[i] is None:
                while parent2.chemin[index_parent2] in enfant:
                    index_parent2 += 1
```

```
                enfant[i] = parent2.chemin[index_parent2]
        enfant.append(enfant[0])  # referme le cycle
        return enfant

    def generer_nouvelle_population(self) -> list:
        """
            Génère une nouvelle population en utilisant
            la sélection, le croisement et la mutation.
        """
        nouvelle_population = []
        for i in range(len(self.chemins)):
            parent1 = self.tournoi() # Une route parent
            parent2 = self.tournoi()
            enfant_liste = self.reproduction(parent1, parent2) # Chemin pour l'enfant
            enfant = Route(self.depart, self.villes) # La route enfant
            enfant.chemin = enfant_liste # Le chemin devient la route enfant
            enfant.muter() # Est-ce que le chemin va muter?
            parent1.evaluer()
            parent2.evaluer()
            enfant.evaluer()
            eva1, eva2, eva3 = parent1.distance, parent2.distance, enfant.distance
            maxi = min([eva1,eva2,eva3])
            if maxi == eva1 :
                nouvelle_population.append(parent1)
            elif maxi == eva2 :
                nouvelle_population.append(parent2)
            else:
                nouvelle_population.append(enfant)

        self.chemins = nouvelle_population
```