RAPPORT DE PROJET PROBLÈME DU VOYAGEUR DE COMMERCE (TSP) INTELLIGENCE ARTIFICIELLE

Réalisé par :

- Nom : TOUATI
- Prénom : Mounia
- N° Étudiant : 20225680
- Groupe : L3CILS

- Nom : ALKHOURI
- Prénom : George
- N° Étudiant : 20214191
- Groupe : L3CILS

2023 - 2024

Table des matières

1- Introduction :

Au cours de la formation Licence 3 Informatique parcours Conception et Intelligence des Logiciels et Systèmes, dans le module d'Intelligence Artificielle, il nous a été demandé de résoudre le Problème du Voyageur de Commerce (TSP) en utilisant des algorithmes génétiques étudié. Ce projet vise à appliquer ces algorithmes, qui simulent la sélection, permettent de trouver des solutions efficaces à des problèmes d'optimisation complexes, offrant des approches viables pour minimiser la distance parcourue dans le TSP de maniéré efficiente.

L'algorithme génétique, par sa capacité à générer et améliorer des itinéraires au fil des générations, s'est révélé être un outil précieux pour aborder cette tâche ardue. Ce travail ne se contente pas de résoudre un problème académique mais met également en lumière des stratégies applicables dans des contextes industriels et logistiques réels.

2- Problème :

Le problème du voyageur de commerce (TSP, de l'anglais Traveling Salesman Problem) est un célèbre problème d'optimisation qui cherche à trouver l'itinéraire le plus court permettant à un voyageur de visiter une série de villes, chacune une seule fois, et de retourner à son point de départ. Il est souvent utilisé pour tester des algorithmes d'optimisation car, malgré sa formulation simple, il est classé parmi les problèmes NP-difficiles, ce qui signifie qu'aucun algorithme connu ne peut le résoudre en temps polynomial pour tous les cas théoriques. Le TSP est non seulement un sujet d'étude fondamental en théorie des graphes et en recherche opérationnelle, mais il trouve aussi des applications pratiques dans des domaines variés tels que la logistique, la planification de routes, la fabrication de circuits intégrés, et plus encore.

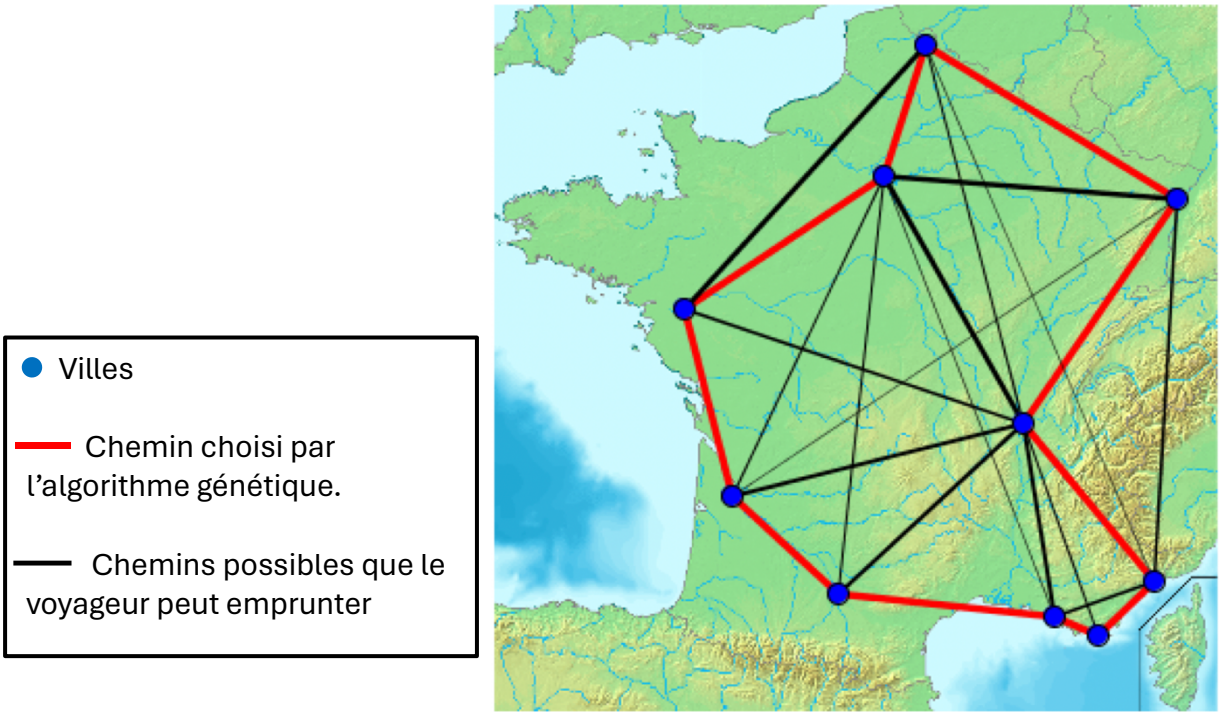

Figure 1 : Exemple de problème TSP sur la carte géographique de

3- Solution :

3.1- Méthodologie :

1. Algorithmes génétiques :

Dans le contexte du problème du voyageur de commerce, qui cherche à minimiser le cout total d'un circuit passant une unique fois par chaque ville et retournant au point de départ, les algorithmes génétiques sont particulièrement pertinents. En effet, c'est ici que les algorithmes génétiques montrent leur utilité. Inspire par la nature, ces algorithmes sont bons pour chercher parmi de nombreuses solutions possibles et trouver de très bonnes approximations rapidement, même pour des problèmes compliques comme le TSP.

- **Initialisation de la population :**

Premièrement, l'algorithme génétique commence par générer une population initiale composée de plusieurs itinéraires possibles (ou "individus"), chacun représentant une solution potentielle au TSP. Ces itinéraires sont typiquement générés de manière aléatoire, ou en utilisant des heuristiques simples pour constituer un point de départ viable. Chaque individu est ensuite évalué pour déterminer sa "fitness", généralement calculée comme étant l'inverse du coût total de l'itinéraire (la distance totale parcourue).

- **Sélection des parents :**

La sélection des parents pour la reproduction est ensuite effectuée via un processus de tournoi (sélection par tournoi). Un nombre fixé de tournois est organisé, chaque tournoi consistant en un choix aléatoire de plusieurs individus. Les individus sont sélectionnés pour la reproduction en fonction de leur fitness, les meilleurs étant plus susceptibles d'être choisis.

- **Croisement et mutation :**

Pour le problème du voyageur de commerce, le croisement ordonné (OX) est utilisé. Ce type de croisement consiste à choisir aléatoirement deux points de coupure dans l'itinéraire du parent, et à copier directement la séquence entre ces points dans l'enfant. Le reste de l'itinéraire de l'enfant est ensuite rempli avec les villes restantes du second parent en respectant l'ordre d'apparition, tout en évitant les doublons. Quant à la mutation, elle augmente la diversité génétique et permet d'explorer de nouvelles solutions. Elle est réalisée par des échanges aléatoires de positions entre deux villes dans l'itinéraire, avec une probabilité définie par le taux de mutation. Cette opération est cruciale pour prévenir la convergence prématurée vers des optima locaux. Le croisement OX et la mutation par échange travaillent ensemble pour bien mélanger et tester de nouvelles solutions, ce qui est crucial pour trouver l'itinéraire le plus court dans le problème du voyageur de commerce.

La population évoluera. Chaque nouvelle génération remplace la précédente, et ce cycle se répète. Ainsi, les générations suivantes sont souvent meilleures, améliorant petit à petit les solutions jusqu'à atteindre le meilleur possible.

2. Interface graphique :

Pour visualiser les solutions générées par l'algorithme génétique, une interface graphique a été développée en utilisant Tkinter, une bibliothèque standard pour l'interface utilisateur graphique en Python. Cette interface affiche les villes et le chemin optimal trouvé. L'interface graphique montre les villes sous forme de points et le chemin sous forme de lignes connectant ces points. Chaque ville est numérotée pour référence. L'itinéraire optimal est mis en évidence et peut être visualisé à mesure que l'algorithme progresse vers une solution.

3.2- Implémentation en Python :

3.2.1- Importations des Bibliothèques :

- **Numpy** : Utilisée pour les opérations sur les tableaux et les calculs mathématiques, essentielle pour manipuler les données des villes et calculer les distances.
- **Radom** : Employée pour générer des nombres aléatoires nécessaires lors des étapes de sélection, de croisement et de mutation dans l'algorithme génétique.
- **Tkinter** : Utilisée pour construire l'interface graphique, permettant de visualiser les villes et les chemins optimisés.

3.2.2- Fonctions de l'algorithme génétique :

- **Distances (Coord_Villes)** : Calculer et retourne une matrice des distances entre les N villes, Cette matrice est essentielle pour évaluer la « fitness » des itinéraires.
- **Init_population (taillep, Coord_villes)** : Génère une population initiale d'itinéraires (solutions), où chaque itinéraire est une permutation des indices des villes.
- **Fitness_idv (individu, distance_villes)** : Calcule la fitness d'un itinéraire en utilisant l'inverse de la distance totale parcourue, encourageant des solutions de plus courtes distances.
- **Tournament_selection (population, fitness, tournament_size, num_parents) :** Sélectionne les parents pour la reproduction via une méthode de tournoi, où un sous-ensemble aléatoire de la population est comparé et les meilleurs sont choisis.
- **Ox_Croisement (parent1, parent2)** : Implémente le croisement ordonné (OX) pour produire un nouvel itinéraire à partir de deux parents, garantissant que toutes les villes apparaissent une seule fois dans l'itinéraire de l'enfant.
- **Mutation (individu, mutation_rate)** : Mute un itinéraire en échangeant aléatoirement deux villes, avec une probabilité définie par le taux de mutation.
- **Algo_Genetique (Coords, taillep, n_generation, mutation_rate, taille_tournament, num_parent)** : Englobe tout le processus de l'algorithme génétique en intégrant la création de la population, l'évaluation, la

sélection, le croisement, la mutation et la mise à jour de la population à travers les générations.

3.2.3- Interface graphique :

- **Dessiner_villes_chemins (canvas, villes, solution)** : Dessine les villes et les chemins entre elles sur un Canvas, illustrant visuellement le chemin optimisé trouvé par l'algorithme.
- **Creation_inteface (villes, solution)** : Configure la fenêtre principale et le canevas, et appelle la fonction pour dessiner les villes et les chemins.

3.2.4- Bloc principal (main) :

Le bloc principal gère l'entrée des paramètres par l'utilisateur (nombre de villes, par exemple), génère des coordonnées aléatoires pour les villes, détermines les paramètres de l'algorithme (aille de la population, nombre de générations, taux de mutation, etc.), et lance l'exécution de l'algorithme génétique. Après l'exécution, il affiche la meilleure solution trouvée et les détails de fitness correspondants.

3.3- Résultats :

Nous avons exécuté l'algorithme sur un ensemble de villes générées aléatoirement et observé la convergence vers un solution satisfaisante qui minimise la distance totale parcourue. Les résultats démontrent l'efficacité de l'algorithme génétique pour le TSP, même avec un nombre relativement élevé de villes. Voici ci – dessous un exemple sur n = 9 villes.

```
Entrez le nombre de villes : 9
Génération 1: Meilleur Individu: [2 4 6 7 1 0 3 8 5], Fitness: 0.0003892498870002234737
Génération 2: Meilleur Individu: [4 8 3 0 1 7 6 2 5], Fitness: 0.0004361247511867377322
Génération 3: Meilleur Individu: [6 7 1 5 2 4 8 3 0], Fitness: 0.0003904116066209890770
Génération 4: Meilleur Individu: [4 8 3 1 0 7 5 6 2], Fitness: 0.0003959016312399713075
Génération 5: Meilleur Individu: [0 3 8 4 6 2 5 7 1], Fitness: 0.0003862517024673959372
Génération 6: Meilleur Individu: [3 8 4 2 5 6 7 0 1], Fitness: 0.0004235404411393469759
Génération 7: Meilleur Individu: [6 7 2 5 4 8 3 0 1], Fitness: 0.0004049524333741008554
Génération 8: Meilleur Individu: [6 5 2 4 8 3 1 0 7], Fitness: 0.0004235404411393469759
Génération 9: Meilleur Individu: [4 2 8 3 0 1 6 7 5], Fitness: 0.0003912110187175226817
Génération 10: Meilleur Individu: [0 3 8 4 2 5 6 7 1], Fitness: 0.0004407354790268873715
Meilleure solution: [0 3 8 4 2 5 6 7 1]
Meilleure fitness: 0.0004407354790268737
```

Figure 2 : Affichage console après une exécution sur 9 villes.

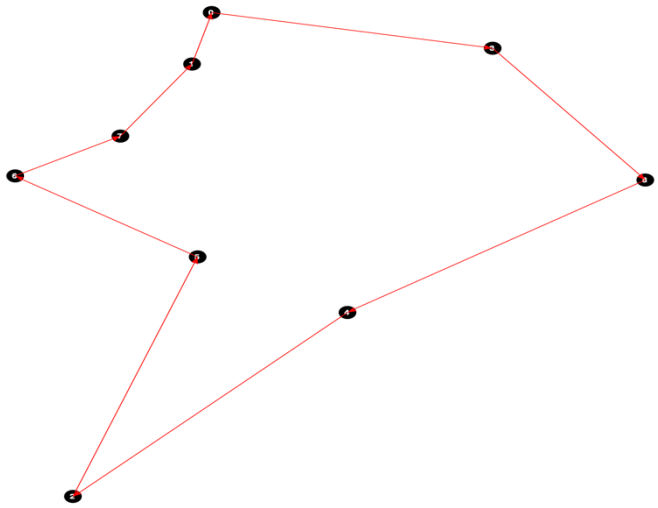

Figure 3 : Affichage de l'interface graphique pour les 9 villes en entrée.

4- Conclusion :

Ce projet nous a permis d'explorer l'application des algorithmes génétiques au problème du voyageur de commerce, un défi classique en optimisation combinatoire. À travers notre étude, nous avons démontré que les algorithmes génétiques sont non seulement capables de trouver des solutions de haute qualité pour ce problème complexe, mais aussi qu'ils peuvent le faire de manière efficace.

L'utilisation d'une interface graphique pour visualiser les itinéraires a renforcé la compréhension du processus d'optimisation et a montré comment les solutions évoluent au fil des générations. Cela a permis non seulement de valider l'efficacité de notre approche, mais aussi de rendre les résultats plus accessibles et compréhensibles.

En conclusion, bien que les algorithmes génétiques ne garantissent pas toujours la solution optimale absolue, ils offrent une méthode puissante et flexible pour aborder des problèmes d'optimisation où les méthodes traditionnelles échouent souvent en raison de la complexité calculatoire. Pour des améliorations futures, explorer des combinaisons avec d'autres techniques heuristiques, comme le recuit simulé, pourrait fournir des résultats encore meilleurs. Ce projet confirme le potentiel des algorithmes génétiques comme un outil précieux dans le domaine de la recherche opérationnelle et de l'optimisation.

5- Annexe :

Dans cette section, vous trouverez le code source du projet – Problème du TSP.

~/Downloads/AI/projetAI.py

```python
#    ------------------------------------------------------------------------------------------------------
#   |                                PROJET TSP - AI - ALGORITHME GENETIQUE                                |
#   |    Realisation : TOUATI MOUNIA  N°20225680 (L3CILS)   -    ALKHOURI George  N° 20214191 (L3CILS)      |
#   |    Année Universitaire : 2023 / 2024                                                                 |
#    ------------------------------------------------------------------------------------------------------

import numpy as np
import random
from tkinter  import *
import tkinter as tk
#_____________________________________METHODES POUR L'ALGORITHMES GENETIQUE_____________________________________
# Fonction pour calculer la matrice de distance entre villes
def distances(Coord_Villes):
    tailleV = len(Coord_Villes)
    dist_villes = np.zeros ((tailleV,tailleV))
    for i in range (tailleV):
        for j in range (tailleV):
           dist_villes [i,j] = np.linalg.norm(Coord_Villes[i]-Coord_Villes[j])
    return dist_villes

#                    ---------------------------------------------------------------------

#Creation d'une population initiale de solutions pour l'algorithme genetique
def init_population(taillep,Coord_Villes):
    return [np.random.permutation(len(Coord_Villes)) for _ in range (taillep)]

#                    ---------------------------------------------------------------------

#Calcule la "fitness" d'une solution basée sur la distance totale parcourue pour l'itinéraire donné
def fitness_idv (individu,distance_villes):
    tailleV = len(distance_villes)
    return 1 / sum (distance_villes[individu[i],individu[(i + 1) % tailleV]] for i in range (tailleV)) #on utilise l'
inverse de la distance totale de l'itinéraire comme valeur de fitness

#                    ---------------------------------------------------------------------

#La selection des parents pour la reproduction en utilisant une méthode de sélection par tournoi.
def tournament_selection(population, fitness,tournament_size,num_parents):
   if tournament_size> len(population):
       raise ValueError ("la taille du tournoi doit etre inferieure ou egale a la taille de la population")
   if num_parents >len(population):
       raise ValueError("Le nombre de parents doit etre inferieur ou egal a la taille de la population")
   parents = []
   for _ in range(num_parents):
        tournament_indices = np.random.choice(len(population), tournament_size, replace=False)
        tournament_fitnesses = [fitness[i] for i in tournament_indices]
        best_index = tournament_indices[np.argmax(tournament_fitnesses)]
        parents.append(population[best_index])
   return parents

#                    ---------------------------------------------------------------------

#Opération de croisement (dans ce probleme le croisement doit etre ordonné en assurant que la solution enfant herite
des génes (indices ville) des deux parents sans répeter aucune ville)
def ox_Croisement(parent1,parent2):
    size = len(parent1)
    enfant =[None]*size
    debut, fin = sorted(random.sample(range(size),2)) #selection aleatoire deux indices dans l'iteneraire qui erviront
de début et de fin pour la section du parent1 à hériter. "sorted" assure que "debut" est avant "fin".
    enfant[debut:fin+1]= parent1[debut:fin+1]#Copie une séquence continue du parent1 dans l'enfant, de l'indice debut
à fin. Cette partie de l'itinéraire de parent1 est directement héritée par l'enfant
    indP2 = 0 #initialisation de l'indice pour commencer a verifier les elemetns de parent2
    for  i in range(size):
         if enfant[i]is None: #Vérifie si la position actuelle dans l'itinéraire de l'enfant est vide
             while parent2[indP2]in enfant : #Tant que l'élément de parent2 à l'indice indP2 est déjà dans l'
itinéraire de l'enfant, continue de passer à l'élément suivant de parent2.
                 indP2+=1
             enfant[i]=parent2[indP2]#Ajoute l'élément courant de parent2 à la position actuelle de l'enfant.
             indP2+=1 # on avance pour la prochaine iteration
    return np.array(enfant)# retourner la tableau numpy d'enfant

#                     ---------------------------------------------------------------------

# Mute une solution en échangeant deux villes de l'itinéraire avec un certain taux de mutation
def mutation (individu,mutation_rate):
    tailleV = len(individu)
    for i in range (tailleV):
```

```
        if random.random()<mutation_rate :
            j= random.randint(0,tailleV-1)
            individu[i],individu [j]=individu[j],individu [i]
    return individu
#                        ------------------------------------------------------------------------
# Fonction algorithme genetique


def algo_Genetique(Coords, taillep, n_generation, mutation_rate, taille_tournament, num_parent):
    distance_m = distances(Coords)  # calculer la matrice des distances entre villes
    population = init_population(taillep, Coords)  # créer une population initiale de solutions (itinéraires)
    best_overall_individu = None
    best_overall_fitness = float('-inf')  # Initialiser avec l'infini négatif pour s'assurer que toute fitness sera plus grande

    for gen in range(n_generation):  # itérer sur le nombre de générations spécifié par n_generation
        fitness_scores = np.array([fitness_idv(ind, distance_m) for ind in population])  # calculer la fitness pour chaque individu de la population
        new_population = []  # initialisation de la nouvelle population pour stocker la nouvelle génération de solutions

        while len(new_population) < taillep:  # dans cette boucle on génère des enfants pour la nouvelle population
            parents = tournament_selection(population, fitness_scores, taille_tournament, num_parent)  # on sélectionne les parents pour la reproduction
            enfant1 = ox_Croisement(parents[0], parents[1])  # génération d'enfant avec la fonction de croisement
            enfant2 = ox_Croisement(parents[1], parents[0])
            new_population.append(mutation(enfant1, mutation_rate))  # applique la mutation sur les deux enfants
            new_population.append(mutation(enfant2, mutation_rate))
        population = new_population  # mise à jour de la population

        # Trouver le meilleur individu de la génération actuelle
        best_individu_gen = max(population, key=lambda x: fitness_idv(x, distance_m))
        best_fitness_gen = fitness_idv(best_individu_gen, distance_m)

        # Comparer avec le meilleur global
        if best_fitness_gen > best_overall_fitness:
            best_overall_fitness = best_fitness_gen
            best_overall_individu = best_individu_gen

        # Imprimer le meilleur individu et la meilleure fitness de la génération actuelle
        print(f"Génération {gen + 1}: Meilleur Individu: {best_individu_gen}, Fitness: {best_fitness_gen:.19f}")

    # Retourne la meilleure solution globale de toutes les générations
    return best_overall_individu


# fonction pour generer des coordonnées aleatoire dans notre canva
def generer_coordonnees_random(n_points, largeur_canvas, hauteur_canvas):
    coordonnees = []
    for _ in range(n_points):
        x = random.randint(10, largeur_canvas - 10)  # 10 est la marge pour que les points ne soient pas collés aux bords
        y = random.randint(10, hauteur_canvas - 10)
        coordonnees.append((x, y))
    return np.array(coordonnees)
#_____________________________________INTERFACE GRAPHIQUE_____________________________________
#fonction pour dessiner les villes et le chemin optimal
def dessiner_villes_chemins(canvas, villes, solution):
    taille_ville = len(villes)
    # Dessiner les villes
    points = []
    for i, (x, y) in enumerate(villes):
        #creation des points pour chaque ville
        point = canvas.create_oval(x - taille_ville, y - taille_ville, x + taille_ville, y + taille_ville, fill='black')
        canvas.create_text(x, y, text=str(i), fill='white', font=('Helvetica', 10, 'bold'))
        points.append((x, y))

    # Dessiner les chemins sous forme d'arcs
    for i in range(len(solution)):
        x1, y1 = villes[solution[i]]
        x2, y2 = villes[solution[(i + 1) % len(solution)]]
        canvas.create_line(x1, y1, x2, y2, fill='red', arrow=tk.LAST)  # Ajouter une flèche pour indiquer la direction


# fonction pour creer la fenetre et la canvas
def creation_interface(villes, solution):
    fenetre = Tk() #creation de la fenetre
    fenetre.title('Visualisation du Problème du Voyageur de Commerce')

```

```
    # Configuration du canvas
    canvas = Canvas(fenetre, width=800, height=800, bg='white')  # Adapter les dimensions selon le besoin
    canvas.pack(expand=YES, fill=BOTH)

    # Dessiner les villes et les chemins
    dessiner_villes_chemins(canvas, villes, solution)

    fenetre.mainloop()




#_____________________________________________MAIN_____________________________________________

nvilles = int(input("Entrez le nombre de villes : "))
coordinates = generer_coordonnees_random(nvilles,800,800)
pop_size = 300
n_generations = 10
mutation_rate = 0.1
tournament_size = 3
num_parents = pop_size

# Exécution de l'algorithme
best_solution = algo_Genetique(coordinates, pop_size, n_generations, mutation_rate, tournament_size, num_parents)
print("Meilleure solution:", best_solution)
print("Meilleure fitness:", fitness_idv(best_solution, distances(coordinates)))

creation_interface(coordinates,best_solution)
```