Rapport de projet

Projet IA

Utilisation algorithme génétique sur problème du voyageur

Participant projet :

-RALAIARINONY Mirindra Nomena Johan (20212409)

-ANAS TALEB (20203008)

Table des matières

I. Introduction

1. Introduction au projet

1-1. Présentation de l'énoncé

Le problème du voyageur de commerce est un défi classique en informatique , impliquant la recherche du chemin le plus court qui passe par un ensemble donné de villes. Dans ce projet, nous abordons ce défi en utilisant l'algorithme génétique, une méthode inspirée de la théorie de l'évolution des espèces.

1-2. Notre compréhension de l’énoncé

Nous avons bien saisi que le problème consiste à trouver le chemin le plus court passant par toutes les villes une seule fois et revenant à la ville de départ. Nous avons également compris que l’algorithme génétique utilise une approche heuristique pour résoudre ce problème et simule le processus de sélection naturelle et d'évolution.

1-3. Explication de l’algorithme génétique

L’algorithme génétique est une technique d'optimisation inspirée de la sélection naturelle et de la génétique. Ils travaillent avec une population d'individus, chaque individu représentant une solution potentielle au problème.

Les étapes clés de l'algorithme génétique comprennent la création d'une population initiale, l'évaluation de la qualité de chaque individu (fitness), la création de nouveaux individus par croisement et mutation, l'ajout de ces nouveaux individus à la population, et la répétition de ce processus jusqu'à ce qu'une solution satisfaisante soit trouvée ou jusqu'à ce qu'un critère d'arrêt soit atteint. Ce processus itératif permet à la population de converger vers des solutions de plus en plus optimales au fil du temps.

II. Notre projet

2. Présentation de notre projet

2-1. Explication du projet

Notre projet vise à résoudre le problème du voyageur de commerce. Pour résoudre ce problème nous avons développé une interface simulant l’utilisation de l’algorithme génétique.

Pour y procédé nous avons décidé de repartir le projet en plusieurs fichier/classes :

Main.py : Ce fichier est le point d'entrée de notre application. Il initialise l'interface graphique et lance le programme c’est dedans qu’on a créé les données test pour le programme .

ListeRoute.py : Ce module contient la classe ListeRoute qui représente une liste de routes possibles entre les villes.

C'est aussi dedans que l'on effectuera la majeur partie de l'algorithme génétique

Region.py : Ce module contient la classe Region qui représente une région géographique contenant un ensemble de villes (on a décidé de nommer un ensemble de ville région pour mieux nous retrouver dans le développement du code.

Route.py : Ce module contient la classe Route qui représente un run de voyageur c'est-à-dire un ensemble aléatoire ou pas de ville parcourus par le voyageur.

Ville.py : Ce module contient la classe Ville qui représente une ville avec ses coordonnées géographiques.

Voyageur.py : Ce module contient la classe Voyageur qui représente un voyageur chargé de trouver un itinéraire optimal pour visiter toutes les villes d'une région donnée.

2-2. Explication par partie du projet

2-2-1 Ville

Cette classe possède un constructeur qui prend en paramètre le nom de la ville ainsi que ses coordonnées.

La méthode distance_vers() calcule la distance entre la ville actuelle et une autre ville donnée en utilisant la distance entre leurs coordonnées.

Cette classe est utilisée pour représenter individuellement chaque ville dans le problème du voyageur de commerce, en permettant le calcul de distances entre les villes

2-2-2 Région

Cette classe possède un constructeur qui soit initialise une région avec un nombre donné de villes, chacun ayant des coordonnées géographiques aléatoire soit initialise une région avec une liste vide de ville .
Il y a également une méthode __len__() qui retourne le nombre de villes dans la région.

Cette classe est utilisée pour représenter une région géographique dans le problème du voyageur de commerce, en facilitant la manipulation des villes qu'elle contient.

2-2-3 Route

Dans cette classe le constructeur initialise une route vide avec des poids initial de 0

La méthode ajouter_ville() ajoute une ville à la route, mettant à jour le poids de la route en fonction de la distance ajoutée.

La méthode ajouter_villes() permet d'ajouter plusieurs villes à la route en appelant la méthode ajouter_ville()

La méthode mutate() effectue une mutation sur la route en échangeant aléatoirement deux villes.

La méthode fitnessReload() recalcule le poids de la route en recalculant la distance totale entre les villes qui la composent. Cela est utile lors de l'évaluation de la qualité d'une solution dans l'algorithme génétique.

Cette classe permet de représenter et manipuler les solutions du problème du voyageur de commerce, ce qui facilite leur évaluation, leur modification et leur utilisation dans le cadre de l'algorithme génétique pour trouver une solution optimale au problème.

2-2-4 Liste Route

Cette classe représente une liste de routes potentielles dans le contexte de l'algorithme génétique. Avec le constructeur qui initialise une liste vide de routes.

La méthode crossover() prend deux routes parentes et effectue un croisement pour produire deux nouvelles routes enfants, avec une probabilité de mutation de 10% pour chaque enfant.

La méthode evolutive_list_route() effectue l'évolution de la liste de routes en appliquant des croisements et des mutations entre les routes existantes pour créer de nouvelles routes, puis en ne gardant que les 50 meilleures routes.

Cette classe permet de manipuler et évoluer les solutions du problème du voyageur de commerce, en appliquant des opérations de croisement et de mutation pour générer de nouvelles solutions et en sélectionnant les meilleures solutions pour la génération suivante.

2-2-5 Voyageur

Cette classe représente un voyageur chargé de trouver un itinéraire optimal pour visiter toutes les villes d'une région donnée. Contenant un constructeur qui prend en paramètre une instance de la classe Région, représentant la région que le voyageur doit visiter.

La méthode voyager() est appelée pour générer un itinéraire de visite des villes de la région.

À chaque étape, le voyageur choisit aléatoirement une ville de la région qui n'a pas encore été visitée.

La méthode retourne finalement l'itinéraire optimal qui passe par toutes les villes de la région une seule fois.

Cela génère aléatoirement une population initiale d'individus, évalue leur qualité (fitness), sélectionne les meilleurs individus pour la reproduction, les croise pour créer de nouveaux individus et les mute occasionnellement. Cet algorithme est exécuté jusqu'à ce qu'une solution satisfaisante soit trouvée ou qu'un nombre maximum d'itérations soit atteint.

2-2-6 Main et Graphics

Le fichier main, gère le processus de résolution du problème du voyageur de commerce en utilisant un algorithme génétique pour trouver un itinéraire optimal. Elle permet aussi l'utilisation d'un algorithme génétique pour résoudre le problème du voyageur de commerce et fournit une représentation visuelle de l'évolution des solutions au fil des générations.

Le fichier Graphics fournit des fonctionnalités pour créer une interface graphique permettant de visualiser les itinéraires de voyage générés par l'algorithme génétique pour résoudre le problème du voyageur de commerce. Elle va d’abord initialiser cette fenêtre, ensuite le dessin des villes puis de l’itinéraire et pour finir l’afficher

III. Remarques et problèmes rencontrées

3.1 Remarques

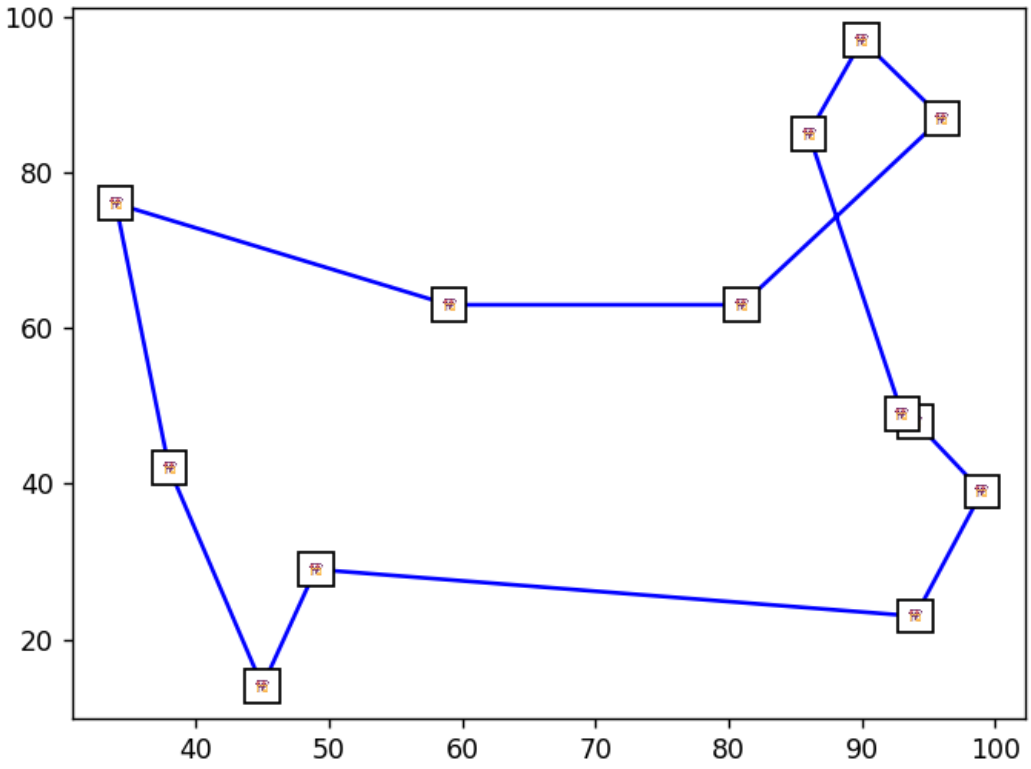
Nous avons trouvé l'exercice assez intéressant dans la forme , elle nous a permis de mieux comprendre le fonctionnement de l'algorithme , en effet la partie simulation via l'évolution humaine se trouve etre assez intriguant , on a réussi à trouver des résultat quoique satisfaisant sans avoir utilisé une réflexion profonde dans le code (ex : algo de Dijkstra) la ou le code agit sans vraiment nous laisser agir .

Par rapport au résultat de l'algorithme , nous avons remarqué que l'algorithme génétique permettait d'avoir un résultat presque optimal mais ne présente pas forcément la meilleure solution a surtout avec un nombre de run limité , (avec une mutation de 10% comme dans notre cas)

3.2 Problèmes rencontrées

Nous avons pas vraiment rencontré énormément de problème dans notre code si ce n'est l'utilisation de reloadfitness() car on a remarqué le calcul de poids automatique ne marchait pas sur les enfants Route

3.3 Exemple lancement du programme



IV. Code en annexe

Ville

```python
class Ville:
    def __init__(self, nom, x, y): # Constructeur
        self.nom = nom
        self.x = x
        self.y = y

    def __str__(self): # Affiche la ville
        return f"Ville: {self.nom}, Coordonnées: ({self.x}, {self.y})"

    def __repr__(self): # Affiche la ville
        return self.__str__()

    def distance_vers(self, autre_ville): # Calcule la distance entre deux villes
        distance_x = abs(self.x - autre_ville.x) # Calcule la distance en x
        distance_y = abs(self.y - autre_ville.y) # Calcule la distance en y
        distance = (distance_x ** 2 + distance_y ** 2) ** 0.5   # Calcule la distance totale
        return distance
```

Région

```python
from Ville import *
import random
class Region:
    def __len__(self):
        return len(self.villes) # Retourne le nombre de villes dans la region
    def __init__(self, number=None): # Constructeur
        self.villes = []
        if number is not None: # Si le nombre de villes est specifie
            for i in range(number): # Creez le nombre de villes specifie
                self.villes.append(Ville("Ville " + str(i), random.randint(0, 100), random.randint(0, 100))) # Creez

    def afficher_villes(self): # Affiche toutes les villes de la region
        for ville in self.villes: # Pour chaque ville dans la liste des villes
            print(ville)

    def ajouter_ville(self, ville): # Ajoute une ville a la region
        self.villes.append(ville)

    def ajouter_villes(self, villes): # Ajoute plusieurs villes a la region
        for ville in villes:
            self.ajouter_ville(ville)

    def enlever_ville(self, ville): # Enleve une ville de la region
        self.villes.remove(ville)

    def choisir_ville_aleatoire(self): # Choisit une ville aleatoire
        return random.choice(self.villes)
```

Route

```python
from Region import *
from Ville import *
import random
class Route:
    def __init__(self): # Constructeur
        self.villes = []
        self.poids = 0

    def ajouter_ville(self, ville): # Ajoute une ville a la route
        self.villes.append(ville) # Ajoute la ville a la liste des villes
        if len(self.villes) >= 2: # Si la route contient au moins 2 villes
            self.poids += ville.distance_vers(self.villes[-2]) # Ajoute le poids de la distance entre la ville et la ville precedente
        else:
            self.poids += 0 # Sinon, le poids est 0

    def ajouter_villes(self, villes): # Ajoute plusieurs villes a la route
        for ville in villes: # Pour chaque ville dans la liste de villes
            self.ajouter_ville(ville) # Ajoute la ville a la route

    def __str__(self): # Affiche la route
        return f"Route: {self.villes}, Poids: {self.poids}"

    def mutate(self):
        # Choisissez deux indices aléatoires
        index1 = random.randint(0, len(self.villes) - 1)
        index2 = random.randint(0, len(self.villes) - 1)

        # Échangez les villes à ces indices
        self.villes[index1], self.villes[index2] = self.villes[index2], self.villes[index1]

    def fitnessReload(self):
        self.poids = 0
        for i in range(len(self.villes) - 1):  # s'arrête à l'avant-dernier élément
            self.poids += self.villes[i].distance_vers(self.villes[i+1]) # ajoute la distance entre la ville i et la ville i+1
```

Liste route

```python
import random
from Region import *
from Ville import *
from Route import *

class ListeRoute:

    def __init__(self): # Constructeur
        self.routes = []

    def crossover(self,route1, route2):
        # Choisissez un point d'index aléatoire
        index = random.randint(1, len(route1.villes) - 1)

        # Créez les premières moitiés des nouvelles routes
        new_route1_villes = route1.villes[:index]
        new_route2_villes = route2.villes[:index]

        # Ajoutez les villes manquantes à la fin des nouvelles routes
        new_route1_villes += [ville for ville in route2.villes if ville not in new_route1_villes]
        new_route2_villes += [ville for ville in route1.villes if ville not in new_route2_villes]

        # Créez les nouvelles routes
        new_route1 = Route()
        new_route1.ajouter_villes(new_route1_villes)
        new_route2 = Route()
        new_route2.ajouter_villes(new_route2_villes)

        if random.randint(0,1000)<10: # 10% de chance de mutation
            new_route1.mutate()
        if random.randint(0,1000)<10: # 10% de chance de mutation
            new_route2.mutate()
        new_route1.fitnessReload() # Recalcul du poids
        new_route2.fitnessReload() # Recalcul du poids
        return new_route1, new_route2 # Retourne les deux nouvelles routes
```

```python
def add_route(self, route):
    self.routes.append(route) # Ajoute une route à la liste

def remove_route(self, route):
    self.routes.remove(route)

def get_routes(self):
    return self.routes

def clear_routes(self):
    self.routes = []

def print_routes(self):
    print("==================================")
    for route in self.routes:

        print(route)
        print("==================================")

def trier_routes(self):
    self.routes.sort(key=lambda x: x.poids)
```

```python
def evolutive_list_route(self,n=50):

    for i in range(n):
        # Sélectionnez deux routes parentes
        parent1 = self.routes[i]
        parent2 = self.routes[i+1]

        # Effectuez le croisement
        child1, child2 = self.crossover(parent1, parent2)

        # Ajoutez les enfants à la liste des routes
        self.add_route(child1)
        self.add_route(child2)

    # Triez les routes par poids
    self.trier_routes()

    # Ne gardez que les 50 meilleures routes
    self.routes = self.routes[:n]

    # Affichez les routes
    #self.print_routes()
```

Voyageur

```python
import random
from Region import *
from Ville import *
from Route import *
class Voyageur:
    def __init__(self, region):
        self.region = region

    def voyager(self):

        # Cree une liste pour stocker les regions visitees
        visited_regions = []

        # Cree une liste pour stocker le trajet
        route = []

        # Tant que toutes les regions n'ont pas ete visitees
        while len(visited_regions) < len(self.region):
            # Choisir une region aleatoire
            random_ville = self.region.choisir_ville_aleatoire()

            # Verifier si la region a deja ete visitee
            if random_ville not in visited_regions:
                # Ajouter la region a la liste des regions visitees
                visited_regions.append(random_ville)

                # Ajouter la region a la liste du trajet
                route.append(random_ville)

        # Print the final route
        #print(route)
        return route
```

Graphics

```python
import matplotlib.pyplot as plt
from matplotlib.offsetbox import OffsetImage, AnnotationBbox
import time
class Graphics:
    def __init__(self): # Constructeur
        self.fig, self.ax = plt.subplots()

    def draw_cities(self, cities, img_path): # Dessine les villes
        img = plt.imread(img_path)
        imagebox = OffsetImage(img, zoom=0.1)

        for city in cities: # Pour chaque ville dans la liste des villes
            ab = AnnotationBbox(imagebox, (city.x, city.y))
            self.ax.add_artist(ab) # Ajoute la ville au graphique

    def draw_route(self, route): # Dessine la route
        x = [city.x for city in route.villes] # Liste des coordonnees x des villes
        y = [city.y for city in route.villes] # Liste des coordonnees y des villes
        self.ax.plot(x + [x[0]], y + [y[0]], 'b-')  # 'b-' signifie une ligne bleue

    def show(self):
        plt.show() # Affiche le graphique

    def update(self, route): # Met a jour le graphique
        self.ax.clear() # Efface le graphique
        self.draw_cities(route.villes, 'img/ville.png') # Dessine les villes
        self.draw_route(route) # Dessine la route
        plt.draw()
        plt.pause(0.5)  # Pause pour permettre la mise à jour du graphique
```

Main

```
from Region import *
from Route import *
from Ville import *
from Voyageur import *
from ListeRoute import *
from Grapics import *
class Main:
    france = Region() # Creez une region

    # Ajoutez des villes a la region
    france.ajouter_ville(Ville("Ville1",random.randint(0,100),random.randint(0,100) ))
    france.ajouter_ville(Ville("Ville2",random.randint(0,100),random.randint(0,100) ))
    france.ajouter_ville(Ville("Ville3",random.randint(0,100),random.randint(0,100) ))
    france.ajouter_ville(Ville("Ville4",random.randint(0,100),random.randint(0,100) ))
    france.ajouter_ville(Ville("Ville5",random.randint(0,100),random.randint(0,100) ))
    france.ajouter_ville(Ville("Ville6",random.randint(0,100),random.randint(0,100) ))
    france.ajouter_ville(Ville("Ville7",random.randint(0,100),random.randint(0,100) ))
    france.ajouter_ville(Ville("Ville8",random.randint(0,100),random.randint(0,100) ))
    france.ajouter_ville(Ville("Ville9",random.randint(0,100),random.randint(0,100) ))
    france.ajouter_ville(Ville("Ville10",random.randint(0,100),random.randint(0,100) ))
    france.ajouter_ville(Ville("Ville11",random.randint(0,100),random.randint(0,100) ))
    france.ajouter_ville(Ville("Ville12",random.randint(0,100),random.randint(0,100) ))
    france.ajouter_ville(Ville("Ville13",random.randint(0,100),random.randint(0,100) ))

    # Creez un voyageur
    jean = Voyageur(france)
    list=ListeRoute()
    # Creez 100 routes
    for i in range(100):
        route=Route()
        route.ajouter_villes(jean.voyager())
        route.fitnessReload()
        list.add_route(route)
```

```
# Creez une fenetre graphique
 graphics = Graphics()
 graphics.draw_cities(france.villes, 'img/ville.png')
 # Boucle d'evolution
 for generation in range(10):
     list.evolutive_list_route(100)
     meilleure_route = list.routes[0]
     graphics.update(meilleure_route)

 list.print_routes()
 graphics.show()
 # Affichez les routes
```