CODING CLUB

EPITECH

Voyageur des plages

DIVERSITY by EPITECH

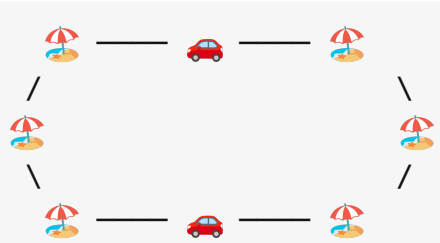
Summer Beaches 🏖️ - Le Voyageur des Plages

Introduction

Cet été, Alex a décidé de faire le road trip de sa vie ! Avec ses amis, il veut découvrir les plus belles plages d'Espagne. Le problème ? L'essence coûte cher, le temps est limité, et il y a tellement de plages magnifiques à voir...

Alex a besoin de ton aide pour planifier l'itinéraire parfait : **visiter toutes les plages de sa liste en parcourant le moins de kilomètres possible** !

Ce défi s'appelle le "Problème du Voyageur de Commerce" (TSP - Traveling Salesman Problem), et c'est l'un des problèmes les plus célèbres en informatique. Prêt à relever le défi ?



Objectifs

- Découvrir les algorithmes d'optimisation
- Manipuler des coordonnées et calculer des distances
- Implémenter différentes stratégies de résolution
- S'amuser avec de la visualisation (bonus)
- Comprendre la complexité algorithmique

Structure du Projet

- `main.py` : Point d'entrée du programme
- `beaches.py` : Données des plages espagnoles

Étape 1 : Les Fondations 🏗️

1.1. Découverte des Plages

Alex a sélectionné 8 plages incontournables en Espagne. Chaque plage a des coordonnées GPS que nous allons simplifier, on peut les retrouver dans beaches.py

TODO 1.1 : Dans le fichier `main.py`, charge ces données et affiche la liste des plages disponibles.

```
from beaches import beaches

def display_beaches():
    print("🏖️  PLAGES DISPONIBLES  🏖️")
    print("-" * 30)
    # TODO: Afficher chaque plage avec ses coordonnées
    pass

if __name__ == "__main__":
    display_beaches()
```

1.2. Calcul de Distance

Pour planifier l'itinéraire, nous devons calculer la distance entre deux plages. Nous utiliserons la **distance euclidienne** (à vol d'oiseau) :

```
import math

def calculate_distance(beach1_coords, beach2_coords):
    """
    Calcule la distance entre deux plages

    Args:
        beach1_coords: tuple (x, y) des coordonnées de la première plage
        beach2_coords: tuple (x, y) des coordonnées de la deuxième plage

    Returns:
        float: distance entre les deux plages
    """
    # TODO: Implémenter la formule de distance euclidienne
    # distance = √((x2-x1)² + (y2-y1)²)
    pass
```

TODO 1.2 : Complétez cette fonction et testez-la avec quelques exemples :

```
# Test de la fonction
print(f"Distance Barcelona -> Valencia: {calculate_distance(beaches['Barcelona'
```

Étape 2 : Premier Algorithme - Force Brute 💪

2.1. Générer tous les Itinéraires Possibles

Pour 3-4 plages, nous pouvons tester **tous les itinéraires possibles** !

```
from itertools import permutations

def generate_all_routes(beach_list):
    """
    Génère toutes les permutations possibles des plages

    Args:
        beach_list: liste des noms des plages

    Returns:
        list: toutes les routes possibles
    """
    # TODO: Utiliser itertools.permutations pour générer toutes les routes
    pass
```

Question : Si nous avons 4 plages, combien y a-t-il d'itinéraires possibles ? Et pour 8 plages ?

2.2. Algorithme Force Brute

```python
def brute_force_tsp(beaches_dict):
    """
    Résout le TSP par force brute (teste toutes les possibilités)

    Args:
        beaches_dict: dictionnaire {nom: (x, y)} des plages

    Returns:
        tuple: (meilleur_itinéraire, distance_totale)
    """
    beach_names = list(beaches_dict.keys())
    best_route = None
    best_distance = float('inf')

    # TODO:
    # 1. Générer toutes les routes possibles
    # 2. Pour chaque route, calculer la distance totale
    # 3. Garder la meilleure route

    return best_route, best_distance
```

TODO 2.2 : Testez votre algorithme avec 4 plages d'abord !

```python
# Test avec un sous-ensemble
small_beaches = {k: beaches[k] for k in list(beaches.keys())[:4]}
route, distance = brute_force_tsp(small_beaches)
print(f"Meilleur itinéraire: {' -> '.join(route)}")
print(f"Distance totale: {distance:.2f} km")
```

Étape 3 : Algorithme Glouton - Plus Proche Voisin 🎯

3.1. Comprendre l'Algorithme

L'algorithme force brute devient rapidement impossible (8! = 40,320 possibilités). L'algorithme du **plus proche voisin** est une heuristique rapide :

1. Commencer à une plage de départ
2. Aller à la plage la plus proche non visitée
3. Répéter jusqu'à avoir visité toutes les plages
4. Revenir au point de départ

3.2. Implémentation

```python
def nearest_neighbor_tsp(beaches_dict, start_beach=None):
    """
    Résout le TSP avec l'algorithme du plus proche voisin

    Args:
        beaches_dict: dictionnaire des plages
        start_beach: plage de départ (si None, prend la première)

    Returns:
        tuple: (itinéraire, distance_totale)
    """
    if start_beach is None:
        start_beach = list(beaches_dict.keys())[0]

    unvisited = set(beaches_dict.keys())
    route = [start_beach]
    unvisited.remove(start_beach)
    current = start_beach

    # TODO: Implémenter l'algorithme du plus proche voisin
    while unvisited:
        # 1. Trouver la plage la plus proche parmi les non visitées
        # 2. L'ajouter à l'itinéraire
        # 3. La retirer des non visitées
        # 4. Mettre à jour la position actuelle
        pass

    # TODO: Calculer la distance totale de l'itinéraire
    total_distance = calculate_route_distance(route, beaches_dict)

    return route, total_distance
```

TODO 3.2 : Testez les deux algorithmes et comparez les résultats !

```python
print("=== COMPARAISON DES ALGORITHMES ===")

# Force brute (seulement pour 6 plages max)
```

```
small_beaches = {k: beaches[k] for k in list(beaches.keys())[:6]}
route_bf, dist_bf = brute_force_tsp(small_beaches)
print(f"Force Brute: {dist_bf:.2f} km")

# Plus proche voisin
route_nn, dist_nn = nearest_neighbor_tsp(beaches)
print(f"Plus Proche Voisin: {dist_nn:.2f} km")
```

Étape 4 : Bonus et Extensions 🎨

4.1. Défis Avancés

Pour les Plus Motivés :

1. **Contraintes Réelles :** Ajoutez des contraintes comme les horaires d'ouverture des plages, la météo, etc.
2. **Algorithme Génétique :** Implémentez un algorithme génétique pour résoudre le TSP : `python def genetic_algorithm_tsp(beaches_dict, population_size=100, generations=500): # TODO: Implémenter l'algorithme génétique pass`
3. **Interface Web :** Créez une interface web avec Flask/Django pour planifier des voyages.
4. **API Réelle :** Utilisez une vraie API de cartographie pour obtenir les distances de route réelles.
5. **Multi-Objectifs :** Optimisez non seulement la distance, mais aussi le coût, le temps, la beauté des plages, etc.

5.4. Données Réelles

Ajoutez plus de plages avec leurs vraies coordonnées :

```
extended_beaches = {
    # Côte Est
    "Barcelona": (2.15, 41.38),
    "Sitges": (1.81, 41.24),
    "Valencia": (0.37, 39.47),
    "Benidorm": (-0.13, 38.54),
    "Alicante": (-0.48, 38.35),
    "Cartagena": (-0.98, 37.60),

    # Côte Sud
    "Almeria": (-2.46, 36.84),
    "Malaga": (-4.42, 36.72),
    "Marbella": (-4.89, 36.51),
    "Cadiz": (-6.29, 36.53),

    # Côte Ouest
    "Huelva": (-6.95, 37.26),
    "Porto": (-8.61, 41.15),  # Portugal, pourquoi pas !
}
```

Conclusion 🎯

Félicitations ! Vous avez découvert :

- **Le problème du TSP** et sa complexité
- **Différents algorithmes** : force brute, heuristiques, optimisation locale
- **L'importance des compromis** entre qualité de solution et temps de calcul
- **La visualisation** de données géographiques

Le TSP apparaît dans de nombreux domaines : logistique, fabrication de circuits, bioinformatique, et même dans l'optimisation des trajets de livraison !

Pour Aller Plus Loin

- Recherchez d'autres algorithmes : Simulated Annealing, Ant Colony, Branch & Bound
- Découvrez la théorie de la complexité (classes P et NP)
- Explorez d'autres problèmes d'optimisation combinatoire

Bon voyage en Espagne ! 🇪🇸🏖️