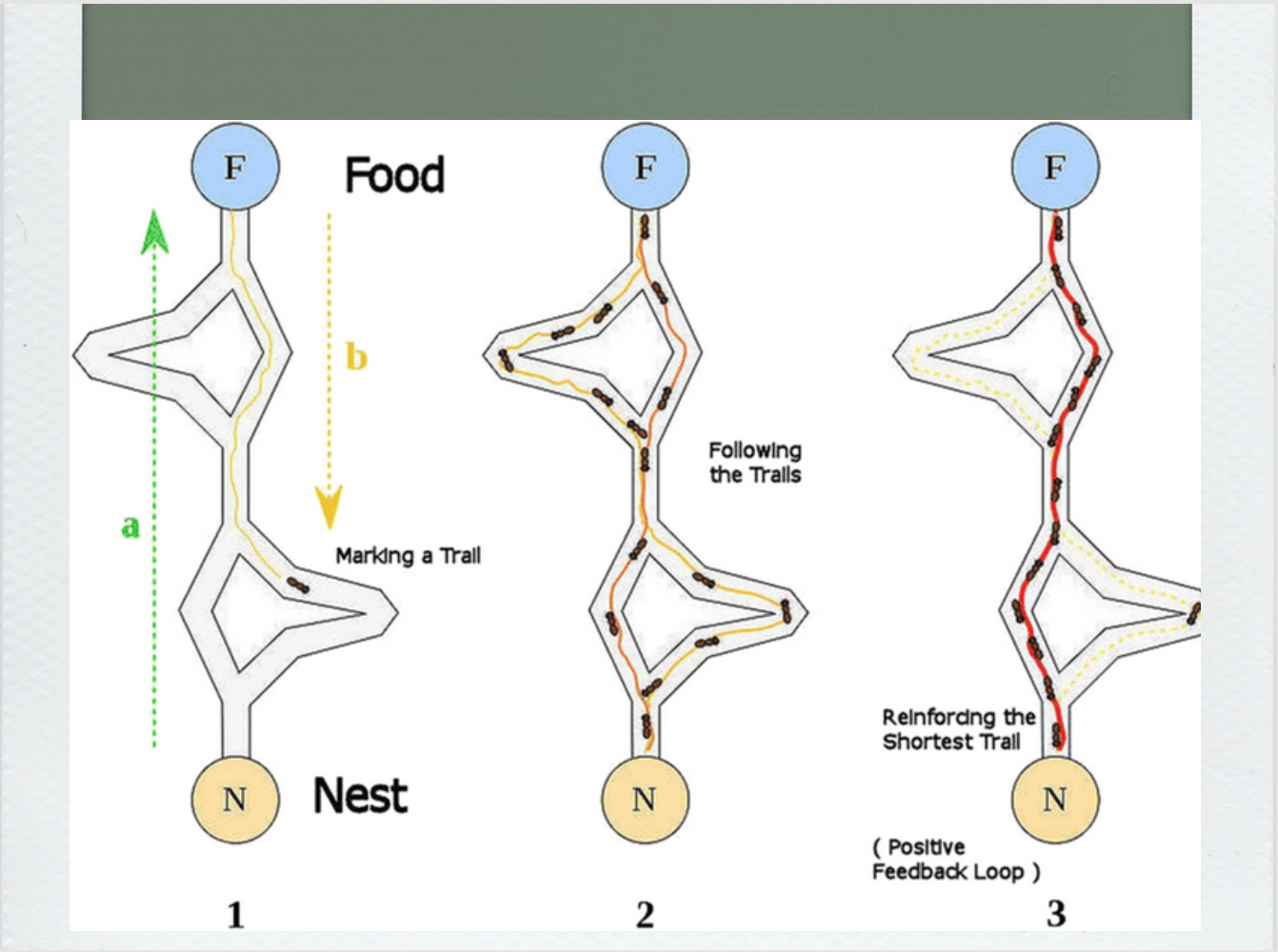


COLONIE DE FOURMIS

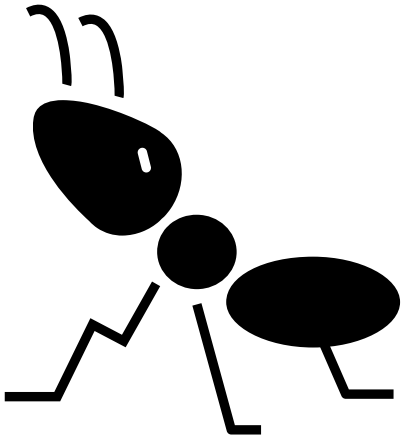
01/07/2025

Prepared BY :
Meryem Badaoui

Prepared for :
El Hilali Alaoui Ahmed

A propos de colonie de fourmis

L'optimisation par colonie de fourmis (ACO pour Ant Colony Optimization) est un algorithme méta-heuristique inspiré du comportement de recherche de nourriture des fourmis. Il est utilisé pour trouver des solutions approximatives à des problèmes d'optimisation difficiles, tels que le problème du voyageur de commerce (TSP). Dans ACO, un groupe de fourmis artificielles cherche une bonne solution au problème d'optimisation en construisant une solution incrémentalement. Chaque fourmi construit une solution en choisissant la prochaine ville à visiter en fonction des concentrations de phéromones sur les arêtes entre les villes et de la distance entre les villes. Les concentrations de phéromones sont mises à jour en fonction de la qualité des solutions trouvées par les fourmis.
ACO a plusieurs paramètres qui peuvent être ajustés pour affiner les performances de l'algorithme, tels que le nombre de fourmis, le nombre d'itérations, le taux d'évaporation des phéromones et l'importance des concentrations de phéromones par rapport à la distance dans le processus de décision des fourmis



Le problème de voyageur de commerce

Le problème du voyageur de commerce (TSP pour Traveling Salesman Problem) est un problème d'optimisation classique dans lequel un commerçant doit visiter un ensemble donné de villes, en visitant chaque ville exactement une fois, et en retournant à la ville de départ. L'objectif est de trouver le plus court itinéraire possible qui visite toutes les villes et retourne à la ville de départ.
Le TSP est un problème bien connu NP-difficile, ce qui signifie qu'il n'existe pas d'algorithme connu qui puisse le résoudre efficacement pour des entrées de grande taille. Cependant, il existe des algorithmes approchés, tels que ACO, qui peuvent trouver de bonnes solutions au TSP en un temps raisonnable.
Dans le TSP, l'entrée consiste en un ensemble de villes et les distances entre elles. La sortie est une liste des villes dans l'ordre où elles doivent être visitées, de sorte que la distance totale parcourue soit minimisée.

Le problème Détaillé

Voici un énoncé plus formel du problème du voyageur de commerce sous forme constatée de problème d'optimisation combinatoire. Soit un graphe complet $G = (V, A, \omega)$ avec V un ensemble de sommets, A un ensemble d'arêtes et ω une fonction de coût sur les arcs. Le problème est de trouver le plus court cycle hamiltonien dans le graphe.
Remarque : Rien n'interdit au graphe donné en entrée d'être orienté. Dans ce cas, on considère qu'un chemin existe dans un sens mais pas dans l'autre (exemple: routes a sens unique).

Étape de l'algorithme :

1. Initialisation des taux d'évaporation de phéromones et la quantité de phéromones sur chaque arête du graphe à une valeur initiale. La quantité de phéromones sur une arête représente l'attractivité de cette arête pour les fourmis.

1. Création d'une certaine quantité de fourmis et les placer au hasard sur les villes du graphe.

1. Pour chaque fourmi, exécutez les étapes suivantes jusqu'à ce qu'elle ait visité toutes les villes : a. Déterminez la prochaine ville à visiter en fonction de la quantité de phéromones sur chaque arête et de la distance entre les villes. Plus il y a de phéromones sur une arête et plus la distance entre les villes est petite, plus la probabilité que la fourmi choisisse cette arête sera grande. b. Visitez la ville suivante et mettez à jour la quantité de phéromones sur les arêtes en ajoutant une quantité de phéromones déterminée par une formule (par exemple, une formule qui prend en compte la distance totale parcourue par la fourmi et le nombre total de villes visitées). c. Répétez les étapes a et b jusqu'à ce que la fourmi ait visité toutes les villes

1. A la fin de l'algorithme, la solution finale sera celle qui a été trouvée par les fourmis et qui a la plus courte distance totale parcourue.

Partie Programmation :

Voici ce qui se passe dans l'algorithme des colonies de fourmis pour résoudre le problème du voyageur de commerce :

```python
import random
from typing import Dict, List

def calcul_distance(city: int, next_city: int, cities: Dict[int, Dict[int, float]]) -> float:
#Calcul la distance entre deux villes dans un graphe donné"""
#city : l'indice de la ville de départ
#nest_city : l'indice de la ville de destination
#cities : un dictionnaire qui représente le graphe, avec les villes en tant que clés et les distances entre les villes en tant que valeurs
    return cities[city][next_city]
```

La fonction calculate_distance prend en entrée 3 arguments :

- city : l'indice de la ville de départ
- next_city : l'indice de la ville de destination
- cities : un dictionnaire qui représente le graphe, avec les villes en tant que clés et les distances entre les villes en tant que valeurs

La fonction retourne la distance entre la ville de départ et la ville de destination en utilisant les valeurs du graphe.
Cette fonction est utilisée pour calculer la distance entre deux villes dans le graphe pour les autres fonctions de l'algorithme

```python
def calculate_probability(city: int, actual_city: int, cities: Dict[int, Dict[int, float]], alpha: float, beta: float) -> float:
    """Calcul la probabilité de se déplacer d'une ville à une autre dans le graphe, en fonction des valeurs de alpha et beta"""
    return (cities[actual_city][city] ** alpha) / (calcul_distance(actual_city, city, cities) ** beta)
```

La fonction calculate_probability prend en entrée 5 arguments :

- city : l'indice de la ville de destination
- actual_city : l'indice de la ville actuelle
- cities : un dictionnaire qui représente le graphe, avec les villes en tant que clés et les distances entre les villes en tant que valeurs
- alpha : un paramètre de l'algorithme qui permet de pondérer l'importance de la quantité de phéromones sur la probabilité de choisir une ville
- beta : un paramètre de l'algorithme qui permet de pondérer l'importance de la distance sur la probabilité de choisir une ville

La fonction retourne la probabilité de se déplacer de la ville actuelle à la ville de destination en utilisant les valeurs alpha et beta et la distance entre les deux villes.
Cette fonction est utilisée pour calculer la probabilité de se déplacer d'une ville à une autre dans le graphe pour les autres fonctions de l'algorithme.

Partie Programmation :

```python
def choose_destination(availaible_city: List[int], actual_city: int, cities: Dict[int, Dict[int, float]], alpha: float, beta: float) -> int:
#Sélectionne aléatoirement la prochaine ville à laquelle se déplacer, en fonction des probabilités calculées"""
#availaible_city : une liste des indices des villes disponibles pour se déplacer
#actual_city : l'indice de la ville actuelle
#cities : un dictionnaire qui représente le graphe, avec les villes en tant que clés et les distances entre les villes en tant que valeurs
#alpha : un paramètre de l'algorithme qui permet de pondérer l'importance de la quantité de phéromones sur la probabilité de choisir une ville
#beta : un paramètre de l'algorithme qui permet de pondérer l'importance de la distance sur la probabilité de choisir une ville

    probabilites = [calculate_probability(v, actual_city, cities, alpha, beta) for v in availaible_city]
    return random.choices(availaible_city, probabilites)[0]
```

La fonction choose_destination prend en entrée 5 arguments :

- availaible_city : une liste des indices des villes disponibles pour se déplacer
- actual_city : l'indice de la ville actuelle
- cities : un dictionnaire qui représente le graphe, avec les villes en tant que clés et les distances entre les villes en tant que valeurs
- alpha : un paramètre de l'algorithme qui permet de pondérer l'importance de la quantité de phéromones sur la probabilité de choisir une ville
- beta : un paramètre de l'algorithme qui permet de pondérer l'importance de la distance sur la probabilité de choisir une ville

La fonction utilise la fonction random.choices pour sélectionner aléatoirement la prochaine ville à laquelle se déplacer, en fonction des probabilités calculées par la fonction calculate_probaility . Elle retourne l'indice de la ville choisie.
Cette fonction est utilisée pour choisir aléatoirement la prochaine ville à laquelle se déplacer pour les autres fonctions de l'algorithme.

```python
[16] def maj_feromones(cities: Dict[int, Dict[int, float]], epsilon: float, Q: float, distance: float) -> None:
    #Met à jour les feromones dans le graphe, en utilisant une valeur donnée de epsilon, Q et la distance totale du circuit actuel"""
    #cities : un dictionnaire qui représente le graphe, avec les villes en tant que clés et les distances entre les villes en tant que valeurs
    #epsilon : un paramètre de l'algorithme qui permet de définir le taux d'évaporation des phéromones
    #Q : un paramètre de l'algorithme qui permet de définir la quantité de phéromones déposée sur le chemin optimal
    #distance : la distance totale du circuit actuel
        for i, j in cities.items():
            for k, l in j.items():
                cities[i][k] = (1 - epsilon) * l + epsilon * (Q / distance)
```

La fonction maj_feromones prend en entrée 4 arguments :

- cities : un dictionnaire qui représente le graphe, avec les villes en tant que clés et les distances entre les villes en tant que valeurs
- epsilon : un paramètre de l'algorithme qui permet de définir le taux d'évaporation des phéromones
- Q : un paramètre de l'algorithme qui permet de définir la quantité de phéromones déposée sur le chemin optimal
- distance : la distance totale du circuit actuel

La fonction met à jour les phéromones dans le graphe en utilisant la formule (1-epsilon)l + epsilon(Q/distance) pour chaque arc du graphe, où l est la quantité de phéromones actuelle sur l'arc. Cette fonction ne retourne aucun résultat.
Cette fonction est utilisée pour mettre à jour les phéromones dans le graphe après chaque itération de l'algorithme pour favoriser les chemins les plus courts.

Partie Programmation :

```python
def iterate(cities: Dict[int, Dict[int, float]], alpha: float, beta: float, epsilon: float, Q: float, K: int) -> List[List[int]]:
#Génère itérativement un circuit en sélectionnant aléatoirement la prochaine ville à laquelle se déplacer, en mettant à jour le graphe, et en enregistrant le circuit.
    circuits = []
    for k in range(K):
        #Pour chaque itération, elle définit la ville actuelle comme la première ville du graphe et crée une liste de villes visitées qui contient uniquement cette ville.
        actual_city = list(cities.keys())[0]
        visited_cities = [actual_city]
        #Elle crée également une liste de villes disponibles qui contient toutes les autres villes du graphe.
        available_city = list(cities.keys())[1:]
        while len(available_city) > 0:
            next_city = choose_destination(available_city, actual_city, cities, alpha, beta)
            available_city.remove(next_city)
            visited_cities.append(next_city)
            actual_city = next_city
#Une fois que toutes les villes ont été visitées, la fonction ajoute la première ville (ville de départ) à la liste de villes visitées pour fermer le circuit.
#Elle utilise la fonction "calcul_distance" pour calculer la distance totale du circuit et la fonction "maj_feromones" pour mettre à jour les feromones dans le graphe.
        visited_cities.append(visited_cities[0])

        distance = sum(calcul_distance(visited_cities[i], visited_cities[i+1], cities) for i in range(len(visited_cities)-1))
        maj_feromones(cities, epsilon, Q, distance)

        circuits.append(visited_cities)
    return circuits
```

La fonction commence par initialiser une liste vide pour stocker les circuits générés. Elle effectue ensuite une boucle K fois pour générer autant de circuits.
Pour chaque itération, elle définit la ville actuelle comme la première ville du graphe et crée une liste de villes visitées qui contient uniquement cette ville. Elle crée également une liste de villes disponibles qui contient toutes les autres villes du graphe.
Ensuite, la fonction entre dans une boucle qui se poursuit tant qu'il reste des villes à visiter. À chaque itération, elle utilise la fonction "choose_destination" pour sélectionner aléatoirement la prochaine ville à visiter, en fonction des probabilités calculées à l'aide de la fonction "calculate_probability". Elle ajoute ensuite cette ville aux villes visitées et la retire des villes disponibles.
Une fois que toutes les villes ont été visitées, la fonction ajoute la première ville (ville de départ) à la liste de villes visitées pour fermer le circuit. Elle utilise la fonction "calculate_distance" pour calculer la distance totale du circuit et la fonction "maj_feromones" pour mettre à jour les feromones dans le graphe.
Enfin, elle ajoute le circuit généré à la liste de circuits et retourne cette liste en fin de fonction.

Partie Programmation :

```
[24] def solve(cities: Dict[int, Dict[int, float]], alpha: float, beta: float, epsilon: float, Q: float, tmax: int, K: int) -> List[List[int]]:
     #Résout le problème en utilisant les fonctions ci-dessus, en utilisant les paramètres donnés, et retourne les circuits générés."""
     #cities : un dictionnaire qui représente le graphe, avec les villes en tant que clés et les distances entre les villes en tant que valeurs
     #alpha : un paramètre de l'algorithme qui permet de pondérer l'importance de la quantité de phéromones sur la probabilité de choisir une ville
     #beta : un paramètre de l'algorithme qui permet de pondérer l'importance de la distance sur la probabilité de choisir une ville
     #epsilon : un paramètre de l'algorithme qui permet de définir le taux d'évaporation des phéromones
     #Q : un paramètre de l'algorithme qui permet de définir la quantité de phéromones déposée sur le chemin optimal
     #tmax : le nombre d'itérations maximales de l'algorithme
     #K : le nombre d'itérations de l'algorithme
         random.seed()
         for t in range(tmax):
             iterate(cities, alpha, beta, epsilon, Q, K)
         return iterate(cities, alpha, beta, epsilon, Q, K)
```

La fonction utilise la fonction iterate pour générer des circuits de manière itérative en utilisant l'algorithme de colonie de fourmis et en utilisant les paramètres donnés. Elle utilise également la fonction random.seed() pour initialiser le générateur de nombres aléatoires. Elle retourne une liste de circuits générés.
Cette fonction est utilisée pour résoudre le problème en utilisant l'algorithme de colonie de fourmis.

```
5] # Définir les données sous forme de graphe
   #0 indique le numero de la ville par exemple je vais affecter le 0 a Rabat
   #1 indique le numero de la ville par exemple je vais affecter le 1 a Casa
   # 2 indique le numero de la ville par exemple je vais affecter le 2  fes
   # 3 indique le numero de la ville par exemple je vais affecter le 3  tanger
   cities = {
       0: {1: 88, 2: 200, 3: 247},
       1: {0: 88, 2: 296, 3: 340},
       2: {0: 200, 1: 296, 3: 400},
       3: {0: 247, 1: 340, 2: 400},
   }

   # Définir les paramètres de l'algorithme
   alpha = 1
   beta = 2
   epsilon = 0.1
   Q = 1
   tmax = 1
   K = 3
   # je vais choisir 5 iterations
   # Appeler la fonction pour résoudre le problème
   path = solve(cities, alpha, beta, epsilon, Q, tmax, K)

   # Afficher le résultat
   print("path",path)
```

Ce code définit d'abord les données sous forme de graphe,**j ai choisi de travailler sur les villes du maroc le dictionnaire presente les distances entre les villes : fes casa rabat tanger**
Il définit ensuite les paramètres de l'algorithme : alpha, beta, epsilon, Q, tmax et K. Ces paramètres sont utilisés pour contrôler le comportement de l'algorithme.
Il appelle ensuite la fonction solve en lui passant le graphe, les paramètres de l'algorithme et les autres arguments requis pour résoudre le problème.
Enfin, il imprime le résultat en utilisant la commande print("path",path). Le résultat sera sous forme de liste de listes contenant des circuits générés par l'algorithme.

```
path [[0, 2, 3, 1, 0], [0, 2, 1, 3, 0], [0, 1, 3, 2, 0]]
```

il me donne comme solution les circuits suivants puisque j'ai travaille avec k=3 donct 3 fourmis

Conclusion:

En conclusion, l'algorithme des colonies de fourmis est une méthode qui peut être utilisée pour résoudre le problème du voyageur de commerce. Il consiste à simuler le comportement de fourmis qui parcourent un graphe en utilisant des phéromones comme indicateur de la qualité des différentes arêtes. La quantité de phéromones sur chaque arête est mise à jour à chaque itération de l'algorithme en fonction de la distance parcourue par les fourmis et de la qualité du circuit obtenu. En répétant ce processus plusieurs fois, on peut espérer obtenir un circuit de bonne qualité qui passe par toutes les villes du graphe.