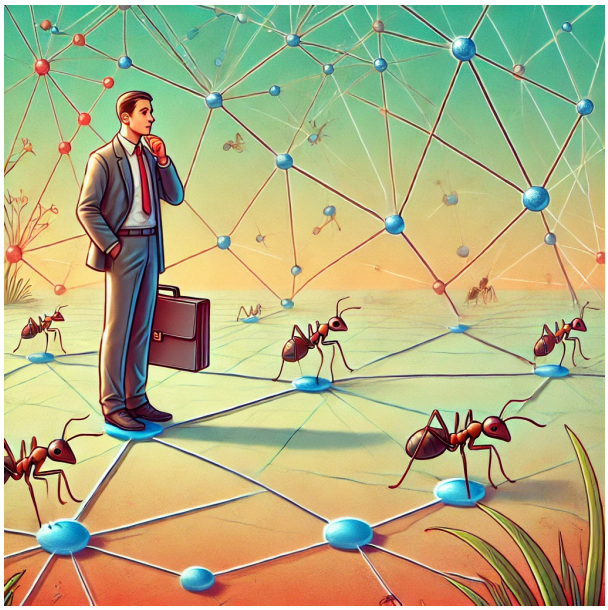
Une colonie de fourmis
au secours d'un voyageur de commerce



1. Le problème TSP

Le problème du voyageur de commerce (TSP, pour "Traveling Salesman Problem") est un problème d'optimisation combinatoire classique en mathématiques et en informatique. Il consiste à trouver le chemin le plus court qu' un vendeur itinérant doit emprunter pour visiter une liste donnée de villes exactement une fois chacune, puis revenir à son point de départ. On considèrera par la suite que toutes les villes sont reliées entre elles.

1.1. Principales caractéristiques :

1. Entrée:
- Une liste de villes (points) et les distances ou coûts entre chaque paire de villes.

2. Objectif:
- Déterminer l'ordre optimal pour visiter toutes les villes afin de minimiser la distance totale parcourue ou le coût total.

3. Contraintes:
- Chaque ville doit être visitée une seule fois.
- Le voyage doit commencer et se terminer au même endroit.

On dit qu'on cherche le plus court cycle Hamiltonien. Un cycle Hamiltonien est un cycle passant une seule fois par chaque sommet du graphe.
Il ne faut pas le confondre avec un cycle Eulérien qui est un cycle passant par chaque arête une seule fois.

1.2. Applications :

Le TSP a des applications dans de nombreux domaines tels que la logistique, la planification de tournées, l'optimisation des routes pour les livraisons, et même la conception de circuits électroniques.

1.3. Modélisation du problème:

Il est aisé de se représenter les villes reliées entre elles par un graphe pondéré par les distances entre les villes. Qui plus est, ce graphe est complet car chaque ville est reliée à toutes les autres.

Approche brute force:

On peut se dire qu'il suffit de générer tous les chemins possibles, de calculer un bête minimum, et l'affaire est pliée.

```
longueur_minimum = +∞
chemin_optimal = ∅
Pour chaque chemin possible:
    si longueur(chemin)<longueur_minimum:
       chemin_optimal = chemin
       longueur_minimum = longueur(chemin_optimal)
```

Étudions sommairement la complexité de cet algorithme.
On s'intéresse à la ligne *Pour chaque chemin possible* car toutes les autres lignes sont en $O(1)$

Combien de chemins possibles existe-t-il donc? C'est un problème classique de dénombrement.

Nous partons d'un sommet du graphe complet. il nous reste n-1 sommets à parcourir. A chaque fois qu'on choisit un sommet, il en reste un de moins à parcourir. Le nombre de sommets choisis successivement sans remise sera donc $(n-1)\times(n-2)\times\ldots\times 2\times 1 \;=\; (n-1)!$
La complexité est donc en $O\big((n-1)!\big)$
En imaginant qu'on ait 26 villes (rien du tout), et qu'on parte de l'une d'entre elles, il y aurait donc $25!$ chemins différents à tester pour calculer le chemin réalisant la longueur minimum.

$$25! = 15\,511\,210\,043\,330\,985\,984\,000\,000 \;\approx\; 1.55\times 10^{25}$$

En imaginant que le traitement relatif à la recherche et au traitement d'un chemin prenne $1\mu s$, je vous laisse faire les calculs, mais il faudrait 492 milliards d'année. (l'âge estimé de l'univers tel que nous le connaissons est de 13 milliards d'année).

Nous pouvons tous convenir du fait que ça n'est pas une approche raisonnable.

2. Métaheuristiques

Il peut être suffisant de ne trouver qu'une solution suffisamment approchée de la solution optimale. Les algorithmes réalisant ceci sont appelés des **métaheuristiques**.

Beaucoup de métaheuristiques sont basés sur des comportements observés dans la nature. On les appelle des algorithmes **bio-inspirés**.

Nous allons nous intéresser à une catégorie bien particulière de ces algorithmes les algorithmes à colonie de

fourmis, ou ACO (Ant Colony Optimization)

3. Les fourmis

Rechercher de la nourriture est un processus qui occupe l'évolution des espèces depuis la nuit des temps. Une espèce ne peut pas survivre si l'énergie apportée par sa nourriture est inférieure à l'énergie nécessaire à la trouver, d'où la nécessité induite d'optimiser ce processus. Voici comment se débrouillent les fourmis, espèce apparue durant le crétacé, il y a quelques 100 millions d'années, dont les humains connaissent actuellement 14000 espèces. Autant vous dire qu'elles ont eu le temps d'affûter leur technique.

Lorsque les fourmis quittent leur nid pour chercher de la nourriture, elles commencent par explorer leur environnement de manière **aléatoire**.

- Les fourmis ne savent pas à l'avance où se trouve la nourriture.
- Elles parcourent leur territoire en laissant une petite quantité de phéromones sur le chemin qu'elles empruntent. Les phéromones ne restent pas indéfiniment dans l'environnement. Elles **s'évaporent** progressivement avec le temps.

Lorsqu'une fourmi trouve une source de nourriture, elle retourne au nid en laissant une traînée de phéromones plus concentrée sur le chemin du retour. Ce chemin devient une indication chimique pour les autres fourmis.

Les autres fourmis, en rencontrant un chemin marqué par des phéromones, sont attirées et plus susceptibles de le suivre. Cependant, **leur choix n'est pas déterministe : elles ont une probabilité plus élevée de suivre un chemin où la concentration de phéromones est forte.**
Elles laissent elles-mêmes des phéromones en empruntant ce chemin, **renforçant encore son attractivité**.

Une conséquence observée de ce processus est qu'au fil du temps, les chemins plus courts vers la nourriture deviennent plus **attractifs**.

- Les fourmis qui prennent un chemin plus court reviennent plus rapidement au nid et augmentent la **concentration de phéromones** sur ce chemin.
- Les phéromones des chemins plus longs s'**évaporent** plus facilement, ce qui réduit leur **attractivité**.

C'est cette dynamique entre renforcement et évaporation qui permet d'aboutir à une très bonne approximation d'un chemin minimum.

4. Les ACO (Ant Colony Optimization)

Nous allons directement nous inspirer des fourmis pour résoudre approximativement le TSP. On va envoyer une par une toutes les fourmis d'une colonie de fourmi faire le tour des sommets du graphe aléatoirement selon ce principe, et on ne gardera que le chemin le plus fréquenté.
Nous nous permettrons quelques adaptations, mais rien de bien sorcier.

L'approche suivante part du comportement de la fourmi à un endroit précis, et on élargit progressivement jusqu'au comportement de la colonie.

4.1. Une fourmi choisit un prochain sommet

Au niveau d'une fourmi, voici ce qu'il va se passer:

- Quand elle est sur un sommet du graphe, elle va aller vers un sommet **qu'elle n'a pas déjà exploré**.
- Comme toute bonne fourmi, elle va choisir un sommet suivant au hasard, **selon une loi bien**

déterminée en fonction de l'attractivité des arêtes.

Voici comment nous allons définir l'**attractivité** d'une arête du graphe. Nous allons faire intervenir 2 notions:

- La visibilité du sommet suivant
- La quantité totale de phéromones résidant sur l'arête

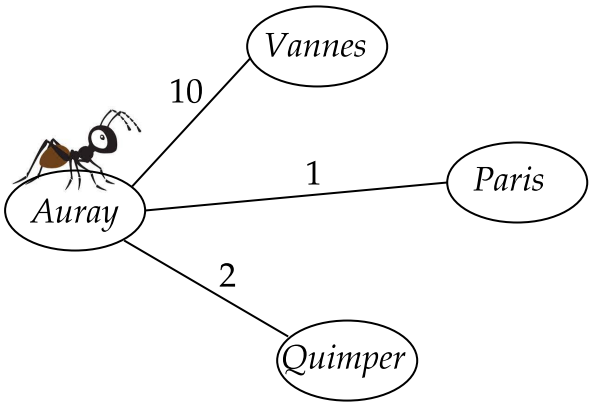
La visibilité est définie en tant qu'inverse de la distance portée par l'arête. Ainsi, si la distance est petite, la visibilité sera grande, et inversement. Ici, nous ajoutons une capacité à la fourmi, car dans la nature, elle ne se base que sur ses phéromones.

On définira donc l'attractivité de l'arête $(i,\ j)$ ainsi:

$$attractivite_{i,j} = pheromones_{i,j}^{\alpha} \times\ visibilite_{i,j}^{\beta}$$

On remarque l'intervention de deux paramètres supplémentaires, α et β. Pas de panique, il s'agit de paramètres d'ajustement du modèle à l'interprétation simple. En choisissant leurs valeurs respectives, on peut choisir d'accorder plus d'importance à la visibilité, ou à la quantité de phéromones. Dans nos test, on utilisera $\alpha = 1$ et $\beta = 2$. On peut dès lors se dire qu'on accordera deux fois plus d'importance à la la visibilité immédiate qu'aux phéromones. En général, on appelle ces paramètres d'ajustement des modèles des hyperparamètres, nous y reviendrons.

La fourmi doit choisir un sommet suivant parmi tous ses sommets inexplorés. Elle le fait aléatoirement en se basant sur l'attractivité des arêtes.



Ainsi, on peut exhiber une loi de probabilité pour le choix aléatoire de la fourmi:

Sommets inexplorés	Vannes	Paris	Quimper
Probabilité	$\frac{10}{13}$	$\frac{1}{13}$	$\frac{2}{13}$

Si on a une liste de sommets S et une liste de leur probabilité correspondante P, on peut choisir aléatoirement un sommet avec l'instruction python `random.choices(S, weight=P)`

4.1.1. Implémentations préliminaire:

Voici notre entête de fichier. La création d'alias de types sert surtout à conserver un code lisible et à ne pas se mélanger les pinceaux dans ce qu'on manipule. On assimilera un graphe à sa matrice d'adjacence.

On ne donne pas d'étiquette aux sommets, on se sert des indices de la matrice pour les identifier.

```
import random
type matrice[T] = list[liste[T]]
type chemin = list[int]
```

Implémentez les fonctions suivantes qui seront utiles dès le départ pour les doctests.

On initialise la matrice des phéromones avec de petites valeurs. En effet, si on l'initialise à 0, l'attractivité sera toujours nulle étant donnée sa définition.

```
def init_pheromones(nb_sommets: int,
                    tau_initial: float=1.0) -> matrice[float]:
    """Initialise la matrice des phéromones à tau_initial.
    tau_initial est la contribution en phéromones de chaque arête"""
```

On créé aussi la matrice de visibilité de chaque arête, qui elle ne changera jamais.

```
def calculer_visibilite(graphe: matrice[float]) -> matrice[float]:
    """Renvoie la matrice de visibilité.
    La visibilité de chaque arête est l'inverse de la distance
    """
```

De plus, on donne le graphe exemple suivant:

```
def get_graphe_exemple() -> matrice[float]:
    return [
        [0, 2, 9, 10],
        [2, 0, 6, 4],
        [9, 6, 0, 8],
        [10, 4, 8, 0]
    ]
```

4.1.2. Implémentation

Ecrire la fonction suivante qui détermine le prochain sommet que choisira la fourmi située au sommet courant:

```
def prochain_sommet(pheromones: matrice[float],
                    visibilite: matrice[float],
                    sommet_courant: int,
                    inexplore: list[int],
                    alpha: float, beta: float) -> int:
    """Renvoit le prochain sommet au hasard en fonction de l attractivité des chemins.
    Notre fourmi est quand même plus intelligente qu'une fourmi classique, elle ne va pas aux
    endroits déjà visités.

    >>> random.seed(54)
    >>> g = get_graphe_exemple()
    >>> n = len(g)
    >>> prochain_sommet(init_pheromones(n), calculer_visibilite(g), 0, list(range(n)), 1, 2)
    1
```

```
    """
```

4.2. Une fourmi parcourt le graphe entier

Ecrire la fonction suivante qui permet de renvoyer le cycle hamiltonien parcouru par une fourmi dans le graphe complet en partant d'un sommet aléatoire.
La fonction renvoie le chemin emprunté, ainsi que la longueur de ce chemin.

```
def parcours_fourmi( graphe: matrice[float],
                     pheromones: matrice[float],
                     visibilite: matrice[float],
                     alpha: float, beta: float) -> tuple[chemin, float]:
    """
    Simule le parcours d'une seule fourmi choisissant au hasard un sommet de départ inexploré.
    Renvoie le chemin ainsi que la longueur de ce chemin.

    >>> random.seed(54)
    >>> g = get_graphe_exemple()
    >>> n = len(g)
    >>> parcours_fourmi(g, init_pheromones(n), calculer_visibilite(g), 1, 2)
    ([1, 0, 2, 3, 1], 23)
    """
```

4.3. Chaque fourmi de la colonie parcourt le graphe entier

Ecrire la fonction suivante qui envoie les fourmis les unes après les autres parcourir le graphe, et qui renvoie la liste des chemins ainsi que la liste de leurs longueurs.

```
def simuler_colonie(graphe: matrice[float],
                    pheromones: matrice[float],
                    visibilite: matrice[float],
                    nb_fourmis: int,
                    alpha: float, beta: float)
                    -> tuple[list[chemin], list[float]]:
    """
    Simule le parcours de nb_fourmis fourmis d'une colonie lancées l'une après l'autre
    dans le graphe. Renvoie la liste des cyles obtenus, ainsi que la liste des distances
    correspondantes

    >>> random.seed(54)
    >>> g = get_graphe_exemple()
    >>> n = len(g)
    >>> simuler_colonie(g, init_pheromones(n), calculer_visibilite(g), 5, 1, 2)
    ([[1, 0, 2, 3, 1], [3, 1, 0, 2, 3], [3, 2, 1, 0, 3], [1, 0, 3, 2, 1], [0, 1, 3, 2, 0]], [23, 23,
26, 26, 23])
    """
```

4.4. Mise à jour des phéromones

Pour l'instant, les fourmis n'ont pas déposé de phéromones. C'est là où nous prenons des libertés avec la réalité de la vie des fourmis. Si nous déposons des phéromones trop tôt, certains chemins non optimaux seront trop tôt renforcés avant qu'une solution bien plus optimale ne soit elle même découverte, et ce renforcement nous éloignera lui-même de la découvrir un jour.
On va alors s'inquiéter du mécanisme de phéromones uniquement après que la colonie entière ait parcouru le graphe, afin de "bien laisser le hasard s'installer".

L'évaporation des phéromones

ρ est un autre hyperparamètre du modèle. Il représente le taux d'évaporation des phéromones. Avant toute chose, on commence donc par diminuer toutes les phéromones de ce taux pour simuler l'évaporation des phéromones.

La contribution en phéromones

Une fois l'évaporation effectuée, on va considérer tous les chemins empruntés par les fourmis, et pour chaque chemin c de longueur d, on calcule la quantité $\frac{Q}{d}$ qu'on appelle la contribution en phéromone au chemin. On ajoutera donc cette quantité à toutes les arêtes du chemin c.
Ainsi, les chemins les plus courts auront beaucoup plus de phéromone ajoutée que les chemins les plus longs.
On peut voir Q comme la contribution de base en phéromones

Implémentation

```
def update_pheromones(  pheromones: matrice[float],
                        chemins: list[chemin],
                        longueurs_chemins: list[float],
                        rho: float, Q: float):
    """
    Met à jour les phéromones sur les arêtes. Dans un premier temps, toutes les phéromones
    diminuent du facteur rho. Dans un deuxième temps, pour chaque chemin de longueur d, toutes
    les arête augmentent en phéromone de Q/d.

    >>> g = get_graphe_exemple()
    >>> chemins = [[1,0,2,3,1], [3,1,0,2,3], [3,2,1,0,3], [1,0,3,2,1], [0,1,3,2,0]]
    >>> distances = [23, 23, 26, 26, 23]
    >>> p = init_pheromones(len(g))
    >>> update_pheromones(p, chemins,  distances, 0.5, 100)
    >>> [round(val) for ligne in p for val in ligne]
    [0, 21, 14, 8, 21, 0, 8, 14, 14, 8, 0, 21, 8, 14, 21, 0]
    """
```

4.5. ACO

On a maintenant tout pour envoyer plusieurs colonies à la suite en mettant à jour les phéromones à chaque fois.

C'est ici qu'on pourra créer la matrice de visibilité ainsi que la matrice de phéromones. On initialisera la matrice des phéromones à celle de l'hyperparamètre τ_0. En effet, si on part sur une base de phéromones nulle, l'attractivité sera toujours nulle d'après sa définition. On prendra τ_0 petit

Ensuite, on répète `nb_colonies` fois le processus:
- Simulation de colonie
- Mise à jour des phéromones .
Ce faisant, on garde la trace du plus court chemin rencontré pendant la procession.

4.5.1. Implémentation

```python
def ant_colony_optimization(graphe: matrice[float],
                            nb_fourmis: int,
                            alpha: float, beta: float,
                            rho: float, Q: float, tau_zero: float=1.0
                            nb_colonies: int) -> tuple[chemin, float]:
    """Algorithme de colonies de fourmis pour résoudre le TSP.
    Il s'agit de faire passer plusieurs colonies dans le graphe.
    Entre chaque passage de colonie, on met à jour les phéromones.
    L'objectif est de renvoyer le meilleur chemin et la meilleure distance"""
```

4.5.2. Test sur de vraies données

Berlin52 est une instance bien connue du TSP issue de la TSPLIB, une bibliothèque de benchmarks pour les algorithmes d'optimisation combinatoire. Elle représente 52 villes dans la périphérie de Berlin.

il faut **`uv add tsplib95`** et importer **`tsplib95`** dans le code.

La fonction suivante génère la matrice d'adjacence à partir du fichier `berlin52.tsp` qui vous a été fourni. (Notez que python va chercher les fichiers relativement au répertoire duquel il a été lancé)

```python
def get_matrice_berlin52() -> matrice[float]:
    """
    Retourne la matrice d'adjacence de berlin52 dont on sait que le plus
    petit chemin fait 7542 unités
    """
    probleme = tsplib95.load('berlin52.tsp')
    sommets = list(probleme.get_nodes())
    return [[problem.get_weight(s1, s2) for s2 in sommets] for s1 in sommets]
```

On étendra la section main ainsi:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()

    random.seed(42)

    graphe: matrice[float] = get_matrice_berlin52()
    nb_fourmis  = 100
    alpha       = 1.06
    beta        = 2.29
    rho         = 0.53
    q           = 309
    nb_colonies = 1000

    # Complétez pour tester l'ACO sur berlin52
```

5. Amélioration

Dans notre approche, on calcule l'attractivité à chaque fois qu'une fourmi choisit un prochain sommet. En réalité, ça n'est pas ici nécessaire. Pourquoi? Quels changements faudrait-il opérer pour ne plus avoir à la faire à chaque fois? Apportez les modifications nécessaires à votre programme.