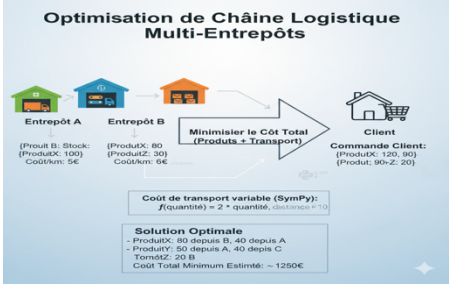
L'École Normale Supérieure de l'Enseignement Technique de Mohammedia

Filières Ingénieure

COMPÉTENCES NUMÉRIQUES ET INFORMATIQUE

DEVOIR LIBRE À RENDRE

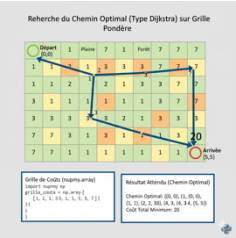
Sujet N°1. 🚚 Optimisation de Chaîne Logistique Multi-Entrepôts



Objectif : Trouver le plan d'approvisionnement le moins cher pour une commande client.

- **Étape 1 : Modélisation des Données (Structures dict)**
  - Créez une structure de données (probablement un dict de dict) pour représenter les entrepôts, leurs stocks par produit, et leur coût de transport de base.
  - Créez un dict pour représenter la commande du client (ex: {'ProduitX': 90, ...}).
- **Étape 2 : Coûts Variables (Introduction à Sympy)**
  - Demandez à l'utilisateur (ou définissez en dur) une fonction de coût variable sous forme de str (ex: "2 * quantite + 5").
  - Utilisez sympy.symbols pour définir la variable quantite.
  - Utilisez sympy.sympify pour convertir la str en une expression symbolique.
  - Utilisez sympy.lambdify pour transformer cette expression en une fonction Python standard, prête à être utilisée pour des calculs numériques.
- **Étape 3 : Algorithme d'Optimisation "Glouton" (Fonctions, for, if)**
  - Créez une fonction trouver_solution(commande, entrepots).
  - Dans cette fonction, initialisez un plan_approvisionnement vide (ex: dict) et un cout_total = 0.
  - *Logique de parcours :* Pour chaque produit de la commande :
    1. Triez les entrepôts qui ont ce produit en stock (par ex: du moins cher au plus cher en coût de transport).
    2. Itérez sur ces entrepôts triés.
    3. Prenez le maximum de quantité possible de l'entrepôt actuel, mettez à jour le stock, et ajoutez cette action au plan_approvisionnement.
    4. Arrêtez pour ce produit dès que la quantité commandée est atteinte.
- **Étape 4 : Calcul Final et Affichage (Intégration, print)**
  - Pendant l'étape 3, à chaque fois que vous ajoutez une action au plan, calculez le coût (coût produit + coût transport) en utilisant votre fonction "lambdifiée" de l'étape 2.
  - Affichez le plan_approvisionnement détaillé (ex: "Prendre 80 de X chez B, ...") et le cout_total calculé.

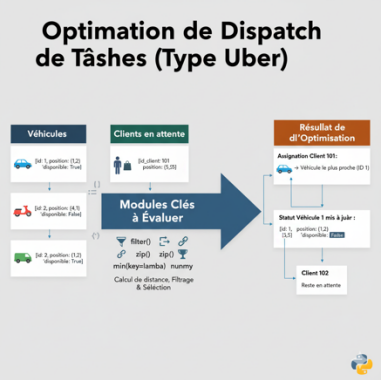
Sujet N°2. 🗺️ Recherche du Chemin Optimal (Type Dijkstra) sur Grille Pondérée



Objectif : Trouver le chemin au coût cumulé le plus faible entre deux points d'une matrice.

- **Étape 1 : Modélisation de la Grille (Numpy)**
  - Créez la grille des coûts en utilisant un numpy.array 2D.
  - Définissez les coordonnées de DEPART et ARRIVEE (ex: (0, 0) et (5, 5)).
- **Étape 2 : Initialisation de l'Algorithme (Structures de Données)**
  - Créez une matrice distances (un np.array de même taille que la grille) initialisée à +inf partout, sauf à la case DEPART (initialisée à 0).
  - Créez un set visites pour stocker les coordonnées des nœuds déjà traités.
  - Créez une list file_priorite contenant initialement (0, DEPART).
  - *Bonus :* Créez un dict predecesseurs pour stocker le chemin (ex: {(1,1): (0,1)}).
- **Étape 3 : Boucle Principale de Dijkstra (while, for, if)**
  - Créez une boucle while qui s'exécute tant que file_priorite n'est pas vide.
  - Dans la boucle :
    1. Triez la file_priorite et extrayez le nœud avec le coût le plus bas (c'est le nœud actuel).
    2. Si actuel est dans visites, continuez (sautez cette itération).
    3. Ajoutez actuel à visites.
    4. Si actuel est ARRIVEE, arrêtez la boucle (break).
    5. *Parcours des voisins :* Pour chaque voisin valide de actuel :
       - Calculez le nouveau_cout = distances[actuel] + grille_couts[voisin].
       - Si nouveau_cout < distances[voisin] :
         - Mettez à jour distances[voisin] = nouveau_cout.
         - Ajoutez (nouveau_cout, voisin) à file_priorite.
         - Mettez à jour predecesseurs[voisin] = actuel.
- **Étape 4 : Reconstruction du Chemin et Affichage (while, list, print)**
  - Affichez le coût total, qui est distances[ARRIVEE].
  - En partant de ARRIVEE, remontez le chemin en utilisant le dict predecesseurs jusqu'à tomber sur DEPART. Stockez ce chemin dans une list et affichez-la.

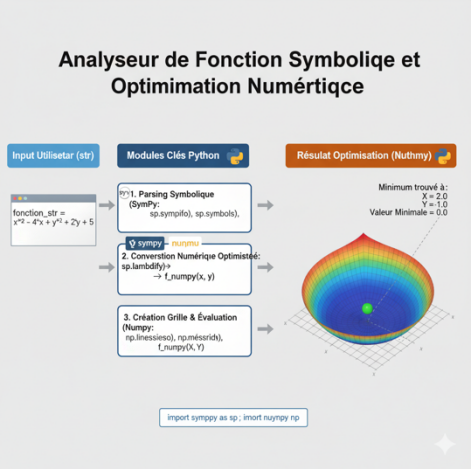
Sujet N°3. 🤖 Optimisation de Dispatch de Tâches (Type Uber)



Objectif : Assigner des clients aux véhicules disponibles les plus proches.

- **Étape 1 : Modélisation des Données (List, Dict, Numpy)**
  - Créez une list de dict pour les vehicules, chaque dict contenant 'id', 'position' (un np.array) et 'disponible' (un bool).
  - Créez une list de dict pour les clients_en_attente, avec 'id_client' et 'position'.
- **Étape 2 : Fonction Utilitaire (Fonction, Numpy)**
  - Créez une fonction calculer_distance(pos1, pos2) qui prend deux np.array et retourne la distance (euclidienne ou Manhattan).
- **Étape 3 : Boucle de Dispatch (Fonctions Avancées, for, min)**
  - Créez une list assignations vide.
  - Itérez sur chaque client dans clients_en_attente (boucle for).
  - Dans la boucle :
    1. **Filtrage :** Utilisez filter() (ou une liste en compréhension) pour créer une list vehicules_dispos à partir de la liste vehicules principale.
    2. Si vehicules_dispos est vide, break (on ne peut plus assigner).
    3. **Calcul des distances :** Utilisez map() pour appliquer calculer_distance entre le client et *chaque* véhicule dans vehicules_dispos.
    4. **Association :** Utilisez zip() pour lier les distances calculées aux vehicules_dispos.
    5. **Optimisation :** Utilisez min(..., key=lambda ...) sur la liste zippée pour trouver le (distance_min, vehicule_elu).
    6. **Mise à jour :**
       - Affichez l'assignation (ex: "Client 101 -> Véhicule 3").
       - Ajoutez-la à la liste assignations.
       - **Important :** Modifiez le statut du vehicule_elu dans la liste *principale* vehicules pour le passer à disponible: False.
- **Étape 4 : Affichage des Résultats (print)**
  - Affichez la liste finale des assignations.
  - Affichez l'état final de la liste vehicules (montrant qui n'est plus disponible).

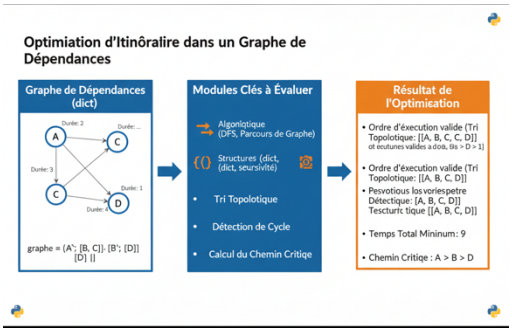
Sujet N°4. 📈 Analyseur de Fonction Symbolique et Optimisation Numérique



Objectif : Trouver le minimum numérique d'une fonction mathématique donnée sous forme de texte.

- **Étape 1 : Entrée Utilisateur et Symbolique (Input, Sympy)**
  - Demandez à l'utilisateur une fonction str (ex: "x**2 + sin(y)").
  - Utilisez sympy.symbols pour définir x et y.
  - Utilisez sympy.sympify pour convertir la str en expression sympy.
- **Étape 2 : Conversion Symbolique vers Numérique (Sympy, lambdify)**
  - Utilisez sympy.lambdify([x, y], expression_sympy) pour créer une f_numpy rapide, prête pour numpy.
- **Étape 3 : Création de la Grille d'Évaluation (Numpy)**
  - Définissez des domaines (ex: de -5 à 5) pour x et y.
  - Créez des vecteurs x_vals = np.linspace(-5, 5, 100) et y_vals = np.linspace(-5, 5, 100).
  - Utilisez np.meshgrid(x_vals, y_vals) pour obtenir deux matrices 2D : X_grid et Y_grid.
- **Étape 4 : Évaluation et Optimisation (Numpy)**
  - Appliquez votre fonction à la grille : Z_results = f_numpy(X_grid, Y_grid).
  - Trouvez la valeur minimale : min_val = np.min(Z_results).
  - Trouvez les *indices* du minimum : indices = np.unravel_index(np.argmin(Z_results), Z_results.shape).
  - Utilisez ces indices pour trouver les valeurs x et y correspondantes :
    - x_min = x_vals[indices[1]]
    - y_min = y_vals[indices[0]]
- **Étape 5 : Affichage (print)**
  - Affichez les résultats : "Minimum trouvé à (x, y) = (x_min, y_min) avec une valeur de min_val."

Sujet N°5. 🧬 Optimisation d'Itinéraire (Tri Topologique)



Objectif : Trouver un ordre d'exécution valide pour des tâches ayant des dépendances.

- **Étape 1 : Modélisation du Graphe (Structures dict)**
  - Représentez le graphe des dépendances à l'aide d'un dict (liste d'adjacence).
  - Exemple : graphe = {'A': ['B', 'C'], 'B': ['D'], 'C': ['D'], 'D': []} (A doit être fait avant B et C, etc.)
- **Étape 2 : Structures de Parcours (Structures de Données)**
  - Créez un set visites (pour suivre les nœuds déjà traités).
  - Créez une list ordre_topologique (pour stocker le résultat).
- **Étape 3 : Algorithme de Parcours (DFS) (Fonction, Récursivité, if)**
  - Définissez une fonction (récursive) parcourir(noeud) :
    1. Ajoutez noeud à visites.
    2. *Parcours des voisins :* Pour chaque voisin dans graphe[noeud] :
       - Si voisin n'est pas dans visites :
         - Appelez récursivement parcourir(voisin).
    3. **Crucial :** *Après* avoir exploré tous les voisins, ajoutez noeud au *début* de la liste ordre_topologique.
- **Étape 4 : Lancement du Tri**
  - Itérez sur chaque clé (noeud) de votre graphe.
  - Si le noeud n'est pas dans visites, appelez parcourir(noeud).
  - (Cette boucle for garantit que tous les nœuds, y compris ceux sans dépendances, sont traités).
- **Étape 5 : Affichage des Résultats**
  - Affichez la ordre_topologique finale.
  - *Bonus :* Si vous aviez des durées, parcourez cet ordre et calculez un temps total.