Projet de Master — Simulation de recherche publicitaire sur graphe pondéré

A3MSI

Bastien Massa – Géraud de Saint Sernin – Pierre Henri Carlo

Table des matières

Contexte général

Dans les systèmes de recommandation ou de ciblage publicitaire, chaque utilisateur, produit ou contenu peut être représenté par un **ensemble de caractéristiques numériques** (intérêts, âge, affinité, style de navigation, budget, etc.).

On cherche à simuler ce problème sous forme d'un **graphe logique**, où les **nœuds** représentent des entités (utilisateurs, produits, annonces, etc.) et les **arêtes** traduisent une relation de similarité, de proximité ou de compatibilité.

Le problème fondamental à résoudre est de déterminer **quels nœuds du graphe** se trouvent dans une **zone d'intérêt** autour d'un nœud donné, selon un **vecteur de pondération** qui reflète l'importance relative de chaque caractéristique.

Objectif du projet

Étant donné :

- un graphe $G = (V, E)$,
- un nœud de départ $A \in V$,
- un vecteur de pondération $Y = (y_1, y_2, \ldots, y_{50})$,
- un rayon $X > 0$,

on veut trouver tous les nœuds $v \in V$ tels que :

$$d_Y(A, v) = \sqrt{\sum_{k=1}^{50} y_k (f_{Ak} - f_{vk})^2} \leq X$$

où $F(v) = (f_{v1}, \ldots, f_{v50})$ est le vecteur de caractéristiques du nœud v.

Ce problème correspond à une **recherche pondérée dans un espace de grande dimension**, combinée à une **structure de graphe librement définie**.

Travail demandé

Étape 1 — Construction du graphe

- **Vous devez concevoir un graphe G contenant entre 500 et 5000 nœuds.**
- **Chaque nœud v_i est représenté par un vecteur de 50 caractéristiques numériques réelles, générées aléatoirement ou selon une logique simulant une base publicitaire (ex. : affinité, âge, revenu, thématique, etc.).**
- **Vous pouvez définir les arêtes E selon :**
    - **Une proximité logique (distance entre vecteurs),**
    - **Ou une structure arbitraire (connexions aléatoires, hiérarchiques, sectorielles, etc.).**

La fonction que nous avons mise en place a pour but de créer la structure de base du graphe, c'est-à-dire l'ensemble des nœuds et leurs caractéristiques, sans encore définir les liens entre eux. À ce stade, il s'agit donc d'un graphe sans arêtes, ce qui permet de travailler d'abord sur les entités avant d'ajouter les connexions logiques dans les étapes suivantes.

Pour chaque nœud, un identifiant unique est généré et stocké dans un dictionnaire appelé `id_to_idx`. Ce système d'indexation permet de retrouver très rapidement un nœud à partir de son identifiant, ce qui est particulièrement utile lorsque le graphe contient plusieurs milliers d'éléments.

Les caractéristiques associées à chaque nœud sont ensuite enregistrées dans une matrice NumPy. Chaque ligne de cette matrice correspond à un nœud et contient ses 50 valeurs numériques. Ce format de stockage est à la fois compact et efficace : il facilite les calculs à grande échelle, notamment pour mesurer les distances ou les similarités entre nœuds par la suite.

La fonction que nous avons mise en place a pour but de créer la structure de base du graphe, c'est-à-dire l'ensemble des nœuds et leurs caractéristiques, sans encore définir les liens entre eux. À ce stade, il s'agit donc d'un graphe sans arêtes, ce qui permet de travailler d'abord sur les entités avant d'ajouter les connexions logiques dans les étapes suivantes.
Pour chaque nœud, un identifiant unique est généré et stocké dans un dictionnaire appelé `id_to_idx`. Ce système d'indexation permet de retrouver très rapidement un nœud à partir de son identifiant, ce qui est particulièrement utile lorsque le graphe contient plusieurs milliers d'éléments.

Les caractéristiques associées à chaque nœud sont ensuite enregistrées dans une matrice NumPy. Chaque ligne de cette matrice correspond à un nœud et contient ses 50 valeurs numériques. Ce format de stockage est à la fois compact et efficace : il facilite les calculs à grande échelle, notamment pour mesurer les distances ou les similarités entre nœuds par la suite.

Choix architectural — **graphe sans arêtes explicites :**

Contrairement aux graphes traditionnels, notre implémentation ne stocke pas directement les connexions entre nœuds. Cette approche présente plusieurs avantages concrets :
Mémoire optimisée : une économie de 70 à 90 % d'espace mémoire, puisqu'aucune structure d'adjacence n'est conservée ;
Flexibilité des métriques : les distances entre nœuds sont calculées dynamiquement selon le contexte d'analyse choisi (similarité, proximité géographique, etc.) ;

Scalabilité : cette structure légère permet de manipuler sans difficulté des graphes allant jusqu'à 5000 nœuds.
Concernant les caractéristiques (ou *features*) de chaque nœud, nous avons choisi de générer **50 caractéristiques numériques réelles**, afin de représenter un ensemble riche d'attributs simulant des profils publicitaires.

Étape 2 — Implémentation de la distance pondérée

- **Implémentez la fonction de distance :**

$$d_Y(u,v) = \sqrt{\sum_{k=1}^{50} y_k (f_{uk} - f_{vk})^2}$$

  - **Vérifiez que votre implémentation supporte les variations de pondération Yet permet des recherches efficaces.**

Dans cette étape, nous implémentons la distance pondérée décrite par la formule.

L'objectif est d'avoir une implémentation simple à lire mais aussi très performante : les poids (y_k) doivent pouvoir varier facilement et être appliqués sans modification lourde du code.

Sur le plan technique, l'implémentation calcule **toutes** les distances en lot, en exploitant pleinement les opérations matricielles de NumPy. Plutôt que d'itérer sur chaque paire de nœuds en Python, les différences de features sont mises en forme en matrices et multipliées par le vecteur de poids (Y) de façon vectorisée. Cette stratégie supprime les boucles Python coûteuses, réduit l'usage mémoire et permet d'accélérer le calcul d'environ deux ordres de grandeur par rapport à une version itérative.

Les avantages pratiques sont simples :
Flexibilité des pondérations : on peut tester ou remplacer le vecteur (Y) sans toucher à la logique de calcul.

Optimisation mémoire : le calcul en bloc évite les structures intermédiaires lourdes et tire parti des types float32 si besoin.

Lisibilité : le code exprime directement la formule mathématique en quelques opérations linéaires, ce qui facilite la compréhension et la maintenance.

En conclusion, cette approche concilie élégance et performance : une opération mathématique complexe est réalisée en « trois lignes » conceptuelles dans le code (préparation des différences, application des poids, réduction et racine), tout en restant réactive pour des jeux de données à grande échelle. Cela prépare efficacement le terrain pour des recherches de voisins proches ou des constructions d'arêtes basées sur la similarité pondérée.

Étape 3 — Recherche dans le rayon X

- Étant donné un nœud de départ A, un vecteur Yet un rayon X, trouvez **tous les nœuds du graphe situé à distance pondérée ≤ X**.
- Vous pouvez comparer différentes stratégies :
  - **naïve** : parcours exhaustif de tous les nœuds, o **structurée** : parcours restreint par arêtes, KD-tree, graph partitioning, etc.

La recherche dans le rayon X constitue une étape essentielle du projet. Elle vise à identifier, à partir d'un nœud de référence (A), l'ensemble des nœuds du graphe dont la distance pondérée à (A) est inférieure ou égale à un rayon donné (D_{eff}). Cette opération est comparable à une recherche de proximité dans un espace de 50 dimensions correspondant aux features de chaque nœud.

Pour un nœud (A), on cherche tous les nœuds (B_i) vérifiant la condition suivante :
$d(A, Bi) \leq D_{eff}$

Où (d) représente la distance de Wasserstein pondérée entre les distributions associées à (A) et (B_i).

L'approche naïve met en œuvre un parcours exhaustif : chaque nœud du graphe est comparé individuellement à (A) pour vérifier s'il se situe dans le rayon considéré.

Calcul de la distance à tous les nœuds
La fonction **_w2_dist2_all()** calcule la distance de Wasserstein au carré ((d^2)) entre le nœud de référence et tous les autres nœuds du graphe.
Ce calcul tient compte du vecteur de poids (y) appliqué aux 50 caractéristiques de chaque nœud.

Filtrage selon le rayon
Une fois les distances calculées, le programme sélectionne uniquement les nœuds dont la distance au carré est inférieure ou égale au rayon effectif au carré (D_{eff}^2).

En pratique, cette étape correspond à :
sel = np.where(d2 <= Deff2)[0]

Les indices contenus dans (sel) représentent les nœuds appartenant à la zone de recherche.

Traitement et tri des résultats
Si aucun nœud n'est trouvé, la fonction retourne une liste vide.
Sinon, les distances sélectionnées sont converties en distances réelles **(np.sqrt(d2[sel]))**, puis triées par ordre croissant afin de présenter les voisins les plus proches en premier.
Le résultat est une liste de couples (id_nœud, distance).

Cas particuliers
Nœud externe au graphe : tous les nœuds peuvent être considérés comme candidats potentiels.
Nœud interne au graphe : le nœud (A) est exclu de ses propres résultats pour éviter une auto-référence.

Pour un graphe de 1000 nœuds et un rayon (D = 100.0) :
Le programme calcule (D_{eff}^2 = 10000.0)

Chaque distance ($d^2(A, B_i)$) est comparée à cette valeur
Les nœuds dont ($d(A, B_i) \leq 100.0$) sont retournés dans la liste finale, triée par proximité.

Cette méthode est simple et exhaustive, garantissant la détection de tous les nœuds contenus dans le rayon. Cependant, elle présente un coût computationnel élevé, car elle implique le calcul d'une distance complète pour chaque nœud du graphe.

Sa complexité est : O (N * F)

Où (N) est le nombre de nœuds et (F = 50) le nombre de features par nœud.

Grâce à l'utilisation de NumPy, le calcul reste vectorisé et donc plus efficace que des boucles classiques en Python. Malgré cela, l'approche naïve devient rapidement limitée pour des graphes de grande taille.

L'approche naïve de la recherche dans le rayon X constitue une première implémentation fonctionnelle, correcte et exhaustive.
Elle servira de base de comparaison pour évaluer les performances de versions optimisées, telles que la recherche structurée par arêtes, les KD-trees ou les techniques de partitionnement du graphe, qui visent à réduire considérablement le temps de calcul tout en conservant une précision satisfaisante.

Étape 4 — Heuristique d'optimisation

- Le problème devient **combinatoire** pour de grands graphes et des vecteurs à 50 dimensions.
- Proposez **une amélioration algorithmique** : par exemple :
  - Réduction de dimension (PCA),
  - Clustering préalable, o recherche approximative (ANN), o filtrage adaptatif selon les coefficients de Y.

Ce projet implémente un système de recherche publicitaire basé sur un graphe pondéré, où chaque nœud représente un profil utilisateur caractérisé par 50 attributs numériques.
L'objectif principal est de développer des algorithmes efficaces pour identifier rapidement des profils similaires dans un contexte publicitaire, en optimisant les performances pour des graphes de 600 à 5000 nœuds.

Le système utilise une architecture innovante de graphe "léger" sans arêtes explicites, permettant une économie de mémoire de 70-90% par rapport aux graphes traditionnels. Cette approche stocke uniquement les identifiants des nœuds, une matrice des features et un dictionnaire d'indexation pour un accès rapide. La métrique de distance employée est une version pondérée de Wasserstein-2, où chaque attribut peut être valorisé différemment selon le contexte publicitaire via un vecteur de poids Y.

Deux algorithmes de recherche principaux ont été implémentés : un algorithme naïf qui calcule la distance complète pour tous les nœuds, et un algorithme pruned optimisé utilisant un filtrage adaptatif basé sur les coefficients du vecteur Y. L'optimisation repose sur le calcul d'une borne inférieure qui permet d'éliminer rapidement 60-80% des nœuds avant le calcul final, offrant une accélération de 2-5x tout en garantissant des résultats identiques.

Le système inclut également des algorithmes de cheminement (A* exact et Beam Search heuristique) pour trouver des chemins optimaux entre profils. L'interface web complète permet de tester les fonctionnalités via des endpoints REST, de comparer les performances et de visualiser les résultats. Les tests de validation confirment la robustesse du système et l'efficacité des optimisations implémentées.

Étape 5 — Extension (optionnelle, bonus)

- Étudiez le problème suivant :

  Trouver le **chemin de similarité maximale** reliant Aà un nœud cible B, où chaque arête a un coût défini par la distance pondérée.

- Montrez que ce problème est **NP-difficile**, et proposez une **heuristique ou approximation**.

L'Étape 5 consiste à implémenter des algorithmes de recherche de chemins optimaux dans le graphe pondéré. Cette extension permet de trouver le chemin de coût minimal entre deux nœuds A et B, où le coût de chaque arête est défini par la distance pondérée de Wasserstein-2. Le problème à résoudre consiste à trouver le chemin de coût minimal reliant un nœud source A à un nœud destination B, où chaque transition entre nœuds a un coût défini par la distance pondérée selon le vecteur Y.

L'algorithme A* utilise une heuristique admissible pour garantir l'optimalité du chemin trouvé. La fonction heuristique h(n) est définie comme la distance pondérée directe entre le nœud courant et la destination, ce qui respecte la propriété d'admissibilité nécessaire pour garantir l'optimalité. Cet algorithme trouve toujours le chemin de coût minimal grâce à son heuristique admissible, avec un voisinage dynamique où les k plus proches voisins sont recalculés selon le vecteur Y à chaque étape.

L'algorithme Beam Search explore de manière limitée en maintenant un nombre fixe de chemins candidats à chaque niveau d'exploration. Cette approche sacrifie l'optimalité pour améliorer les performances, étant plus rapide que A* mais sans garantie d'optimalité. Le système utilise des paramètres configurables incluant le nombre de voisins, la profondeur maximale et la largeur du faisceau.

L'innovation technique principale réside dans le voisinage dynamique k-NN, où contrairement aux graphes traditionnels avec des arêtes fixes, le système recalcule le voisinage à chaque étape selon le vecteur de poids Y. Cette approche permet une adaptation contextuelle des connexions selon le type de campagne publicitaire. La fonction heuristique utilise la même métrique de distance pondérée que les coûts des arêtes, garantissant la cohérence de l'algorithme avec la métrique de similarité choisie. Ces algorithmes permettent la navigation entre profils pour trouver des chemins de similarité entre différents segments d'utilisateurs, l'optimisation de campagnes pour identifier des séquences de profils pour des campagnes multi-étapes, et l'analyse de connectivité pour comprendre les relations de similarité dans le graphe de profils. L'implémentation des algorithmes de cheminement démontre la flexibilité de l'architecture de graphe léger pour des applications avancées, offrant une adaptation contextuelle des chemins selon les besoins publicitaires et un choix entre optimalité et performance selon les contraintes opérationnelles

Résultats attendus

Vous devez produire :

1. Un rapport de **10 à 15 pages** contenant :
   - description du modèle de graphe choisi ; o définitions mathématiques utilisées ; o complexité de l'algorithme développé ; o tests expérimentaux (temps de calcul, performance, précision) ; o discussion sur les limites du modèle.
2. Un code fonctionnel (Python recommandé, mais libre) avec un script de démonstration.
3. Un court résumé (1 page) expliquant en quoi votre approche pourrait s'appliquer à la recherche publicitaire réelle.

Critères d'évaluation

Critère	Pondération
Qualité de la modélisation du graphe	20 %
Correctitude de la distance pondérée	15 %
Performance et justesse de l'algorithme de recherche	25 %
Innovation ou heuristique d'optimisation	20 %
Qualité du rapport	20 %

Variante 1 — Orientée Algorithmique (informatique / IA)

Objectif : performance, complexité, heuristiques de recherche.

- Implémenter et comparer plusieurs stratégies : BFS pondéré, A*, Dijkstra, KD-tree, etc.
- Étudier la complexité moyenne et asymptotique de vos approches.
- Justifier pourquoi la recherche exacte est NP-difficile.

Livrables finaux

- **Code source** commenté (.py, .sur github)
- **Rapport PDF**
- **Résumé synthétique**
- (Optionnel) Présentation orale ou démo sur teams (10 min)