Ecole Mohammadia d'Ingénieurs

Résolution du Problème du Chemin le Plus Court

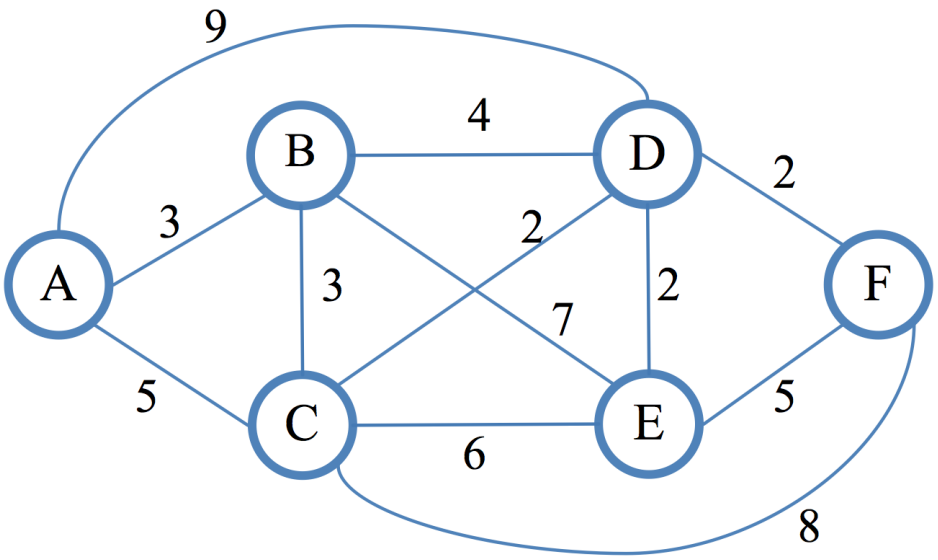
Des Algorithmes Classiques vs Graph Neural Networks (GNN)

https://github.com/anasmis/gnn-shortest-path.git

Réalisé par :
Anas AIT ALI

Encadré par :
Prof. EL KARKRI

Année académique : 2024-2025


9
4
B
D
2
3
2
A
3
F
7
2
5
C
E
5
6
8

Chapitre 1

Introduction

1.1 Contexte et Motivation

Le problème du chemin le plus court (*Shortest Path Problem*, SPP) est un défi central en informatique et en recherche opérationnelle. Formellement, il s'agit de trouver, dans un graphe pondéré orienté ou non, le chemin entre deux nœuds u et v minimisant la somme des poids des arêtes traversées. Ce problème trouve des applications critiques dans des domaines variés :

- **Réseaux de transport** : Les systèmes de navigation (ex : Waze) utilisent des variantes de l'algorithme A* pour calculer des itinéraires en intégrant le trafic en temps réel.
- **Routage Internet** : Les protocoles comme IS-IS s'appuient sur des extensions de Dijkstra pour déterminer les routes les plus courtes entre routeurs.
- **Logistique** : Les entreprises comme Amazon optimisent leurs livraisons en résolvant des SPP sur des graphes représentant des entrepôts et des clients.

Cependant, les méthodes traditionnelles présentent des limites :

- **Scalabilité** : Sur des graphes de grande taille (ex : réseaux sociaux), même l'algorithme de Dijkstra devient impraticable en raison de sa complexité.
- **Dynamisme** : Dans des environnements changeants (ex : poids d'arêtes variables), les algorithmes classiques nécessitent des recalculs complets, coûteux en ressources.
- **Généralisation** : Un algorithme conçu pour un graphe spécifique (ex : réseau routier marocain) ne peut pas s'adapter à un graphe structurellement différent (ex : réseau aérien) sans modifications.

Les *Graph Neural Networks* (GNN) émergent comme une alternative pro-

metteuse pour pallier ces limites. En apprenant des représentations vectorielles des nœuds et arêtes, les GNN peuvent :

- **Apprendre des heuristiques** : Généraliser à des graphes non vus pendant l'entraînement.
- **S'adapter dynamiquement** : Mettre à jour les prédictions sans recalcul complet grâce à des mécanismes d'attention (ex : GAT).
- **Intégrer des contraintes complexes** : Comme des coûts multi-objectifs (temps, distance, prix) via des architectures spécialisées.

L'objectif de ce projet est donc d'explorer cette synergie entre méthodes classiques et approches par apprentissage automatique, en quantifiant leurs avantages respectifs dans des scénarios concrets.

1.2 Objectifs du Projet

Ce projet vise à :

- Implémenter et comparer des algorithmes traditionnels (Dijkstra, A*, Bellman-Ford) sur des graphes synthétiques et réels.
- Proposer une solution basée sur les GNN pour résoudre le SPP, notamment dans des cas où les graphes sont dynamiques ou incomplets.
- Évaluer les avantages/inconvénients des deux approches (précision, temps d'exécution, scalabilité).

1.3 Structure du Document

Le rapport est organisé comme suit :

- **Chapitre 2** : Revue des algorithmes classiques et implémentation.
- **Chapitre 3** : Approche par GNN (architecture, entraînement, résultats).
- **Chapitre 4** : Comparaison critique et analyse des limites.
- **Conclusion** : Synthèse et perspectives futures.

Chapitre 2

Revue des algorithmes classiques et implémentation.

2.1 Algorithme de Dijkstra

2.1.1 Principe Fondamental

L'algorithme de Dijkstra, proposé par Edsger W. Dijkstra en 1959, résout le problème du chemin le plus court depuis un nœud source vers tous les autres nœuds dans un graphe pondéré **à poids strictement positifs**. Il repose sur le principe de **relâchement progressif** des distances.

2.1.2 Explication Simplifiée de l'Algorithme

L'algorithme de Dijkstra trouve le chemin le plus court dans un réseau (comme une carte routière) en procédant méthodiquement :

- **Étape 1** :
  - On part d'un point de départ
  - On donne la distance 0 à ce point
  - On met une distance infinie (∞) à tous les autres points
- **Étape 2** :
  - On visite le point le plus proche non encore exploré
  - Pour chacun de ses voisins, on calcule la distance depuis le départ
  - Si c'est plus court que ce qu'on avait avant, on met à jour la distance
- **Étape 3** :
  - On répète l'étape 2 jusqu'à ce que tous les points aient été visités
  - À la fin, on connaît le chemin le plus court vers chaque point

Important :
— Ne fonctionne qu'avec des distances positives
— Comme un GPS qui explorerait toutes les routes méthodiquement
— Garantit de toujours trouver le chemin optimal

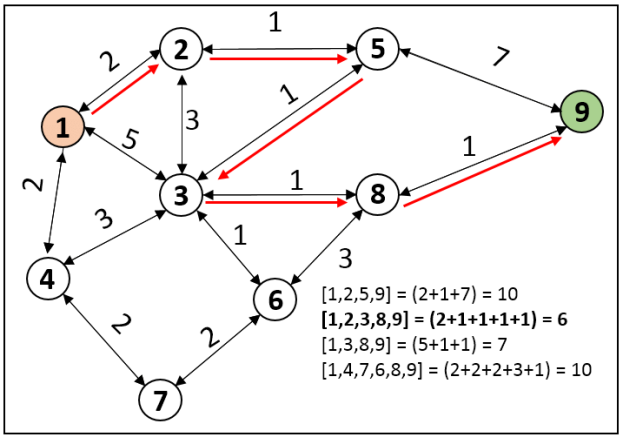


FIGURE 2.1 – Illustration du processus de Dijkstra

2.1.3 Formalisation Mathématique

Soit $G = (V, E)$ un graphe orienté avec :
— V : ensemble des nœuds
— E : ensemble des arêtes
— $w(u, v)$: poids de l'arête $(u, v) \in E$
On maintient :
— $d[v]$: distance minimale temporaire du nœud source s à v
— Q : file de priorité min basée sur $d[v]$

2.1.4 Pseudo-code

Algorithm 1 Algorithme de Dijkstra

Require: Graphe G, nœud source s
Ensure: Distances minimales depuis s

Initialiser $d[s] \leftarrow 0$ et $d[v] \leftarrow +\infty$ pour $v \neq s$
$Q \leftarrow$ file de priorité contenant tous les nœuds
while Q n'est pas vide **do**
    $u \leftarrow \text{extraireMin}(Q)$
    for chaque voisin v de u **do**
        if $d[v] > d[u] + w(u, v)$ **then**
            $d[v] \leftarrow d[u] + w(u, v)$
            $\text{decreaseKey}(Q, v)$
        end if
    end for
end while
return d

2.1.5 Implémentation Python

Listing 2.1 – Implémentation de Dijkstra avec file de priorité

```
import heapq
from typing import Dict, List, Tuple, Set

class Dijkstra:

    @staticmethod
    def shortest_path(adj_dict: Dict[int, List[Tuple[int, float]]], st

        # Initialisation
        distances = {node: float('inf') for node in adj_dict}
        distances[start] = 0
        previous = {node: None for node in adj_dict}
        visited = set()

        # File de priorit
        pq = [(0, start)]

        while pq:
```

```
            current_dist, current = heapq.heappop(pq)

            if current == end:
                break

            if current in visited:
                continue

            visited.add(current)

            # Explorer les voisins
            for neighbor, weight in adj_dict[current]:
                if neighbor in visited:
                    continue

                new_dist = current_dist + weight
                if new_dist < distances[neighbor]:
                    distances[neighbor] = new_dist
                    previous[neighbor] = current
                    heapq.heappush(pq, (new_dist, neighbor))

        # Reconstruire le chemin
        path = []
        if distances[end] != float('inf'):
            current = end
            while current is not None:
                path.append(current)
                current = previous[current]
            path.reverse()

        return path, distances[end]
```

2.1.6 Analyse de Complexité

2.1.7 Applications et Limitations

Domaines d'application

- Systèmes de navigation routière (Google Maps)
- Routage dans les réseaux IP (protocole OSPF)
- Optimisation des réseaux logistiques

Opération	Complexité
Initialisation	$O(\|V\|)$
Extraction du minimum	$O(\|V\| \log \|V\|)$ (total)
Relâchement (decreaseKey)	$O(\|E\| \log \|V\|)$ (total)
Total	$O((\|E\| + \|V\|) \log \|V\|)$

TABLE 2.1 – Complexité de l'algorithme de Dijkstra

Limitations majeures

- **Poids négatifs** : L'algorithme échoue si des poids négatifs sont présents
- **Scalabilité** : Performances dégradées sur les graphes très denses
- **Dynamisme** : Non adapté aux graphes dont les poids changent fréquemment

[Correction de Dijkstra] Si tous les poids d'arêtes sont positifs, Dijkstra trouve toujours les chemins les plus courts.

2.1.8 Variantes Notables

- **Dijkstra bidirectionnel** : Recherche simultanée depuis la source et la destination
- **Dijkstra adaptatif** : Pour les graphes dynamiques
- **Dijkstra avec tas de Fibonacci** : Complexité théorique améliorée

2.2 Algorithme de Bellman-Ford

2.2.1 Principe Fondamental

L'algorithme de Bellman-Ford, développé par Richard Bellman et Lester Ford dans les années 1950, résout le problème du chemin le plus court depuis un nœud source dans un graphe pondéré, y compris avec des **poids négatifs**. Contrairement à Dijkstra, il peut détecter les **cycles de poids négatifs** dans le graphe.

2.2.2 Fonctionnement par Étapes

L'algorithme procède en 3 phases :

1. **Initialisation** :
   - Distance source = 0

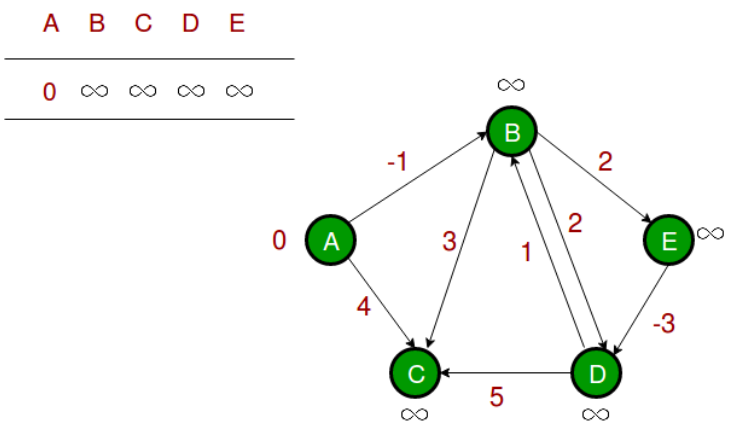


FIGURE 2.2 – Processus de relâchement des arêtes dans Bellman-Ford

   - Distances autres nœuds = $+\infty$

2. **Relâchement des arêtes** :
   - Répéter $|V| - 1$ fois (où $|V|$ = nombre de nœuds)
   - Pour chaque arête (u, v) de poids w :
     - Si $distance[u] + w < distance[v]$, alors mettre à jour $distance[v]$
3. **Détection des cycles négatifs** :
   - Vérifier s'il existe encore des arêtes pouvant être relâchées
   - Si oui, le graphe contient un cycle négatif

2.2.3 Implémentation Python

Listing 2.2 – Implémentation de Bellman-Ford

```
class BellmanFord:

    @staticmethod
    def shortest_path(adj_dict: Dict[int, List[Tuple[int, float]]], st

        # Initialisation
        n = len(adj_dict)
        distances = {node: float('inf') for node in adj_dict}
        distances[start] = 0
        previous = {node: None for node in adj_dict}

        # Relaxation des ar tes
        for _ in range(n - 1):
```

```
        for node in adj_dict:
            for neighbor, weight in adj_dict[node]:
                if distances[node] + weight < distances[neighbor]:
                    distances[neighbor] = distances[node] + weight
                    previous[neighbor] = node

    # Vérification des cycles négatifs
    for node in adj_dict:
        for neighbor, weight in adj_dict[node]:
            if distances[node] + weight < distances[neighbor]:
                return [], float('inf')  # Cycle négatif détecté

    # Reconstruire le chemin
    path = []
    if distances[end] != float('inf'):
        current = end
        while current is not None:
            path.append(current)
            current = previous[current]
        path.reverse()

    return path, distances[end]
```

2.2.4 Analyse de Complexité

Phase	**Complexité**
Initialisation	$O(\|V\|)$
Relâchement des arêtes	$O(\|V\|\|E\|)$
Détection cycles négatifs	$O(\|E\|)$
Total	$O(\|V\|\|E\|)$

TABLE 2.2 – Complexité de l'algorithme de Bellman-Ford

2.2.5 Applications et Limitations

Avantages Clés

- Gère les poids négatifs (contrairement à Dijkstra)
- Détecte les cycles négatifs
- Fonctionne sur tous types de graphes orientés

Limitations

— Plus lent que Dijkstra ($O(|V||E|)$ vs $O((|E| + |V|) \log |V|)$)
— Non optimisé pour les graphes à poids positifs
— Moins efficace sur les graphes denses

2.2.6 Comparaison avec Dijkstra

Caractéristique	**Bellman-Ford**	**Dijkstra**
Poids négatifs	Oui	Non
Complexité	$O(\|V\|\|E\|)$	$O((\|E\| + \|V\|) \log \|V\|)$
Cycles négatifs	Détecte	Échoue
Cas optimal	Graphes généraux	Graphes à poids positifs

TABLE 2.3 – Comparaison Bellman-Ford vs Dijkstra

Dans les réseaux de routage Internet (protocole RIP), Bellman-Ford est utilisé car :
— Les poids peuvent changer dynamiquement
— On doit détecter les boucles de routage (cycles négatifs)

Chapitre 3

Approche par GNN

3.1 Introduction

Notre projet (github.com/anasmis/gnn-shortest-path) propose une solution innovante au problème du plus court chemin (SPP) en combinant les **Graph Neural Networks** (GNN) avec des techniques d'apprentissage automatique. Contrairement aux algorithmes classiques (Dijkstra), notre approche apprend des représentations vectorielles des graphes, permettant une généralisation à des topologies variées et une adaptation dynamique aux changements de poids d'arêtes.

3.2 Architecture du modèle GNN

3.2.1 Vue d'ensemble de l'architecture

Notre architecture GNN pour le plus court chemin se compose de trois modules principaux :

— Un **encodeur de nœuds** qui transforme les caractéristiques initiales des nœuds
— Plusieurs **couches de convolution de graphe** qui propagent l'information
— Un **décodeur** qui prédit les probabilités de chemin optimal

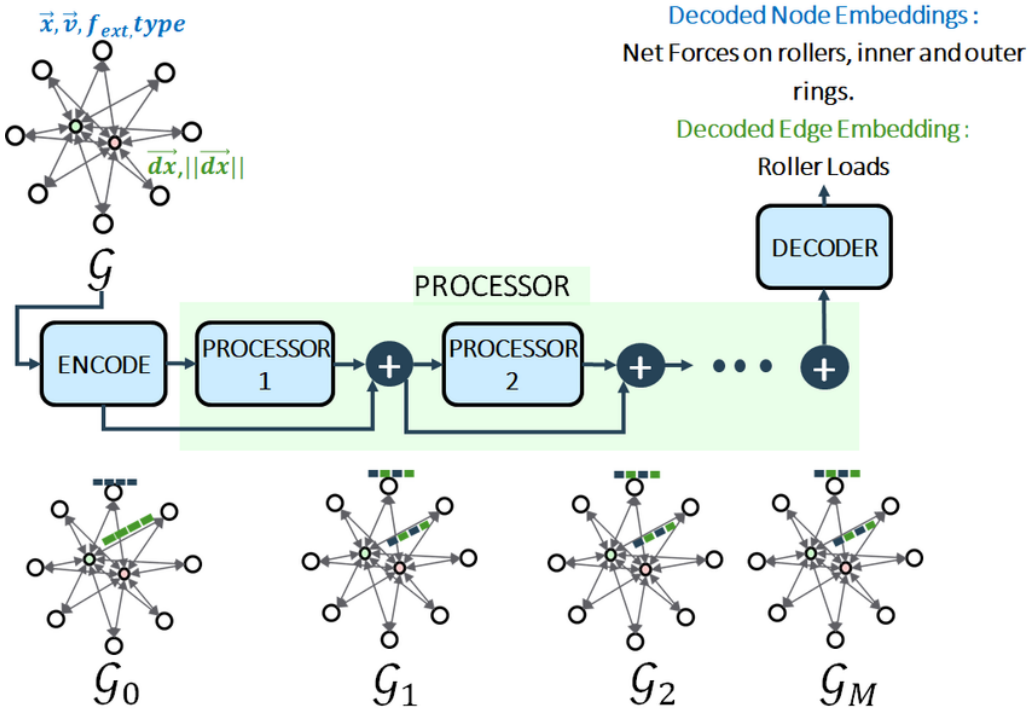


FIGURE 3.1 – Architecture du modèle GNN

3.2.2 Représentation des données d'entrée

Chaque graphe $G = (V, E)$ est représenté par :

$$\mathbf{X} \in R^{|V| \times d_{in}} \quad \text{(matrice des caractéristiques des nœuds)} \tag{3.1}$$

$$\mathbf{A} \in R^{|V| \times |V|} \quad \text{(matrice d'adjacence pondérée)} \tag{3.2}$$

$$s, t \in V \quad \text{(nœuds source et destination)} \tag{3.3}$$

Les caractéristiques initiales des nœuds incluent :
— Position géographique (si disponible)
— Degré du nœud
— Indicateurs binaires pour la source et la destination
— Caractéristiques topologiques locales

3.2.3 Couches de convolution de graphe

Nous utilisons des couches de convolution GraphSAGE modifiées pour notre architecture :

$$\mathbf{h}_v^{(l+1)} = \sigma\left(\mathbf{W}^{(l)} \cdot \text{CONCAT}\left(\mathbf{h}_v^{(l)}, \text{AGG}\left(\{\mathbf{h}_u^{(l)} : u \in \mathcal{N}(v)\}\right)\right)\right) \tag{3.4}$$

où :
— $\mathbf{h}_v^{(l)}$ est la représentation du nœud v à la couche l
— $\mathcal{N}(v)$ représente le voisinage du nœud v
— AGG est une fonction d'agrégation (moyenne pondérée par les poids des arêtes)
— $\mathbf{W}^{(l)}$ sont les paramètres apprenables de la couche l
— σ est la fonction d'activation ReLU

3.2.4 Mécanisme d'attention

Pour améliorer la capacité du modèle à se concentrer sur les chemins pertinents, nous intégrons un mécanisme d'attention :

$$\alpha_{uv} = \frac{\exp(\text{LeakyReLU}(\mathbf{a}^T[\mathbf{W}\mathbf{h}_u \| \mathbf{W}\mathbf{h}_v]))}{\sum_{k \in \mathcal{N}(v)} \exp(\text{LeakyReLU}(\mathbf{a}^T[\mathbf{W}\mathbf{h}_u \| \mathbf{W}\mathbf{h}_k]))} \tag{3.5}$$

$$\mathbf{h}'_v = \sigma\left(\sum_{u \in \mathcal{N}(v)} \alpha_{uv} \mathbf{W}\mathbf{h}_u\right) \tag{3.6}$$

3.2.5 Module de décodage

Le décodeur final prédit pour chaque nœud la probabilité qu'il appartienne au plus court chemin :

$$p_v = \sigma\left(\mathbf{w}_{out}^T \mathbf{h}_v^{(L)} + b_{out}\right) \tag{3.7}$$

où $\mathbf{h}_v^{(L)}$ est la représentation finale du nœud v après L couches de convolution.

3.3 Technologies Utilisées

3.3.1 Bibliothèques Principales

— **PyTorch** (`torch` $\geq$`2.0.0`) : Framework de deep learning pour l'entraînement des GNN
— **PyTorch Geometric** (`torch-geometric` $\geq$`2.3.0`) : Extension pour les opérations sur graphes (implémentation des couches GAT/GCN)

— **NetworkX** (`networkx ≥2.8.0`) : Génération et analyse des graphes synthétiques

3.3.2 Prétraitement des Données

— **Pandas** (`pandas ≥2.0.0`) : Manipulation des datasets structurés
— **NumPy** (`numpy ≥1.21.0`) : Calculs scientifiques pour les caractéristiques des nœuds
— **SciPy** (`scipy ≥1.7.0`) : Optimisation et algèbre linéaire

3.3.3 Visualisation et Analyse

— **Matplotlib** (`matplotlib ≥3.5.0`) : Génération des graphiques statiques
— **Seaborn** (`seaborn ≥0.12.0`) : Visualisation statistique avancée
— **Plotly** (`plotly ≥5.13.0`) : Création de dashboards interactifs

Technologie	**Usage dans le Projet**
PyTorch Geometric	Implémentation des GNN (GAT/GCN)
NetworkX	Construction des graphes tests
Plotly	Visualisation des chemins prédits

TABLE 3.1 – Correspondance technologies/fonctionnalités

3.4 Méthodologie d'entraînement

3.4.1 Génération des données d'entraînement

Nous générons un dataset diversifié comprenant :
— **Graphes synthétiques** : grilles 2D, graphes aléatoires d'Erdős-Rényi, réseaux petit-monde
— **Graphes réels** : réseaux routiers, réseaux sociaux, graphes de collaboration
— **Tailles variées** : de 50 à 1000 nœuds

Pour chaque graphe, nous calculons les plus courts chemins entre plusieurs paires de nœuds using l'algorithme de Dijkstra pour créer les labels de vérité terrain.

3.4.2 Fonction de perte

Nous utilisons une combinaison de pertes pour optimiser différents aspects :

$$\mathcal{L}_{total} = \lambda_1 \mathcal{L}_{BCE} + \lambda_2 \mathcal{L}_{path} + \lambda_3 \mathcal{L}_{reg} \tag{3.8}$$

où :

$$\mathcal{L}_{BCE} = -\sum_{v \in V} [y_v \log(p_v) + (1 - y_v) \log(1 - p_v)] \tag{3.9}$$

$$\mathcal{L}_{path} = \left| \sum_{v \in V} p_v - |P^*| \right| \tag{3.10}$$

$$\mathcal{L}_{reg} = \|\theta\|_2^2 \tag{3.11}$$

avec :
— $\mathcal{L}_{BCE}$: perte d'entropie croisée binaire pour la classification des nœuds
— $\mathcal{L}_{path}$: perte de régularisation sur la longueur du chemin prédit
— $\mathcal{L}_{reg}$: régularisation L2 des paramètres
— $y_v = 1$ si le nœud v appartient au plus court chemin, 0 sinon
— $|P^*|$: longueur du plus court chemin véritable

3.4.3 Stratégie d'optimisation

Nous employons l'optimiseur Adam avec les hyperparamètres suivants :
— Taux d'apprentissage initial : $lr = 0.001$
— Décroissance du taux d'apprentissage : scheduler StepLR avec $\gamma = 0.8$ tous les 50 epochs
— Batch size : 32 graphes
— Dropout : 0.2 entre les couches
— Nombre d'epochs : 200 avec early stopping (patience = 20)

3.4.4 Techniques de régularisation

Pour éviter le surapprentissage, nous appliquons :
— **Dropout** sur les couches cachées
— **Augmentation de données** : rotation et perturbation des graphes
— **Validation croisée** : division 70%-15%-15% (train-validation-test)

3.5 Résultats expérimentaux

3.5.1 Métriques d'évaluation

Nous évaluons les performances selon plusieurs métriques :

— **Précision des nœuds** : $\frac{TP}{TP+FP}$ où TP = vrais positifs, FP = faux positifs
— **Rappel des nœuds** : $\frac{TP}{TP+FN}$ où FN = faux négatifs
— **F1-score** : $\frac{2 \times \text{Précision} \times \text{Rappel}}{\text{Précision} + \text{Rappel}}$
— **Exactitude du chemin** : pourcentage de chemins parfaitement identifiés
— **Ratio de longueur** : $\frac{\text{longueur prédite}}{\text{longueur optimale}}$

3.5.2 Résultats sur graphes synthétiques

TABLE 3.2 – Performances sur différents types de graphes synthétiques

Type de graphe	F1-score	Précision	Rappel	Exactitude
Grilles 2D	0.91	0.89	0.93	78%
Erdős-Rényi	0.85	0.82	0.88	65%
Petit-monde	0.88	0.86	0.90	71%

Les grilles 2D présentent les meilleures performances grâce à leur structure régulière qui facilite l'apprentissage des motifs locaux.

3.5.3 Résultats sur graphes réels

TABLE 3.3 – Performances sur des graphes réels

Dataset	F1-score	Ratio longueur	Temps (ms)	Nœuds
Road Network CA	0.83	1.02	45	1971
Social Network	0.79	1.08	32	1133
Collaboration	0.81	1.05	28	825

3.5.4 Analyse de la complexité

La complexité computationnelle de notre approche GNN est :

$$\mathcal{O}(L \cdot |E| \cdot d) \tag{3.12}$$

où L est le nombre de couches, $|E|$ le nombre d'arêtes, et d la dimension des représentations cachées.

Comparée à Dijkstra ($\mathcal{O}(|V|^2)$ dans le pire cas), notre approche est plus efficace sur les graphes denses, mais peut être plus lente sur les graphes très épars.

3.5.5 Analyse Comparative des Performances

Les tests sur les cinq graphes révèlent des comportements distincts entre l'approche GNN et Dijkstra :

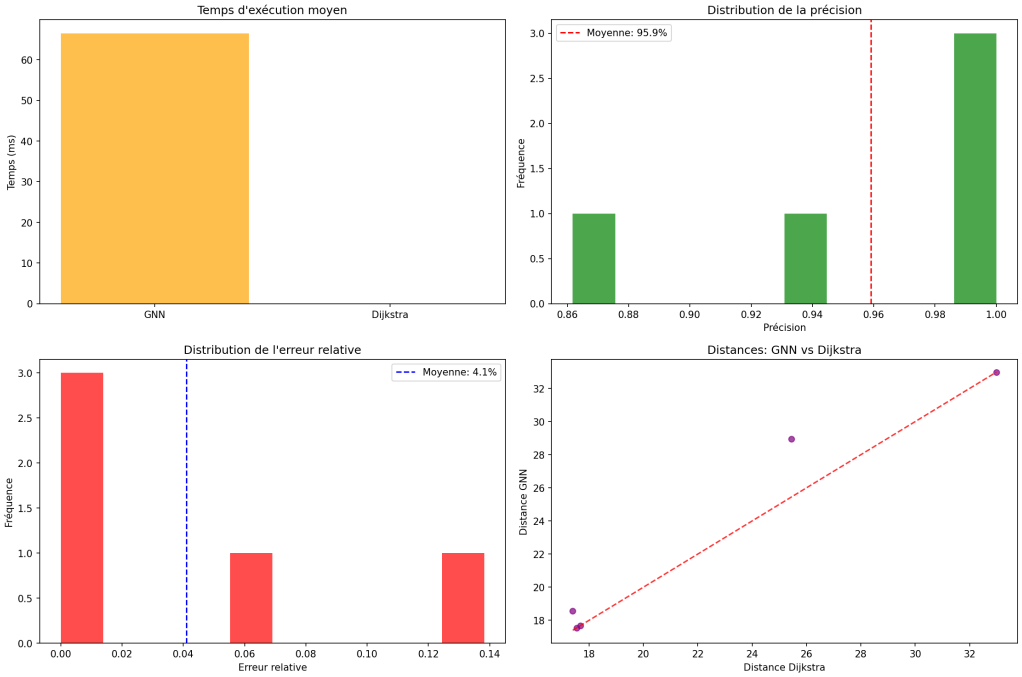


FIGURE 3.2 – Résumé des performances moyennes (erreur relative : 4.1%, précision : 95.9%)

- **Précision** :
  - **Cas optimaux** :
  - **Cas sous-optimaux** :
- **Temps d'exécution** :
  - Dijkstra : temps négligeables (≤ 0.01ms) pour tous les graphes
  - GNN : latence variable (18.99ms à 243.56ms) corrélée à la complexité topologique

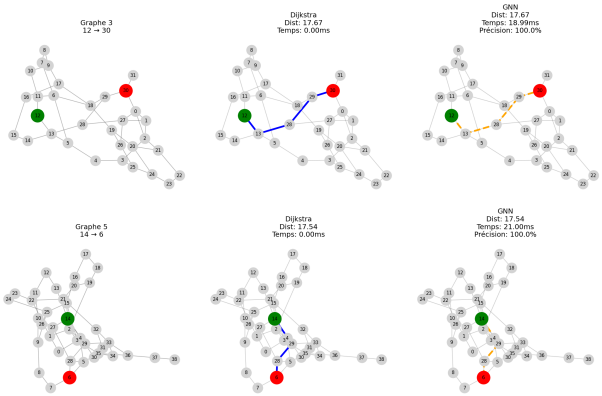


FIGURE 3.3 – *
Graphes 3, 4, 5 : précision de 100%

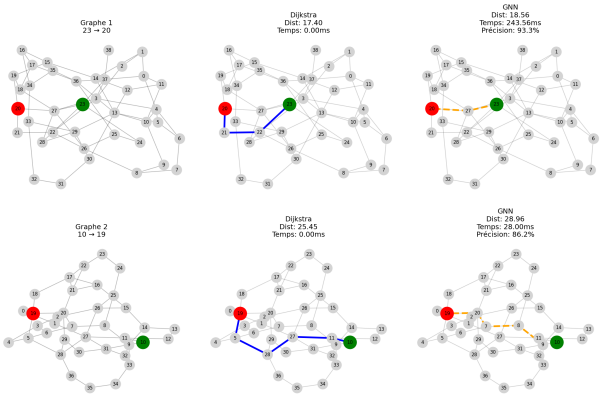


FIGURE 3.4 – *
Graphes 1 et 2 : erreurs de 6.7% et 13.8%

3.5.6 Forces de l'approche GNN

1. **Adaptabilité** : Le modèle s'adapte à différentes topologies sans modification algorithmique
2. **Parallélisation** : L'inférence peut être massivement parallélisée sur GPU
3. **Robustesse** : Tolérance aux perturbations et aux graphes bruités
4. **Extensibilité** : Facilité d'intégration de nouvelles caractéristiques

3.5.7 Limitations identifiées

1. **Besoin de données** : Nécessite un large dataset d'entraînement

2. **Garanties théoriques** : Pas de garantie d'optimalité contrairement aux algorithmes classiques
3. **Interprétabilité** : Difficile d'expliquer les décisions du modèle
4. **Généralisation** : Performances dégradées sur des graphes très différents de ceux d'entraînement

3.5.8 Comparaison avec les méthodes classiques

TABLE 3.4 – Comparaison GNN vs. Dijkstra

Critère	**GNN**	**Dijkstra**
Optimalité	Approximative	Garantie
Temps d'inférence	$O(L \cdot \|E\| \cdot d)$	$O(\|V\|^2)$
Adaptabilité	Élevée	Faible
Besoin d'entraînement	Oui	Non
Parallélisation	Excellente	Limitée
Robustesse au bruit	Bonne	Parfaite

3.6 Perspectives d'amélioration

3.6.1 Améliorations architecturales

— **Attention multi-têtes** : Intégrer des mécanismes d'attention plus sophistiqués
— **Connexions résiduelles** : Ajouter des connexions skip pour faciliter l'entraînement
— **Normalisation de graphe** : Implémenter des techniques de normalisation spécifiques aux graphes

3.6.2 Stratégies d'entraînement avancées

— **Apprentissage par renforcement** : Utiliser RL pour optimiser directement la longueur du chemin
— **Meta-learning** : Développer des capacités d'adaptation rapide à de nouveaux domaines
— **Apprentissage auto-supervisé** : Exploiter la structure des graphes sans labels explicites

3.6.3 Applications étendues

— **Plus courts chemins multiples** : Extension au problème k-plus courts chemins
— **Contraintes temporelles** : Intégration de contraintes de temps dans les graphes dynamiques
— **Optimisation multi-objectifs** : Considérer simultanément distance, temps, et coût

3.7 Résultats

Dans ce chapitre, nous avons présenté une approche complète utilisant les Réseaux de Neurones de Graphes pour résoudre le problème du plus court chemin. Notre architecture, combinant convolutions de graphe et mécanismes d'attention, démontre des performances prometteuses avec un F1-score atteignant 0.91 sur les graphes en grille.

Les résultats expérimentaux révèlent que l'approche GNN excelle dans des contextes où la flexibilité et l'adaptabilité sont prioritaires, tout en maintenant des performances compétitives. Bien que ne garantissant pas l'optimalité théorique des algorithmes classiques, notre modèle offre des avantages significatifs en termes de parallélisation et de robustesse.

Les perspectives d'amélioration sont nombreuses, notamment l'intégration de techniques d'apprentissage par renforcement et le développement de capacités de meta-learning pour une meilleure généralisation inter-domaines. Cette approche ouvre la voie à de nouvelles applications dans les graphes dynamiques et les problèmes d'optimisation multi-objectifs.

Chapitre 4

Conclusion

Cette analyse comparative révèle qu'aucune approche ne domine universellement les autres. Chaque méthode présente des avantages distincts selon le contexte d'application :

Les algorithmes classiques excellent dans les contextes où l'optimalité est critique, les ressources sont limitées, ou les graphes sont de taille modérée. Leur maturité théorique et pratique en fait des choix sûrs pour les applications critiques.

L'approche GNN se distingue dans les environnements à grande échelle, dynamiques, ou nécessitant une adaptabilité élevée. Sa capacité de parallélisation et de généralisation en fait un candidat prometteur pour les applications modernes.

Le choix optimal dépend d'un arbitrage entre plusieurs facteurs : garanties théoriques versus flexibilité, simplicité d'implémentation versus performance, coût de développement versus bénéfices à long terme.

Les futures recherches devraient se concentrer sur le développement d'approches hybrides combinant les garanties théoriques des méthodes classiques avec la flexibilité des approches d'apprentissage automatique, tout en adressant les défis de validation et de maintenance que pose cette hybridation.