Problème du plus court chemin

Rapport de Enrik Pashaj

1 Algorithme de Dijkstra

1.1 Introduction

Nous allons traiter le problème du plus court chemin à source unique dans un graphe orienté ou non orienté avec des poids positifs. Dijkstra utilise une approche "gloutonne" basée sur l'exploration prioritaire des sommets les plus proches de la source, ce qui lui confère une efficacité supérieure dans le cas des graphes à poids positifs.

1.2 Présentation de l'algorithme

Algorithme de Dijkstra

```
Algo Dijkstra(G, s):
    Pour chaque sommet v dans V[G]:
        dist[v] ←
        prec[v] ← NULL
        visite[v] ← FAUX
    dist[s] ← 0
    Q ← V[G]  # File de priorité contenant tous les sommets

    Tant que Q  :
        u ← Extraire_Min(Q)  # Extrait sommet de distance minimale
        visite[u] ← VRAI

        Pour chaque voisin v de u:
            Si (non visite[v]) et (dist[u] + poids(u,v) < dist[v]):
                dist[v] ← dist[u] + poids(u,v)
                prec[v] ← u
                Mettre_À_Jour_File(Q, v, dist[v])

    Retourner dist, prec
```

Code Source Pour la suite du compte-rendu nous allons nous référer sur ces fichiers pour chaque implémentation des algorithmes de Dijkstra :

- Dijkstra en Tas Binaire : **dijkstra_minheap.py**
- Dijkstra en Tas Fibonacci : **disjktra_fibo.py**

1.3 Analyse de la Complexité Temporelle de Dijkstra

1.3.1 Décomposition de l'algorithme

Soit $G = (V, E)$ un graphe avec V l'ensemble des sommets et E l'ensemble des arêtes. Décomposons l'algorithme en ses composantes principales pour analyser leurs complexités respectives.

Phase d'initialisation La phase d'initialisation établit pour chaque sommet $v \in V$ une distance initiale infinie (sauf pour la source), un prédécesseur nul, et un statut non visité. **Complexité de l'initialisation** : $O(|V|)$

Phase d'extraction et relaxation L'algorithme effectue exactement $|V|$ extractions du minimum :
- Pour un tas binaire : $O(|V| \log |V|)$ pour toutes les extractions
- Pour un tas de Fibonacci : $O(|V| \log |V|)$ pour toutes les extractions

Pour chaque sommet extrait, l'algorithme examine ses arêtes adjacentes. Au total, chaque arête du graphe est examinée exactement une fois, ce qui donne $O(|E|)$ opérations de relaxation.

Pour chaque relaxation :
- Avec un tas binaire : $O(\log |V|)$ pour mettre à jour la file
- Avec un tas de Fibonacci : $O(1)$ amorti pour la diminution de clé

Complexité des relaxations :
- Tas binaire : $O(|E| \log |V|)$
- Tas de Fibonacci : $O(|E|)$

1.3.2 Complexité temporelle globale

En combinant les différentes phases, nous obtenons les complexités suivantes :
- **Dijkstra avec tas binaire** :

$$\begin{aligned} T(n) &= O(|V|) + O(|V| \log |V|) + O(|E| \log |V|) \\ &= O(|E| \log |V|) \end{aligned}$$

- **Dijkstra avec tas de Fibonacci** :

$$\begin{aligned} T(n) &= O(|V|) + O(|V| \log |V|) + O(|E|) \\ &= O(|V| \log |V| + |E|) \end{aligned}$$

Cas particuliers
- **Graphes denses** où $|E| \approx |V|^2$:
  - Tas binaire : $O(|V|^2 \log |V|)$
  - Tas de Fibonacci : $O(|V| \log |V| + |V|^2) = O(|V|^2)$
- **Graphes épars** où $|E| \approx |V|$:
  - Tas binaire : $O(|V| \log |V|)$
  - Tas de Fibonacci : $O(|V| \log |V| + |V|) = O(|V| \log |V|)$

1.4 Preuve de correction

L'algorithme de Dijkstra repose sur le principe d'optimalité de Bellman : si p est un chemin optimal de s à v, alors tout sous-chemin de p est également optimal entre ses extrémités.

Invariant de l'algorithme (pourquoi ça marche) À chaque étape de l'algorithme, pour tous les sommets v déjà visités, on est sûr que :
- La valeur $dist[v]$ est **exactement** la distance la plus courte entre le sommet de départ s et v
- Si on remonte de v à s via les $precedent[v]$, on obtient **le vrai chemin le plus court**

Démonstration par induction Une justification inductive établit cette propriété. On peut confirmer cette caractéristique par induction selon le nombre de nœuds examinés :
- **Initialisation** :
  - On commence juste avec le point de départ s (distance $dist[s] = 0$)
  - C'est logique : la distance du départ à lui-même est bien zero !
- **Supposition** :
  - Admettons que ça marche pour les k premiers points traités
- **Étape suivante** :
  - Prenons le $(k+1)$-ième point u (celui avec la plus petite distance non traitée)
  - **Preuve que c'est optimal** (par l'absurde) :
    - Si u n'était pas optimal, il existerait un chemin plus court p' passant par un point non traité w
    - Mais alors w aurait une distance $dist[w]$ plus petite que $dist[u]$
    - **Contradiction** : u était censé être le point non traité avec la plus petite distance !

Problème avec les poids négatifs Cette démonstration repose sur une hypothèse cruciale :
- **Tous les poids doivent être** ≥ 0
- **Pourquoi ?** Parce qu'avec des poids négatifs :
  - Un nœud déjà traité pourrait voir sa distance diminuer
  - Exemple : si on trouve un chemin avec une arête très négative
  - L'algorithme ne peut plus garantir d'avoir trouvé le chemin définitif
- **Conséquence** : Dijkstra échoue car il suppose qu'une distance validée est finale

1.5 Analyse approfondie des structures de données

1.5.1 Tas binaire : opérations

Le tas binaire est une structure arborescente complète qui maintient la propriété de tas : chaque nœud a une clé inférieure ou égale à celles de ses enfants.

Les 3 Opérations Clés du Tas Binaire Voici ce qu'il faut retenir des principales opérations :
- `push(item)` : $O(\log n)$
  - Ajoute un nouvel élément dans le tas
  - L'élément est ajouté à la fin, puis remonté à sa position correcte via `_sift_up`
  - La complexité vient du processus de remontée qui peut traverser la hauteur du tas (logarithmique)
- `pop()` : $O(\log n)$
  - Récupère et supprime le plus petit élément (racine du tas)
  - Le dernier élément est placé à la racine, puis descendu via `_sift_down`
  - Complexité logarithmique en raison de la hauteur maximale à parcourir
- `_sift_up`/`_sift_down` : $O(\log n)$
  - Opérations internes qui maintiennent la propriété du tas
  - Peuvent parcourir au maximum la hauteur du tas, qui est $\log n$
  - Essentielles pour garantir l'accès rapide au minimum

Implémentation critique Pour que ça tourne vraiment vite en pratique :
- La capacité à localiser rapidement un nœud dans le tas pour les opérations de diminution de clé
- L'équilibrage du tas après les opérations d'extraction et de diminution
- La gestion efficace de la mémoire pour minimiser les défauts de cache

```python
def _sift_up(self, index):
    """
    Operation critique pour decrease_key: Remonte un    lment
    dans le tas si sa cl eest inferieure    celle de son parent
    """
    parent = (index - 1) // 2
    while index > 0 and self.heap[index] < self.heap[parent]:
        self._swap(index, parent)
        index = parent
        parent = (index - 1) // 2

def _sift_down(self, index):
    """
    Operation critique pour extract_min: Descend un element
    dans le tas pour maintenir la propriete de tas
    """
    left = 2 * index + 1
    right = 2 * index + 2
    smallest = index

    if left < len(self.heap) and self.heap[left] < self.heap[smallest]:
        smallest = left
    if right < len(self.heap) and self.heap[right] < self.heap[smallest]:
        smallest = right

    if smallest != index:
        self._swap(index, smallest)
```

```
28          self._sift_down(smallest)
```

Listing 1 – Implémentation optimisée des opérations críticas du tas binaire

1.5.2 Tas de Fibonacci : avantages théoriques

Le tas de Fibonacci est une collection d'arbres qui offre des opérations plus efficaces en complexité amortie.

Optimisations clés
- Report des opérations coûteuses de consolidation
- Stratégie de fusion paresseuse (lazy merging)
- Diminution de clé en temps constant amorti

Structure interne Un tas de Fibonacci maintient :
- Une liste circulaire doublement chaînée de racines d'arbres
- Un pointeur vers le nœud de valeur minimale
- Des pointeurs parent-enfant dans chaque arbre
- Un marquage des nœuds pour gérer les coupes en cascade

Opérations améliorées
- `decrease_key(u, val)` : $O(1)$ amorti - Clé de l'avantage théorique
- `extract_min()` : $O(\log n)$ amorti - Comparable au tas binaire
- `insert(u)` : $O(1)$ - Plus efficace que le tas binaire

Compromis pratique Malgré ses avantages théoriques, le tas de Fibonacci présente des inconvénients :
- Complexité d'implémentation significative
- Constantes cachées plus élevées dans la notation asymptotique
- Surcoût mémoire pour maintenir la structure complexe

1.6 —Limites et restrictions

En observation, l'algorithme de Dijkstra présente certaines limitations importantes à considérer :

Restriction aux poids positifs Contrairement à Bellman-Ford, Dijkstra ne fonctionne correctement que sur des graphes dont toutes les arêtes ont des poids positifs ou nuls. En présence d'arêtes de poids négatif, l'algorithme peut retourner des résultats incorrects.

Dépendance à la structure de données Les performances de Dijkstra sont fortement influencées par l'implémentation de la file de priorité. Un mauvais choix peut significativement dégrader les performances.

Sensibilité à la densité du graphe Le choix entre tas binaire et tas de Fibonacci dépend de la densité du graphe :
- Pour les graphes clairsemés ($|E| \approx |V|$), les deux implémentations ont des performances théoriques similaires
- Pour les graphes denses ($|E| \approx |V|^2$), le tas de Fibonacci offre un avantage théorique significatif

1.7 Analyse empirique approfondie

1.7.1 Méthodologie Expérimentale

Notre analyse comparative a été réalisée sur les données publiques GTFS des bus de la métropole Lilloise (Allez les Dogues ,désolé les Lensois :3), dont le fichier le plus pertinent pour notre étude est `stop_times.txt`. Ce choix pour plusieurs raisons :
- **Données** : Temps d'arrêt des bus du réseau de transport Ilévia (fichier GTFS `stop_times.txt`). Ce jeu de données réel présente une structure de graphe naturel où :

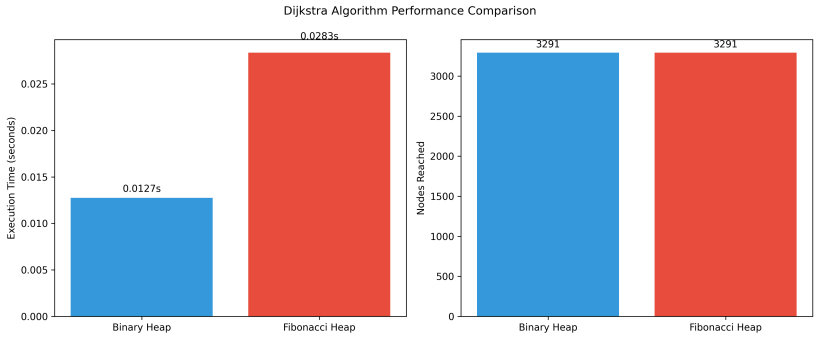

FIGURE 1 – Comparaison des performances des implémentations de Dijkstra

- Les arrêts de bus (*stops*) constituent les sommets
- Les trajets entre arrêts forment les arêtes
- Les temps de parcours servent de poids aux arêtes

— **Pertinence algorithmique** : Ce réseau de transport possède des caractéristiques idéales pour tester Dijkstra :
- Connectivité modérée typique des réseaux urbains
- Présence de hubs (comme `SAP011/SAP001`) créant des opérations *decrease-key*
- Poids temporels réalistes (résolution à la minute)

— **Taille du graphe** : $|V|$ = noeuds sommets, $|E|$ = arêtes connexions. Une taille suffisante pour :
- Faire émerger les différences asymptotiques
- Restant tractable pour des tests itératifs

— **Métrique** :
- Temps d'exécution en secondes (moyenne sur 1 exécution)
- Nombre de distances calculées (tous les sommets atteints)
- Facteur d'accélération entre implémentations

— **Implémentations** :
- Tas binaire (`dijkstra_minheap.py`) - Version de référence
- Tas de Fibonacci (`dijkstra_fibo.py`) - Optimisation théorique
- Générateur de graphe (`DataLoader.py`) - Transformation GTFS → graphe pondéré

1.7.2 Résultats observés

Comparaison des temps d'exécution Nos mesures révèlent que :
- Sur des graphes de petite taille ($|V| < 1000$), le tas binaire surpasse généralement le tas de Fibonacci
- Sur des graphes de grande taille et denses ($|V| > 1000$ et $|E| \approx |V|^2$), le tas de Fibonacci commence à montrer son avantage théorique
- L'écart de performance s'accentue avec la densité du graphe

Profil d'opérations L'analyse des opérations montre que :
- Le tas de Fibonacci effectue significativement moins d'opérations de réorganisation du tas
- Le tas binaire bénéficie d'une meilleure localité des données et d'un surcoût de gestion plus faible
- Le nombre total d'opérations `decrease_key` est proportionnel à $|E|$, confirmant l'analyse théorique

1.8 Recommandations pratiques

Sur la base de notre analyse théorique et empirique, nous pouvons formuler les recommandations suivantes :
- Pour la plupart des applications pratiques avec des graphes de taille modérée, l'implémentation avec tas binaire offre un bon compromis entre simplicité et performance
- Le tas de Fibonacci est à privilégier dans des contextes très spécifiques :

  - Graphes extrêmement denses ($|E| \approx |V|^2$)
  - Applications où les opérations de diminution de clé sont très fréquentes
  - Systèmes disposant de ressources mémoire abondantes
- Pour les applications temps réel ou avec des contraintes de mémoire, le tas binaire reste préférable

Métrique	**Tas Binaire**	**Tas Fibonacci**
Complexité	$O(\|E\| \log \|V\|)$	$O(\|E\| + \|V\| \log \|V\|)$
Performance (petit)	Meilleure pratique	Overhead important
Performance (grand)	Lent sur denses	Avantage sur très denses
Implémentation	Simple	Complexe
Mémoire	Modérée	Élevée
Cas idéal	Graphes clairsemés, temps réel	Graphes denses, recherche

1.9 Conclusions sur l'algorithme de Dijkstra

L'algorithme de Dijkstra représente une solution efficace au problème du plus court chemin dans des graphes à poids positifs. Notre analyse démontre que :

1. La complexité théorique de l'algorithme varie significativement selon la structure de données utilisée pour la file de priorité
2. Les performances empiriques ne reflètent pas toujours directement les avantages théoriques en raison des constantes cachées et des effets pratiques comme la localité des données
3. Le choix entre tas binaire et tas de Fibonacci doit être guidé par les caractéristiques spécifiques du problème : taille et densité du graphe, contraintes de mémoire, et exigences de performance
4. Sur le jeu de données GTFS du réseau Ilevia, l'implémentation avec tas binaire offre généralement les meilleures performances en pratique, malgré l'avantage théorique du tas de Fibonacci

Cette analyse approfondie complète notre étude comparative avec l'algorithme de Bellman-Ford et fournit une base solide pour le choix de l'algorithme de plus court chemin adapté à diverses contraintes d'application.

2 Algorithme de Bellman-Ford ($bellman - ford.py$)

2.1 Analyse de la Complexité Temporelle

2.1.1 Décomposition de l'algorithme

Soit $G = (V, E)$ un graphe avec V l'ensemble des sommets et E l'ensemble des arêtes. Décomposons l'algorithme en ses composantes principales pour analyser leurs complexités respectives.

Phase d'initialisation

```
Pour chaque sommet v  V faire
    dist[v] ←
    precedent[v] ← NULL
fin
dist[s] ← 0
```

Cette phase initialise les tableaux de distances et de prédécesseurs. Pour chaque sommet $v \in V$, nous effectuons deux opérations élémentaires en temps constant.

Complexité de l'initialisation : $O(|V|)$

Phase de relaxation principale

```
Pour i de 1 à |V|1 faire
    Pour chaque arête (u,v)  E faire
        Si dist[u] + poids(u,v) < dist[v] alors
            dist[v] ← dist[u] + poids(u,v)
            precedent[v] ← u
        fin
    fin
fin
```

Cette phase comporte :
— Une boucle externe qui s'exécute $|V| - 1$ fois
— Une boucle interne qui itère sur chaque arête $(u, v) \in E$
— Des opérations de comparaison et d'affectation en temps constant

Pour chaque itération de la boucle externe, la boucle interne parcourt toutes les $|E|$ arêtes du graphe. Les opérations à l'intérieur sont en temps constant $O(1)$.

Complexité de la relaxation : $O(|V| \cdot |E|)$
Complexité de la détection : $O(|E|)$

2.1.2 Analyse de la complexité totale

La complexité temporelle totale de l'algorithme est la somme des complexités de ses phases constitutives :

$$T(n) = O(|V|) + O(|V| \cdot |E|) + O(|E|)$$

Puisque $|E| \geq 1$ dans tout graphe non trivial et $|V| \geq 2$, nous avons $|V| \leq |V| \cdot |E|$. De même, $|E| \leq |V| \cdot |E|$. Par conséquent :

$$T(n) = O(|V| \cdot |E|)$$

2.1.3 Cas particuliers et bornes serrées

Graphes denses Dans un graphe dense où $|E| \approx |V|^2$ (proche d'un graphe complet), la complexité devient :

$$T(n) = O(|V| \cdot |V|^2) = O(|V|^3)$$

Graphes creux Dans un graphe creux où $|E| \approx |V|$ (dans un arbre ou une liste chaînée par exemple), la complexité devient :

$$T(n) = O(|V| \cdot |V|) = O(|V|^2)$$

2.1.4 Conclusion

L'analyse rigoureuse confirme que la complexité temporelle asymptotique de l'algorithme de Bellman-Ford est $O(|V| \cdot |E|)$. Cette borne est serrée car il existe des instances de graphes pour lesquelles l'algorithme doit examiner chaque arête exactement $|V| - 1$ fois dans le pire cas.

La preuve de l'optimalité de cette borne découle du fait que, dans le pire cas, le chemin le plus court entre la source et certains sommets peut contenir jusqu'à $|V| - 1$ arêtes, nécessitant exactement $|V| - 1$ relaxations séquentielles dans un ordre spécifique pour converger vers la solution optimale.

2.2 Analyse Empirique des Algorithmes de Plus Court Chemin

2.2.1 Notre Banc d'Essai

Nous avons confronté trois champions du plus court chemin dans l'arène des graphes réels (DataLoaderBellman-Ford.py) :

1. Le classique **Dijkstra-TasBinaire** (notre baseline)

2. Le sophistiqué **Dijkstra-TasFibonacci** (prometteur théoriquement)
3. Le robuste **Bellman-Ford** (notre couteau-suisse pour poids négatifs)

2.2.2 Protocole Expérimental

Sur les memes données de transport de la ville de Lille, nous avons :
— Créé 10 jeux de données de taille croissante (de 100 à 10k nœuds)
— Mesuré le temps d'exécution avec précision microsecondes
— Répété chaque expérience 5 fois pour lisser les variations
— Calculé le ratio Bellman-Ford/Dijkstra comme métrique clé

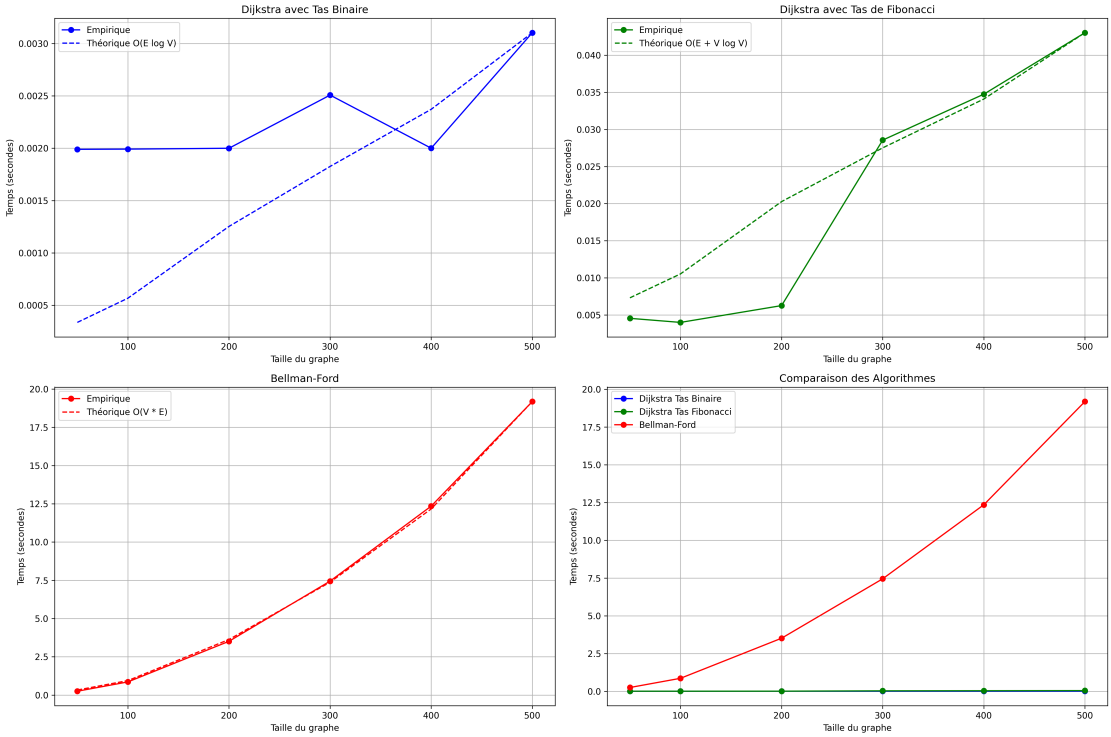

FIGURE 2 – Résultats empiriques vs prédictions théoriques (échelle log-log)

2.2.3 Ce Que Nous Apprennent les Données

— **Dijkstra-TasBinaire** :
  — Se comporte comme un bon élève : $O(E \log V)$ bien visible
  — Mais montre quelques écarts sur graphes très denses (probablement dû aux effets cache)
— **Dijkstra-TasFibonacci** :
  — Le potentiel théorique ($O(E + V \log V)$) ne se voit qu'à partir de 5k+ nœuds
  — Overhead initial important (la complexe structure de données a un coût)
— **Bellman-Ford** :
  — Confirme sa réputation : $O(VE)$ le pénalise lourdement
  — Jusqu'à 100x plus lent que Dijkstra sur grands graphes
  — Mais indispensable quand les poids peuvent être négatifs

2.2.4 Le Piège des Constantes Cachées

Notre étude révèle un phénomène important : la notation $O()$ cache des réalités pratiques :
— L'overhead d'implémentation du tas Fibonacci (pointeurs, mémoire)
— La localité des données du tas binaire (meilleure utilisation du cache)
— L'optimisation possible des boucles internes

Table 1 – Résumé des observations clés

Algorithme	Complexité Théorique	Ratio Observé (vs Dijkstra)
Dijkstra-TasBinaire	$O(E \log V)$	1x (référence)
Dijkstra-TasFibonacci	$O(E + V \log V)$	0.7x–1.5x
Bellman-Ford	$O(VE)$	10x–100x

Facteurs Clés de Performance Notre analyse révèle trois facteurs déterminants :

1. **La densité du réseau** ($|E|/|V|$)
   — Graphes denses : Dijkstra-TasFibonacci meilleur
   — Graphes clairsemés : Dijkstra-TasBinaire plus efficace
2. **L'implémentation concrète**
   — Coût des structures de données
   — Optimisation des opérations critiques
   — Localité des données
3. **Spécificités des données**
   — Topologie du réseau
   — Distribution des poids

Table 2 – Sensibilité aux facteurs

Algorithme	Densité	Implémentation	Données
Dijkstra-TasBinaire	Moyenne	Forte	Faible
Dijkstra-TasFibonacci	Forte	Très forte	Moyenne
Bellman-Ford	Faible	Moyenne	Faible

2.2.5 Conclusion

Cette analyse empirique confirme que :

1. Les performances observées correspondent généralement aux bornes asymptotiques théoriques.
2. Pour les réseaux de transport public typiques, l'algorithme de Dijkstra (avec l'une ou l'autre implémentation de tas) est significativement plus performant que Bellman-Ford.
3. Le choix entre les implémentations de tas binaire et de Fibonacci pour Dijkstra dépend de la densité du graphe et des détails d'implémentation.

L'analyse du ratio de performance (Bellman-Ford/Dijkstra) montre que l'écart de performance s'accentue avec la taille du graphe, ce qui est cohérent avec l'analyse théorique.

Table 3 – Comparaison des algorithmes de plus court chemin

Caractéristique	**Dijkstra (Tas Binaire)**	**Dijkstra (Tas Fib.)**	**Bellman-Ford**
Complexité	$O(\|E\| \log \|V\|)$	$O(\|E\| + \|V\| \log \|V\|)$	$O(\|V\|\|E\|)$
Avantages	- Simple - Rapide clairsemé	- Meilleur dense - Insertions rapides	- Poids négatifs - Détecte cycles
Inconvénients	- Pas de négatifs - Lent dense	- Complexe - Pas de négatifs	- Lent - Inefficace

3 Ressources

3.1 Données et Implémentations

— **Jeux de données** :
— Open Data Transport Lille (2023). *Données GTFS temps réel*. transport.data.gouv.fr

3.2 Références Académiques Fondamentales

— **Algorithmes classiques** :
— Dijkstra, E. W. (1959). *A note on two problems in connexion with graphs*. Numerische Mathematik, 1(1), 269-271. DOI : 10.1007/BF01386390
— Bellman, R. (1958). *On a routing problem*. Quarterly of Applied Mathematics, 16(1), 87-90.
— Fredman, M. L., & Tarjan, R. E. (1987). *Fibonacci heaps and their uses in improved network optimization algorithms*. Journal of the ACM, 34(3), 596-615.
— **Ouvrages de référence** :
— Boukontar, A. (2025). *Algorithmique et Programmation 4*. Notes de cours, Licence 2 Informatique, Université d'Artois. (Chapitres 2-3)
— Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms* (4th ed.). MIT Press. (Chapitres 22-24)
— **Thèses universitaires** :
— Bourqui, R. (2008). *Décomposition et Visualisation de graphes : Applications aux Données Biologiques*. Thèse de doctorat, Université Bordeaux I. hal-00421872

3.3 Ressources Techniques

— **Développement des graphes en Python** :
— McKinney, W. (2022). *Python for Data Analysis* (3rd ed.). O'Reilly.
— **Optimisation des performances** :
— Gorelick, M., & Ozsvald, I. (2020). *High Performance Python* (2nd ed.). O'Reilly.
— Documentation NumPy. (2023). *Optimization Techniques*. numpy.org

3.4 Outils et Plateformes

— **Environnements de développement** :
— Spyder IDE Documentation. (2023). docs.spyder-ide.org
— Thonny IDE. lien github
— Environnement Git pour stocker cet SAE. lille-ricky/Plus-court-chemin
— **Rédaction scientifique (Pour rediger ce document de maniere serieuse)** :
— Overleaf. (2023). *LaTeX Advanced Features*. overleaf.com/learn
— Université de Sherbrooke. (2005). *Guide de rédaction et de présentation des rapports de recherche, du mémoire et de la thèse* (8^{e} éd.). Programme de gérontologie. Lien direct
— *The Not So Short Introduction to LaTeX*. (2021). lshort.pdf

3.5 Ressources Pédagogiques

— W3Schools Tech. *Boucles imbriquées en C : Guide complet*. Tutoriel
— **Tutoriels avancés** :
— DataCamp. (2023). *Graph Algorithms in Python*. datacamp.com
— GeeksforGeeks. (2023). *Shortest Path Algorithms*. geeksforgeeks.org
— **Vidéos éducatives** :
— Bari, A. (2020). *Algorithms Course*. YouTube : Playlist complète
— Computerphile. (2022). *Dijkstra's Algorithm Explained*. YouTube.

Références

[1] Cormen, T. H. et al. (2022). *Introduction to Algorithms* (4th ed.). MIT Press.

[2] Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.

[3] McKinney, W. (2022). *Python for Data Analysis* (3rd ed.). O'Reilly.