

Comparaison d'algorithmes de plus courts chemins dans un graphe pondéré

Ben Said Nadhir et Said Anas

Encadrante : Fanny Pascual
2024–2025

Résumé

Ce rapport s'inscrit dans le cadre du projet 12 « Comparaison d'algorithmes construisant des arbres couvrants de coût minimum ». Toutefois, nous avons choisi d'explorer et de comparer des algorithmes fondamentaux de la théorie des graphes, tels que Dijkstra et Bellman-Ford, qui résolvent un problème étroitement lié : " le calcul des plus courts chemins dans un graphe pondéré ", problème fréquemment rencontré en routage, réseaux et optimisation. Notre étude inclut une analyse théorique des complexités, des implémentations commentées, ainsi qu'une comparaison des temps d'exécution pratique .

De surcroît, nous avons développé une application utilisant les algorithmes de Dijkstra (avec et sans tas), Bellman-Ford et A* pour calculer les plus courts chemins. Enfin, nous avons abordé le problème du voyageur de commerce en intégrant son code dans notre application IOS.

Table des matières

1	Introduction	3
1.1	Contexte et problématique	3
2	Algorithme de Dijkstra	3
2.0.1	Principe	3
2.1	Algorithme de Dijkstra — pseudocode	3
2.1.1	Déroulement de l'algorithme	3
2.2	Remarque	4
3	Algorithme de Bellman-Ford	4
3.1	Principe	4
3.2	Algorithme de Bellman-Ford — pseudocode	5
4	Algorithme A* (A Star)	5
5	Comparaison théorique	6
6	Comparaison expérimentale	6
6.1	Graphes peu denses ($p = 0.1$)	7
6.2	Graphes denses ($p = 0.8$)	7
6.3	Influence de la densité ($n = 500$)	8
6.4	Comparaison entre A* et Dijkstra avec tas	8
7	Analyse Comparative des Algorithmes de Plus Court Chemin	9
7.1	Impact de la Taille du Graphe (n) à Densité Fixe	9
7.2	Effet de la Densité (p) pour $n = 500$	10
8	Recommandations d'Utilisation	10
9	Conclusion	11
10	Application Flutter	12
10.1	Remarque importante	15

1 Introduction

1.1 Contexte et problématique

Le problème du plus court chemin dans un graphe pondéré consiste à trouver un chemin entre deux sommets de sorte que la somme des poids des arcs empruntés soit minimale. Ce problème est fondamental en algorithmique et intervient dans de nombreux domaines tels que la logistique, les réseaux informatiques, les transports et l'intelligence artificielle.

Il s'agit d'un problème central car il permet d'optimiser des trajets ou des flux dans des systèmes complexes, facilitant ainsi la prise de décisions efficaces dans des contextes variés, comme le routage dans les réseaux ou la planification d'itinéraires.

En théorie des graphes, le problème du plus court chemin est défini comme la recherche d'un chemin entre un sommet de départ et un sommet d'arrivée dans un graphe pondéré, de manière à minimiser la somme des poids des arcs qui composent ce chemin.

2 Algorithme de Dijkstra

2.0.1 Principe

L'algorithme de Dijkstra sert à résoudre le problème du plus court chemin. Il calcule des plus courts chemins à partir d'une source vers tous les autres sommets. On peut aussi l'utiliser pour calculer un plus court chemin entre un sommet de départ et un sommet d'arrivée. Dijkstra explore toujours le sommet le plus proche non visité, en maintenant des distances temporaires. Son fonctionnement nécessite des poids **non négatifs**.

2.1 Algorithme de Dijkstra — pseudocode

Cette méthode d'implémentation de l'algorithme de Dijkstra, qui utilise un ensemble P et une recherche du sommet non visité à distance minimale par parcours linéaire, est une implémentation naïve dont la complexité totale est en

$$\mathcal{O}(|S|^2 + |A|) \approx \mathcal{O}(|S|^2),$$

car en pratique, le terme $|S|^2$ domine $|A|$.

2.1.1 Déroulement de l'algorithme

L'algorithme de Dijkstra calcule les plus courts chemins depuis un noeud source vers tous les autres dans un graphe pondéré à poids positifs. Il utilise une file de priorité (ou ensemble de noeuds non visités) et suit le principe suivant :

- Initialiser les distances : 0 pour la source, ∞ pour les autres.
- Tant qu'il reste des sommets non visités :
 - Sélectionner le sommet non visité avec la plus petite distance.
 - Pour chacun de ses voisins, mettre à jour la distance si un chemin plus court est trouvé.
 - Marquer le sommet comme visité.

À chaque étape, on enregistre aussi le prédécesseur permettant de reconstruire le plus court chemin à la fin.

Algorithm 1 Algorithme de Dijkstra pour les plus courts chemins

Entrée: Un graphe pondéré $G = (S, A)$ avec poids positifs, un sommet source s_{deb}

Sortie : Distances minimales d et prédecesseurs $pred$

```
Initialisation :  $P \leftarrow \emptyset$                                 // Ensemble des sommets visités
 $d \leftarrow$  tableau avec  $d[a] = +\infty$  pour tout sommet  $a \in S$ 
 $pred \leftarrow$  tableau avec  $pred[a] = \text{None}$  pour tout sommet  $a \in S$ 
 $d[s_{deb}] \leftarrow 0$ 

while  $P \neq S$  do
|   Choisir  $a \in S \setminus P$  tel que  $d[a]$  est minimal           // Sommet non visité avec distance
|   minimale
|    $P \leftarrow P \cup \{a\}$ 
|   foreach voisin  $b$  de  $a$  tel que  $b \notin P$  do
|   |   nouvelle_distance  $\leftarrow d[a] + \text{poids}(a, b)$           // Distance alternative
|   |   if nouvelle_distance <  $d[b]$  then
|   |   |    $d[b] \leftarrow$  nouvelle_distance
|   |   |    $pred[b] \leftarrow a$ 
|   |   end
|   end
| end
return  $(d, pred)$                                          // Retourne les distances et prédecesseurs
```

2.2 Remarque

Dans la version naïve de l'algorithme de Dijkstra, le sommet de distance minimale est recherché manuellement à chaque itération parmi tous les sommets non visités, ce qui coûte $\mathcal{O}(n)$ par sélection. Cela rend l'algorithme globalement en $\mathcal{O}(n^2)$ pour un graphe à n sommets.

L'optimisation consiste à utiliser un *tas de priorité*, qui permet de sélectionner et de mettre à jour efficacement les sommets avec la plus petite distance en $\mathcal{O}(\log n)$.

Chaque sommet est extrait du tas au plus une fois (soit n extractions), et chaque arête peut entraîner une mise à jour de distance donc une insertion dans le tas (soit m insertions), ce qui donne au total $\mathcal{O}((n + m) \log n)$ opérations.

Cela améliore fortement l'efficacité sur les grands graphes, en particulier lorsqu'ils sont peu denses (c'est-à-dire lorsque le nombre d'arêtes m est beaucoup plus petit que n^2).

3 Algorithme de Bellman-Ford

3.1 Principe

L'algorithme de Bellman-Ford calcule les plus courts chemins depuis un sommet source dans un graphe orienté pondéré, y compris lorsque certaines arêtes ont des poids négatifs, ce que l'algorithme de Dijkstra ne peut pas gérer.

Il fonctionne en relâchant toutes les arêtes du graphe à plusieurs reprises (jusqu'à $n - 1$ fois, où n est le nombre de sommets), en mettant à jour progressivement les distances les plus courtes. Cette méthode garantit que même les chemins passant par des arêtes à poids négatif sont pris en compte.

De plus, Bellman-Ford peut détecter l'existence de circuit de poids négatif : si, après $n - 1$ itérations, une nouvelle relaxation améliore encore une distance, cela signifie qu'un circuit négatif est présent.

Bien que plus coûteux en temps, avec une complexité en $\mathcal{O}(n \times m)$ (où m est le nombre d'arêtes), que Dijkstra, Bellman-Ford est particulièrement utile dans les contextes où les graphes peuvent comporter des poids négatifs, comme dans certains problèmes d'optimisation ou réseaux financiers.

3.2 Algorithme de Bellman-Ford — pseudocode

Déroulement de l'algorithme

L'algorithme de Bellman-Ford suit le principe suivant :

- Initialiser les distances : 0 pour la source, ∞ pour les autres.
- Répéter $(n - 1)$ fois, où n est le nombre de sommets :
 - Pour chaque arête (u, v) , mettre à jour la distance de v si un chemin plus court via u est trouvé et mettre à jour le prédecesseur de v .
- Vérifier l'existence de cycles de poids négatif :
 - Si une distance peut encore être améliorée, alors un cycle négatif est présent (le problème n'a pas de solution).

Algorithm 2 Algorithme de Bellman-Ford

Entrée: Un graphe $G = (S, A, w)$ et un sommet source s

Sortie : Distances minimales d ou détection d'un circuit absorbant

```

foreach  $v \in S$  do
  |  $d[v] \leftarrow +\infty$   $pred[v] \leftarrow \text{None}$ 
end
 $d[s] \leftarrow 0$ 
for  $i \leftarrow 1$  to  $|S| - 1$  do
  | foreach  $(u, v) \in A$  do
    |   | if  $d[u] + w(u, v) < d[v]$  then
    |   |   |  $d[v] \leftarrow d[u] + w(u, v)$ 
    |   |   |  $pred[v] \leftarrow u$ 
    |   | end
  | end
end
foreach  $(u, v) \in A$  do
  | if  $d[u] + w(u, v) < d[v]$  then
  |   | return "Circuit absorbant détecté"
  | end
end
return  $d, pred$ 

```

4 Algorithme A* (A Star)

Déroulement de l'algorithme

L'algorithme A* (*A star*) calcule le plus court chemin depuis un noeud source vers un noeud but dans un graphe pondéré, en s'appuyant sur une heuristique pour guider la recherche. Il suit le principe suivant :

- Initialiser les distances : 0 pour la source, ∞ pour les autres.

- Initialiser une file de priorité contenant la source, avec pour priorité $f(n) = g(n) + h(n)$:
 - $g(n)$ est le coût réel pour atteindre n depuis la source.
 - $h(n)$ est une estimation heuristique du coût pour aller de n jusqu’au but.
- Tant que la file n’est pas vide :
 - Extraire le noeud avec le plus petit $f(n)$.
 - Si ce noeud est le but, reconstruire et retourner le chemin.
 - Sinon, pour chacun de ses voisins :
 - Calculer le coût temporaire g pour atteindre ce voisin.
 - Si ce coût est plus faible que le précédent, mettre à jour la distance, le prédecesseur et ajouter (ou mettre à jour) ce voisin dans la file avec la nouvelle priorité f .
- L’heuristique $h(n)$ doit être **admissible** (ne jamais surestimer le vrai coût) pour garantir l’optimalité. Dans notre projet on a trouvé l’heuristique euclidienne (ou la distance euclidienne) comme une des heuristiques les plus naturelles, simple et efficace au même temps.

5 Comparaison théorique

TABLE 1 – Comparaison des algorithmes de plus court chemin

Propriété	Dijkstra	Dijkstra + tas	Bellman-Ford
Poids négatifs	Non	Non	Oui
Complexité	$\mathcal{O}(S ^2 + A) \approx \mathcal{O}(S ^2)$	$\mathcal{O}((S + A) \log S)$	$\mathcal{O}(S A)$
Cas optimal	Graphes denses avec poids positifs	Graphes peu denses avec poids positifs	Graphes avec poids négatifs ou peu denses
Détection de circuits	Non	Non	Oui

6 Comparaison expérimentale

Nous avons mesuré les temps d’exécution des algorithmes dans différents scénarios :

6.1 Graphes peu denses ($p = 0.1$)

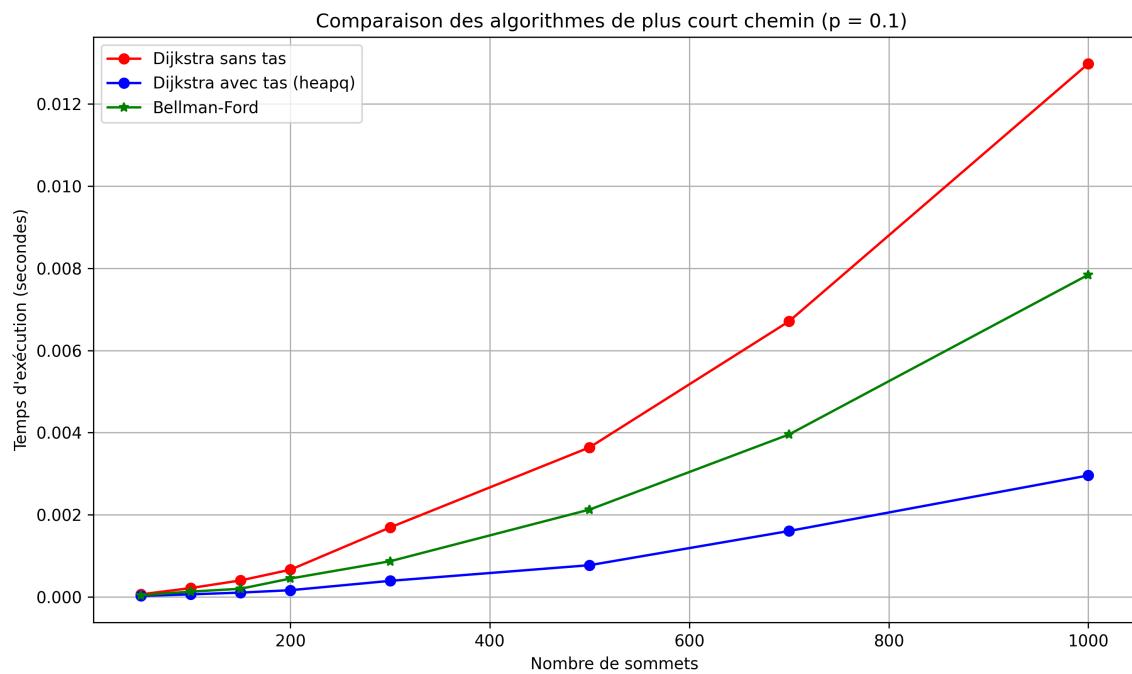


FIGURE 1 – Temps d’exécution des algorithmes en fonction du nombre de sommets avec (probabilité d’arête $p = 0.1$).

6.2 Graphes denses ($p = 0.8$)

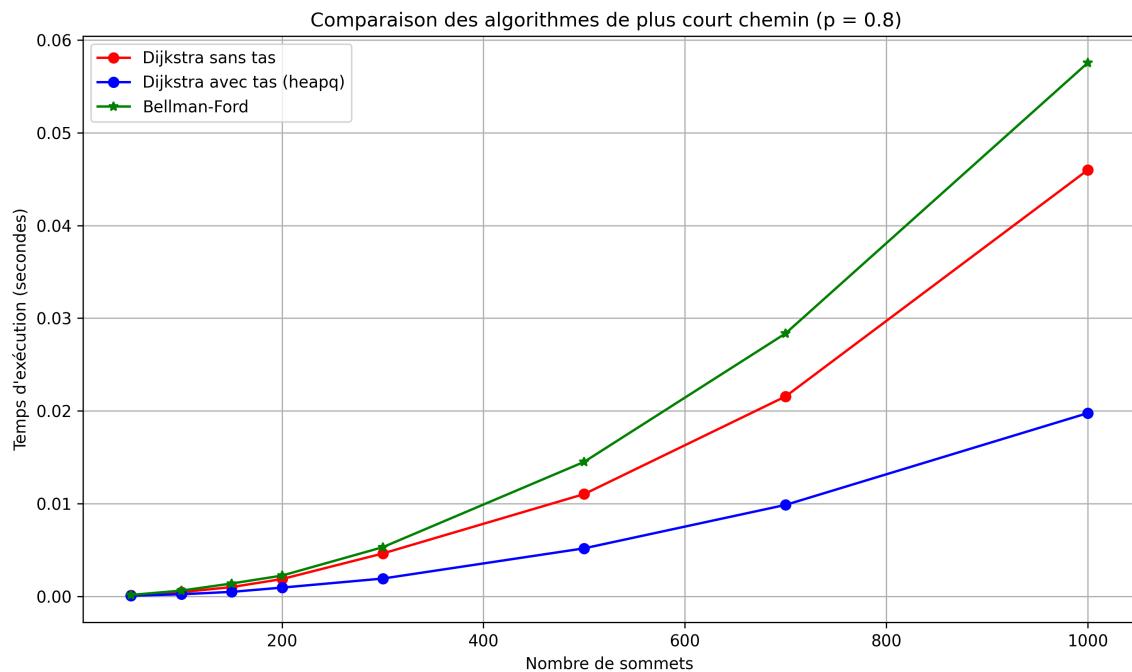


FIGURE 2 – Temps d’exécution des algorithmes en fonction du nombre de sommets avec (probabilité d’arête $p = 0.8$).

6.3 Influence de la densité ($n = 500$)

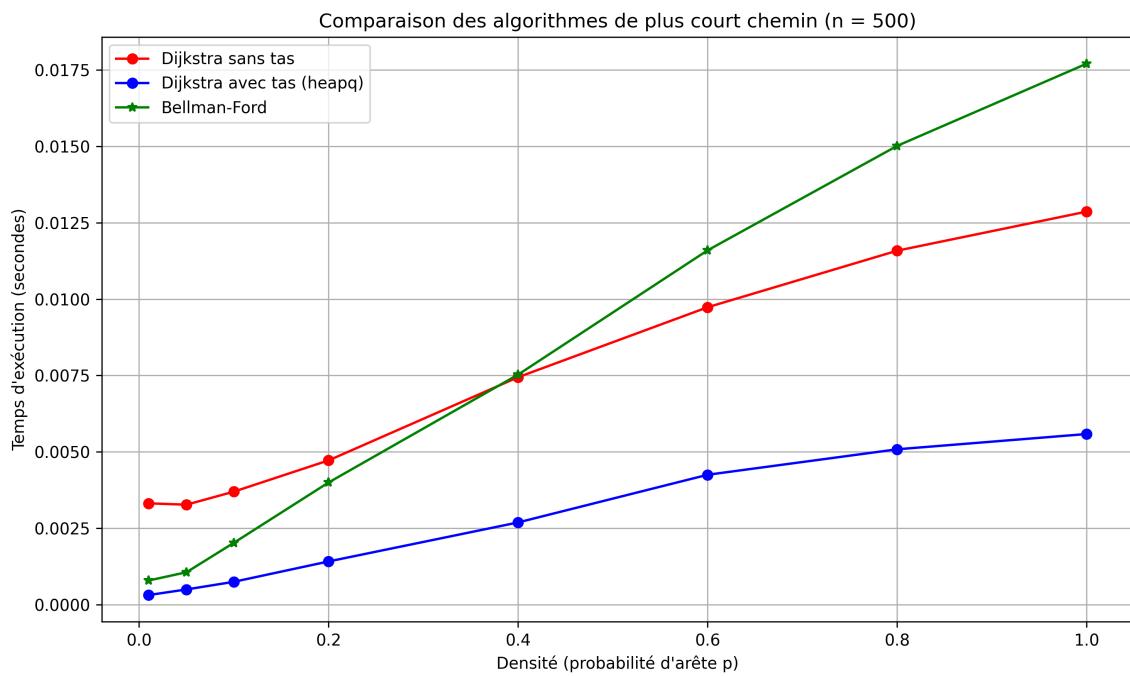


FIGURE 3 – Évolution des performances en fonction de la densité du graphe (nombre de sommets fixé à 500).

6.4 Comparaison entre A* et Dijkstra avec tas

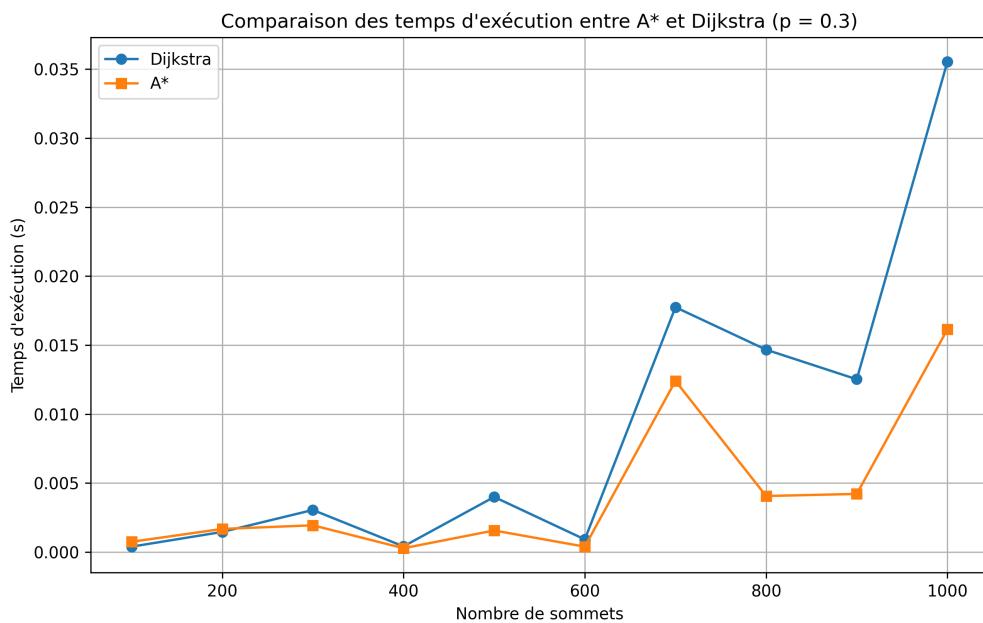


FIGURE 4 – Évolution des performances en fonction du nombre de sommet du graphe (probabilité d'arête $p = 0.3$).

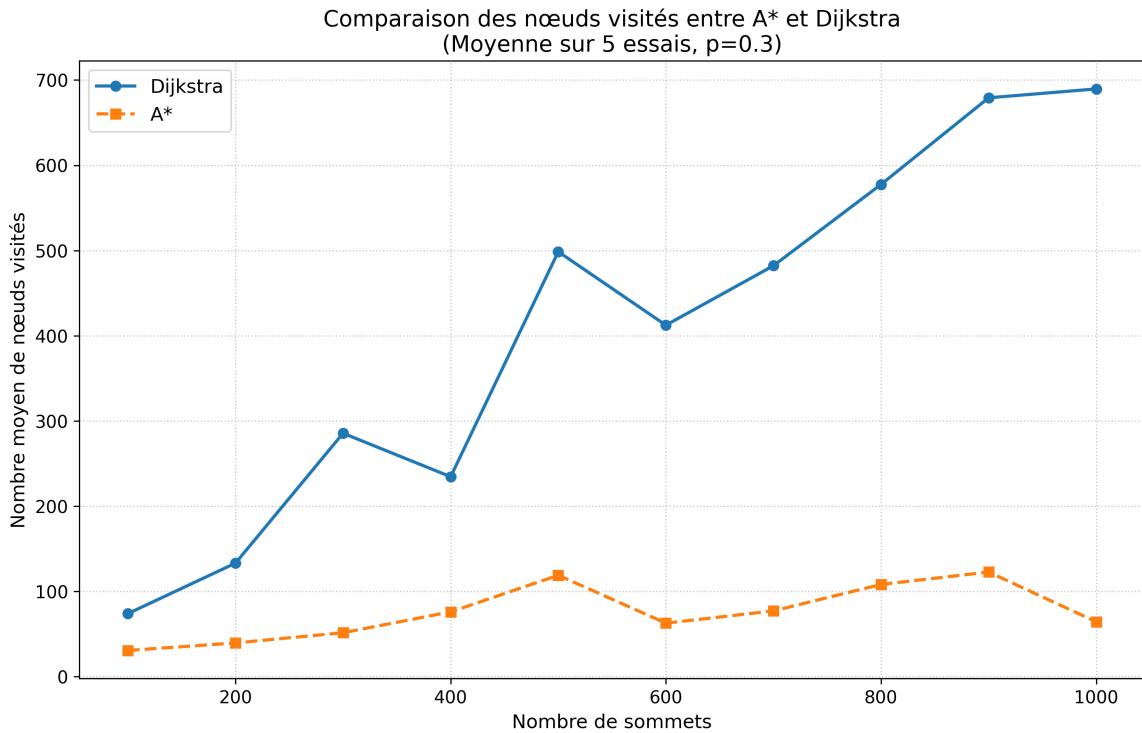


FIGURE 5 – Le nombre de sommets visités en fonction du nombre de sommet du graphe.

7 Analyse Comparative des Algorithmes de Plus Court Chemin

7.1 Impact de la Taille du Graphe (n) à Densité Fixe

Pour un graphe moyennement dense ($p = 0,1$)

- Dijkstra sans tas est plus pénalisant que Bellman-Ford avec *Flag* (arrêt prématuré si aucune relaxation n'est effectuée).
- Cela s'explique par la complexité théorique :
 - Dijkstra sans tas : $\mathcal{O}(n^2)$ (peu efficace pour n grand, même si m est faible).
 - Bellman-Ford (optimisé avec *Flag*) : $\mathcal{O}(nm)$, mais comme $p = 0,1$, on a $m \approx 0,1 \cdot n^2$, donc en pratique aussi $\mathcal{O}(n^3)$.
- Cependant, l'implémentation avec *Flag* peut réduire le temps d'exécution.

Pour un graphe dense ($p = 0,8$)

- Bellman-Ford (avec *Flag*) devient plus lent que Dijkstra sans tas, bien que les deux aient une complexité $\mathcal{O}(n^2)$ (car $m \approx 0,8 \cdot n^3$).
- Dijkstra avec tas (`heapq`) montre une évolution quasi-linéaire : $\mathcal{O}(\cdot n^2 \log n)$, ce qui le rend bien plus efficace pour les grands graphes denses.

Observation clé

- Pour les graphes peu denses ($p = 0,1$), Bellman-Ford (avec *Flag*) peut être plus rapide que Dijkstra sans tas, bien que les deux soient en $\mathcal{O}(n^2)$.
- Pour les graphes denses ($p = 0,8$), Dijkstra sans tas surpasse Bellman-Ford, mais Dijkstra avec tas reste le meilleur choix.

7.2 Effet de la Densité (p) pour $n = 500$

Pour $p < 0,4$ (graphe peu dense)

- Bellman-Ford (avec *Flag*) est plus approprié que Dijkstra sans tas car $\mathcal{O}(nm)$ reste acceptable lorsque m est faible.
- Dijkstra avec tas est légèrement plus lent que Bellman-Ford pour de très faibles densités (à cause du coût du tas).

Pour $p > 0,4$ (graphe dense)

- Bellman-Ford devient désavantageux : $m \approx \mathcal{O}(n^2)$ et donc $\mathcal{O}(nm) \approx \mathcal{O}(n^3)$.
- Dijkstra sans tas reste stable ($\mathcal{O}(n^2)$), mais Dijkstra avec tas ($\mathcal{O}(m \log n)$) devient clairement le plus efficace.

Remarque sur Bellman-Ford sans *Flag*

- Sans optimisation (pas d'arrêt précoce), Bellman-Ford est lent et imprévisible, car il effectue systématiquement $n - 1$ itérations même si aucune relaxation n'est nécessaire.
Pour un graphe de $n = 1000$ sommets et une densité $p = 0,8$, l'algorithme de Bellman-Ford s'exécute en :
 - Avec l'optimisation par flag : 0,06 secondes
 - Sans l'optimisation par flag : 19,8406 secondes

Comparaison entre A* et Dijkstra avec tas

On remarque que l'algorithme A* avec l'heuristique distance euclidienne est plus rapide que Dijkstra avec tas. En effet, A* améliore Dijkstra en ajoutant une heuristique $h(n)$ qui estime la distance restante jusqu'à la cible. Il explore d'abord les sommets qui semblent les plus prometteurs, ce qui le rend souvent plus rapide en pratique que Dijkstra, à condition que l'heuristique soit admissible (elle ne surestime jamais le coût). A* explore moins de sommets "inutiles" que Dijkstra. En résumé, A* est à privilégier lorsqu'on connaît la cible et qu'une bonne estimation de la distance est possible, tandis que Dijkstra est plus général et robuste lorsqu'aucune heuristique fiable n'est disponible.

8 Recommandations d'Utilisation

Pour les graphes denses ($p > 0,4$)

- Dijkstra avec tas est toujours le meilleur choix.
- Éviter Bellman-Ford, sauf en cas d'arêtes négatives.

Pour les graphes peu denses ($p < 0,4$)

- Bellman-Ford (avec *Flag*) peut être plus rapide que Dijkstra sans tas.
- Dijkstra avec tas reste une bonne option, mais son coût $\mathcal{O}(m \log n)$ peut être légèrement moins performant que Bellman-Ford pour de très petites densités.

Cas particuliers

- Si le graphe contient des arêtes de poids négatifs, Bellman-Ford (avec *Flag*) est obligatoire (Dijkstra ne fonctionne pas).
- Pour les très petits graphes ($n < 200$), Dijkstra sans tas peut être suffisant.
- L'algorithme A* est dirigé vers la cible et est souvent plus efficace que l'algorithme de Dijkstra dans les cas où l'on cherche un chemin vers une destination spécifique.

9 Conclusion

- Dijkstra avec tas est l'algorithme le plus polyvalent pour les graphes sans arêtes négatives car le tas minimise le coût d'extraction du sommet minimum. Il est le plus efficace parmi les trois algorithmes (Dijkstra avec/sans tas et Bellmann-Ford). Mais l'algorithme A*, avec une heuristique admissible et bien choisie, peut surpasser Dijkstra en vitesse d'exécution, tout en garantissant le bon résultat.
- Bellman-Ford (avec *Flag*) est utile pour les graphes peu denses ou avec poids négatifs, mais à éviter pour les graphes denses avec poids positifs.
- Dijkstra sans tas l'algorithme le plus efficace dans des cas très spécifiques, petits graphes par exemple grâce à sa simplicité d'implémentation.

Algorithme	Avantage	Inconvénient	Meilleur contexte d'utilisation
Dijkstra avec tas	Très rapide même sur grands graphes	Nécessite une structure de tas	Graphes grands et/ou denses avec poids +
Dijkstra sans tas	Implémentation simple	Moins efficace sur graphes denses	Petits graphes avec poids + ou pour l'enseignement
Bellman-Ford	Gère les poids négatifs	Très lent pour grands/denses graphes	Graphes avec arêtes de poids négatif
A*	Très rapide même sur grands graphes et est dirigé vers la cible	Il faut choisir une heuristique admissible et convenable	Graphes avec arêtes de poids positifs et recherche d'un chemin entre deux points

TABLE 2 – Comparaison qualitative des algorithmes de plus court chemin

10 Application Flutter

Présentation de l'application *Touristo*

Nous avons développé une application mobile nommée **Touristo** sous Flutter, visant à calculer et visualiser les itinéraires optimaux entre plusieurs musées à Paris. L'utilisateur peut choisir un point de départ, une destination, et ajouter des arrêts intermédiaires. L'application exploite les algorithmes de plus court chemin tels que Dijkstra (avec ou sans tas), Bellman-Ford et A*, implémentés dans notre projet.

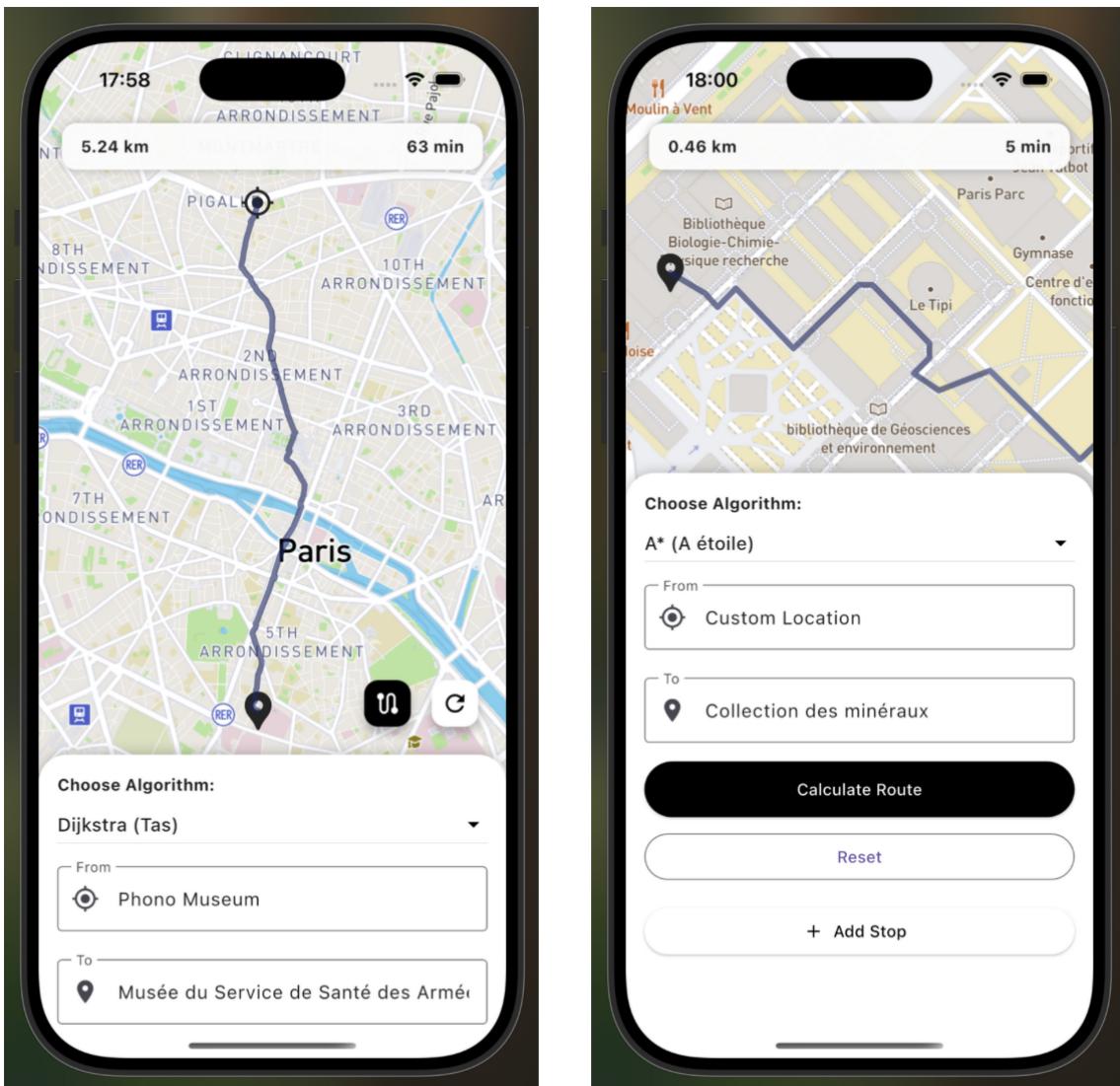


FIGURE 6 – Écran principal de l'application *Touristo*

Touristo offre une carte interactive affichant les musées, les trajets optimaux, les distances estimées et le temps de parcours. L'utilisateur peut également déposer un point de départ personnalisé et zoomer automatiquement sur l'itinéraire calculé. Des boutons permettent de calculer ou de réinitialiser l'itinéraire facilement. (Voir la video dans l'annexe sur github)

Cette première version de *Touristo* constitue une base fonctionnelle pour naviguer entre les musées parisiens. Notre objectif est de la compléter afin d'en faire une application complète, intégrant des services de géolocalisation, des suggestions touristiques, un support multilingue et une interface plus riche pour offrir une excellente expérience.

Représentation du graphe dans l'application

Dans notre application, le graphe est constitué de noeuds représentant principalement les intersections des rues piétonnes de la ville. Certains de ces noeuds possèdent un label spécial de type "musée", ce qui permet de les distinguer comme des points d'intérêt touristiques. Les coordonnées géographiques (latitude et longitude) de ces noeuds ont été extraites à partir de données OpenStreetMap, en utilisant une bibliothèque Python dédiée, puis transformées au format JSON. Ce fichier JSON a ensuite été importé dans l'application et converti en objets Dart correspondant à la structure de notre classe `GraphNode`.

La carte de l'application repose sur une architecture en trois couches :

- 1/ Une couche de fond représentant le plan de la ville sous forme vectorielle (fournie par une bibliothèque comme flutter-map).
- 2/ Une couche graphique intermédiaire où sont tracées les lignes reliant les noeuds selon leurs coordonnées GPS (arêtes).
- 3/ Une couche supérieure interactive, affichant les positions des noeuds (musées, intersections, départ/arrivée) à l'aide d'icônes interactifs.

Tous les codes sources, les données JSON, ainsi que les scripts de génération du graphe sont disponibles dans le dépôt GitHub de l'application mentionné dans l'annexe du rapport.

Problème du voyageur de commerce (TSP)

Le problème du voyageur de commerce consiste à trouver le plus court chemin passant exactement une fois par chaque ville d'un ensemble de n villes, puis revenant à la ville de départ. Ce problème revient à chercher le cycle hamiltonien de coût minimal dans un graphe pondéré. Il est bien connu pour être **NP-difficile**.

Dans le cadre de notre projet, nous utilisons ce problème dans notre application *Touristo* pour planifier une visite optimale entre plusieurs musées. L'utilisateur peut sélectionner plusieurs musées, et l'application calcule l'itinéraire le plus court qui permet de les visiter tous une seule fois, avant de revenir au point de départ ou sans.

Déroulement de l'algorithme de force brute

- Fixer une ville de départ (par exemple la ville 0).
- Générer toutes les permutations possibles des $k - 1$ autres villes.
- Pour chaque permutation :
 - Calculer la distance totale du circuit (en visitant chaque ville dans l'ordre de la permutation, puis en revenant à la ville de départ).
- Conserver la permutation donnant la distance minimale.

Nombre total de permutations testées : $(k - 1)!$

Complexité : $\mathcal{O}((k - 1)! * k * [(n + m)\log(n)])$

avec K : le nombre de musées à visités et $\mathcal{O}((n + m)\log(n))$ la complexité pire cas de A* ou Dijkstra

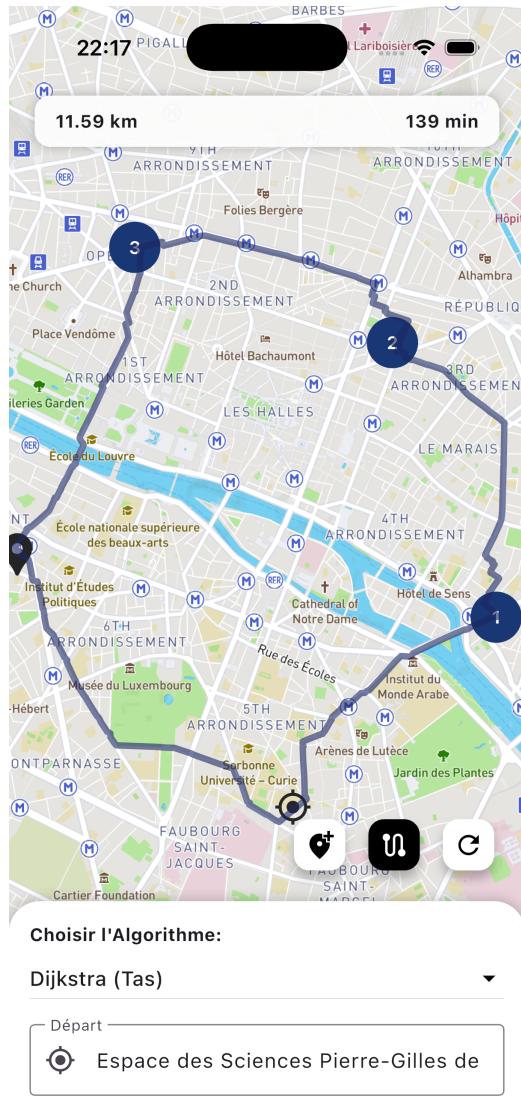


FIGURE 7 – Illustration d'un exemple du voyageur de commerce

Algorithme personnalisé basé sur la distance euclidienne

Nous avons également conçu une version simplifiée de l'algorithme de force brute, dans laquelle les distances entre les points sont estimées à l'aide de la distance euclidienne, selon la formule suivante :

$$d(p_1, p_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Cet algorithme est utilisé dans notre application pour identifier rapidement le musée le plus proche parmi ceux qui restent à visiter, en comparant leur position à notre position actuelle. Cette opération, de complexité $\mathcal{O}(1)$, permet de sélectionner à chaque étape la prochaine destination de manière efficace.

Une fois le musée le plus proche sélectionné, un seul appel à un algorithme de plus court chemin (comme A*) est nécessaire pour tracer l'itinéraire précis entre les deux points. Cette méthode réduit considérablement le nombre total d'appels à l'algorithme, contrairement à l'approche de force brute qui explore toutes les permutations possibles (de complexité $(k - 1)!$).

Bien que cette stratégie ne garantisse pas le circuit optimal global, elle offre une solution beaucoup plus rapide, avec un compromis acceptable entre performance et précision. Le nombre total d'appels à l'algorithme de plus court chemin devient linéaire par rapport au nombre de musées à visiter.

Ici, avec une heuristique du plus proche voisin combinée à un seul appel à A* à chaque étape, on obtient une complexité totale en

$$\sum_{i=0}^{k-1} (k-i) = \sum_{j=1}^k j = \frac{k(k+1)}{2} = \mathcal{O}(k^2)$$

donc nettement plus rapide, mais non optimale globalement.

Les deux algorithmes retournent une liste ordonnée des musées à visiter, puis dans le programme principal, on applique l'algorithme A* ($k - 1$) fois pour relier les musées consécutifs dans le chemin.

10.1 Remarque importante

la version brute force garantit la meilleure solution possible, mais son coût est factoriel : $\mathcal{O}((k - 1)!)$ permutations, donc inutilisable pour $k > 8$ ou 9 . la version gloutonne (plus proche voisin) est très rapide $O(k^2)$, mais greedy, donc souvent sous-optimal. Cependant, il donne souvent de bons résultats (70–95 % d'efficacité selon les cas).

Références

- [1] Wikipedia *Algorithme de Dijkstra*. https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra.
- [2] Wikipedia *Algorithme de Bellman-Ford*. https://fr.wikipedia.org/wiki/Algorithme_de_Bellman-Ford.
- [3] Wikipedia *Algorithme de A star*. https://fr.wikipedia.org/wiki/Algorithme_A*.
- [4] Wikipedia *Voyageur de commerce*. https://fr.wikipedia.org/wiki/Problème_du_voyageur_de_commerce.
- [5] *Code source des tests avec Python* : <https://github.com/NBBS20/NBBS/blob/main>
- [6] *Code source de Touristo et Video* : https://github.com/Anassddd/Touristo_app.git