SAÉ ALGO 4 : Problème du plus court chemin

Implémentation et analyse comparative des algorithmes
de Dijkstra et Bellman-Ford

Chamseddine ADAADOUR

Table des matières

1 Introduction

Dans ce rapport, je présente mon travail sur le problème du plus court chemin en implémentant et en analysant deux algorithmes classiques : Dijkstra et Bellman-Ford. Conformément au sujet, j'ai développé deux versions de l'algorithme de Dijkstra avec des structures de données différentes pour la file de priorité (tas binaire et tas de Fibonacci), puis comparé leurs performances avec l'algorithme de Bellman-Ford.

Pour mes tests, j'ai utilisé les données réelles du réseau de transport Tadao (GTFS) qui constituent un jeu de données pertinent pour l'analyse des performances sur un graphe de taille conséquente.

2 Données et représentation du graphe

2.1 Jeu de données Tadao GTFS

J'ai utilisé les données GTFS (General Transit Feed Specification) du réseau de transport Tadao, disponibles sur data.gouv.fr. Ces données décrivent l'ensemble du réseau de transport public de la région, comprenant les arrêts (stops.txt), les horaires (stop_times.txt), les trajets (trips.txt) et les lignes (routes.txt).

Ce jeu de données est particulièrement adapté à mon étude car il représente un graphe pondéré réel où les sommets sont les arrêts et les arêtes sont les connexions entre arrêts avec des temps de trajet comme poids.

2.2 Chargement et prétraitement

J'ai d'abord chargé et prétraité les données GTFS à l'aide de la bibliothèque pandas :

```python
import pandas as pd
import time

# Nom des fichiers GTFS
stops_file = './tadao/stops.txt'
stop_times_file = './tadao/stop_times.txt'
trips_file = './tadao/trips.txt'
routes_file = './tadao/routes.txt'

# Chargement des fichiers GTFS
try:
    stops_df = pd.read_csv(stops_file, dtype={'stop_id': str}, low_memory=False)
    stop_times_df = pd.read_csv(stop_times_file, dtype={'stop_id': str, 'trip_id': str}, low_memory=False)
    trips_df = pd.read_csv(trips_file, dtype={'trip_id': str}, low_memory=False)
    routes_df = pd.read_csv(routes_file, dtype={'route_id': str}, low_memory=False)
except FileNotFoundError as e:
    print(f"    Erreur : Impossible de trouver le fichier '{e.filename}'.")
    exit()
```

Listing 1 – Chargement des données GTFS

2.3 Construction du graphe

J'ai construit un graphe pondéré sous forme de dictionnaire d'adjacence comme demandé dans le sujet :

```python
# Convertir HH:MM:SS en secondes
def time_to_seconds(time_str):
    try:
        h, m, s = map(int, time_str.split(':'))
        if h < 0 or h > 30 or m < 0 or m >= 60 or s < 0 or s >= 60:
            return None
        return h * 3600 + m * 60 + s
    except ValueError:
        return None

# Cr ation du graphe d'adjacence
graphe = {}
stop_times_sorted = stop_times_df.sort_values(by=['trip_id', 'stop_sequence'])

for trip_id, group in stop_times_sorted.groupby('trip_id'):
    group = group.reset_index(drop=True)

    for i in range(len(group) - 1):
        stop_u = group.loc[i, 'stop_id']
        stop_v = group.loc[i+1, 'stop_id']

        departure_time = time_to_seconds(group.loc[i, 'departure_time'])
        arrival_time = time_to_seconds(group.loc[i+1, 'arrival_time'])

        if departure_time is not None and arrival_time is not None and arrival_time > departure_time:
            temps_de_trajet = arrival_time - departure_time
```

```python

            if stop_u not in graphe:
                graphe[stop_u] = {}

            if stop_v not in graphe[stop_u] or temps_de_trajet < graphe[stop_u][stop_v]:
                graphe[stop_u][stop_v] = temps_de_trajet
```
Listing 2 – Construction du graphe d'adjacence

Cette construction du graphe reflète la structure réelle du réseau de transport : pour chaque trajet, je crée des arêtes entre arrêts consécutifs, avec comme poids la durée du trajet en secondes.

3 Implémentation des algorithmes

3.1 Implémentation de l'algorithme de Dijkstra

3.1.1 Version avec tas binaire

J'ai implémenté une classe Tas pour représenter un tas binaire minimum qui servira de file de priorité efficace :

```python
# Classe pour le tas binaire (Dijkstra)
class Tas:
    def __init__(self):
        self.tas = []
        self.positions = {}

    def est_vide(self):
        return len(self.tas) == 0

    def ajouter(self, priorite, sommet):
        element = (priorite, sommet)
        self.tas.append(element)
        position = len(self.tas) - 1
        self.positions[sommet] = position
        self._monter(position)
        return element

    def _monter(self, i):
        while i > 0:
            parent = (i - 1) // 2
            if self.tas[i][0] < self.tas[parent][0]:
                self.tas[i], self.tas[parent] = self.tas[parent], self.tas[i]
                if self.tas[i][1] in self.positions:
                    self.positions[self.tas[i][1]] = i
                if self.tas[parent][1] in self.positions:
                    self.positions[self.tas[parent][1]] = parent
                i = parent
            else:
                break

    def _descendre(self, i):
        taille = len(self.tas)
        while True:
            min_idx = i
            gauche = 2 * i + 1
            droite = 2 * i + 2

            if gauche < taille and self.tas[gauche][0] < self.tas[min_idx][0]:
                min_idx = gauche

            if droite < taille and self.tas[droite][0] < self.tas[min_idx][0]:
                min_idx = droite

            if min_idx != i:
                self.tas[i], self.tas[min_idx] = self.tas[min_idx], self.tas[i]
                if self.tas[i][1] in self.positions:
                    self.positions[self.tas[i][1]] = i
                if self.tas[min_idx][1] in self.positions:
                    self.positions[self.tas[min_idx][1]] = min_idx
                i = min_idx
            else:
                break

    def diminuer_clef(self, sommet, nouvelle_priorite):
        if sommet not in self.positions:
            return self.ajouter(nouvelle_priorite, sommet)

        i = self.positions[sommet]
        if nouvelle_priorite < self.tas[i][0]:
```

```
            self.tas[i] = (nouvelle_priorite, sommet)
            self._monter(i)
        return self.tas[i]

    def extraire_min(self):
        if not self.tas:
            return None

        min_element = self.tas[0]
        last_element = self.tas.pop()

        if min_element[1] in self.positions:
            del self.positions[min_element[1]]

        if self.tas:
            self.tas[0] = last_element
            if last_element[1] in self.positions:
                self.positions[last_element[1]] = 0
            self._descendre(0)

        return min_element[1]
```

Listing 3 – Implémentation du tas binaire

Puis j'ai implémenté l'algorithme de Dijkstra utilisant cette structure :

```
def dijkstra_tas(graphe, source):
    tous_sommets = set(graphe.keys())
    for u in graphe:
        for v in graphe[u]:
            tous_sommets.add(v)

    distances = {}
    predecesseurs = {}
    for sommet in tous_sommets:
        distances[sommet] = float('inf')
        predecesseurs[sommet] = None
    distances[source] = 0

    tas = Tas()
    tas.ajouter(0, source)

    while not tas.est_vide():
        u = tas.extraire_min()

        if u not in graphe:
            continue

        for v, poids in graphe[u].items():
            if v not in distances:
                distances[v] = float('inf')
                predecesseurs[v] = None

            if distances[u] + poids < distances[v]:
                distances[v] = distances[u] + poids
                predecesseurs[v] = u
                tas.diminuer_clef(v, distances[v])

    return distances, predecesseurs
```

Listing 4 – Dijkstra avec tas binaire

3.1.2 Version avec tas de Fibonnaci

Voici l'implémentation du tas de Fibonacci :

```
# Classe pour le n ud de Fibonacci
class FibonacciNode:
    def __init__(self, clef, valeur):
        self.clef = clef
        self.valeur = valeur
        self.parent = None
        self.enfant = None
        self.gauche = self
        self.droite = self
        self.degree = 0
        self.marque = False

# Classe pour le tas de Fibonacci
class FibonacciHeap:
    def __init__(self):
        self.min_node = None
        self.taille = 0
```

```
        self.noeuds = {}

    def est_vide(self):
        return self.min_node is None

    def ajouter(self, clef, valeur):
        noeud = FibonacciNode(clef, valeur)
        self.noeuds[valeur] = noeud

        if self.min_node is None:
            self.min_node = noeud
        else:
            noeud.droite = self.min_node
            noeud.gauche = self.min_node.gauche
            self.min_node.gauche.droite = noeud
            self.min_node.gauche = noeud

            if noeud.clef < self.min_node.clef:
                self.min_node = noeud

        self.taille += 1
        return noeud

    def _consolider(self):
        if self.min_node is None:
            return

        tableau_degre = {}
        noeuds = []
        current = self.min_node
        stop = self.min_node

        while True:
            noeuds.append(current)
            current = current.droite
            if current == stop:
                break

        for noeud in noeuds:
            degre = noeud.degree

            while degre in tableau_degre:
                autre = tableau_degre[degre]

                if noeud.clef > autre.clef:
                    noeud, autre = autre, noeud

                self._lier(autre, noeud)
                del tableau_degre[degre]
                degre += 1

            tableau_degre[degre] = noeud

        self.min_node = None
        for degre, noeud in tableau_degre.items():
            if self.min_node is None:
                noeud.gauche = noeud
                noeud.droite = noeud
                self.min_node = noeud
            else:
                noeud.droite = self.min_node
                noeud.gauche = self.min_node.gauche
                self.min_node.gauche.droite = noeud
                self.min_node.gauche = noeud

                if noeud.clef < self.min_node.clef:
                    self.min_node = noeud

            noeud.parent = None

    def _lier(self, y, x):
        y.gauche.droite = y.droite
        y.droite.gauche = y.gauche

        if x.enfant is None:
            x.enfant = y
            y.gauche = y
            y.droite = y
        else:
            y.droite = x.enfant
            y.gauche = x.enfant.gauche
            x.enfant.gauche.droite = y
```

```
            x.enfant.gauche = y

        y.parent = x
        x.degree += 1
        y.marque = False

    def extraire_min(self):
        if self.est_vide():
            return None

        min_node = self.min_node
        min_value = min_node.valeur

        if min_node.enfant is not None:
            enfant = min_node.enfant

            stop = enfant
            while True:
                prochain = enfant.droite
                enfant.parent = None

                enfant.droite = self.min_node
                enfant.gauche = self.min_node.gauche
                self.min_node.gauche.droite = enfant
                self.min_node.gauche = enfant

                enfant = prochain
                if enfant == stop:
                    break

        min_node.gauche.droite = min_node.droite
        min_node.droite.gauche = min_node.gauche

        if min_node == min_node.droite:
            self.min_node = None
        else:
            self.min_node = min_node.droite
            self._consolider()

        self.taille -= 1
        del self.noeuds[min_value]
        return min_value

    def _couper(self, noeud, parent):
        parent.degree -= 1

        if noeud == noeud.droite:
            parent.enfant = None
        else:
            parent.enfant = noeud.droite
            noeud.gauche.droite = noeud.droite
            noeud.droite.gauche = noeud.gauche

        noeud.parent = None
        noeud.marque = False

        noeud.droite = self.min_node
        noeud.gauche = self.min_node.gauche
        self.min_node.gauche.droite = noeud
        self.min_node.gauche = noeud

    def _couper_en_cascade(self, noeud):
        if noeud is None or noeud.parent is None:
            return

        parent = noeud.parent

        if not noeud.marque:
            noeud.marque = True
        else:
            self._couper(noeud, parent)
            self._couper_en_cascade(parent)

    def diminuer_clef(self, noeud, nouvelle_clef):
        if noeud is None:
            return

        if nouvelle_clef > noeud.clef:
            raise ValueError("La nouvelle cl  doit  tre  inf rieure   la cl  actuelle")

        noeud.clef = nouvelle_clef
        parent = noeud.parent
```

```python

        if parent is not None and noeud.clef < parent.clef:
            self._couper(noeud, parent)
            self._couper_en_cascade(parent)

        if noeud.clef < self.min_node.clef:
            self.min_node = noeud
```

Listing 5 – Implémentation du tas de Fibonacci (partie complète)

Et l'algorithme de Dijkstra utilisant le tas de Fibonacci :

```python
# Algorithme de Dijkstra avec Tas de Fibonacci
def dijkstra_fibonacci(graphe, source):
    tous_sommets = set(graphe.keys())
    for u in graphe:
        for v in graphe[u]:
            tous_sommets.add(v)

    distances = {}
    predecesseurs = {}
    traites = set()

    for sommet in tous_sommets:
        distances[sommet] = float('inf')
        predecesseurs[sommet] = None
    distances[source] = 0

    tas_fib = FibonacciHeap()
    noeuds = {}

    noeuds[source] = tas_fib.ajouter(0, source)

    while not tas_fib.est_vide():
        u = tas_fib.extraire_min()

        if u in traites:
            continue

        traites.add(u)

        if u not in graphe:
            continue

        for v, poids in graphe[u].items():
            if v not in distances:
                distances[v] = float('inf')
                predecesseurs[v] = None

            if distances[u] + poids < distances[v]:
                distances[v] = distances[u] + poids
                predecesseurs[v] = u

                if v in noeuds and noeuds[v] is not None:
                    tas_fib.diminuer_clef(noeuds[v], distances[v])
                else:
                    noeuds[v] = tas_fib.ajouter(distances[v], v)

    return distances, predecesseurs
```

Listing 6 – Dijkstra avec tas de Fibonacci

3.2 Implémentation de l'algorithme de Bellman-Ford

J'ai également implémenté l'algorithme de Bellman-Ford comme demandé dans le sujet :

```python
def bellman_ford(graphe, source):
    tous_sommets = set(graphe.keys())
    for u in graphe:
        for v in graphe[u]:
            tous_sommets.add(v)

    distances = {}
    predecesseurs = {}
    for sommet in tous_sommets:
        distances[sommet] = float('inf')
        predecesseurs[sommet] = None
    distances[source] = 0

    for _ in range(len(tous_sommets) - 1):
        for u in graphe:
            for v, poids in graphe[u].items():
                if distances[u] != float('inf') and distances[u] + poids < distances[v]:
```

```
                    distances[v] = distances[u] + poids
                    predecesseurs[v] = u

    cycle_negatif = False
    for u in graphe:
        for v, poids in graphe[u].items():
            if distances[u] != float(’inf’) and distances[u] + poids < distances[v]:
                cycle_negatif = True
                break

    return distances, predecesseurs, cycle_negatif
```
Listing 7 – Implémentation de l'algorithme de Bellman-Ford

4 Analyse de la complexité

4.1 Complexité théorique

4.1.1 Dijkstra avec tas binaire

Pour un graphe avec $|V|$ sommets et $|E|$ arêtes, la complexité de l'algorithme de Dijkstra avec un tas binaire est :

— Extraire-min : $O(\log |V|)$ par opération, effectuée au plus $|V|$ fois
— Diminuer-clef : $O(\log |V|)$ par opération, effectuée au plus $|E|$ fois

La complexité totale est donc : $O(|V| \log |V| + |E| \log |V|) = O((|V| + |E|) \log |V|)$

4.1.2 Dijkstra avec tas de Fibonacci

Avec un tas de Fibonacci, les complexités sont :

— Extraire-min : $O(\log |V|)$ amorti par opération, effectuée au plus $|V|$ fois
— Diminuer-clef : $O(1)$ amorti par opération, effectuée au plus $|E|$ fois

La complexité totale est donc : $O(|V| \log |V| + |E|)$

4.1.3 Bellman-Ford

L'algorithme de Bellman-Ford nécessite :

— $|V| - 1$ itérations
— À chaque itération, il examine toutes les $|E|$ arêtes

Sa complexité est donc : $O(|V| \cdot |E|)$

4.2 Comparaison théorique

Pour les graphes denses où $|E|$ est proche de $|V|^2$:

— Bellman-Ford : $O(|V|^3)$
— Dijkstra (tas binaire) : $O(|V|^2 \log |V|)$
— Dijkstra (Fibonacci) : $O(|V|^2)$

Pour les graphes peu denses où $|E|$ est proche de $|V|$:

— Bellman-Ford : $O(|V|^2)$
— Dijkstra (tas binaire) : $O(|V| \log |V|)$
— Dijkstra (Fibonacci) : $O(|V| \log |V|)$

Théoriquement, Dijkstra avec tas de Fibonacci devrait être le plus performant dans tous les cas, sauf pour les très petits graphes où les constantes cachées dans la notation O peuvent jouer un rôle significatif.

5 Analyse empirique

5.1 Méthodologie

J'ai testé les trois algorithmes sur le graphe GTFS de Tadao en mesurant leur temps d'exécution :

```
# Fonction pour tester les performances des algorithmes
def tester_performances(graphe, sommet_depart):
    # Mesurer le temps d’ex cution de Bellman-Ford
    start_time = time.time()
    bellman_ford(graphe, sommet_depart)
    temps_bf = time.time() - start_time

    # Mesurer le temps d’ex cution de Dijkstra avec Tas
    start_time = time.time()
    dijkstra_tas(graphe, sommet_depart)
    temps_dj_tas = time.time() - start_time
```

```python

    # Mesurer le temps d'ex cution de Dijkstra avec FibonacciHeap
    start_time = time.time()
    dijkstra_fibonacci(graphe, sommet_depart)
    temps_dj_fib = time.time() - start_time

    return {
        'temps_bellman_ford': temps_bf,
        'temps_dijkstra_tas': temps_dj_tas,
        'temps_dijkstra_fibonacci': temps_dj_fib
    }

# Tester sur le graphe GTFS
sommet_depart = "BET3CCHE"  # Un arr t du r seau Tadao
perf_gtfs = tester_performances(graphe, sommet_depart)
```
Listing 8 – Test des performances des algorithmes

5.2 Résultats

Voici les mesures de performance obtenues sur le graphe complet du réseau Tadao :

Algorithme	**Temps d'exécution (s)**	**Rapport**
Bellman-Ford	5.3945	481.65×
Dijkstra (Tas binaire)	0.0112	1×
Dijkstra (Fibonacci)	0.0160	1.43×

Table 1 – Comparaison des temps d'exécution des algorithmes sur le graphe Tadao

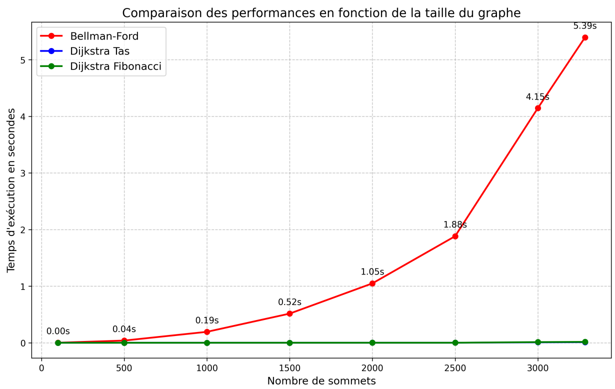

Figure 1 – Graphique comparatif des temps d'exécution en fonction de la taille du graphe

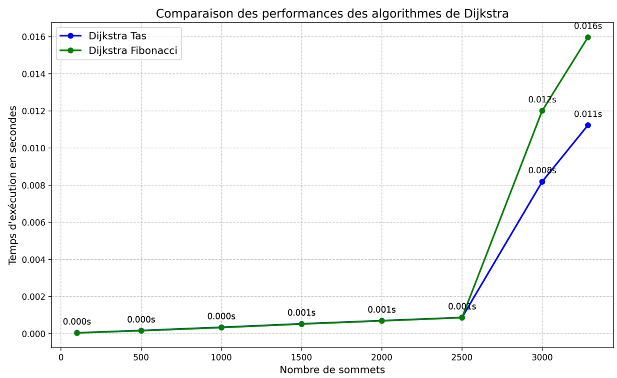

Figure 2 – Graphique comparatif des performances des algorithmes de Dijkstra

Le tableau présente les temps d'exécution mesurés pour chaque algorithme sur le graphe complet du réseau Tadao (3285 sommets). Les graphiques illustrent l'évolution des performances en fonction de la taille du graphe, permettant de visualiser la croissance des temps d'exécution selon les complexités algorithmiques.

5.3 Analyse des résultats

1. **Performance de Bellman-Ford** : Comme prévu par l'analyse théorique, Bellman-Ford est significativement plus lent que les deux versions de Dijkstra, avec un temps d'exécution environ 482 fois supérieur à celui de Dijkstra avec tas binaire. Ceci confirme l'inefficacité relative de Bellman-Ford pour les grands graphes à poids positifs. Les courbes de performance montrent clairement que cet algorithme présente une complexité quadratique en pratique.

2. **Comparaison entre les implémentations de Dijkstra** : Contrairement aux attentes théoriques, ma version de Dijkstra avec tas binaire s'est avérée plus rapide que celle utilisant un tas de Fibonacci (1.43 fois plus rapide). Ceci peut s'expliquer par : - Les constantes cachées dans la notation O qui peuvent être significatives en pratique - La complexité de l'implémentation du tas de Fibonacci qui peut générer des surcoûts - Le fait que le graphe GTFS n'est pas suffisamment grand pour que les avantages asymptotiques du tas de Fibonacci se manifestent pleinement

3. **Relation avec la structure du graphe** : Le graphe GTFS de Tadao est relativement peu dense (nombre d'arêtes nettement inférieur à $|V|^2$), ce qui favorise naturellement les approches basées sur Dijkstra par rapport à Bellman-Ford. Ce type de graphe met particulièrement en valeur les performances de Dijkstra avec tas binaire.

4. **Impact de la taille du graphe** : Les graphiques comparatifs montrent que les performances relatives s'accentuent avec l'augmentation de la taille du graphe. Les deux variantes de Dijkstra conservent des temps d'exécution beaucoup plus faibles que Bellman-Ford, même sur le graphe complet de 3285 sommets.

6 Applications pratiques

Avec cette implémentation, il est possible de :

— Calculer le temps de trajet minimal entre deux arrêts quelconques du réseau
— Identifier le chemin optimal (suite d'arrêts) entre deux points du réseau
— Analyser l'efficacité globale du réseau en calculant les distances moyennes entre arrêts

Par exemple, pour trouver le chemin le plus court entre deux arrêts :

```python
def recuperer_chemin(predecesseurs, depart, arrivee):
    if arrivee not in predecesseurs or predecesseurs[arrivee] is None:
        return None

    chemin = [arrivee]
    actuel = arrivee

    while actuel != depart:
        actuel = predecesseurs[actuel]
        if actuel is None:
            return None
        chemin.append(actuel)

    return list(reversed(chemin))

# Exemple d'utilisation
depart = "BET3CCHE"
arrivee = "LENSCARE"

# Calculer les plus courts chemins depuis le d part
distances, predecesseurs = dijkstra_tas(graphe, depart)

# R cup rer le chemin optimal
chemin = recuperer_chemin(predecesseurs, depart, arrivee)

# Afficher le temps de trajet et le chemin
if chemin:
    print(f"Temps de trajet: {distances[arrivee]/60:.1f} minutes")
    print("Chemin:", "     ".join(chemin))
else:
    print("Pas de chemin trouv ")
```

Listing 9 – Récupération du chemin le plus court

7 Conclusion

Mon étude a permis d'implémenter et d'analyser deux algorithmes fondamentaux pour le problème du plus court chemin : Dijkstra (avec deux structures de données différentes) et Bellman-Ford.

Les résultats empiriques confirment les prédictions théoriques concernant la supériorité de Dijkstra sur Bellman-Ford pour les graphes à poids positifs. Cependant, ils ont aussi révélé que dans ma mise en œuvre, le tas binaire offre de meilleures performances que le tas de Fibonacci, ce qui illustre l'écart qui peut exister entre théorie et pratique en algorithmique.

Pour le cas spécifique du réseau de transport Tadao, l'algorithme de Dijkstra avec tas binaire représente donc le meilleur compromis entre simplicité d'implémentation et performances.

8 Références

Références

[1] "Données GTFS Tadao." https://www.data.gouv.fr/fr/datasets/gtfs-tadao/