1A CI INFO

Algorithme de Dijkstra : Cours détaillé

Introduction

Contexte et objectifs du cours

L'algorithme de Dijkstra est un algorithme fondamental en informatique pour résoudre le problème du plus court chemin dans un graphe pondéré à arêtes non négatives. Il est largement utilisé dans diverses applications telles que les réseaux informatiques, la navigation GPS, la planification de réseaux logistiques, etc.

L'objectif de ce cours est de vous fournir une compréhension approfondie de l'algorithme de Dijkstra, y compris sa justification théorique, son fonctionnement détaillé, son implémentation et ses applications pratiques.

Plan du cours

1. **Introduction à la théorie des graphes**
   - Définitions de base
   - Types de graphes
   - Représentation des graphes en informatique
2. **Le problème du plus court chemin**
   - Formulation du problème
   - Applications pratiques
   - Contraintes et considérations
3. **Présentation de l'algorithme de Dijkstra**
   - Historique
   - Principe de base
   - Conditions d'utilisation
4. **Description détaillée de l'algorithme**
   - Fonctionnement pas à pas
   - Pseudocode
   - Analyse de complexité
5. **Exemples d'application**
   - Résolution d'un exemple simple
   - Analyse étape par étape
   - Visualisation graphique
6. **Implémentation pratique**
   - Structures de données utilisées
   - Optimisations possibles
   - Code exemple (pseudo-code ou langage de programmation)
7. **Limitations et alternatives**
   - Cas des poids négatifs
   - Comparaison avec l'algorithme de Bellman-Ford
   - Autres algorithmes de plus court chemin
8. **Applications avancées**
   - Algorithmes pour les plus courts chemins multiples

- Utilisation dans les systèmes de routage

9. **Conclusion**
   - Récapitulation des points clés
   - Perspectives futures
   - Questions et discussions

---

1. Introduction à la théorie des graphes

1.1 Définitions de base

- **Graphe** : Un graphe G=(V,E) est composé d'un ensemble de sommets V et d'un ensemble d'arêtes E qui relient ces sommets.
- **Sommet (ou nœud)** : Un point du graphe.
- **Arête** : Une connexion entre deux sommets. Peut être orientée (dans un graphe orienté) ou non orientée.
- **Poids** : Une valeur numérique associée à une arête, représentant un coût, une distance, un temps, etc.

1.2 Types de graphes

- **Graphe non orienté** : Les arêtes n'ont pas de direction.
- **Graphe orienté (ou digraphe)** : Les arêtes ont une direction, allant d'un sommet à un autre.
- **Graphe pondéré** : Les arêtes sont associées à des poids.

1.3 Représentation des graphes en informatique

- **Liste d'adjacence** : Pour chaque sommet, on maintient une liste des sommets
- adjacents.
- **Matrice d'adjacence** : Une matrice n×n où n est le nombre de sommets,
- et l'entrée a_{ij} indique la présence (et éventuellement le poids) d'une arête de i vers j.

---

2. Le problème du plus court chemin

2.1 Formulation du problème

Trouver le chemin de coût minimal entre un sommet source S et un sommet destination T dans un graphe pondéré, en cumulant les poids des arêtes traversées.

2.2 Applications pratiques

- **Réseaux routiers** : Trouver le trajet le plus court ou le plus rapide.
- **Réseaux de communication** : Optimisation des routes pour la transmission de données.
- **Logistique** : Planification des itinéraires de livraison.

2.3 Contraintes et considérations

- Les poids doivent être non négatifs pour l'algorithme de Dijkstra.
- Le graphe peut être orienté ou non orienté.

---

3. Présentation de l'algorithme de Dijkstra

3.1 Historique

- Proposé par Edsger W. Dijkstra en 1956 et publié en 1959.

3.2 Principe de base

- L'algorithme maintient un ensemble de sommets pour lesquels le plus court chemin à partir de la source est déjà connu.
- À chaque étape, il sélectionne le sommet non visité avec la distance minimale provisoire et met à jour les distances des sommets adjacents.

3.3 Conditions d'utilisation

- Les poids des arêtes doivent être non négatifs.

---

4. Description détaillée de l'algorithme

4.1 Fonctionnement pas à pas

1. **Initialisation**
   - Attribuer à chaque sommet une distance initiale : 0 pour la source, infini (∞) pour les autres.
   - Placer tous les sommets dans un ensemble non visité.
2. **Sélection du sommet courant**
   - Sélectionner le sommet non visité avec la distance minimale (au départ, la source).
3. **Mise à jour des distances**
   - Pour chaque voisin non visité du sommet courant :
     - Calculer une distance provisoire : distance du sommet courant + poids de l'arête vers le voisin.
     - Si cette distance est inférieure à la distance actuellement enregistrée pour le voisin, mettre à jour la distance.
4. **Marquer le sommet courant comme visité**
   - Retirer le sommet courant de l'ensemble non visité.
5. **Répéter**
   - Répéter les étapes 2 à 4 jusqu'à ce que tous les sommets soient visités ou que la distance minimale parmi les sommets non visités soit infinie.

4.2 Pseudocode

```
Fonction Dijkstra(Graphe, Source):
  pour chaque sommet v dans Graphe:
    distance[v] ← ∞
    précédent[v] ← non défini
  distance[Source] ← 0
  Q ← ensemble de tous les sommets de Graphe
  tant que Q n'est pas vide:
    u ← sommet dans Q avec la distance minimale
    retirer u de Q
    pour chaque voisin v de u:
```

```
        alt ← distance[u] + poids(u, v)
        si alt < distance[v]:
            distance[v] ← alt
            précédent[v] ← u
    retourner distance, précédent
```

4.3 Analyse de complexité

- **Complexité en temps** :
  - Utilisation d'une file de priorité simple : $O(V^2)$, où V est le nombre de sommets.
  - Utilisation d'une file de priorité avec un tas (heap) : O(ElogV), où E est le nombre d'arêtes.
- **Complexité en espace** : O(V) pour stocker les distances et les prédécesseurs.

---

5. Exemples d'application

5.1 Exemple simple

Considérons le graphe suivant :

- Sommets : A, B, C, D, E
- Arêtes et poids :
  - A-B (4)
  - A-C (2)
  - B-C (5)
  - B-D (10)
  - C-E (3)
  - E-D (4)

5.2 Exécution étape par étape

1. **Initialisation**
   - distance[A]=0, distance[B]=∞, distance[C]=∞, distance[D]=∞, distance[E]=∞
   - Non visités : {A, B, C, D, E}
2. **Sélection de A (distance minimale = 0)**
   - Voisins de A : B, C
     - distance[B]=min(∞, 0+4)=4, précédent[B]=A
     - distance[C]=min(∞, 0+2)=2, précédent[C]=A
   - Visités : {A}
3. **Sélection de C (distance minimale = 2)**
   - Voisins de C : B, E
     - distance[B]=min(4, 2+5)=4 (reste inchangé)
     - distance[E]=min(∞, 2+3)=5, précédent[E]=C
   - Visités : {A, C}
4. **Sélection de B (distance minimale = 4)**
   - Voisins de B : D
     - distance[D]=min(∞, 4+10)=14, précédent[D]=B
   - Visités : {A, C, B}
5. **Sélection de E (distance minimale = 5)**
   - Voisins de E : D
     - distance[D]=min(14, 5+4)=9, précédent[D]=E
   - Visités : {A, C, B, E}

6. **Sélection de D (distance minimale = 9)**
    - Pas de voisins non visités.
    - Visités : {A, C, B, E, D}
7. **Fin de l'algorithme**

5.3 Résultats

- Distances finales depuis A :
    - distance[A]=0
    - distance[B]=4
    - distance[C]=2
    - distance[E]=5
    - distance[D]=9
- Chemins les plus courts :
    - A → B : A-B
    - A → C : A-C
    - A → E : A-C-E
    - A → D : A-C-E-D

---

6. Implémentation pratique

6.1 Structures de données utilisées

- **File de priorité (tas)** : Pour sélectionner efficacement le sommet non visité avec la distance minimale.
- **Tableaux ou dictionnaires** : Pour stocker les distances et les prédécesseurs.

6.2 Optimisations possibles

- Utilisation d'un tas à mise à jour (par exemple, un tas de Fibonacci) pour améliorer la complexité temporelle.
- Éviter les cycles en ne mettant à jour que les distances des sommets non visités.

6.3 Exemple de code en Python

```
import heapq

def dijkstra(graph, source):
    distance = {vertex: float('infinity') for vertex in graph}
    distance[source] = 0
    previous = {vertex: None for vertex in graph}
    queue = [(0, source)]

    while queue:
        current_distance, u = heapq.heappop(queue)
        if current_distance > distance[u]:
            continue
        for neighbor, weight in graph[u].items():
            alt = distance[u] + weight
            if alt < distance[neighbor]:
                distance[neighbor] = alt
```

```
            previous[neighbor] = u
            heapq.heappush(queue, (alt, neighbor))

    return distance, previous
```

6.4 Explication du code

- **Initialisation** : Les distances sont initialisées à l'infini, sauf pour la source.
- **File de priorité** : Utilisation du module heapq pour gérer le tas.
- **Boucle principale** : À chaque itération, on extrait le sommet avec la distance minimale.
- **Relaxation des arêtes** : On met à jour les distances des voisins si un chemin plus court est trouvé.

---

7. Limitations et alternatives

7.1 Cas des poids négatifs

- L'algorithme de Dijkstra ne fonctionne pas correctement si le graphe contient des arêtes avec des poids négatifs.

7.2 Algorithme de Bellman-Ford

- Peut gérer les poids négatifs.
- Complexité temporelle en O(VE).

7.3 Autres algorithmes

- **Algorithme de Floyd-Warshall** : Trouve les plus courts chemins entre toutes les paires de sommets.
- **Algorithme A*** : Utilise une heuristique pour améliorer l'efficacité sur des graphes spécifiques.

---

8. Applications avancées

8.1 Plus courts chemins multiples

- Modifications de l'algorithme pour trouver les k plus courts chemins.

8.2 Systèmes de routage

- Utilisation dans les protocoles de routage comme OSPF (Open Shortest Path First).
- Calcul des chemins dans les réseaux IP.

8.3 Planification et optimisation

- Applications dans la planification des transports, la gestion de l'énergie, etc.

---

9. Conclusion

9.1 Récapitulation des points clés

- L'algorithme de Dijkstra est essentiel pour résoudre le problème du plus court chemin dans les graphes pondérés à poids non négatifs.
- Il utilise une approche gloutonne pour construire progressivement le chemin le plus court vers chaque sommet.
- L'efficacité de l'algorithme dépend des structures de données utilisées, notamment de l'utilisation d'une file de priorité.

Références

- **Edsger W. Dijkstra**, "A Note on Two Problems in Connexion with Graphs", Numerische Mathematik, 1959.
- **Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein**, "Introduction à l'algorithme", MIT Press.

Exercices recommandés

1. **Exercice 1** : Appliquer l'algorithme de Dijkstra sur un graphe donné et tracer les étapes.
2. **Exercice 2** : Implémenter l'algorithme en utilisant différentes structures de données et comparer les performances.
3. **Exercice 3** : Adapter l'algorithme pour trouver le plus court chemin entre toutes les paires de sommets.