Jacquet Samuel
UNamur
IHDCB232 : Algorithmique 1
Travail individuel
Dépôt sur GitHub - Dijkstra

Année académique 2024-2025

Table des matières

Introduction

A.1 Algorithme de Dijkstra : L'algorithme de Dijkstra est utile pour trouver le chemin le moins coûteux entre deux nœuds d'un graphe pondéré. Il nécessite l'utilisation d'une structure de données efficace comme une file de priorité (souvent implémentée avec un tas). Il est important de prouver que les nœuds déjà visités ont leur plus court chemin correctement calculé.

Compréhension de l'algorithme de Dijkstra

> 1. Effectuez des recherches pour comprendre l'algorithme choisi en profondeur. Vous devrez utiliser des ressources académiques ou en ligne pour mieux comprendre son fonctionnement et les théories qui le sous-tendent. Listez vos sources, expliquez en quelques mots pourquoi vous avez le choix de cet algorithme en particulier, et donnez son contexte d'utilisation (dans quels cas concrets est-il typiquement utilisé ?).

Recherche et compréhension

L'algorithme de Dijkstra est un algorithme de recherche de chemin le plus court dans un graphe pondéré. Il a été développé par Edsger W. Dijkstra en 1956 et est largement utilisé en informatique et en mathématiques pour résoudre des problèmes de plus court chemin à partir d'un sommet source vers tous les autres sommets d'un graphe.

Choix de l'algorithme

L'algorithme de Dijkstra est très efficace pour trouver le chemin le plus court dans des graphes pondérés avec des poids non négatifs. Il ne fonctionne pas avec des poids négatifs mais est plus éfficace que Dijkstra en termes de compléxité temporelle, avec une complexité de (O(VE)) contre (O(V^2)) pour Dijkstra (ou (O(E + V \log V)) avec une file de priorité)

L'algorithme est relativement simple à comprendre et à implémenter. Il n'utilise pas d'heuristique pour guider la recherche comme pour le A*. La performance ne dépend pas de l'heuristique utilisée. Il est utilisé dans de nombreux domaines, ce qui en fait un outil polyvalent.

Contexte d'utilisation

L'algorithme de Dijkstra est typiquement utilisé dans les cas suivants :

- **Navigation GPS** : Pour trouver le chemin le plus court entre deux points.
- **Réseaux de télécommunications** : Pour minimiser les délais de transmission des données.
- **Gestion du trafic** : Pour éviter les embouteillages en trouvant les routes les plus rapides.
- **Optimisation fiscale** : Pour optimiser les routes fiscales des multinationales (Delhaye, V., 2015).

En bref, l'algorithme de Dijkstra est un outil puissant et polyvalent pour résoudre des problèmes de plus court chemin dans divers contextes pratiques.

Spécification

> 2. Spécifiez le problème : explicitez les préconditions et postconditions formellement (en respectant les notations mathématiques vues au cours).

environnement

```
int V, E, src;
Graph* graph; // Représentation du graphe sous forme de liste d'adjacence
int dist[V]; // Tableau des distances
int pred[V]; // Tableau des prédécesseurs
```

Soit (G = (V, E)) un graphe où (V) est l'ensemble des sommets (($|V| = V_0$)) et (E \subseteq V \times V) l'ensemble des arêtes pondérées par une fonction (w : E \to \mathbb{R}^+).

Précondition (P)

$$\text{pré} \equiv \text{Graph } G = (V, E) \text{ représenté par une liste d'adjacence, avec } V = V_0 > 0, E = E_0 \geq 0, \qquad \text{src} \in V, \quad \forall (u, v) \in E : w(u, v) \geq 0, \ \text{dist}[\text{src}] = 0, \quad \forall v \neq$$

- V_0 : Nombre de sommets dans le graphe ($V_0 > 0$).
- E_0 : Nombre d'arêtes ($E_0 \geq 0$).
- src : Le sommet source, $\text{src} \in [0, V - 1]$.
- $\text{dist}[\text{src}]$: La distance minimale estimée depuis le sommet source vers lui-même, initialisée à 0.
- $\text{dist}[\text{v}]$: Les distances minimales estimées des autres sommets, initialisées à ∞.
- $\text{sptSet}[\text{v}]$: Booléen indiquant si le sommet v a été traité dans l'arbre des plus courts chemins, initialisé à false.
- $\text{Adj}[\text{u}]$: Liste des voisins du sommet u, utilisée pour représenter les arcs sortants. Les poids des arêtes $w(u, v)$ sont non négatifs ($w(u, v) \geq 0$).
  Cette formulation se base uniquement sur les arêtes existantes dans E_0, ce qui correspond à l'utilisation d'une liste d'adjacence comme structure de données.

> **Remarque** : Je choisis la liste d'adjacence car la matrice d'adjacence :
>
> $$V_0 > 0 \land \forall i, j : 0 \leq i, j < V_0 : \text{graph}[i][j] \geq 0$$

> nécessite $O({V_0}^2)$ d'espace. Cela est dû au fait que `graph[i][j]` représente le poids de l'arête entre `i` et `j`, ou ∞ si l'arête est absente, ce qui permet un accès direct mais utilise beaucoup d'espace. En revanche, la liste d'adjacence :
>
> $$V_0 > 0 \land \forall (u, v) \in E_0 : w(u, v) \geq 0$$

> utilise $O(V_0 + |E_0|)$ d'espace, ce qui est optimal pour les graphes creux (sparse) car seules les arêtes existantes sont stockées.

Orientation du graphe

Dans mon implémentation, je travaille avec des graphes orientés ou non orientés. Un graphe orienté signifie que les arêtes ont une direction, tandis qu'un graphe non orienté n'a pas de direction spécifique pour ses arêtes. Cela se reflète dans les structures de données et les préconditions utilisées dans le code.

Graphe orienté

Un graphe orienté est défini comme suit :

$$G = (V, E)$$

où :

- V est l'ensemble des sommets.
- $E \subseteq V \times V$ est l'ensemble des arêtes orientées.

Pour un graphe orienté, une arête $(v_1, v_2) \in E$ n'implique pas nécessairement que $(v_2, v_1) \in E$. Cela signifie que l'arête (v_1, v_2) existe, mais l'arête inverse (v_2, v_1) n'existe pas nécessairement.

Graphe non orienté

Un graphe non orienté est défini comme suit :

$$G = (V, E)$$

où :

- V est l'ensemble des sommets.
- $E \subseteq v_1, v_2 \mid v_1, v_2 \in V$ est l'ensemble des arêtes non orientées.

Pour un graphe non orienté, une arête $v_1, v_2 \in E$ implique que $v_2, v_1 \in E$. Cela signifie que l'arête entre v_1 et v_2 n'a pas de direction spécifique.

Implémentation en C

Dans le code, la création d'un graphe orienté ou non orienté est déterminée par un booléen `is_oriented` :

```
Graph* g = new_graph(NUM_VERTICES, true); // Graphe orienté
Graph* g = new_graph(NUM_VERTICES, false); // Graphe non orienté
```

Lors de l'ajout d'une arête, le code vérifie si le graphe est orienté ou non pour décider d'ajouter l'arête dans les deux sens ou non :

```
void add_edge(Graph *g, int src, int dest, int weight) {
    NodeList* newNode = add_node(dest, weight);
    newNode->next = g->tab_neighbours[src].head;
    g->tab_neighbours[src].head = newNode;

    if (!g->is_oriented) {
        newNode = add_node(src, weight);
        newNode->next = g->tab_neighbours[dest].head;
        g->tab_neighbours[dest].head = newNode;
    }
}
```

Cette distinction permet de gérer les graphes orientés et non orientés de manière flexible et efficace.

Postcondition (Q)

$$\mathrm{Q} \equiv \forall v \in V : \left\{ \varphi[v] = \min\left(\sum_{k=0}^{|P|-2} w(u_k, u_{k+1})\right), \quad \text{si un chemin } P \text{ existe de } src \text{ à } v, \ \varphi[v] = +\infty, \quad \text{si aucun chemin n'existe.}\right.$$

Pour chaque sommet v :

- Si un chemin P existe de src à v, alors la distance minimale $\phi[v]$ correspond à la somme des poids des arêtes d'un chemin P optimal de src à v.

> Indices de la somme : $k = 0$ à $|P| - 2$ parcourt les $|P| - 1$ sommets du chemin $P = (u_0, u_1, \ldots, u_{|P|-1})$. Chaque arête du chemin P est représentée par $w($

- Si aucun chemin P n'existe, alors $\varphi[v] = +\infty$.

Formulation pour une matrice d'adjacence

Si le graphe est représenté par une matrice d'adjacence, la postcondition devient :

Q \equiv \forall v \in V : \left\{ \phi[v] = \min \left(\sum_{k=0}^{\left| P \right| - 2} \text{graph}[u_k][u_{k+1}] \right), \text{ si un chemin } P \text{ existe de src à } v, \phi[v] =

Exemple :

$$\text{graph} = \begin{bmatrix} 0 & 2 & +\infty & 4 & +\infty & 0 & 1 & +\infty & +\infty & +\infty & 0 & 3 & +\infty & +\infty & +\infty & 0 \end{bmatrix}$$

Pour un chemin $P = (0, 1, 2, 3)$, la somme des poids est donnée par :

$$\sum_{k=0}^{|P|-2} \text{graph}[u_k][u_{k+1}] = \text{graph}[0][1] + \text{graph}[1][2] + \text{graph}[2][3] = 2 + 1 + 3 = 6.$$

Ainsi, la distance minimale $\phi[3]$ est 6.

Vérification dans le code

La distance initiale des sommets est définie comme ∞ : $dist[i] = \text{INT\_MAX}$. Les distances sont mises à jour à l'aide de la fonction explicite $min()$:

```
if (!sptSet[v] && dist[u] != INT_MAX) {
    int new_dist = dist[u] + tmp->weight;
    dist[v] = min(dist[v], new_dist);
```

```
    if (dist[v] == new_dist) {
        pred[v] = u; // Mise à jour du prédécesseur
    }
}
```

Les sommets inatteignables conservent leur valeur ∞, et $pred[v]$ est mis à jour uniquement si une distance minimale est trouvée.

Implémentation

> Implémentez l'algorithme en C. Vous êtes autorisés à vous aider des ressources que vous trouvez, mais l'implémentation doit être votre propre travail : vous devez vous l'approprier et en maîtriser tous les détails. Dans le rapport, vous donnerez le code (commenté) de votre implémentation, vous détaillerez les structures de données utilisées et expliquer pourquoi elles sont adaptées à l'algorithme choisi.

```c
#include "dijkstra.h"

int min(int a, int b);

/**
 * Algorithme de Dijkstra sans Min-Heap.
 * @param graph Graph sous forme de liste d'adjacence.
 * @param src Sommet source pour les plus courts chemins.
 */
void dijkstra_simple(Graph *graph, int src) {
    int V = graph->nb_vertices;
    int dist[V]; // ϕ[v]
    int pred[V]; // ρ[v]
    Boolean sptSet[V]; // Q

    // Initialiser toutes les distances comme infinies et sptSet[] comme faux
    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX; // ϕ[v]=+∞
        sptSet[i] = false; // Q
        pred[i] = -1; // ρ[v]=?
    }

    // La distance du sommet source à lui-même est toujours 0
    dist[src] = 0; // ϕ[s]=0

    // Trouver le chemin le plus court pour tous les sommets
    for (int count = 0; count < V - 1; count++) {
        // Choisir le sommet de distance minimale parmi ceux qui ne sont pas encore traités
        int u = -1; // u ← un sommet de Q avec ϕ[u] minimum
        int min_dist = INT_MAX; // Initialiser à la valeur maximale
        for (int v = 0; v < V; v++) {
            if (!sptSet[v] && dist[v] <= min_dist) {
                min_dist = dist[v]; // màj min_dist
                u = v; // màj le sommet avec la distance minimale
            }
        }

        if(u == -1) break; // Si aucun sommet n'a été trouvé, sortir de la boucle
        sptSet[u] = true; // Retirer u de Q

        // Pointeur temporaire pour parcourir la liste d'adjacence des sommets adjacents au sommet choisi.
        NodeList* tmp = graph->tab_neighbours[u].head;
        while (tmp != NULL) {
            int v = tmp->dest; // Pour tout v ∈ Γ⁺(u)
            if (!sptSet[v] && dist[u] != INT_MAX) {
                int new_dist = dist[u] + tmp->weight; // ϕ[u] + w(u, v)

                // Mise à jour de la distance minimale vers v si le chemin via u est plus court
                dist[v] = min(dist[v], new_dist); // ϕ[v] = min(ϕ[v], ϕ[u] + w(u, v))
                if (dist[v] == new_dist) {
                    pred[v] = u; // ρ[v] = u
                }
            }
            tmp = tmp->next;
        }
    }

    // Afficher les distances finales et les prédécesseurs
    print_shortest_paths(graph, dist, pred);
```

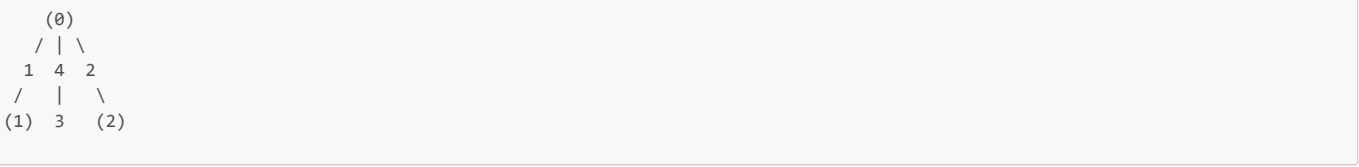
```
    // Afficher les distances
    print_arr(dist, V);
}

/**
 * Renvoie un ternaire du minimum entre deux entiers.
 * @param a Premier entier.
 * @param b Deuxième entier.
 * @return Le plus petit des deux entiers.
 */
int min(int a, int b) {
    return (a < b) ? a : b;
}
```

- V : Nombre total de sommets.
- E : Nombre total d'arêtes.
- src : Sommet source pour Dijkstra.
- graph : Représentation du graphe en liste d'adjacence.
- dist : Tableau des distances minimales.
- pred : Tableau des prédécesseurs des sommets.
- sptSet : Ensemble des sommets traités (Shortest Path Tree Set).
- NodeList : Liste des voisins d'un sommet.

Exemple 1 :

Considérons le graphe suivant :



Avec les poids des arêtes :

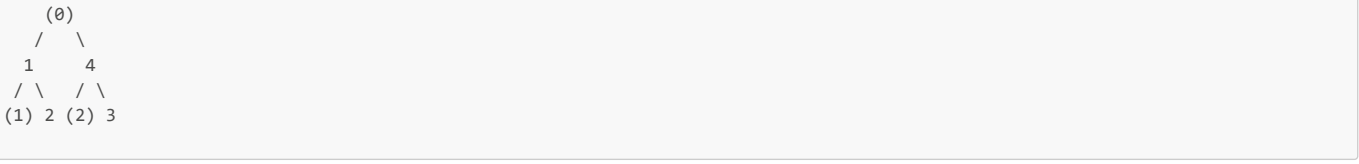
$(0, 1) = 1$
$(0, 2) = 2$
$(0, 3) = 4$
$(1, 3) = 3$
$(2, 3) = 2$

Si $\text{src} = 0$, alors les distances minimales calculées par l'algorithme de Dijkstra sont :

$\text{dist}[0] = 0$
$\text{dist}[1] = 1$
$\text{dist}[2] = 2$
$\text{dist}[3] = 4$

Exemple 2 :

Considérons un autre graphe :



Avec les poids des arêtes :

$(0, 1) = 1$
$(0, 2) = 4$
$(1, 2) = 2$
$(1, 3) = 5$
$(2, 3) = 1$

Si $\text{src} = 0$, alors les distances minimales calculées par l'algorithme de Dijkstra sont :

$\mathrm{dist}[0] = 0$
$\mathrm{dist}[1] = 1$
$\mathrm{dist}[2] = 3$
$\mathrm{dist}[3] = 4$

4. En vos propres mots, expliquez comment fonctionne l'algorithme. Décrivez de manière intuitive pourquoi l'implémentation produit un résultat correct par rapport à vos spécifications.

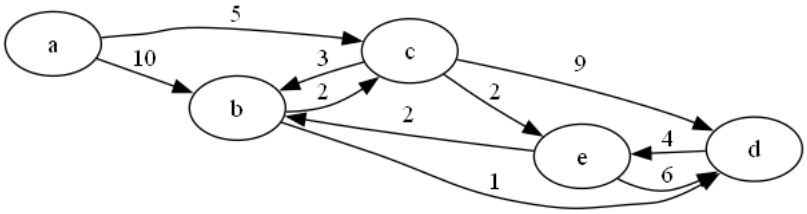
L'algorithme de Dijkstra permet de trouver les plus courts chemins depuis un sommet source src vers tous les autres sommets d'un graphe $G = (V, E)$, à condition que les poids des arêtes soient non négatifs ($w(u, v) \geq 0, \forall (u, v) \in E$).

Description

On pose $\phi[v]$ comme la distance estimée actuelle du sommet v depuis la source, équivalente à dist[v] dans le code.
On pose $\rho[v]$ comme le prédécesseur de v sur le chemin minimal, équivalent à pred[v] dans le code.
On pose $\mathrm{sptSet}[v]$ comme un indicateur booléen indiquant si v a été traité, équivalent à sptSet[v] dans le code.
Lorsque $\mathrm{sptSet}[v] = \mathrm{true}$, cela signifie que le plus court chemin vers v a été trouvé.
On pose $\Gamma^+(u)$ comme l'ensemble des voisins directs (successeurs) du sommet u, équivalent à graph->tab_neighbours[u] dans le code.

- Initialisation :
  - $\phi[v] = +\infty$ pour tous les sommets sauf src, où $\phi[\mathrm{src}] = 0$.
  - $\rho[v] =?$ pour tous les sommets.
  - Ensemble Q contenant tous les sommets ($Q = V$).
- Traitement principal :
  - À chaque itération, sélectionnez $u \in Q$ avec la plus petite distance $\phi[u]$.
  - Marquez u comme traité ($\mathrm{sptSet}[u] = \mathrm{true}$).
  - Mettez à jour les distances des voisins de u :
  - Si $\phi[u] + w(u, v) < \phi[v]$, alors $\phi[v] = \phi[u] + w(u, v)$.
- Condition d'arrêt : L'algorithme s'arrête lorsque Q est vide ou que toutes les distances finales sont calculées.
- Résultats finaux :
  - $\phi[v]$ contient la distance minimale depuis src jusqu'à v.
  - $\rho[v]$ permet de reconstruire les chemins minimaux.

Exemple



Exemple de grapge :

- Un graphe orienté $G = (V, E)$ avec $V = a, b, c, d, e$.
- Une source $\mathrm{src} \in V$ avec $\mathrm{src} = a$.
- Une fonction positive de pondération des arcs $w : E \to \mathbb{R}^+$ avec \
  $w(a, b) = 10, w(b, c) = 2, w(c, d) = 9, w(d, e) = 4, w(a, c) = 5, w(b, d) = 1, w(c, b) = 3, w(c, e) = 2, w(e, b) = 2, w(e, d) = 6.$

Problème : Trouver un plus court chemin de src vers tout autre sommet de G.

0. Données initiales : Distances : $\forall v \in V, \phi[v] = +\infty$
   Prédéceseurs : $\forall v \in V, \rho[v] = -1$
   Source : $\phi[\mathrm{src}] = 0$
   Sommets non-traités : $Q = a, b, c, d, e$

1. Initialisation Avant la première itération :

u	Q	Γ⁺(u)	a	b	c	d	e
-	{a, b, c, d, e}	-	0, ?	+∞, ?	+∞, ?	+∞, ?	+∞, ?

Sélectionner u avec la plus petite $\phi[u]$ dans Q. Retirer u de Q et le marquer comme traité. Mettre à jour $\phi[v]$ pour chaque voisin v de u. Si mise à jour, définir $\rho[v] = u$.

2. Première itération ($u = a$)

- Sélection de $u = a$ car $\phi[a] = 0$.
- Exploration des voisins : $\Gamma^+(a) = b, c$.
    - Mise à jour pour b : $\phi[b] = 0 + 10 = 10, \rho[b] = a$.
    - Mise à jour pour c : $\phi[c] = 0 + 5 = 5, \rho[c] = a$.

u	Q	Γ⁺(u)	a	b	c	d	e
a	{b, c, d, e}	{b, c}	0, ?	10, a	5, a	+∞, ?	+∞, ?

3. Deuxième itération $u = c$

Sélection de $u = c$ car $\phi[c] = 5$.

- Exploration des voisins : $\Gamma^+(c) = b, d, e$.
- Mise à jour pour b : $\phi[b] = \min(10, 5 + 3) = 8, \rho[b] = c$.
    - Mise à jour pour d : $\phi[d] = 5 + 9 = 14, \rho[d] = c$.
    - Mise à jour pour e : $\phi[e] = 5 + 2 = 7, \rho[e] = c$.

u	Q	Γ⁺(u)	a	b	c	d	e
c	{b, d, e}	{b, d, e}	0, ?	8, c	5, a	14, c	7, c

4. Troisième itération $u = e$

- Sélection de $u = e$ car $\phi[e] = 7$.
- Exploration des voisins : $\Gamma^+(e) = b, d$.
    - Mise à jour pour d : $\phi[d] = \min(14, 7 + 6) = 13, \rho[d] = e$.

u	Q	Γ⁺(u)	a	b	c	d	e
e	{b, d}	{b, d}	0, ?	8, c	5, a	13, e	7, c

5. Quatrième itération $u = b$

- Sélection de $u = b$ car $\phi[b] = 8$.
- Exploration des voisins : $\Gamma^+(b) = d$.
    - Mise à jour pour d : $\phi[d] = \min(13, 8 + 1) = 9, \rho[d] = b$.

u	Q	Γ⁺(u)	a	b	c	d	e
b	{d}	{d}	0, ?	8, c	5, a	9, b	7, c

6. Cinquième itération $u = d$

Le sommet $u = d$ est sélectionné car $\phi[d] = 9$.
Comme d n'a pas de voisins non traités ($\Gamma^+(d) = \emptyset$), aucune mise à jour n'est effectuée.

7. Terminaison

> L'algorithme s'arrête lorsque Q est vide, c'est-à-dire que tous les sommets atteignables ont été traités.

L'ensemble Q est désormais vide. Les valeurs finales de ϕ et ρ représentent les distances minimales et les prédécesseurs pour les plus courts chemins depuis a.

8. Résultat final :

> $\phi[v]$ contient la distance minimale de src à v.
> $\rho[v]$ permet de reconstituer les chemins minimaux.

Distances minimales ($\phi[v]$) :
$\phi[a] = 0$,
$\phi[b] = 8$,
$\phi[c] = 5$,
$\phi[d] = 9$,
$\phi[e] = 7$.

Chemins minimaux $\rho[v]$:

$b \leftarrow c \leftarrow a$
$c \leftarrow a$
$d \leftarrow b \leftarrow c \leftarrow a$
$e \leftarrow c \leftarrow a$

Invariant de boucle pertinent pour l'algorithme de Dijkstra

> 5. Identifiez un invariant de boucle pertinent pour l'algorithme. Formulez cet invariant et démontrez, de manière formelle, qu'il est vérifié à chaque itération de la boucle concernée. Expliquez en quoi il permettrait, dans une preuve de programme plus complète, de faire le pont entre pré- et post- conditions.

Elicitation de l'invariant

L'invariant proposé est :

$$I \equiv \forall v \in V : \begin{cases} \varphi[v] = \sum_{k=0}^{|P|-1} w(u_k, u_{k+1}), & \text{si } v \in sptSet,\ \varphi[v] \geq \min\left(\sum_{k=0}^{|P|-1} w(u_k, u_{k+1})\right), \\ & \text{si } v \notin sptSet,\ \varphi[v] = +\infty, \\ & \text{si } v \text{ est inaccessible.} \end{cases}$$

Dans cet invariant, $\varphi[v]$ représente la distance estimée actuelle du sommet v depuis la source src dans l'algorithme de Dijkstra. P est un chemin spécifique reliant la source src au sommet v, composé de sommets $u_0, u_1, \ldots, u_k, \ldots, u_{|P|-1}$. La longueur de ce chemin est $|P| - 1$, c'est-à-dire le nombre d'arêtes dans P. L'indice k représente les arêtes dans le chemin P, chaque arête étant notée $w(u_k, u_{k+1})$. L'ensemble $sptSet$ contient les sommets déjà traités, inclus dans l'arbre des plus courts chemins.

Méthodologie structurée de la boucle

L'algorithme de Dijkstra suit cette structure :

```
{R} // Précondition de la boucle
INIT;
{I} // Invariant vrai avant de rentrer dans la boucle
while (B) {
  ITER;
  {I} // Invariant préservé à la fin du tour de boucle
}
{I et \neg B} // Invariant encore vrai, condition d'arrêt atteinte
CLOT;
{T} // Postcondition de la boucle
```

Condition d'arrêt ($\neg B$)

$$F = \emptyset,$$

où F est l'ensemble des sommets non encore traités. Lorsque F est vide, l'algorithme s'arrête.

6. Analysez la complexité temporelle et spatiale « pire des cas » de l'algorithme. Justifiez votre analyse en fonction des différentes étapes de l'algorithme et des structures de données utilisées.

Complexité temporelle

L'algorithme de Dijkstra sans **Min-Heap** effectue les étapes suivantes :

1. **Initialisation** :
$$O(V)$$
2. **Recherche du sommet minimal (u)** :
À chaque itération, l'algorithme parcourt tous les sommets restants pour trouver celui ayant la plus petite distance :
$$O(V), (\text{par itération}) \quad \Rightarrow \quad O(V^2), (\text{pour (V) sommets}).$$
3. **Mise à jour des voisins de (u)** :
Pour chaque sommet (u), tous ses voisins sont explorés. Le coût total est proportionnel au nombre d'arêtes :
$$O(E).$$

Complexité temporelle totale :

$$O(V^2 + E) = O(V^2).$$

Complexité spatiale

L'algorithme utilise :

1. $\text{dist}[], \text{pred}[], \text{sptSet}[]$:
$$O(V).$$
2. Le graphe en liste d'adjacence :
$$O(V + E).$$

Complexité spatiale totale :

$$O(V + E).$$

Conclusion :

- **Temporelle** :

$$O(V^2)$$

- **Spatiale** :

$$O(V + E)$$

.

L'algorithme de Dijkstra sans Min-Heap a une complexité temporelle de

$$O(V^2)$$

car chaque sommet doit être recherché, prenant

$$O(V)$$

temps. En utilisant une structure de données Min-Heap ou Fibonacci-Heap, la recherche est réduite à

$$O(\log V)$$

, améliorant la complexité à

$$O(V \cdot \log V + E)$$

, où

$$E$$

est le nombre d'arêtes. Cette amélioration est bénéfique pour les graphes grands et clairsemés. Pour les graphes denses, l'implémentation avec Fibonacci-Heap est plus efficace.

Version récursive de l'algorithme et hypothèse d'induction

> 7. Proposez une version récursive de l'algorithme (ou d'une partie de celui-ci, si cela est pertinent). Formulez une hypothèse d'induction qui servira à démontrer la correction de l'algorithme sur la base des appels récursifs. Seule l'hypothèse d'induction, et son impact sur la correction de l'implémentation récursive, doit être formulée formellement ; les autres calculs peuvent être considérés corrects.

Partie choisie : Sélection et mise à jour des distances des voisins Une partie clé pour intégrer la récursivité dans l'algorithme de Dijkstra est la sélection du sommet ayant la plus petite distance et la mise à jour des distances des voisins. Nous explorons ici deux approches :

- Une récursion pour sélectionner le sommet minimal.
- Une récursion pour parcourir les voisins et mettre à jour leurs distances.

1. Sélection récursive du sommet minimal

La sélection du sommet minimal parmi les sommets non traités peut remplacer une boucle par une récursion. Voici la fonction récursive théorique :

```
int recursiveMinDistance(int dist[], Boolean sptSet[], int V, int i, int minIndex, int minVal) {
    if (i == V) return minIndex; // Cas de base : tous les sommets ont été parcourus.

    if (!sptSet[i] && dist[i] < minVal) {
        return recursiveMinDistance(dist, sptSet, V, i + 1, i, dist[i]); // Mise à jour du minimum.
    }
    return recursiveMinDistance(dist, sptSet, V, i + 1, minIndex, minVal); // Passage au sommet suivant.
}
```

Hypothèse d'induction

Pour tout $i \in [0, V-1]$, la fonction récursive garantit que le sommet ayant la distance minimale parmi les $i+1$ premiers sommets non traités est retourné.

Démonstration mathématique

1. Cas de base : Si $i = 0$, la fonction vérifie uniquement le sommet initial. Elle retourne son indice si les conditions $\text{dist}[i] < \text{minVal}$ et $\text{sptSet}[i] = \text{false}$ sont remplies. Sinon, elle passe au sommet suivant.

2. Pas inductif : Supposons que la fonction retourne correctement minIndex pour i. À $i+1$:

- Si $\text{dist}[i+1] < \text{minVal}$, alors minIndex est mis à jour avec $i+1$.
- Sinon, minIndex reste inchangé. La condition est préservée pour $i+1$.

Complexité

- Temporelle : $O(V)$.
- Spatiale : $O(V)$ en raison de la pile d'appels récursifs.

2. Mise à jour récursive des distances des voisins

Après avoir sélectionné un sommet u, la mise à jour des distances de ses voisins peut également être effectuée récursivement.

```
void recursiveUpdate(NodeList* tmp, int u, int dist[], int pred[], Boolean sptSet[]) {
    if (tmp == NULL) return; // Cas de base : plus de voisins à traiter.

    int v = tmp->dest;
    if (!sptSet[v] && dist[u] != INT_MAX) {
        int new_dist = dist[u] + tmp->weight;
        dist[v] = (dist[v] > new_dist) ? new_dist : dist[v]; // Mise à jour de la distance minimale.
        if (dist[v] == new_dist) pred[v] = u; // Mise à jour du prédécesseur.
    }

    recursiveUpdate(tmp->next, u, dist, pred, sptSet); // Appel récursif pour le voisin suivant.
}
```

Hypothèse d'induction

Pour tout $v \in \Gamma^+(u)$, où $\Gamma^+(u)$ est l'ensemble des voisins de u :

$$\phi[v] = \min(\phi[v], \phi[u] + w(u,v))$$

Démonstration mathématique

1. Cas de base : Si $\Gamma^+(u) = \emptyset$, la fonction retourne immédiatement, ce qui est correct.

2. Pas inductif : Supposons que la mise à jour est correcte pour les k premiers voisins. Pour le $(k+1)$-ème voisin v :

Si $\phi[u] + w(u,v) < \phi[v]$, alors $\phi[v]$ est mis à jour correctement. Sinon, $\phi[v]$ reste inchangé.

Complexité

- Temporelle : $O(d(u))$, où $d(u)$ est le degré de u.
- Spatiale : $O(d(u))$ appels récursifs au maximum.

Intégration théorique dans Dijkstra

Ces deux fonctions récursives peuvent être appelées dans l'algorithme principal de Dijkstra comme suit :

```
while (count < V - 1) {
    int u = recursiveMinDistance(dist, sptSet, V, 0, -1, INT_MAX); // Sélection récursive.
    if (u == -1) break;

    sptSet[u] = true; // Marquer comme traité.
    recursiveUpdate(graph->tab_neighbours[u].head, u, dist, pred, sptSet); // Mise à jour récursive.
    count++;
}
```

Comparaison récursivité vs boucle

	Non récursif	**Récursif**
Lisibilité	Simple	Plus complexe
Temporelle	$O(V^2)$	$O(V^2)$
Spatiale	$O(1)$	$O(V)$ (pile d'appels)
Flexibilité	Limitée	Plus modulaire

Recommandation

Bien que l'approche récursive apporte une modularité et une clarté théorique, elle n'est pas optimale en termes de performances pratiques (risques liés à la pile). Une implémentation non récursive est donc préférable pour des graphes de grande taille.

Conclusion

Ces approches récursives permettent d'explorer les fondements mathématiques et la structure de Dijkstra, mais elles restent une adaptation académique. En pratique, elles sont utiles pour enseigner les concepts ou pour des cas très spécifiques où la récursivité apporte un avantage en termes de modularité ou de clarté algorithmique.

Informations pertinentes et caractéristiques de l'algorithme choisi

> 8. Enfin, en fonction de l'algorithme que vous aurez choisi, ajoutez des informations qui vous semblent pertinentes ou sur des aspects caractéristiques à cet algorithme qui sont précisés dans la description de l'algorithme.

L'algorithme de Dijkstra résout efficacement des problèmes de chemin le plus court dans des graphes pondérés.

L'étude de Delhaye (2015) utilise l'algorithme de Dijkstra pour modéliser les circulations fiscales entre dix juridictions. Dans cette approche, chaque pays est un nœud $v \in V$ et chaque transfert financier est une arête $(u, v) \in E$, pondérée par un coût $w(u, v)$ représentant le taux d'imposition effectif.

L'objectif est de minimiser le coût total d'un transfert fiscal entre une société filiale src et une société mère $dest$, en tenant compte des juridictions intermédiaires. La formulation explicite pour chaque $v \in V$ est donnée par :

$$\phi[v] = \{\, 0, \quad \text{si } v = src, \;\; \min \phi[u] + w(u, v) \mid (u, v) \in E, \quad \text{si } v \neq src, \;\; +\infty, \quad \text{si aucun chemin n'existe.}$$

Ici :

$\phi[v]$: le coût fiscal minimal pour atteindre v depuis src.
$w(u, v)$: le taux d'imposition effectif pour transférer des revenus de u à v.

L'algorithme itère sur tous les nœuds $u \in V$, en mettant à jour $\phi[v]$ via la règle de relaxation :

$$\phi[v] \leftarrow \min \phi[v], \phi[u] + w(u, v).$$

Exemple concret
Prenons un réseau fiscal simplifié avec les nœuds suivants :

A : Allemagne ($w(A, B) = 15$),
B : Belgique ($w(B, C) = 10$),
C : Irlande ($w(A, C) = 12.5$).

Soit $src = A$ et $dest = C$. Nous calculons $\phi[C]$ (le coût fiscal minimal pour transférer des revenus de A à C).

Initialisation :
$\phi[A] = 0$,
$\phi[B] = +\infty$,
$\phi[C] = +\infty$.

Itération 1 ($u = A$) :
$\phi[B] \leftarrow \min(\phi[B], \phi[A] + w(A, B)) = \min(+\infty, 0 + 0.15) = 0.15$,
$\phi[C] \leftarrow \min(\phi[C], \phi[A] + w(A, C)) = \min(+\infty, 0 + 0.125) = 0.125$.

Itération 2 ($u = C$) :
$\phi[C]$ ne change pas car $dest = C$ est atteint directement.

Résultat final :
$\phi[C] = 0.125$. Le chemin optimal est direct ($A \to C$) avec un coût fiscal de 12,5 %.

Vérification avec détour profitable
Supposons que $w(A, B) = 10$ et $w(B, C) = 8$. Alors :

$\phi[C] = \min(0.125, \phi[A] + w(A, B) + w(B, C)) = \min(0.125, 0 + 0.10 + 0.08) = 0.10 + 0.08 = 0.18$.
Le chemin direct ($A \to C$) reste optimal.

Ainsi, on peut conclure que cet exemple montre que l'algorithme de Dijkstra modélise efficacement les stratégies fiscales en minimisant le coût total de circulation. L'approche repose sur des calculs explicites pour trouver le chemin optimal dans un réseau fiscal pondéré.

Sources

1. Cours et documents académiques

- **Techniques de programmation**
  Support du cours pour étudiants en horaire décalé.
  Assistante : M. Barkallah, Assistant : G. Yernaux.
  Année académique 2020-2021.

- **Algorithmique 1**
  W. Vanhoof, *Université de Namur, Faculté d'informatique*.
  Édition 2019. Code de l'UE : IHDCB232.
  Professeur : Vanhoof Wim.

- **Mathématiques discrètes 1 : Théorie et algorithmique des graphes**
  Blondel, V. (2014). Technical report, UCL/EPL. Cours LINMA1691.
  Une présentation détaillée des graphes orientés pondérés et des fondements théoriques de l'algorithme de Dijkstra.

- **Generic Dijkstra: Correctness and Tractability**
  Szczesniak, I., & Woźna-Szcześniak, B. (n.d.). Czestochowa University of Technology, Department of Computer Science & Jan Długosz University in Czestochowa, Department of Mathematics and Computer Science.
  arXiv. Une exploration approfondie de la correction et de la tractabilité de l'algorithme.

- **Application économique de l'algorithme de Dijkstra**
  Delhaye, V. (2015). *Le plus court chemin d'imposition des multinationales*.
  UCLouvain, Louvain School of Management. https://dial.uclouvain.be/memoire/ucl/object/thesis:2776

---

2. Livres de référence

- Rivest, R., Stein, C., Cormen, T., & Leiserson, C. (2010). *Algorithmique, 3ème édition*. Dunod.
  Une référence incontournable couvrant les graphes, les structures de données et Dijkstra.

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.
  Disponible via la Bibliothèque UNamur.

---

3. Tutoriels en ligne et implémentations

- **GeeksforGeeks**
  - Binary Heap.
  - Heap Data Structure.
  - Dijkstra's Algorithm (Adjacency List).

- **Autres ressources :**
  - Binary Heap Implementation.
  - TechCodeView.

---

4. Cours en ligne et vidéos pédagogiques

- Champagne, J. (2023). *Langage C #22 - Graphes*. YouTube.
- Champagne, J. (2024). *Architecture - Graphe*. YouTube.
- Série sur l'algorithme de Dijkstra :
  - L'exécution (Partie 1).
  - Preuve d'optimalité (Partie 2).

---

5. Articles et ressources académiques

- Parreaux, J. *Analyse de l'algorithme de Dijkstra*. ENS Rennes.
- **Cours complémentaires :**
  - ENS Rennes : Complexité - Cours 4.
  - Cours PCSI - Chapitre 22 : Dijkstra.

---

6. Ressources généralistes

- Wikipédia :
  - Algorithme de Dijkstra.
  - Liste d'adjacence.
  - Matrice d'adjacence.

---

7. Bibliothèques et outils

- Graphviz Documentation.
  Documentation officielle pour visualiser les graphes.

- Plateformes académiques :

- Bibliothèque BUMP - UNamur.
- UCLouvain Library.