

Plus courts chemins et algorithme A^*

Quentin Fortier

November 24, 2024

Soit $G = (S, A)$ un graphe orienté, pondéré par $p : A \rightarrow \mathbb{R}$.

On note $n = |S|$ et $p = |A|$.

Définition

Soient $u, v \in S$.

- 1 Un chemin de u à v est une suite de sommets $u = u_0, u_1, \dots, u_k = v$ telle que $\forall i \in \{0, \dots, k-1\}, (u_i, u_{i+1}) \in A$.
- 2 Le poids d'un chemin C , noté $p(C)$, est la somme des poids de ses arêtes.
- 3 Un chemin de u à v est un plus court chemin s'il n'existe pas de chemin de u à v de poids plus petit.
- 4 La distance $d(u, v)$ est le poids d'un plus court chemin de u à v .
Autrement dit : $d(u, v) = \inf\{p(C) \mid C \text{ est un chemin de } u \text{ à } v\}$.
S'il n'existe pas de chemin de u à v , on pose $d(u, v) = +\infty$.
S'il y a un cycle de poids négatif, on peut avoir $d(u, v) = -\infty$.

Problème 1

Entrée : Graphe orienté $G = (S, A)$ pondéré par
 $p : A \rightarrow \mathbb{R}^+, s \in S$

Sortie : Tableau d tel que $d[v] = d(s, v)$

Problème 2

Entrée : Graphe non-orienté $G = (S, A)$ pondéré par
 $p : A \rightarrow \mathbb{R}^+, s \in S$

Sortie : Tableau d tel que $d[v] = d(s, v)$

Théorème

Supposons que l'on puisse résoudre le problème 1 en complexité $O(f(n, p))$.

Alors on peut résoudre le problème 2 en complexité $O(f(n, p))$.

Inégalité triangulaire

Si $u, v, w \in S$, alors : $d(u, v) \leq d(u, w) + d(w, v)$

Sous-optimalité

Soit C un plus court chemin de u à v et u', v' deux sommets de C .
Alors le sous-chemin C' de C de u' à v' est aussi un plus court chemin.

Lemme

Soient $u, v, w \in S$. Alors :

$$d(u, v) = \min_{(w,v) \in A} d(u, w) + p(w, v)$$

Remarques :

- Cette équation n'est pas utilisable en l'état car calculer $d(u, v)$ est aussi difficile que calculer $d(u, w)$.
- Pour la rendre utilisable, on peut ajouter un paramètre supplémentaire : nombre d'arêtes (Bellman-Ford) ou numéros des sommets utilisables (Floyd-Warshall).

Entrée : Graphe orienté $G = (S, A)$ pondéré par $p : A \rightarrow \mathbb{R}$,
 $s \in S$

Sortie : Tableau d tel que $d[v] = d(s, v)$

Pour retrouver les plus courts chemins, on peut conserver un tableau `pere` tel que `pere[v]` contient le prédécesseur de v dans un plus court chemin de s à v .

Un parcours en largeur permet de résoudre le problème suivant :

Entrée : $G = (S, A)$ avec des poids unitaires, $s \in S$

Sortie : Tableau d tel que $d[v] = d(s, v)$

Complexité :

- $O(n + p)$ si G est représenté par une liste d'adjacence
- $O(n^2)$ si G est représenté par une matrice d'adjacence

```
int* bfs(int s, int n, int** g) {
    // g est une matrice d'adjacence avec n sommets
    int* d = malloc(n * sizeof(int));
    for(int i = 0; i < n; i++) d[i] = -1;
    d[s] = 0;
    int* q = malloc(n * sizeof(int)); // file
    int deb = 0, fin = 1;
    q[fin] = s;
    while (deb < fin) {
        int u = q[deb++];
        for(int v = 0; v < n; v++)
            if(g[u][v] && d[v] == -1) {
                d[v] = d[u] + 1;
                q[fin++] = v;
            }
    }
    free(q);
    return d;
}
```

Graphe orienté acyclique (HP)

Entrée : $G = (S, A)$ acyclique et $s \in S$

Sortie : Tableau d tel que $d[v] = d(s, v)$

Résolution en $O(n + p)$:

- 1 Calculer un tri topologique des sommets avec Kosaraju.
- 2 Calculer les distances dans l'ordre topologique, en utilisant :

$$d(s, v) = \min_{(u,v) \in A} d(s, u) + p(u, v)$$

Remarque : On ne peut pas utiliser cette dernière formule si G contient des cycles.

```
int* sp_dag(int s, int** g, int n) {
// g[i][j] = poids de l'arc i -> j ou 0 si pas d'arc
    int* d = malloc(n * sizeof(int));
    for(int i = 0; i < n; i++) d[i] = -1;
    d[s] = 0;
    int* tri = tri_topologique(g, n);
    for(int i = 0; i < n; i++) {
        int v = tri[i];
        for(int j = 0; j < i; j++) {
            int u = tri[j];
            if(g[u][v] && (d[v] == -1 || d[v] > d[u] + g[u][v]))
                d[v] = d[u] + g[u][v];
        }
    }
    free(ordre);
    return d;
}
```

Algorithme de Dijkstra

L'algorithme de Dijkstra permet de résoudre le problème suivant :

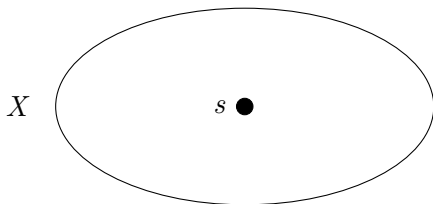
Entrée : $G = (S, A)$ pondéré par $p : A \rightarrow \mathbb{R}^+$, $s \in S$

Sortie : Tableau d tel que $d[v] = d(s, v)$

Algorithme de Dijkstra

Idée : Calculer les distances par ordre croissant depuis s .

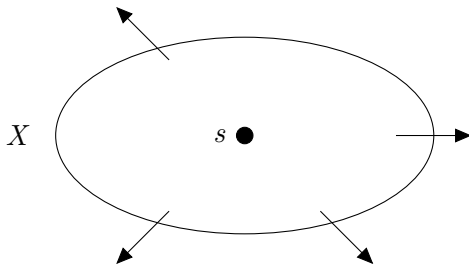
Soit $X \subset S$ l'ensemble des sommets de distance connue.



Algorithme de Dijkstra

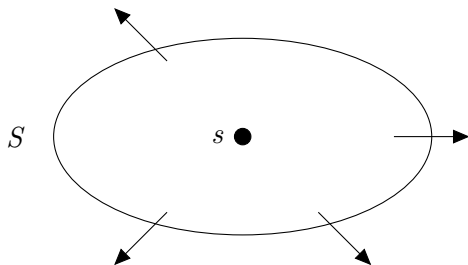
Idée : Calculer les distances par ordre croissant depuis s .

Soit $X \subset S$ l'ensemble des sommets de distance connue.



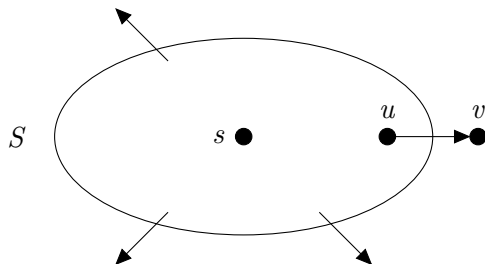
À chaque étape, on déduit la distance à un nouveau sommet (qu'on ajoute à X).

Algorithme de Dijkstra



Soit $(u, v) \in A$ tel que $v \notin X$ et $d(s, u) + p(u, v)$ est minimum.

Algorithme de Dijkstra

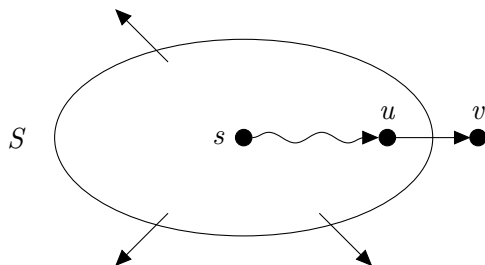


Soit $(u, v) \in A$ tel que $v \notin X$ et $d(s, u) + p(u, v)$ est minimum.

Alors :

$$d(s, v) = d(s, u) + p(u, v)$$

Algorithme de Dijkstra



Soit $(u, v) \in A$ tel que $v \notin X$ et $d(s, u) + p(u, v)$ est minimum.

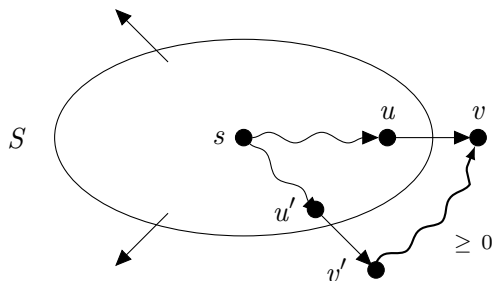
Alors :

$$d(s, v) = d(s, u) + p(u, v)$$

Preuve :

- 1 Il existe un chemin de longueur $d(s, u) + p(u, v)$.

Algorithme de Dijkstra



Soit $(u, v) \in A$ tel que $v \notin X$ et $d(s, u) + p(u, v)$ est minimum.

Alors :

$$d(s, v) = d(s, u) + p(u, v)$$

Preuve :

- ② Un chemin C de s à v doit sortir de S avec un arc (u', v') . Comme les poids sont ≥ 0 :

$$p(C) \geq d(s, u') + p(u', v') \geq d(s, u) + p(u, v)$$

Algorithme de Dijkstra

On stocke les sommets restants à visiter dans q ($= \bar{X}$) et on conserve un tableau d des distances estimées avec l'invariant de boucle suivant :

$$\textcircled{1} \quad \forall v \notin q : d[v] = d(s, v).$$

$$\textcircled{2} \quad \forall v \in q : d[v] = \min_{u \notin q} d(s, u) + p(u, v).$$

Algorithme de Dijkstra

Entrée : $G = (S, A)$ pondéré par $p : A \rightarrow \mathbb{R}^+$, $s \in S$

Sortie : Tableau d tel que $d[v] = d(s, v)$

$q \leftarrow$ file de priorité contenant tous les sommets

$d \leftarrow [\infty, \dots, \infty]$

$d[s] \leftarrow 0$

Tant que $q \neq \emptyset$:

 Extraire u de q tel que $d[u]$ soit minimum

Pour tout voisin v de u :

Si $d[u] + p(u, v) < d[v]$:

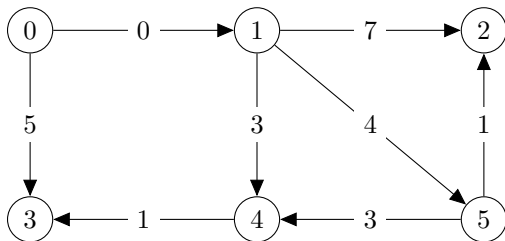
$d[v] \leftarrow d[u] + p(u, v)$

Renvoyer d

Algorithme de Dijkstra

Exercice

Appliquer l'algorithme de Dijkstra depuis $s = 0$ sur le graphe suivant, en mettant à jour les valeurs $d[v]$ à côté de chaque sommet v :



Algorithme de Dijkstra

Complexité de Dijkstra si q est implémenté par un tas :

Algorithme de Dijkstra

Complexité de Dijkstra si q est implémenté par un tas :

- ① n extractions du minimum en $O(n \log(n))$
- ② au plus p mises à jour en $O(p \log(n))$

Total : $O(n \log(n)) + O(p \log(n)) = \boxed{O(p \log(n))}$.

Algorithme de Dijkstra

On suppose avoir une file de priorité min avec les fonctions suivantes :

(create () renvoie une file de priorité vide *)*

`create : unit -> 'a priority_queue`

(add q e p ajout e avec priorité p dans q *)*

`add : 'a priority_queue -> 'a -> int -> unit`

(empty q détermine si q est vide *)*

`empty : 'a priority_queue -> bool`

(extract_min q extrait l'élément de q de priorité minimum *)*

`extract_min : 'a priority_queue -> 'a * int`

Problème : si on veut implémenter la file de priorité avec un tas, la fonction de mise à jour d'un élément dans un tas demande de connaître l'indice de l'élément à modifier.

Il faudrait maintenir un tableau qui donne l'indice (dans le tas) d'un sommet. C'est fastidieux.

Algorithme de Dijkstra

Problème : si on veut implémenter la file de priorité avec un tas, la fonction de mise à jour d'un élément dans un tas demande de connaître l'indice de l'élément à modifier.

Il faudrait maintenir un tableau qui donne l'indice (dans le tas) d'un sommet. C'est fastidieux.

Solution plus simple : ajouter des couples (distance estimée de v , v) sans jamais mettre à jour les éléments de la file de priorité (qui peut donc contenir plusieurs fois le même sommet).

Algorithme de Dijkstra

Algorithme de Dijkstra

Entrée : $G = (S, A)$ pondéré par $p : A \rightarrow \mathbb{R}^+$, $s \in S$

Sortie : Tableau d tel que $d[v] = d(s, v)$

$q \leftarrow$ file de priorité vide

Ajouter s à q avec priorité 0

$d \leftarrow [\infty, \dots, \infty]$

$d[s] \leftarrow 0$

Tant que $q \neq \emptyset$:

 Extraire u de q avec priorité minimum du

Si $d[u] = \infty$:

$d[u] \leftarrow \infty$

Pour tout voisin v de u :

 Ajouter v à q avec priorité $d[u] + p(u, v)$

Renvoyer d

Algorithme de Dijkstra

```
let dijkstra g p s =
  let n = Array.length g in
  let q = create () in
  let d = Array.make n max_int in
  add q s 0;
  while not (is_empty q) do
    let u, du = extract_min q in
    if d.(u) = max_int then (
      d.(u) <- du;
      List.iter (fun v -> add q v (du + p u v)) g.(u)
    )
  done;
  d
```

Souvent, on ne cherche pas les plus courts chemins à tous les autres sommets, mais seulement à un autre sommet fixé.

Exemples :

- GPS : Trouver un chemin le plus court d'une ville à une autre.
- Jeu vidéo : Déplacer un personnage d'un point à un autre.

Algorithme A^*

Entrée : $G = (S, A)$ pondéré par $p : A \rightarrow \mathbb{R}^+$, $s \in S$,
 $t \in S$
Sortie : La distance $d(s, t)$ de s à t

Principe de l'algorithme A^* :

- 1 Définir une heuristique h telle que $h(v)$ soit une estimation de la distance de v à t .
- 2 Modifier l'algorithme de Dijkstra en utilisant $d[u] + p(u, v) + h(v)$ comme priorité, au lieu de $d[u] + p(u, v)$.

L'algorithme A^* visite donc en priorité les sommets v tels que $h(v)$ est petit.

Algorithme A^*

Entrée : $G = (S, A)$ pondéré par $p : A \rightarrow \mathbb{R}^+$, $s, t \in S$ et
 $h : S \rightarrow \mathbb{R}^+$ une heuristique cohérente

Sortie : La distance $d(s, t)$ de s à t

$q \leftarrow$ file de priorité vide

Ajouter s à q avec priorité 0

$d \leftarrow [\infty, \dots, \infty]$

$d[s] \leftarrow 0$

Tant que $d[s] = \infty$:

 Extraire u de q de priorité minimum du

Si $d[u] = \infty$:

$d[u] \leftarrow d$

Pour tout voisin v de u :

 Ajouter v à q avec priorité

$d[u] + p(u, v) + h(v)$

Définition

Une heuristique h est dite :

- Admissible si $h(v) \leq d(v, t)$ pour tout $v \in S$.
- Cohérente (ou : monotone) si $h(v) \leq p(u, v) + h(u)$ pour tout $(u, v) \in A$.

Définition

Une heuristique h est dite :

- Admissible si $h(v) \leq d(v, t)$ pour tout $v \in S$.
- Cohérente (ou : monotone) si $h(v) \leq p(u, v) + h(u)$ pour tout $(u, v) \in A$.

Théorème (admis)

- Si h est admissible alors A^* renvoie la distance de s à t .
- Si h est admissible et cohérente et si la file de priorité q est implémentée avec un tas, alors A^* renvoie la distance de s à t en complexité $O(p \log(n))$.

On a donc théoriquement rien gagné en complexité par rapport à Dijkstra, mais en choisissant bien h on peut réduire le nombre de sommets explorés, en pratique.

Algorithme A^*

Idéalement, une heuristique doit être rapide à calculer ($O(1)$), admissible, cohérente et proche de la distance réelle (ce qui permet de se rapprocher plus rapidement de t).

Algorithme A^*

Idéalement, une heuristique doit être rapide à calculer ($O(1)$), admissible, cohérente et proche de la distance réelle (ce qui permet de se rapprocher plus rapidement de t).

Exemples :

- $h = 0$: on retrouve l'algorithme de Dijkstra.
- $h : v \rightarrow d(v, t)$: on parcourt exactement le plus court chemin de s à t mais on ne connaît pas $d(v, t)$ (c'est ce qu'on cherche).

- Distance euclidienne : si $v, t \in \mathbb{R}^k$, $h(v) = \sum_{i=1}^k \sqrt{(v_i - t_i)^2}$.

- Distance de Manhattan : si $v, t \in \mathbb{R}^k$, $h(v) = \sum_{i=1}^k |v_i - t_i|$.

Exercice

Montrer que l'algorithme A^* ne renvoie pas nécessairement la distance de s à t si h n'est pas admissible.

Algorithme de Bellman-Ford (HP)

L'algorithme de Bellman-Ford permet de résoudre le problème suivant par programmation dynamique :

Entrée : $G = (S, A)$ pondéré par $p : A \rightarrow \mathbb{R}$, $s \in S$

Sortie : Tableau d tel que $d[v] = d(s, v)$

Équation de Bellman-Ford

Soit $d_k(v)$ le poids minimum d'un chemin de s à v utilisant au plus k arêtes. Alors :

Algorithme de Bellman-Ford (HP)

L'algorithme de Bellman-Ford permet de résoudre le problème suivant par programmation dynamique :

Entrée : $G = (S, A)$ pondéré par $p : A \rightarrow \mathbb{R}$, $s \in S$

Sortie : Tableau d tel que $d[v] = d(s, v)$

Équation de Bellman-Ford

Soit $d_k(v)$ le poids minimum d'un chemin de s à v utilisant au plus k arêtes. Alors :

$$d_{k+1}(v) = \min_{(u,v) \in A} d_k(u) + w(u, v)$$

Si G ne contient pas de cycle de poids négatif alors $d_{n-1}(v) = d(v)$.

Remarque : on peut détecter un cycle de poids négatif en testant si $d_n(v) < d_{n-1}(v)$.

Algorithme de Bellman-Ford (HP)

Algorithme de Bellman-Ford

Entrée : $G = (S, A)$ pondéré par p et $s \in S$.

Sortie : d tel que $d[v]$ soit la distance de s à v .

Initialiser $d[s][k] = 0$ et $d[v][k] = \infty$ pour $v \neq s$

Pour $k \in \llbracket 0, n - 2 \rrbracket$:

Pour $v \in V$:

Pour $(u, v) \in A$:

$d[v][k + 1] = \min(d[v][k + 1], d[u][k] + w(u, v))$

Renvoyer d

Algorithme de Bellman-Ford (HP)

Parcourir tous les sommets puis tous les arcs (u, v) entrants dans v revient à parcourir tous les arcs du graphe :

Algorithme de Bellman-Ford

Entrée : $G = (S, A)$ pondéré par p et $s \in S$.

Sortie : d tel que $d[v]$ soit la distance de s à v .

Initialiser $d[s][k] = 0$ et $d[v][k] = \infty$ pour $v \neq s$

Pour $k \in \llbracket 0, n - 2 \rrbracket$:

Pour $v \in V$:

Pour $(u, v) \in A$:

$d[v][k + 1] = \min(d[v][k + 1], d[u][k] + w(u, v))$

Renvoyer d

Algorithme de Bellman-Ford (HP)

Comme on a juste besoin de stocker $d[\dots][k - 1]$ pour calculer $d[\dots][k]$:

Algorithme de Bellman-Ford

Entrée : $G = (S, A)$ pondéré par p et $s \in S$.

Sortie : d tel que $d[v]$ soit la distance de s à v .

Initialiser $d[s] = 0$ et $d[v] = \infty$ pour $v \neq s$

Pour $k \in \llbracket 0, n - 2 \rrbracket$:

Pour $(u, v) \in A$:

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

Renvoyer d

Complexité : $O(np)$ si G est représenté par une liste d'adjacence.

Algorithme de Bellman-Ford (HP)

```
int* bellman_ford(int s, int** g, int n) {
// g[i][j] = p si (i, j) est un arc de poids p, infini sinon
int* d = malloc(n * sizeof(int));
for(int i = 0; i < n; i++) d[i] = i == s ? 0 : -1;
for(int k = 0; k < n - 1; k++)
    for(int u = 0; u < n; u++)
        for(int v = 0; v < n; v++) {
            int x = d[u] + g[u][v];
            if(g[u][v] != 0 && (d[v] == -1 || d[v] > x))
                d[v] = x;
        }
return d;
}
```

Algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall permet de résoudre le problème suivant par programmation dynamique :

Entrée : $G = (S, A)$ pondéré par $p : A \rightarrow \mathbb{R}$

Sortie : Matrice d telle que $d[u][v] = \text{distance de } u \text{ à } v$

Remarque : Contrairement aux autres algorithmes, Floyd-Warshall calcule les distances depuis n'importe quel sommet à n'importe quel autre sommet.

Algorithme de Floyd-Warshall

Équation de Floyd-Warshall

Soit $d_k(u, v)$ la longueur d'un plus court chemin de u à v n'utilisant que des sommets intermédiaires de numéro $< k$ (∞ s'il n'existe pas). Alors :

Algorithme de Floyd-Warshall

Équation de Floyd-Warshall

Soit $d_k(u, v)$ la longueur d'un plus court chemin de u à v n'utilisant que des sommets intermédiaires de numéro $< k$ (∞ s'il n'existe pas). Alors :

$$d_{k+1}(u, v) = \min(d_k(u, v), d_k(u, k) + d_k(k, v))$$

Si G ne contient pas de cycle de poids négatif alors $d_n(u, v) = d(u, v)$.

Algorithme de Floyd-Warshall

Équation de Floyd-Warshall

Soit $d_k(u, v)$ la longueur d'un plus court chemin de u à v n'utilisant que des sommets intermédiaires de numéro $< k$ (∞ s'il n'existe pas). Alors :

$$d_{k+1}(u, v) = \min(d_k(u, v), d_k(u, k) + d_k(k, v))$$

Si G ne contient pas de cycle de poids négatif alors $d_n(u, v) = d(u, v)$.

Remarques :

- Comme pour Bellman-Ford, on peut utiliser une matrice $d[u][v]$ pour stocker la dernière valeur calculée de $d_k(u, v)$.
- On peut détecter un cycle de poids négatif en testant s'il existe u tel que $d_n(u, u) < 0$.
- Si G est représenté par matrice d'adjacence pondérée, on peut l'utiliser pour stocker $d_k(u, v)$ et initialiser $d_0(u, v)$.

Algorithme de Floyd-Warshall

Algorithme de Floyd-Warshall

Entrée : $G = (S, A)$ pondéré par p

Sortie : Matrice d telle que $d[u][v]$ = distance de u à v

Initialiser $d[u][v]$ à 0 si $u = v$ et à $p(u, v)$ sinon

Pour $k \in S$:

Pour $u \in S$:

Pour $v \in S$:

$d[u][v] = \min(d[u][v], d[u][k] + d[k][v])$

Renvoyer d

Complexité : $O(n^3)$ si G est représenté par une matrice d'adjacence.

Algorithme de Floyd-Warshall

Si G est représenté par matrice d'adjacence pondérée, on peut l'utiliser pour stocker $d_k(u, v)$ et initialiser $d_0(u, v)$:

```
void floyd_warshall(int n, int** g, int n) {  
    // g[i][j] = p si (i, j) est un arc de poids p, infini sinon  
    for(int k = 0; k < n; k++)  
        for(int u = 0; u < n; u++)  
            for(int v = 0; v < n; v++) {  
                int x = g[u][k] + g[k][v];  
                if(x < g[u][v])  
                    g[u][v] = x;  
            }  
}
```
