

Théorie des graphes et plus courts chemins

Algorithmes et structures de données, 2025-2026

P. Albuquerque (B410) et O. Malaspinas (A401), ISC, HEPIA

2026-05-27

En partie inspiré des supports de cours de P. Albuquerque

Les graphes

Exercice

- Établir la liste d'adjacence et appliquer l'algorithme de parcours en profondeur/largeur au graphe

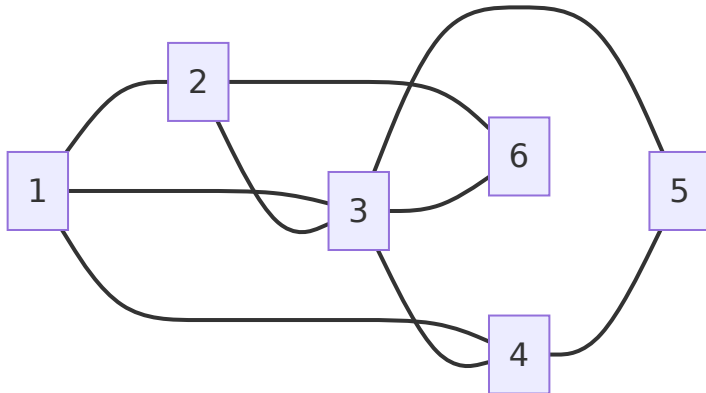


Illustration : parcours en profondeur

Parcours
en profondeur

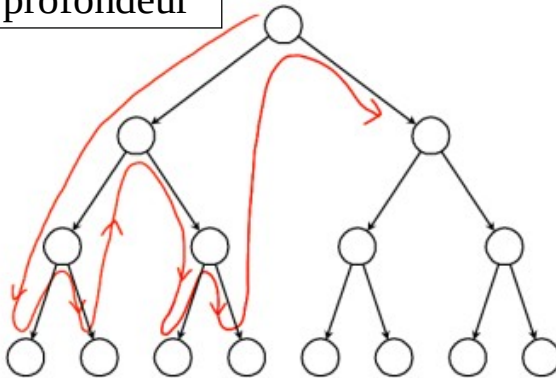


Figure 1 : Le parcours en profondeur. À quel parcours d'arbre cela ressemble-t-il ?

Idée générale

- Initialiser les sommets comme non-lus
- Visiter un sommet
- Pour chaque sommet visité, on visite un sommet adjacent s'il n'est pas encore visité, et ce récursivement.

Remarque

- La récursivité est équivalente à l'utilisation d'une **pile**.

Parcours en profondeur

Pseudo-code (5min)

Parcours en profondeur

Pseudo-code (5min)

```
initialiser(graphe) // tous les sommets sont non-visités
visiter(sommet, pile) // on choisit un sommet du graphe
tant que !est_vide(pile)
    dépiler(pile, (v,u))
    si u != visité
        ajouter (v,u) à arbre T
        visiter(u, pile)
```

Que fait visiter ?

Parcours en profondeur

Pseudo-code (5min)

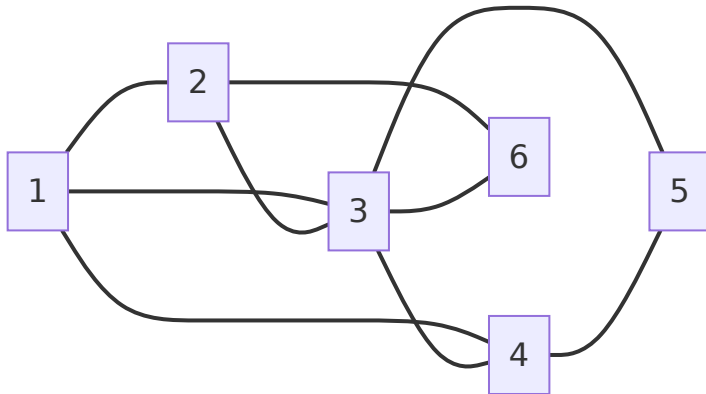
```
initialiser(graphe) // tous les sommets sont non-visités
visiter(sommet, pile) // on choisit un sommet du graphe
tant que !est_vide(pile)
    dépiler(pile, (v,u))
    si u != visité
        ajouter (v,u) à arbre T
        visiter(u, pile)
```

Que fait visiter ?

```
rien visiter(x, pile)
    marquer x comme visité
    pour chaque arête (x,w)
        si w != visité
            empiler(pile, (x,w))
```


Exercice

- Établir la liste d'adjacence et appliquer l'algorithme de parcours en profondeur au graphe



Interprétation des parcours

- Un graphe vu comme espace d'états (sommet : état, arête : action) ;
 - Labyrinthe ;
 - Arbre des coups d'un jeu.

Interprétation des parcours

- Un graphe vu comme espace d'états (sommet : état, arête : action) ;
 - Labyrinthe ;
 - Arbre des coups d'un jeu.
- BFS (Breadth-First) ou DFS (Depth-First) parcourent l'espace des états à la recherche du meilleur mouvement.
 - Les deux parcourent *tout* l'espace ;
 - Mais si l'arbre est grand, l'espace est gigantesque !

Interprétation des parcours

- Un graphe vu comme espace d'états (sommet : état, arête : action) ;
 - Labyrinthe ;
 - Arbre des coups d'un jeu.
- BFS (Breadth-First) ou DFS (Depth-First) parcourent l'espace des états à la recherche du meilleur mouvement.
 - Les deux parcourent *tout* l'espace ;
 - Mais si l'arbre est grand, l'espace est gigantesque !
- Quand on a un temps limité
 - BFS explore beaucoup de coups dans un futur proche ;
 - DFS explore peu de coups dans un futur lointain.

Plus courts chemins

Contexte : les réseaux (informatique, transport, etc.)

- Graphe orienté ;
- Source : sommet s ;
- Destination : sommet t ;
- Les arêtes ont des poids (coût d'utilisation, distance, etc.) ;
- Le coût d'un chemin est la somme des poids des arêtes d'un chemin.

Problème à résoudre

- Quel est le plus court chemin entre s et t ?

Exemples d'application de plus courts chemins

Devenir riches !

- On part d'un tableau de taux de change entre devises.
- Quelle est la meilleure façon de convertir l'or en dollar ?

Currency	£	Euro	¥	Franc	\$	Gold
UK Pound	1.0000	0.6853	0.005290	0.4569	0.6368	208.100
Euro	1.4599	1.0000	0.007721	0.6677	0.9303	304.028
Japanese Yen	189.050	129.520	1.0000	85.4694	120.400	39346.7
Swiss Franc	2.1904	1.4978	0.011574	1.0000	1.3929	455.200
US Dollar	1.5714	1.0752	0.008309	0.7182	1.0000	327.250
Gold	0.004816	0.003295	0.0000255	0.002201	0.003065	1.0000

Figure 2 : Taux de change.

Exemples d'application de plus courts chemins

Devenir riches !

- On part d'un tableau de taux de change entre devises.
- Quelle est la meilleure façon de convertir l'or en dollar ?

Currency	£	Euro	¥	Franc	\$	Gold
UK Pound	1.0000	0.6853	0.005290	0.4569	0.6368	208.100
Euro	1.4599	1.0000	0.007721	0.6677	0.9303	304.028
Japanese Yen	189.050	129.520	1.0000	85.4694	120.400	39346.7
Swiss Franc	2.1904	1.4978	0.011574	1.0000	1.3929	455.200
US Dollar	1.5714	1.0752	0.008309	0.7182	1.0000	327.250
Gold	0.004816	0.003295	0.0000255	0.002201	0.003065	1.0000

Figure 2 : Taux de change.

- 1kg d'or \Rightarrow 327.25 dollars
- 1kg d'or \Rightarrow 208.1 livres \Rightarrow 327 dollars
- 1kg d'or \Rightarrow 455.2 francs \Rightarrow 304.39 euros \Rightarrow 327.28 dollars

Exemples d'application de plus courts chemins

Formulation sous forme d'un graphe : Comment (3min) ?

Currency	£	Euro	¥	Franc	\$	Gold
UK Pound	1.0000	0.6853	0.005290	0.4569	0.6368	208.100
Euro	1.4599	1.0000	0.007721	0.6677	0.9303	304.028
Japanese Yen	189.050	129.520	1.0000	85.4694	120.400	39346.7
Swiss Franc	2.1904	1.4978	0.011574	1.0000	1.3929	455.200
US Dollar	1.5714	1.0752	0.008309	0.7182	1.0000	327.250
Gold	0.004816	0.003295	0.0000255	0.002201	0.003065	1.0000

Figure 3 : Taux de change.

Exemples d'application de plus courts chemins

Formulation sous forme d'un graphe : Comment (3min) ?

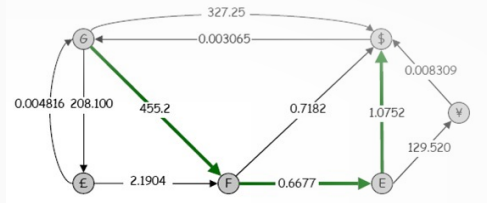


Figure 4 : Graphes des taux de change.

- Un sommet par devise ;
- Une arête orientée par transaction possible avec le poids égal au taux de change ;
- Trouver le chemin qui maximise le produit des poids.

Exemples d'application de plus courts chemins

Formulation sous forme d'un graphe : Comment (3min) ?

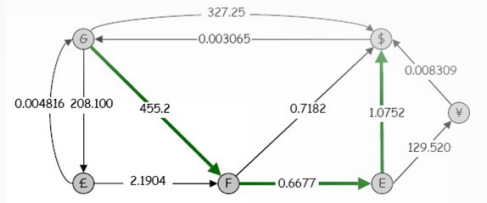


Figure 4 : Graphes des taux de change.

- Un sommet par devise ;
- Une arête orientée par transaction possible avec le poids égal au taux de change ;
- Trouver le chemin qui maximise le produit des poids.

Problème

- On aimerait plutôt avoir une somme...

Exemples d'application de plus courts chemins

Conversion du problème en plus courts chemins

- Soit $\text{taux}(u, v)$ le taux de change entre la devise u et v .
- On pose $w(u, v) = -\log(\text{taux}(u, v))$
- Trouver le chemin poids minimal pour les poids w .

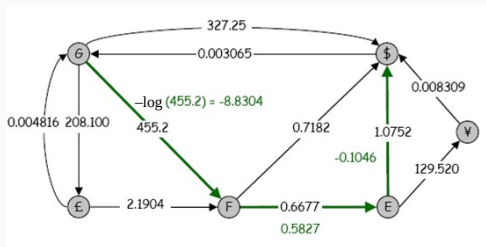


Figure 5 : Graphe des taux de change avec logs.

- Cette conversion se base sur l'idée que

$$\log(u \cdot v) = \log(u) + \log(v).$$

Applications de plus courts chemins

Quelles applications voyez-vous ?

Quelles applications voyez-vous ?

- Déplacement d'un robot ;
- Planification de trajet / trafic urbain ;
- Routage de télécommunications ;
- Réseau électrique optimal ;
- ...

Algorithmes de plus courts chemins

Rappel de la problématique

- Graphe orienté ;
- Source : sommet s ;
- Destination : sommet t ;
- Les arêtes ont des poids (coût d'utilisation, distance, etc.) ;
- Le coût d'un chemin est la somme des poids des arêtes d'un chemin.

Problème à résoudre

- Quel est le plus court chemin entre s et t .

Plus courts chemins à source unique

- Soit un graphe, $G = (V, E)$, une fonction de pondération $w : E \rightarrow \mathbb{R}$, et un sommet $s \in V$
 - Trouver pour tout sommet $v \in V$, le chemin de poids minimal reliant s à v .
- Algorithmes standards :
 - Dijkstra (arêtes de poids positif seulement) ;
 - Bellman-Ford (arêtes de poids positifs ou négatifs, mais sans cycles négatifs).
- Comment résoudre le problèmes si tous les poids sont les mêmes ?

Plus courts chemins à source unique

- Soit un graphe, $G = (V, E)$, une fonction de pondération $w : E \rightarrow \mathbb{R}$, et un sommet $s \in V$
 - Trouver pour tout sommet $v \in V$, le chemin de poids minimal reliant s à v .
- Algorithmes standards :
 - Dijkstra (arêtes de poids positif seulement) ;
 - Bellman-Ford (arêtes de poids positifs ou négatifs, mais sans cycles négatifs).
- Comment résoudre le problèmes si tous les poids sont les mêmes ?
- Un parcours en largeur !

Algorithme de Dijkstra

Comment chercher pour un plus court chemin ?

Algorithme de Dijkstra

Comment chercher pour un plus court chemin ?

si $\text{distance}(u,v) > \text{distance}(u,w) + \text{distance}(w,v)$
on passe par w plutôt qu'aller directement

Algorithme de Dijkstra (1 à 5)

- D est le tableau des distances au sommet 1 : $D[7]$ est la distance de 1 à 7.
- Le chemin est pas forcément direct.
- S est le tableau des sommets visités.

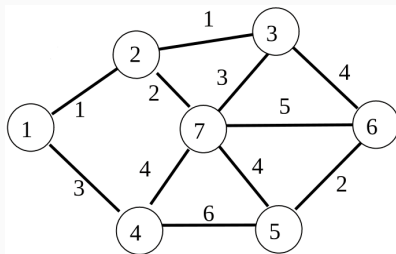


Figure 6 : Initialisation.

Algorithme de Dijkstra (1 à 5)

- D est le tableau des distances au sommet 1 : $D[7]$ est la distance de 1 à 7.
- Le chemin est pas forcément direct.
- S est le tableau des sommets visités.

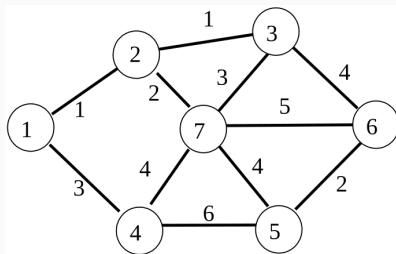


Figure 6 : Initialisation.

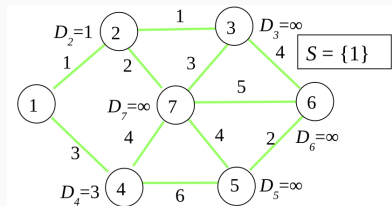


Figure 7 : 1 visité, $D[2]=1$, $D[4]=3$.

Algorithme de Dijkstra (1 à 5)

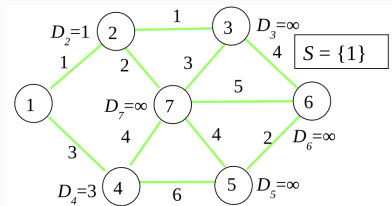


Figure 8 : Plus court est 2.

Algorithme de Dijkstra (1 à 5)

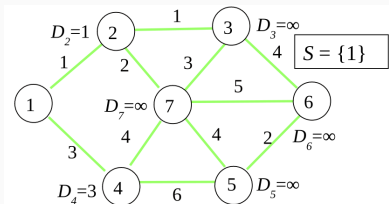


Figure 8 : Plus court est 2.

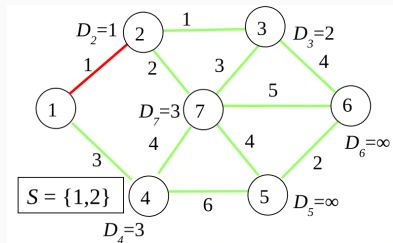


Figure 9 : 2 visité, $D[3] = 2$, $D[7] = 3$.

Algorithme de Dijkstra (1 à 5)

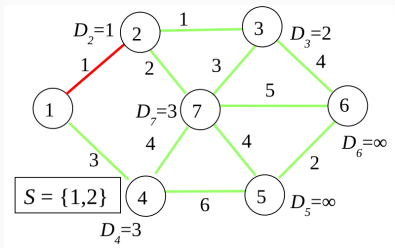


Figure 10 : Plus court est 3.

Algorithme de Dijkstra (1 à 5)

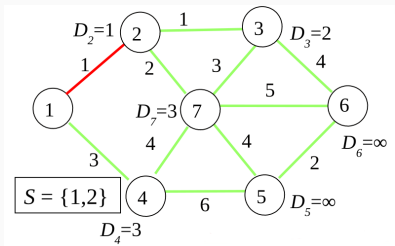


Figure 10 : Plus court est 3.

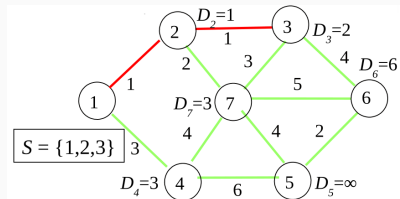


Figure 11 : 3 visité, $D[7]=3$ inchangé, $D[6]=6$.

Algorithme de Dijkstra (1 à 5)

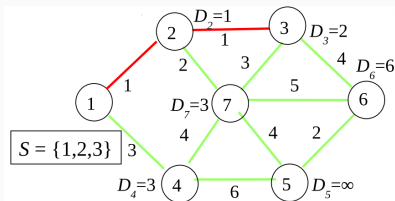


Figure 12 : Plus court est 4 ou 7.

Algorithme de Dijkstra (1 à 5)

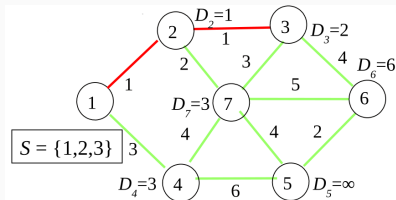


Figure 12 : Plus court est 4 ou 7.

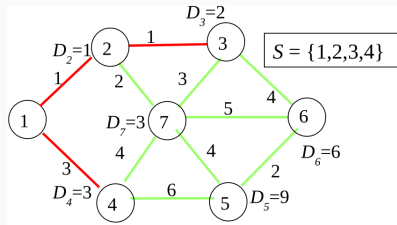


Figure 13 : 4 visité, $D[7]=3$ inchangé, $D[5]=9$.

Algorithme de Dijkstra (1 à 5)

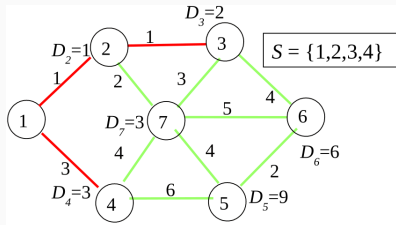


Figure 14 : Plus court est 7.

Algorithme de Dijkstra (1 à 5)

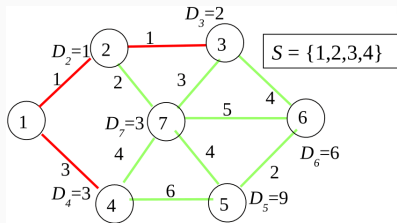


Figure 14 : Plus court est 7.

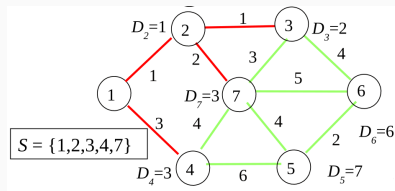


Figure 15 : 7 visité, $D[5]=7$, $D[6]=6$ inchangé.

Algorithme de Dijkstra (1 à 5)

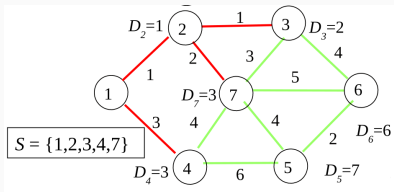


Figure 16 : Plus court est 6.

Algorithme de Dijkstra (1 à 5)

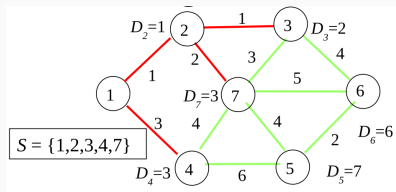


Figure 16 : Plus court est 6.

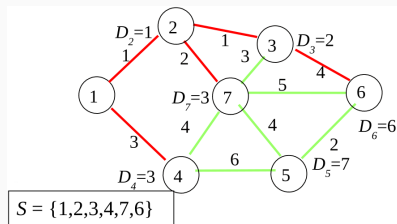


Figure 17 : 6 visité, $D[5]=7$ inchangé.

Algorithme de Dijkstra (1 à 5)

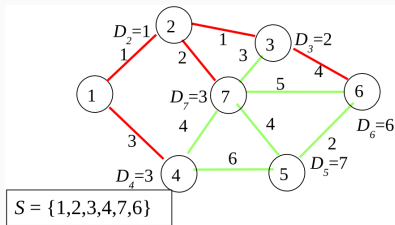


Figure 18 : Plus court est 5 et c'est la cible.

Algorithme de Dijkstra (1 à 5)

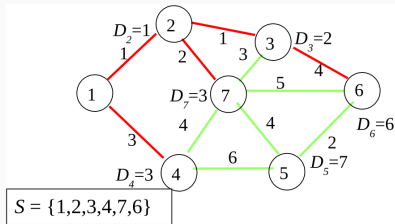


Figure 18 : Plus court est 5 et c'est la cible.

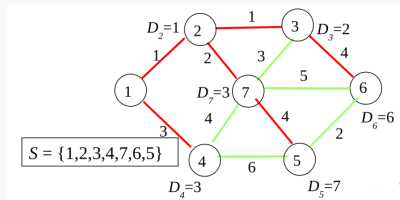


Figure 19 : The end, tous les sommets ont été visités.

Algorithme de Dijkstra

Idée générale

- On assigne à chaque noeud une distance 0 pour s , ∞ pour les autres.
- Tous les noeuds sont marqués non-visités.
- Depuis le noeud courant, on suit chaque arête du noeud vers un sommet non visité et on calcule le poids du chemin à chaque voisin et on met à jour sa distance si elle est plus petite que la distance du noeud.
- Quand tous les voisins du noeud courant ont été visités, le noeud est mis à visité (il ne sera plus jamais visité).
- Continuer avec le noeud à la distance la plus faible.
- L'algorithme est terminé lorsque le noeud de destination est marqué comme visité, ou qu'on n'a plus de noeuds qu'on peut visiter et que leur distance est infinie.

Algorithme de Dijkstra

Pseudo-code (5min, matrix)

Algorithme de Dijkstra

Pseudo-code (5min, matrix)

```
tab dijkstra(graph, s, t)
  pour chaque v dans graphe
    distance[v] = infini
  distance[s] = 0
  q = ajouter(q, s) // q est une liste
  tant que non_vide(q)
    // sélection de u t.q. la distance dans q est min
    u = min(q, distance)
    si u == t // on a atteint la cible
      retourne distance
    q = remove(q, u)
    // voisin de u encore dans q
    pour chaque v dans voisinage(u, q)
      // on met à jour la distance du voisin en passant par u
      n_distance = distance[u] + w(u, v)
      si n_distance < distance[v]
        distance[v] = n_distance
  retourne distance
```

Algorithme de Dijkstra

- Cet algorithme, nous donne le plus court chemin mais...
- ne nous donne pas le chemin !

Comment modifier l'algorithme pour avoir le chemin ?

Algorithme de Dijkstra

- Cet algorithme, nous donne le plus court chemin mais...
- ne nous donne pas le chemin !

Comment modifier l'algorithme pour avoir le chemin ?

- Pour chaque nouveau noeud à visiter, il suffit d'enregistrer d'où on est venu !
- On a besoin d'un tableau precedent.

Modifier le pseudo-code ci-dessus pour ce faire (3min matrix)

Algorithme de Dijkstra

```
tab, tab dijkstra(graph, s, t)
  pour chaque v dans graphe
    distance[v] = infini
    precedent[v] = indéfini
  distance[s] = 0
  q = ajouter(q, s)
  tant que non_vide(q)
    // sélection de u t.q. la distance dans q est min
    u = min(q, distance)
    si u == t
      retourne distance, precedent
    q = remove(q, u)
    // voisin de u encore dans q
    pour chaque v dans voisinage(u, q)
      n_distance = distance[u] + w(u, v)
      si n_distance < distance[v]
        distance[v] = n_distance
        precedent[v] = u
  retourne distance, precedent
```


Algorithme de Dijkstra

Comment reconstruire un chemin ?

Algorithme de Dijkstra

Comment reconstruire un chemin ?

```
pile parcours(precedent, s, t)
    sommets = vide
    u = t
    // on a atteint t ou on ne connait pas de chemin
    si u != s && precedent[u] != indéfini
        tant que vrai
            sommets = empiler(sommets, u)
            u = precedent[u]
        si u == s // la source est atteinte
            retourne sommets
    retourne sommets
```

Algorithme de Dijkstra amélioré

On peut améliorer l'algorithme

- Avec une file de priorité !

Une file de priorité est

- Une file dont chaque élément possède une priorité,
- Elle existe en deux saveurs : `min` ou `max` :
 - File `min` : les éléments les plus petits sont retirés en premier.
 - File `max` : les éléments les plus grands sont retirés en premier.
- On regarde l'implémentation de la `max`.

Comment on fait ça ?

Algorithme de Dijkstra amélioré

On peut améliorer l'algorithme

- Avec une file de priorité !

Une file de priorité est

- Une file dont chaque élément possède une priorité,
- Elle existe en deux saveurs : `min` ou `max` :
 - File `min` : les éléments les plus petits sont retirés en premier.
 - File `max` : les éléments les plus grands sont retirés en premier.
- On regarde l'implémentation de la `max`.

Comment on fait ça ?

- On insère les éléments à haute priorité tout devant dans la file !

Les files de priorité

Trois fonction principales

```
booléen est_vide(element) // triviale  
element enfiler(element, data, priorite)  
data defiler(element)  
rien changer_priorite(element, data, priorite)  
nombre priorite(element) // utilitaire
```

Pseudo-implémentation : structure (1min)

Les files de priorité

Trois fonction principales

```
booléen est_vide(element) // triviale  
element enfiler(element, data, priorite)  
data defiler(element)  
rien changer_priorite(element, data, priorite)  
nombre priorite(element) // utilitaire
```

Pseudo-implémentation : structure (1min)

```
struct element  
    data  
    priorite  
    element suivant
```

Les files de priorité

Pseudo-implémentation : enfiler (2min)

Les files de priorité

Pseudo-implémentation : enfiler (2min)

```
element enfiler(element, data, priorite)
    n_element = creer_element(data, priorite)
    si est_vide(element)
        retourne n_element
    si priorite(n_element) > priorite(element)
        n_element.suivant = element
        retourne n_element
    sinon
        tmp = element
        prec = element
        tant que !est_vide(tmp)
            && priorite(n_element) < priorite(tmp)
                prec = tmp
                tmp = tmp.suivant
        prec.suivant = n_element
        n_element.suivant = tmp
```


Pseudo-implémentation : defiler (2min)

Pseudo-implémentation : defiler (2min)

```
data, element defiler(element)
    si est_vide(element)
        retourne AARGL!
    sinon
        tmp = element.data
        n_element = element.suivant
        liberer(element)
        retourne tmp, n_element
```

Algorithme de Dijkstra avec file de priorité min

```
distance, precedent dijkstra(graphe, s, t):  
    fp = file_p_vide()  
    distance[s] = 0  
    pour v dans sommets(graphe)  
        si v != s  
            distance[v] = infini  
            precedent[v] = indéfini  
            fp = enfiler(fp, v, distance[v])  
    tant que !est_vide(fp)  
        u, fp = defiler(fp)  
        si u == t  
            retourne distance, precedent  
        pour v dans voisinage de u  
            n_distance = distance[u] + w(u, v)  
            si n_distance < distance[v]  
                distance[v] = n_distance  
                precedent[v] = u  
                fp = changer_priorite(fp, u, n_distance)
```

Algorithme de Dijkstra avec file

```
distance dijkstra(graphe, s, t)
----- $\mathcal{O}(V \cdot V)$ -----
    distance[s] = 0
    fp = file_p_vide()
    pour v dans sommets(graphe) //  $\mathcal{O}(|V|)$ 
        si v != s
            distance[v] = infini
    fp = enfiler(fp, s, distance[s]) //  $\mathcal{O}(|V|)$ 
    ----- $\mathcal{O}(V * V)$ -----
    tant que !est_vide(fp)
        u, fp = defiler(fp) //  $\mathcal{O}(1)$ 
        -----
        pour v dans voisinage de u //  $\mathcal{O}(|E|)$ 
            n_distance = distance[u] + w(u, v)
            si n_distance < distance[v]
                distance[v] = n_distance
                fp = changer_priorite(fp, v, n_distance) //  $\mathcal{O}(|V|)$ 
        -----
    retourne distance
```

- Total : $\mathcal{O}(|V|^2 + |E| \cdot |V|)$:
 - Graphe dense : $\mathcal{O}(|V|^3)$

Algorithme de Dijkstra avec file

On peut faire mieux

- Avec une meilleure implémentation de la file de priorité :
 - Tas binaire : $\mathcal{O}(|V| \log |V| + |E| \log |V|)$.
 - Tas de Fibonnacci : $\mathcal{O}(|V| + |E| \log |V|)$
- Graphe dense : $\mathcal{O}(|V|^2 \log |V|)$.
- Graphe peu dense : $\mathcal{O}(|V| \log |V|)$.

Algorithme de Dijkstra (exercice, 5min)

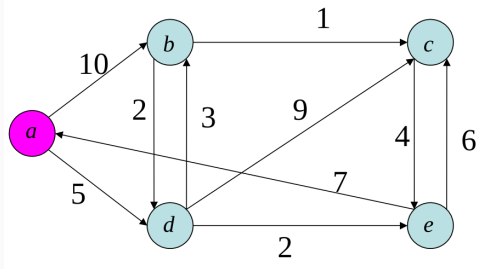


Figure 20 : L'exercice.

- Donner la liste de priorité, puis...

A chaque étape donner :

- Le tableau des distances à a ;
- Le tableau des prédécesseurs ;
- L'état de la file de priorité.

Algorithme de Dijkstra (corrigé)

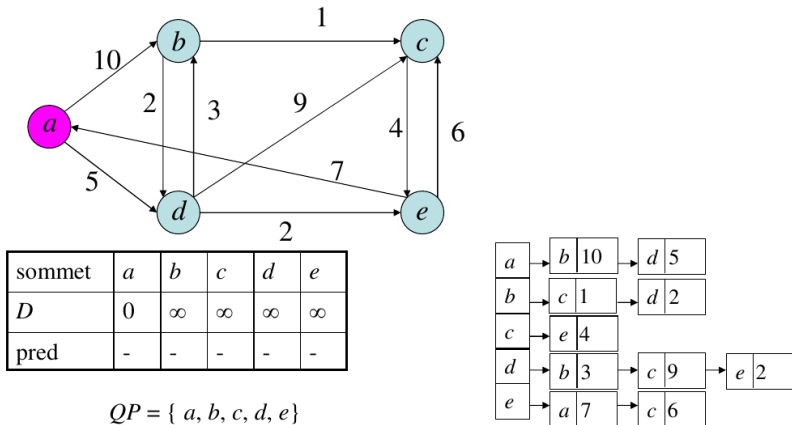


Figure 21 : Le corrigé partie 1.

Algorithme de Dijkstra (corrigé)

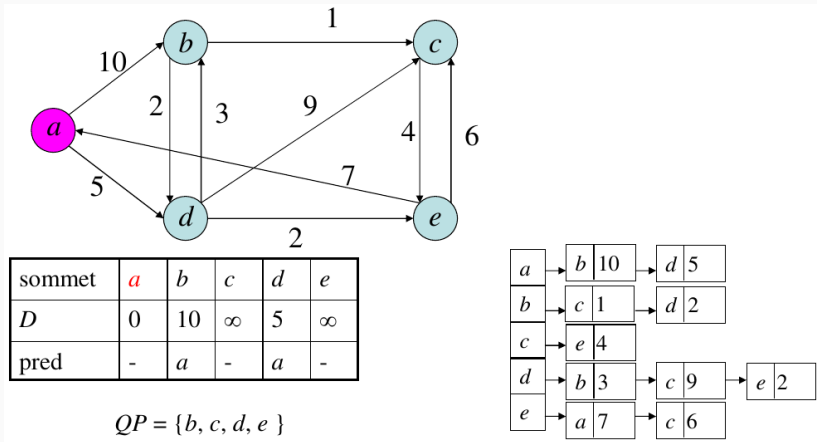


Figure 22 : Le corrigé partie 2.

Algorithme de Dijkstra (corrigé)

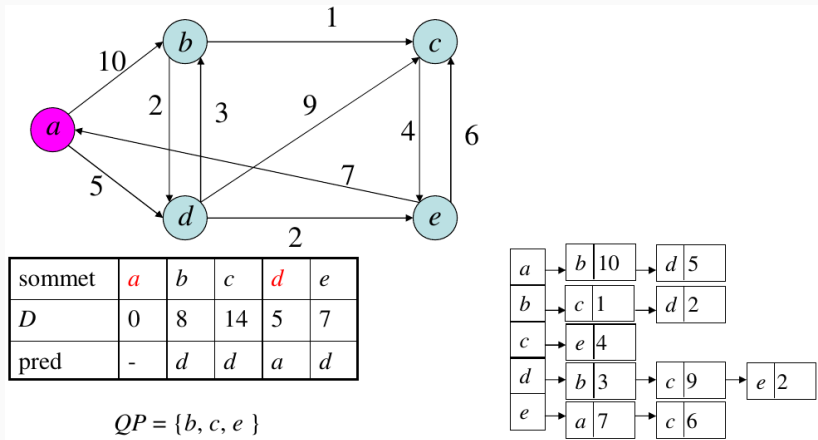


Figure 23 : Le corrigé partie 3.

Algorithme de Dijkstra (corrigé)

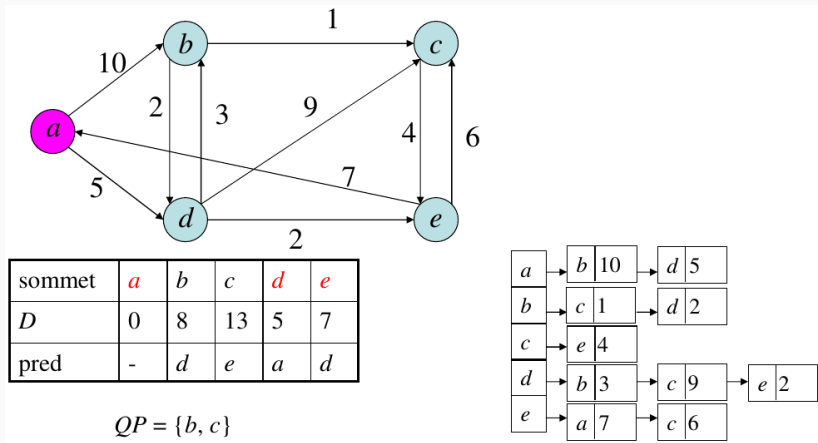


Figure 24 : Le corrigé partie 4.

Algorithme de Dijkstra (corrigé)

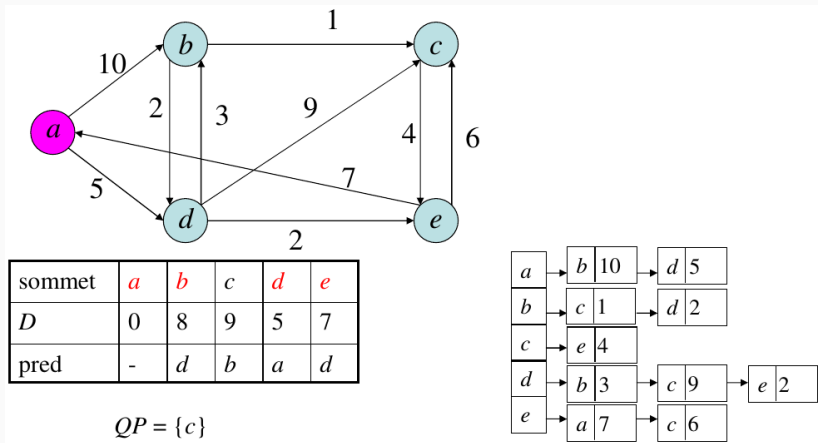


Figure 25 : Le corrigé partie 5.

Algorithme de Dijkstra (corrigé)

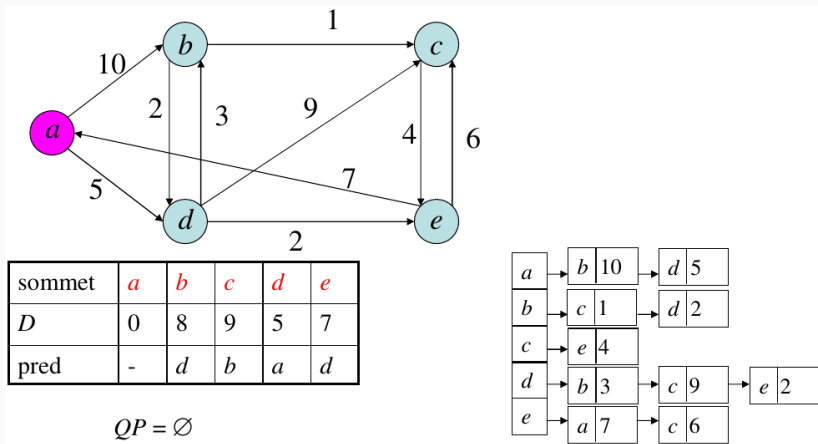


Figure 26 : Le corrigé partie 6.

Limitation de l'algorithme de Dijkstra

Que se passe-t-il pour ?

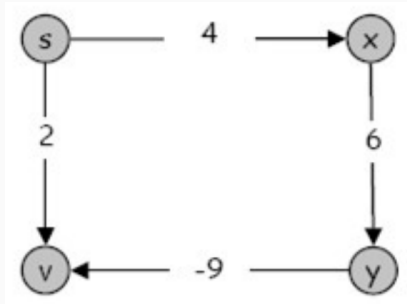


Figure 27 : Exemple.

Quel est le problème ?

Limitation de l'algorithme de Dijkstra

Que se passe-t-il pour ?

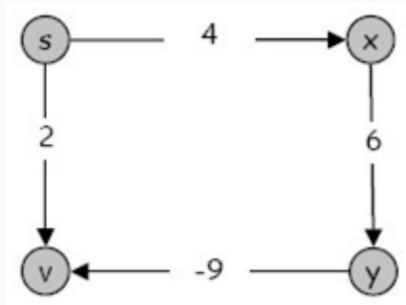


Figure 27 : Exemple.

Quel est le problème ?

- L'algorithme n'essaiera jamais le chemin $s \rightarrow x \rightarrow y \rightarrow v$ et prendra direct $s \rightarrow v$.
- Ce problème n'apparaît que s'il y a des poids négatifs.

Plus cours chemin pour toute paire de sommets

Comment faire pour avoir toutes les paires ?

Plus cours chemin pour toute paire de sommets

Comment faire pour avoir toutes les paires ?

- Appliquer Dijkstra sur tous les sommets d'origine.
- Complexité :
 - Graphe dense : $\mathcal{O}(|V|)\mathcal{O}(|V|^2 \log |V|) = \mathcal{O}(|V|^3 \log |V|)$.
 - Graphe peu dense : $\mathcal{O}(|V|)\mathcal{O}(|V| \log |V|) = \mathcal{O}(|V|^2 \log |V|)$.

Plus cours chemin pour toute paire de sommets

Comment faire pour avoir toutes les paires ?

- Appliquer Dijkstra sur tous les sommets d'origine.
- Complexité :
 - Graphe dense : $\mathcal{O}(|V|)\mathcal{O}(|V|^2 \log |V|) = \mathcal{O}(|V|^3 \log |V|)$.
 - Graphe peu dense : $\mathcal{O}(|V|)\mathcal{O}(|V| \log |V|) = \mathcal{O}(|V|^2 \log |V|)$.

Solution alternative : Floyd–Warshall

- Pour toutes paires de sommets $u, v \in V$, trouver le chemin de poids minimal reliant u à v .
- Complexité $\mathcal{O}(|V|^3)$, indiqué pour graphes denses.
- Fonctionne avec la matrice d'adjacence.