Chapitre 2 : Les graphes

Option Informatique – MP

Lycée Chateaubriand

Table des matières

Plan

Définition 1 (Graphe pondéré)

On appelle graphe pondéré un graphe $G = (S, A)$ muni d'une application $p : A \to X$ où X est l'ensemble des valeurs (la plupart du temps $\mathbb{R}$)

Cela revient à affecter un « poids » à chaque arête.

Poids d'un chemin

Définition 2

Soit $G = (S, A, p)$ un graphe pondéré. Soit $x_0, x_1, \ldots, x_n$ un chemin du graphe, le poids du chemin est la somme des poids des arêtes qui constituent le chemin.

Poids d'un chemin

Définition 2

Soit $G = (S, A, p)$ un graphe pondéré. Soit $x_0, x_1, \ldots, x_n$ un chemin du graphe, le poids du chemin est la somme des poids des arêtes qui constituent le chemin.

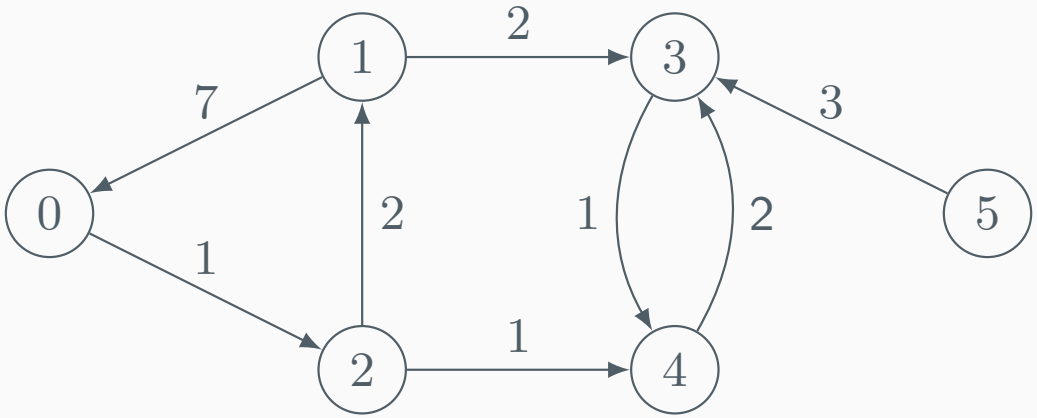
Dans le cas où notre graphe représente des routes, la pondération sera la longueur ou le temps nécessaire au parcours de la route. Dans ce cadre, on voit que la recherche d'un chemin de poids minimum entre deux nœuds est crucial.

Attention

Attention

Ne pas confondre la longueur (nombre d'arêtes) d'un chemin et son poids (somme des poids des arêtes).

Exemple



Pour aller de 0 à 3 on a un chemin de poids 4 et un chemin de poids 5.

Graphe pondérés

On veut implémenter la matrice d'adjacence d'un graphe pondéré. Il y a deux approches :

Graphe pondérés

On veut implémenter la matrice d'adjacence d'un graphe pondéré. Il y a deux approches :

- On travaille avec une matrice dont les éléments sont des couples. Le premier élément est un booléen qui dit s'il l'arête existe et le deuxième est le poids de l'arête. On a donc
  `type graphePondere = (bool*float) array array`

Graphe pondérés

On veut implémenter la matrice d'adjacence d'un graphe pondéré. Il y a deux approches :

- On travaille avec une matrice dont les éléments sont des couples. Le premier élément est un booléen qui dit s'il l'arête existe et le deuxième est le poids de l'arête. On a donc
  `type graphePondere = (bool*float) array array`
- On travaille avec une matrice d'entiers ou flottants mais en ayant défini une valeur `Inf`pour le cas où l'arête n'existe pas

Graphe pondérés

```
type int_inf = Inf | Entier of int;;

let add x y =


;;

let mini x y =


;;

type graph_pond_mat = int_inf array array;;
```

Graphe pondérés

```
type int_inf = Inf | Entier of int;;

let add x y = match x,y with
  |Inf,_ -> Inf
  |_, Inf -> Inf
  |Entier(a),Entier(b) -> Entier(a+b);;

let mini x y = match x,y with
  |Inf,_-> y
  |_,Inf -> x
  |Entier(a),Entier(b) -> Entier(min a b);;

type graph_pond_mat = int_inf array array;;
```

Graphes pondérés

Il y a plusieurs manières de représenter un graphe pondéré avec des listes d'adjacences.

Graphes pondérés

Il y a plusieurs manières de représenter un graphe pondéré avec des listes d'adjacences.

- On peut considérer des listes de couples `(int * float)`où la première coordonnée est le voisin et la deuxième le poids du chemin.

Graphes pondérés

Il y a plusieurs manières de représenter un graphe pondéré avec des listes d'adjacences.

- On peut considérer des listes de couples `(int * float)`où la première coordonnée est le voisin et la deuxième le poids du chemin.
- On peut considérer deux listes. La liste des adjacences et la liste des poids.

Définition du problème

On considère un graphe connexe pondéré.

On se donne deux sommets et on veut déterminer un/le chemin de poids minimal entre ces deux sommets.

Il y a une condition nécessaire évidente pour qu'un tel chemin existe.

Définition du problème

On considère un graphe connexe pondéré.

On se donne deux sommets et on veut déterminer un/le chemin de poids minimal entre ces deux sommets.

Il y a une condition nécessaire évidente pour qu'un tel chemin existe.

Il faut qu'il n'existe pas de cycles dont le poids est strictement négatif.

Définition du problème

Ce problème modélise par exemple la recherche du plus court chemin entre deux points par un logiciel de GPS.

Il existe deux algorithmes classiques :

- l'algorithme de Floyd-Warshall (1959)
- l'algorithme de Dijkstra (1959)

Algorithme de Floyd Warshall

Pour cet algorithme il est plus simple de représenter le graphe par une matrice d'adjacence.

On notera donc $0, 1, 2 \ldots, (n-1)$ les sommets du graphe et $A = (a_{ij}) \in \mathscr{M}_n(\overline{\mathbb{R}})$ la matrice d'adjacence où pour tout couple (i, j), a_{ij} est le poids de l'arête allant du sommet i au sommet j en prenant la convention $a_{ij} = +\infty$ s'il n'y a pas d'arête entre i et j.

Ici les indices des éléments de la matrice vont de 0 à $n-1$ contrairement à l'usage en mathématique.

Algorithme de Floyd Warshall

L'idée est alors de calculer de proche en proche les matrices $A^{(k)} = (a_{ij}^{(k)})$ où $a_{ij}^{(k)}$ est le poids minimal d'un chemin allant de i à j **et ne passant autrement que par des sommets** $l \leq k$.

Algorithme de Floyd Warshall

Théorème

Avec les notations précédentes

- $A^{(-1)} = A$
- $\forall (i,j) \in [\![0, n-1]\!]^2, \forall k \in [\![-1, n-2]\!]$,

$$a_{ij}^{(k+1)} =$$

Algorithme de Floyd Warshall

Théorème

Avec les notations précédentes

- $A^{(-1)} = A$
- $\forall (i,j) \in [\![0, n-1]\!]^2, \forall k \in [\![-1, n-2]\!]$,

$$a_{ij}^{(k+1)} = \min\left(a_{ij}^{(k)}, a_{i,k+1}^{(k)} + a_{k+1,j}^{(k)}\right)$$

Démonstration

Il suffit de voir qu'un chemin réalisant le minimum en allant de i à j en ne passant que par des sommets de $[\![0, k+1]\!]$ peut :

Démonstration

Il suffit de voir qu'un chemin réalisant le minimum en allant de i à j en ne passant que par des sommets de $[\![0, k+1]\!]$ peut :

- Ne pas passer par $k+1$: dans ce cas $a_{ij}^{(k+1)} = a_{ij}^{(k)}$

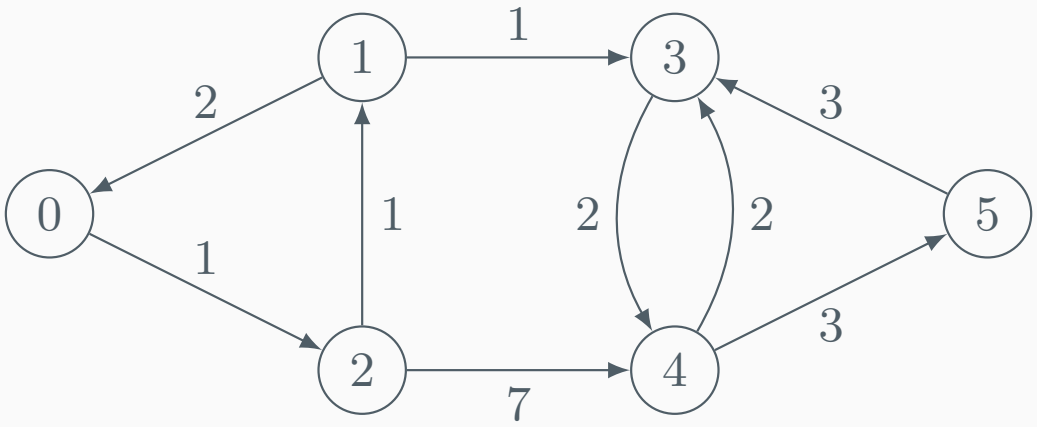
Démonstration

Il suffit de voir qu'un chemin réalisant le minimum en allant de i à j en ne passant que par des sommets de $[\![0, k+1]\!]$ peut :

- Ne pas passer par $k+1$: dans ce cas $a_{ij}^{(k+1)} = a_{ij}^{(k)}$
- Passer par $k+1$: dans ce cas, il ne passe qu'une fois par $k+1$ car, entre deux éventuels passages par $k+1$, le chemin décrit un cycle qui est nécessairement de poids positif. De ce fait $a_{ij}^{(k+1)} = a_{i,k+1}^{(k)} + a_{k+1,j}^{(k)}$.

Exemple

On considère le graphe pondéré suivant

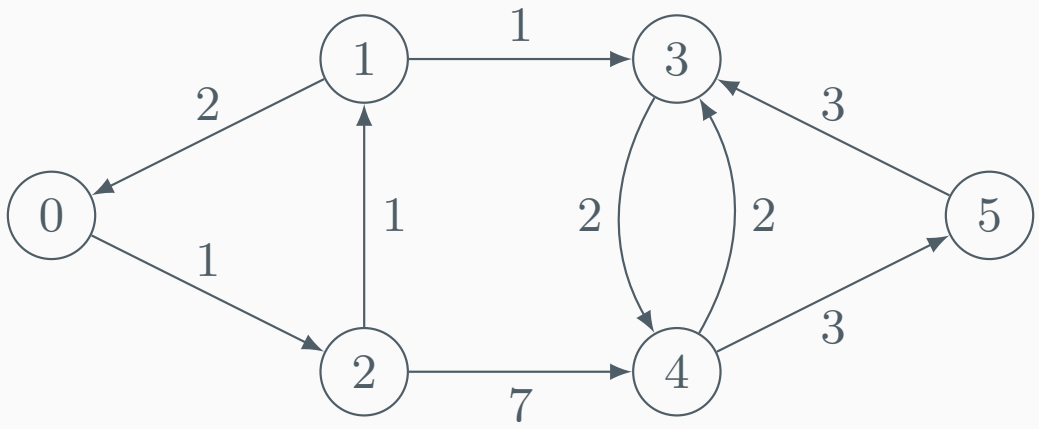


On a donc

$$A^{(-1)} = \begin{pmatrix} \infty & \infty & 1 & \infty & \infty & \infty \\ 2 & \infty & \infty & 1 & \infty & \infty \\ \infty & 1 & \infty & \infty & 7 & \infty \\ \infty & \infty & \infty & \infty & 2 & \infty \\ \infty & \infty & \infty & 2 & \infty & 3 \\ \infty & \infty & \infty & 3 & \infty & \infty \end{pmatrix}.$$

Exemple

On considère le graphe pondéré suivant

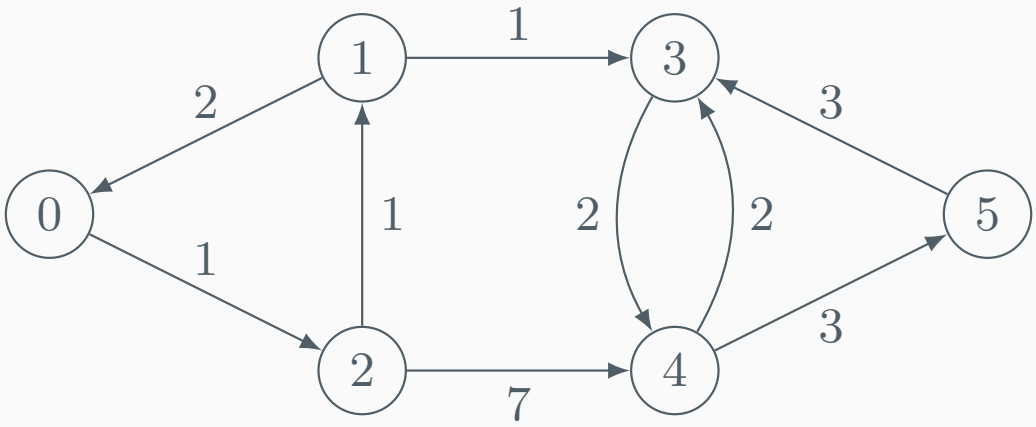


puis

$$A^{(0)} = \begin{pmatrix} \infty & \infty & 1 & \infty & \infty & \infty \\ 2 & \infty & \mathbf{3} & 1 & \infty & \infty \\ \infty & 1 & \infty & \infty & 7 & \infty \\ \infty & \infty & \infty & \infty & 2 & \infty \\ \infty & \infty & \infty & 2 & \infty & 3 \\ \infty & \infty & \infty & 3 & \infty & \infty \end{pmatrix}.$$

Exemple

On considère le graphe pondéré suivant

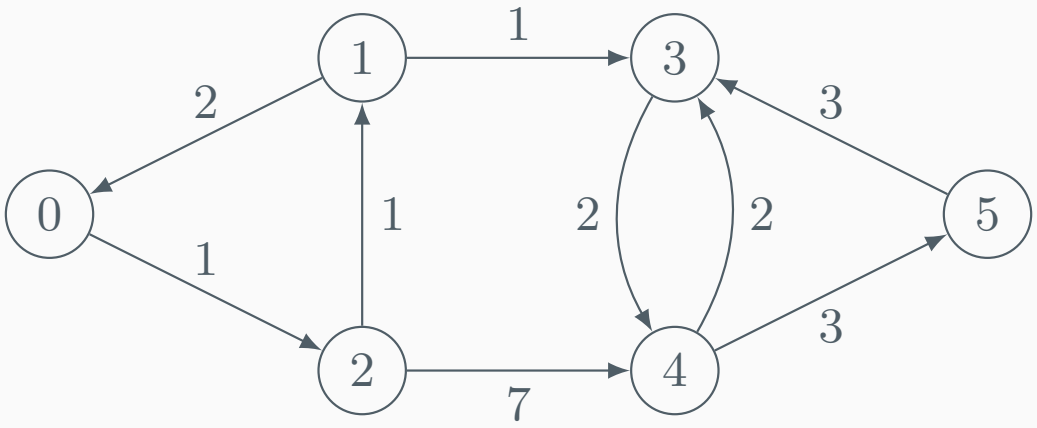


puis

$$A^{(1)} =$$

Exemple

On considère le graphe pondéré suivant

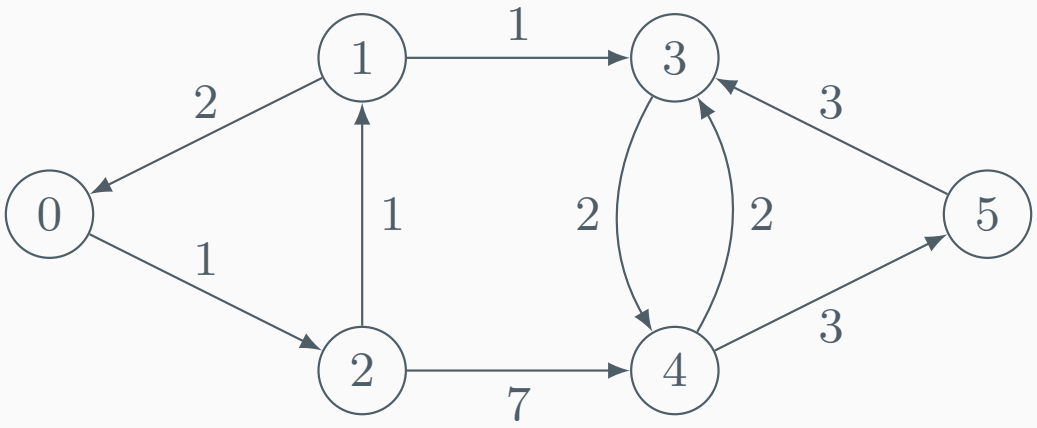


puis

$$A^{(1)} = \begin{pmatrix} \infty & \infty & 1 & \infty & \infty & \infty \\ 2 & \infty & 3 & 1 & \infty & \infty \\ 3 & 1 & 4 & 2 & 7 & \infty \\ \infty & \infty & \infty & \infty & 2 & \infty \\ \infty & \infty & \infty & 2 & \infty & 3 \\ \infty & \infty & \infty & 3 & \infty & \infty \end{pmatrix}.$$

Exemple

On considère le graphe pondéré suivant

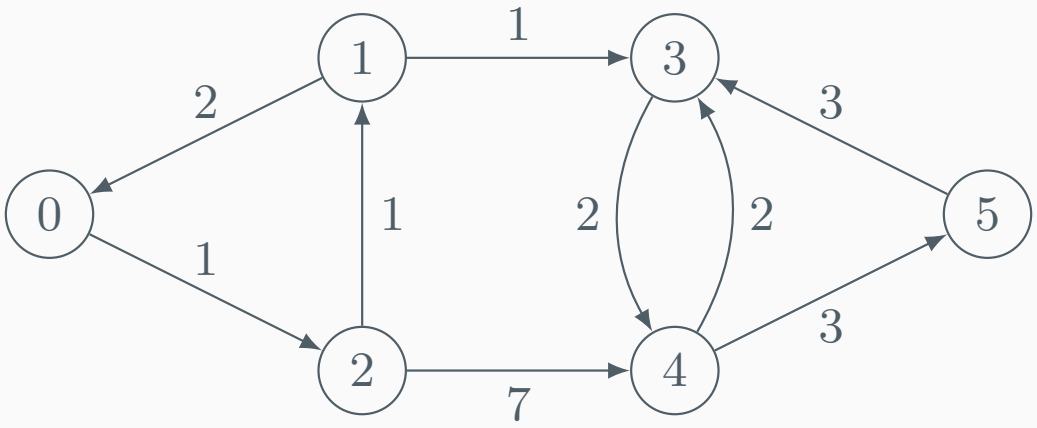


puis

$$A^{(2)} =$$

Exemple

On considère le graphe pondéré suivant

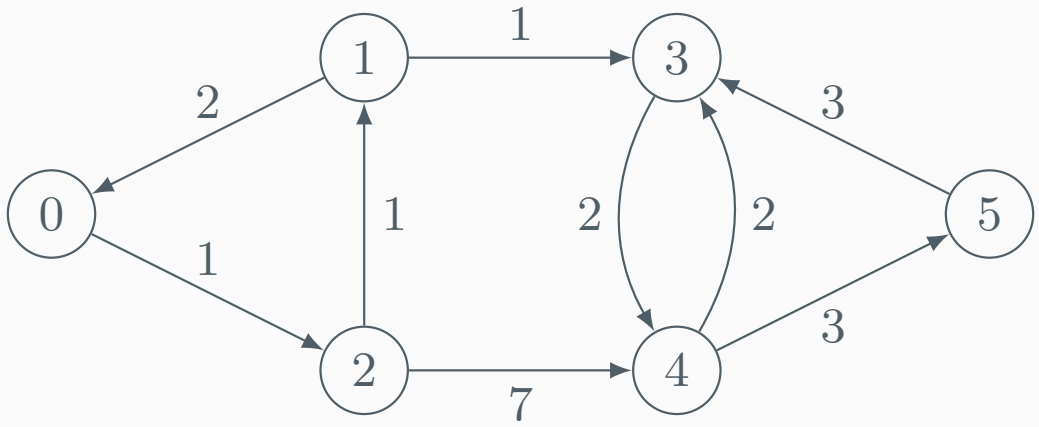


puis

$$A^{(2)} = \begin{pmatrix} 4 & 2 & 1 & 3 & 8 & \infty \\ 2 & 4 & 3 & 1 & 10 & \infty \\ 3 & 1 & 4 & 2 & 7 & \infty \\ \infty & \infty & \infty & \infty & 2 & \infty \\ \infty & \infty & \infty & 2 & \infty & 3 \\ \infty & \infty & \infty & 3 & \infty & \infty \end{pmatrix}.$$

Exemple

On considère le graphe pondéré suivant



Finalement

$$A^{(5)} = \begin{pmatrix} 4 & 2 & 1 & 3 & 5 & 8 \\ 2 & 4 & 3 & 1 & 3 & 6 \\ 3 & 1 & 4 & 2 & 4 & 7 \\ \infty & \infty & \infty & 4 & 2 & 5 \\ \infty & \infty & \infty & 2 & 4 & 3 \\ \infty & \infty & \infty & 3 & 5 & 8 \end{pmatrix}.$$

Implémentation

On rappelle que l'on utilise le type et les fonctions :

```
type int_inf = Inf | Entier of int;;

let add x y = match x,y with
  |Inf,_ -> Inf
  |_, Inf -> Inf
  |Entier(a),Entier(b) -> Entier(a+b);;

let mini x y = match x,y with
  |Inf,_-> y
  |_,Inf -> x
  |Entier(a),Entier(b) -> Entier(min a b);;
```

Implémentation

On peut implémenter ce algorithme.

```
let floyd-warshall g =
   let n = Array.length g in
   let a = Array.make_matrix n n Entier(0) in
   for i = 0 to (n-1) do
         a.(i) <- Array.copy g.(i)
   done;
   for k = 0 to (n-1) do
      for i = 0 to (n-1) do
         for j = 0 to (n-1) do
            a.(i).(j) <- mini a.(i).(j) (add a.(i).(k)  a.(k).(j))
         done;
      done;
   done;
   a;;
```

Implémentation

On peut implémenter ce algorithme.

```
let floyd-warshall g =
   let n = Array.length g in
   let a = Array.make_matrix n n Entier(0) in
   for i = 0 to (n-1) do
         a.(i) <- Array.copy g.(i)
   done;
   for k = 0 to (n-1) do
      for i = 0 to (n-1) do
         for j = 0 to (n-1) do
            a.(i).(j) <- mini a.(i).(j) (add a.(i).(k)  a.(k).(j))
         done;
      done;
   done;
   a;;
```

Dans le programme précédent il n'est pas nécessaire de « sauvegarder » `a.(i).(j)`, lors du calcul, le fait qu'il n'y ait pas de cycle de poids négatif nous permet de remplacer $a_{i,k+1}^{(k)}$ et $a_{k+1,j}^{(k)}$ par $a_{i,k+1}^{(k+1)}$ et $a_{k+1,j}^{(k+1)}$.

Complexité de l'algorithme de Floyd-Warshall

Complexité de l'algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall est quadratique en espace ($O(|S|^2)$) et cubique en temps ($O(|S|^3)$).

Exercice

En utilisant l'algorithme de Floyd-Warshall écrivez une fonction `chemin`telle que `chemin g i j`renvoie le poids d'un chemin de poids minimal allant de i à j ainsi que le chemin sous-forme d'une liste.

Exercice

En utilisant l'algorithme de Floyd-Warshall écrivez une fonction `chemin`telle que `chemin g i j`renvoie le poids d'un chemin de poids minimal allant de i à j ainsi que le chemin sous-forme d'une liste.

- On pourra modifier l'algorithme de Floyd-Warshall afin de garder en mémoire le sommet précédent du chemin
- On reconstruira à la fin de chemin

Exercice

```
let fwChemin g =
  let n = Array.length g in
  let a = Array.make_matrix n n (-1,Inf) in
  for i = 0 to n-1 do
    for j = 0 to (n-1) do
      match g.(i).(j) with
      |Inf -> a.(i).(j) <- (-1, Inf)
      |Entier(x) -> a.(i).(j) <- (i,g.(i).(j))
    done;
  done;
```

Exercice

```
let fwChemin g =
  let n = Array.length g in
  let a = Array.make_matrix n n (-1,Inf) in
  for i = 0 to n-1 do
    for j = 0 to (n-1) do
      match g.(i).(j) with
      |Inf -> a.(i).(j) <- (-1, Inf)
      |Entier(x) -> a.(i).(j) <- (i,g.(i).(j))
    done;
  done;


  for k = 0 to n-1 do
    for i = 0 to (n-1) do
      for j = 0 to (n-1) do
         let (pij,dij) = a.(i).(j) in
         let (pik,dik) = a.(i).(k) in
         let (pkj,dkj) = a.(k).(j) in
         let minche =  mini dij (add dik dkj) in
         if minche <> dij then a.(i).(j) <- (pkj, minche)
      done;
    done;
  done;
  a;;
```

Exercice

On peut alors reconstruire le chemin

```
let cheminFW g i j =
let a = fwChemin g in
let rec reconstruit i k acc =
  if k = i then acc else
    let (pij,Entier(dij)) = a.(i).(k) in
    reconstruit i pij (k::acc)
in (i::(reconstruit i j []));;
```

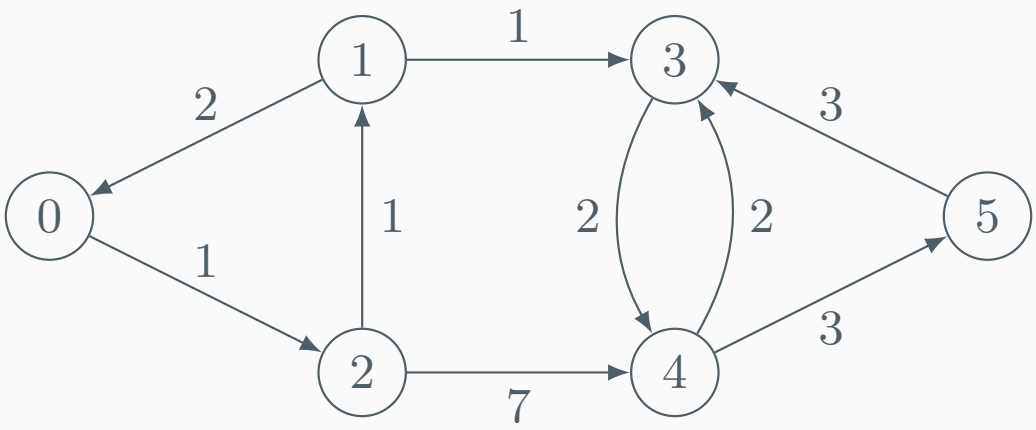
Algorithme de Dijkstra

Nous allons étudier un autre algorithme pour déterminer le chemin de poids minimal dans un graphe. Nous étudierons à la fin les avantages et les inconvénients de chacun.

On suppose maintenant que tous les poids sont positifs.

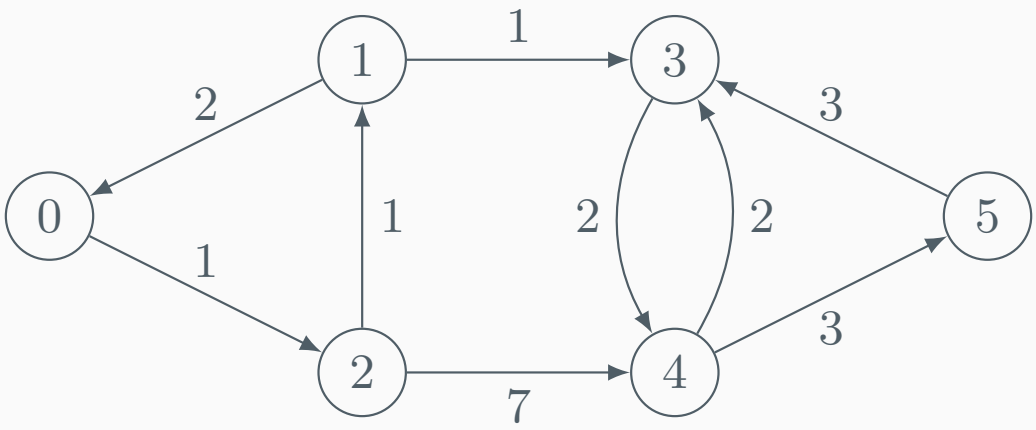
On considère cette fois le graphe donné par des listes d'adjacences. Précisément on suppose que les sommets du graphe sont les entiers compris entre 0 et $n-1$ et que l'on dispose d'un tableau de listes : `adjacenteoù adjacente.(i)`est la liste des couples (t, pt) où t est un voisin de `i`et pt le poids de l'arête de i à t.

Exemple



On obtient

Exemple



On obtient

adjacence	0	1	2	3	4	5
	$[(2,1)]$	$[(0,2);(3,1)]$	$[(1,1);(4,7)]$	$[(4,2)]$	$[(3,2);(5,3)]$	$[(3,3)]$

Algorithme de Dijkstra

On se donne un sommet de départ (que nous noterons `i0`)et le but est de construire un tableau `poidsChemin telque poidsChemin.(j) soit`le poids minimal d'un chemin de `i0 àj.`

L'algorithme est le suivant :

Algorithme de Dijkstra

Algorithme 1 : Algorithme de Dijkstra

Entrée Graphe $G = (S, A)$; sommet de départ `i0`

:

1 $F \leftarrow S$
2 `poidsChemin.(j)` $\leftarrow \infty$ pout tout sommet j
3 `poidsChemin.(i0)` $\leftarrow 0$
4 **tant que** $F \neq \emptyset$ **faire**
5 Choisir un sommet k dans F tel que `PoidsChemin.(k)` est minimal
6 Enlever k de F
7 **pour** $l \in V_k$ **faire**
8 `PoidsChemin.(l)` $\leftarrow \min($`PoidsChemin.(l)` , `PoidsChemin.(k)` $+$ `poids(k,l)` $)$
9 **fin**
10 **fin**

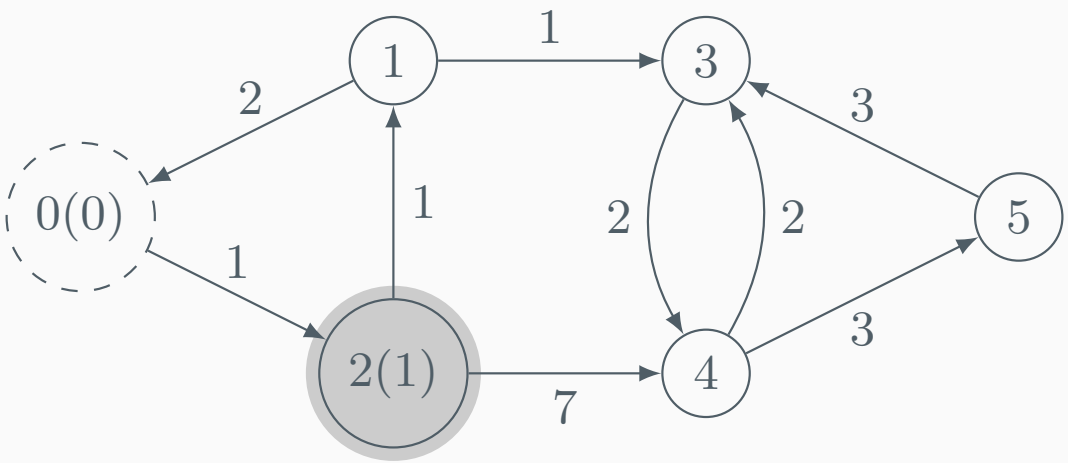
Algorithme de Dijkstra

Théorème 1

A la fin de l'exécution de l'algorithme ci-dessus, `poidsChemin.(j)` contient le poids minimal d'un chemin de `i0` à `j`.

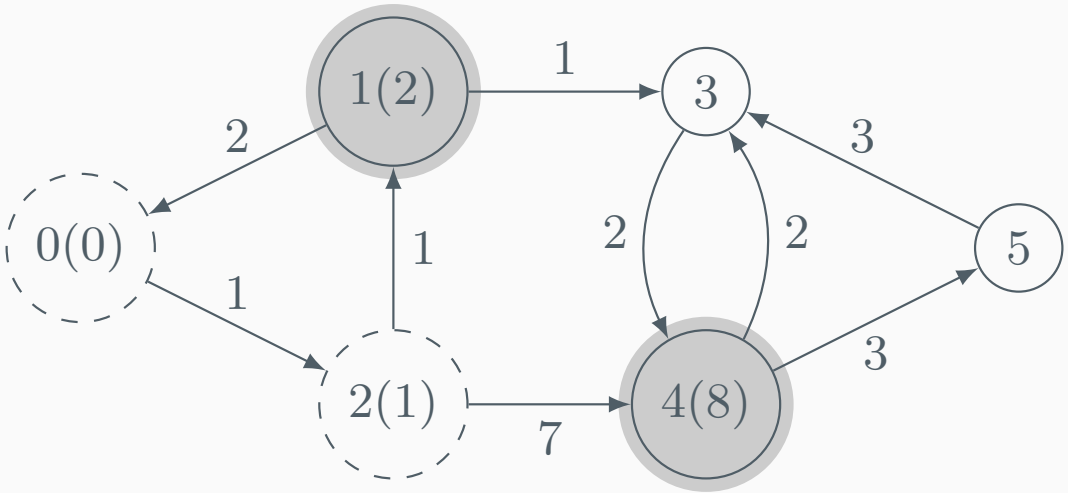
Exemple

Appliquons l'algorithme de Dijkstra sur notre exemple. Les sommets visités deviendrons en pointillés alors que les sommets de la frontière vont être grisés. On indique entre parenthèses le poids du chemin.



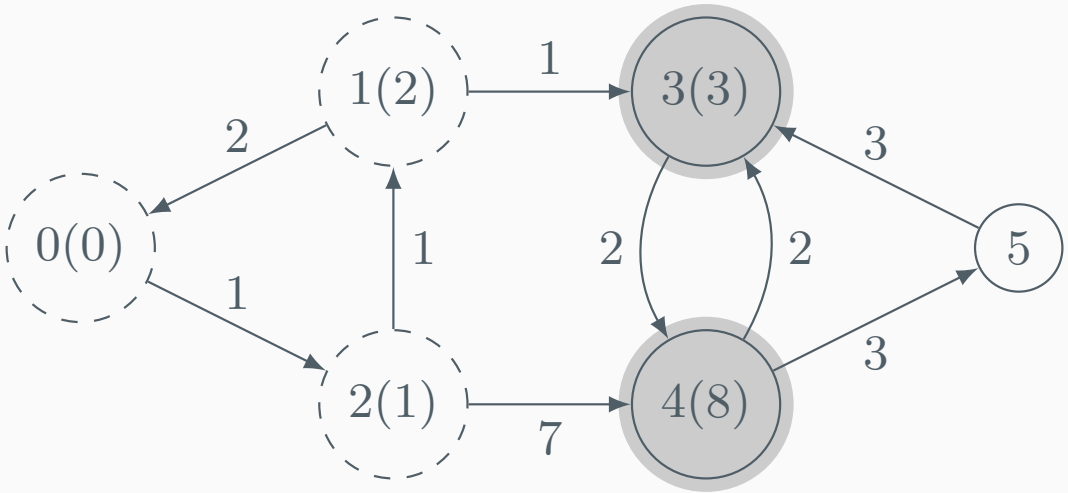
Exemple

Appliquons l'algorithme de Dijkstra sur notre exemple. Les sommets visités deviendrons en pointillés alors que les sommets de la frontière vont être grisés. On indique entre parenthèses le poids du chemin.



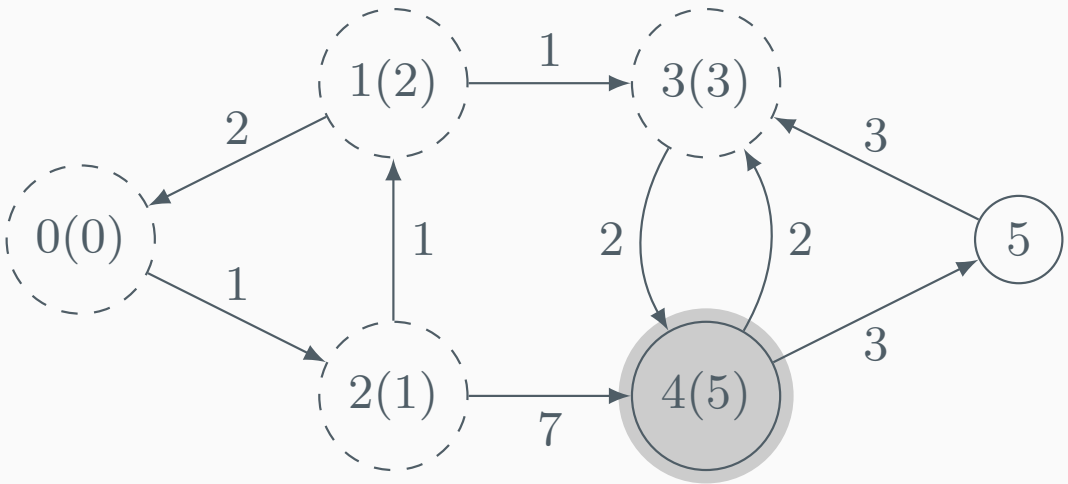
Exemple

Appliquons l'algorithme de Dijkstra sur notre exemple. Les sommets visités deviendrons en pointillés alors que les sommets de la frontière vont être grisés. On indique entre parenthèses le poids du chemin.



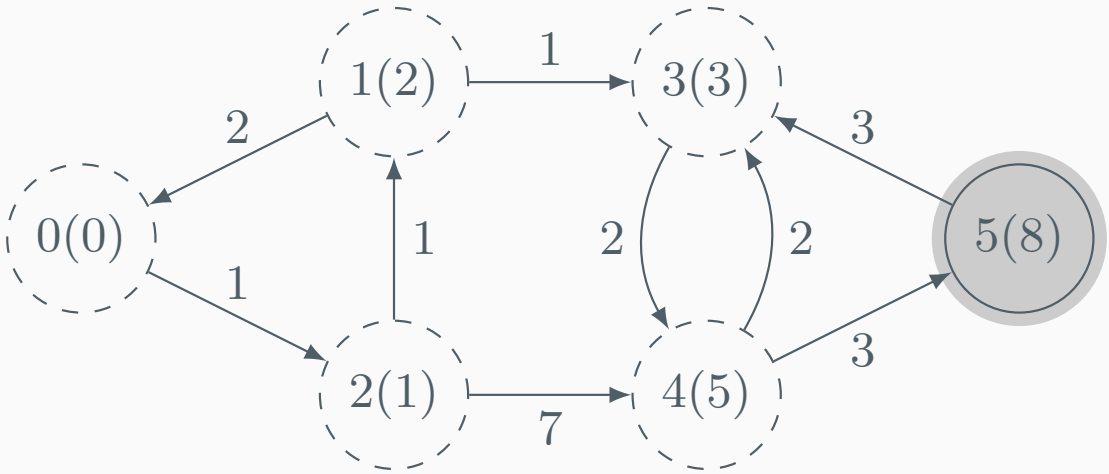
Exemple

Appliquons l'algorithme de Dijkstra sur notre exemple. Les sommets visités deviendrons en pointillés alors que les sommets de la frontière vont être grisés. On indique entre parenthèses le poids du chemin.



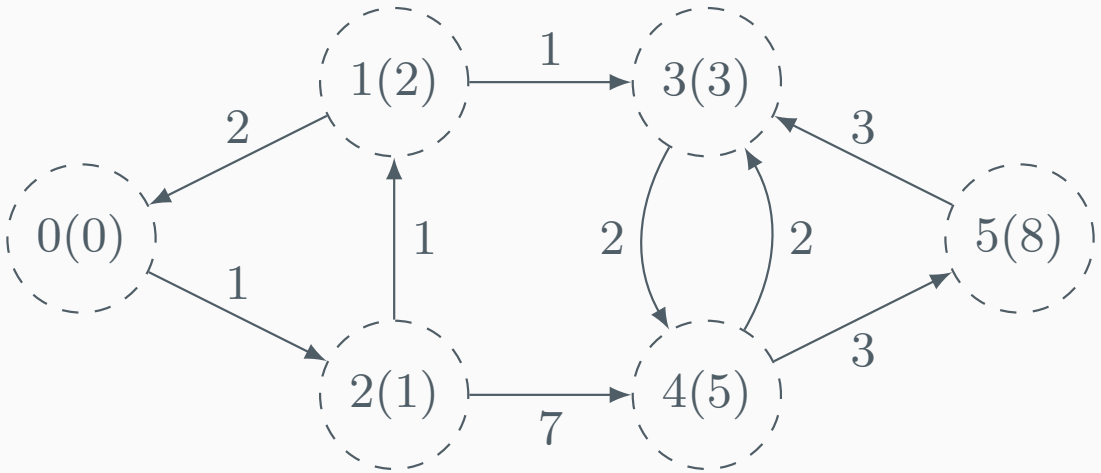
Exemple

Appliquons l'algorithme de Dijkstra sur notre exemple. Les sommets visités deviendrons en pointillés alors que les sommets de la frontière vont être grisés. On indique entre parenthèses le poids du chemin.



Exemple

Appliquons l'algorithme de Dijkstra sur notre exemple. Les sommets visités deviendrons en pointillés alors que les sommets de la frontière vont être grisés. On indique entre parenthèses le poids du chemin.



Démonstration

On notera $E = V \setminus F$ l'ensemble des sommets qui ne sont plus dans F. C'est-à-dire les sommets qui ont été traités

On remarque , qu'à tout moment, `PoidsChemin.(j)` $\geq \delta(i0, j)$ où $\delta(i0, j)$ est le poids du chemin de plus faible poids entre $i0$ et j.

Démonstration

On utilise l'invariant de boucle suivant :

Au début de chaque itération de la boucle Tant que,

pour tout j qui est dans E, `poidsChemin.(j)` est le poids minimal d'un chemin de `i0` à `j`

Démonstration

- Initialisation : Au début, $E = \emptyset$ donc l'invariant est vrai.

Démonstration

- Initialisation : Au début, $E = \emptyset$ donc l'invariant est vrai.
- Supposons par l'absurde qu'il existe un élément k tel que l'invariant de boucle était vrai avant d'ajouter k et que l'invariant ne soit plus vérifié après l'ajout de l'élément k. On peut de plus supposer que k est le « premier » sommet vérifiant cela. On note E l'ensemble des sommets traités avant d'ajouter k et $E' = E \cup \{k\}$.

Démonstration

- Pour tous les sommets j qui ont été ajoutés à E avant k, on a `PoidsChemin.(j)` $= \delta(i0, j)$.

Démonstration

- Pour tous les sommets j qui ont été ajoutés à E avant k, on a `PoidsChemin.(j)` $= \delta(i0, j)$.
- À la fin de la boucle, `PoidsChemin.(k)` $\neq \delta(i0, k)$ et donc $k \neq i0$.
  On considère un chemin p de plus faible poids de $i0$ à k. Ce chemin allant de $i0$ (qui est dans E) à k (qui n'est pas dans E), on peut considérer y le premier sommet du chemin qui n'est pas dans E ainsi que x le sommet qui précède dans le chemin. On a donc

$$p : i0 \xrightarrow{p_1} x \to y \xrightarrow{p_2} k.$$

  où p_1 est un chemin intégralement dans E (éventuellement vide) et p_2 est un chemin qui peut passer (ou pas) dans E (lui aussi éventuellement vide).

Démonstration

Maintenant, **au moment où x a été ajouté à E**, on avait `PoidsChemin.(x)` $= \delta(i0, x)$ car cette égalité est vraie après l'ajout de x et que l'on ne modifie pas `PoidsChemin.(x)` à ce moment.

On en déduit qu'après cet ajout

$$\begin{aligned} \texttt{PoidsChemin.(y)} &\leq \texttt{PoidsChemin.(x)} + \texttt{poids(x,y)} \\ &\leq \delta(i0, x) + \texttt{poids(x,y)} \\ &\leq \delta(i0, y) \end{aligned}$$

La dernière inégalité vient du fait que p est un chemin de poids minimal donc tout sous-chemin est encore de poids minimal.

Démonstration

On a donc

$$\texttt{PoidsChemin.(y)} \leq \delta(i0, y) \leq \delta(i0, k) \leq \texttt{PoidsChemin.(k)}.$$

Comme, par définition du choix de k, $\texttt{PoidsChemin.(k)} \leq \texttt{PoidsChemin.(y)}$, on obtient que $\texttt{PoidsChemin.(k)} = \delta(i0, k)$ ce qui contredit notre hypothèse de départ.

Démonstration

On a donc

$$\texttt{PoidsChemin.(y)} \leq \delta(i0, y) \leq \delta(i0, k) \leq \texttt{PoidsChemin.(k)}.$$

Comme, par définition du choix de k, $\texttt{PoidsChemin.(k)} \leq \texttt{PoidsChemin.(y)}$, on obtient que $\texttt{PoidsChemin.(k)} = \delta(i0, k)$ ce qui contredit notre hypothèse de départ.

Démonstration

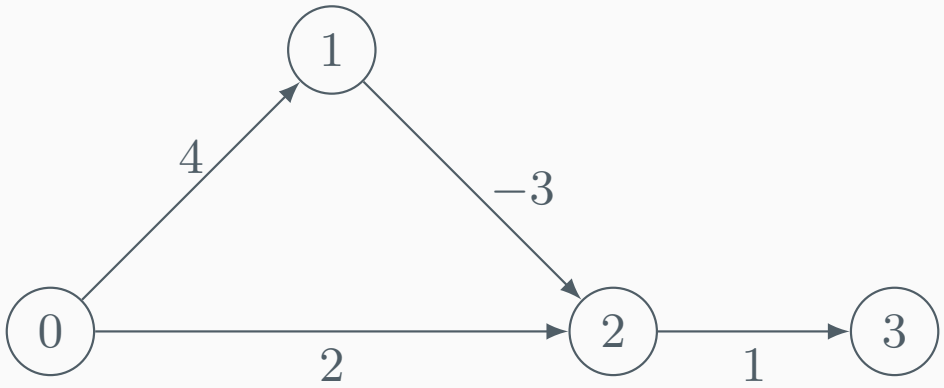
L'invariant de boucle reste donc vrai. A la fin de la boucle, tous les sommets sont dans E et `poidsChemin.(j)`est le poids minimal d'un chemin de `i0`à `j`.

Contre-exemple

S'il y a des arêtes avec un poids négatif, l'algorithme de Dijkstra ne donnera pas nécessairement le chemin de plus faible poids.

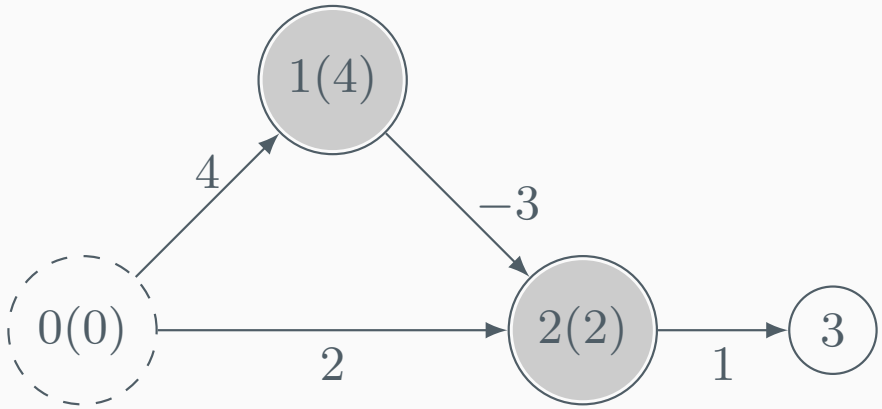
Contre-exemple

S'il y a des arêtes avec un poids négatif, l'algorithme de Dijkstra ne donnera pas nécessairement le chemin de plus faible poids.



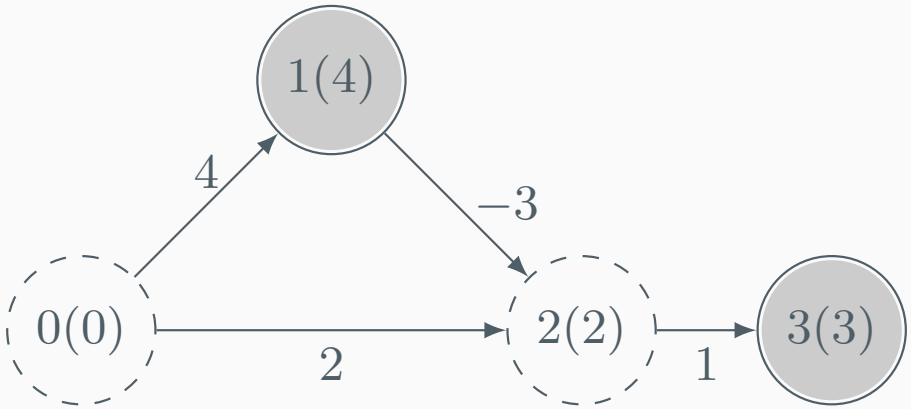
Contre-exemple

S'il y a des arêtes avec un poids négatif, l'algorithme de Dijkstra ne donnera pas nécessairement le chemin de plus faible poids.



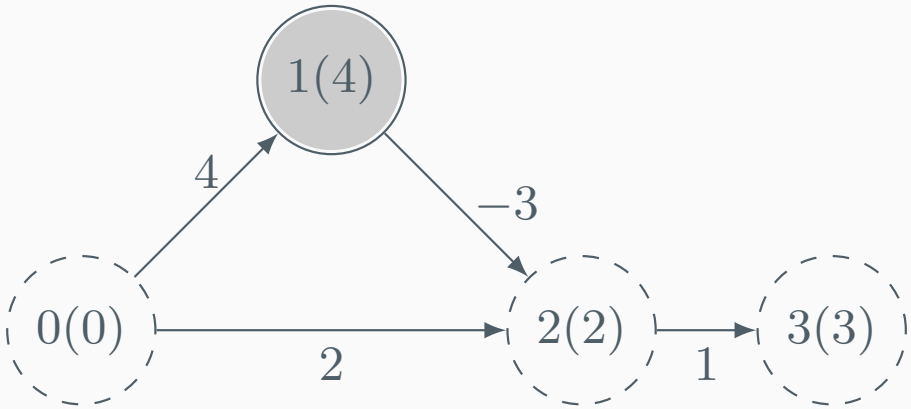
Contre-exemple

S'il y a des arêtes avec un poids négatif, l'algorithme de Dijkstra ne donnera pas nécessairement le chemin de plus faible poids.



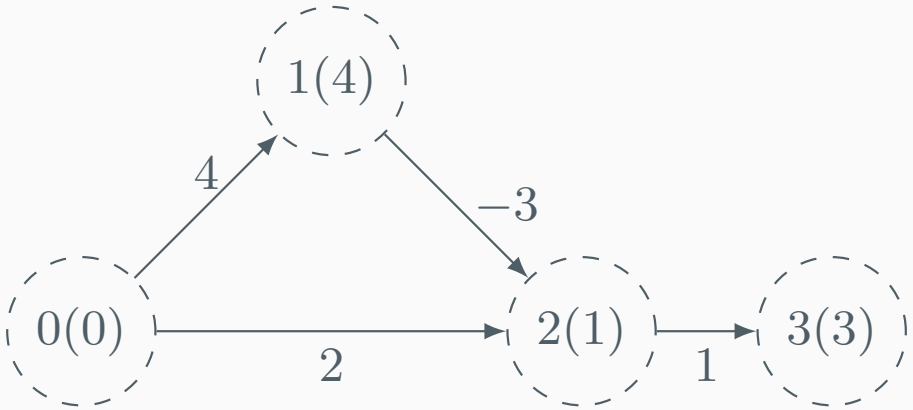
Contre-exemple

S'il y a des arêtes avec un poids négatif, l'algorithme de Dijkstra ne donnera pas nécessairement le chemin de plus faible poids.



Contre-exemple

S'il y a des arêtes avec un poids négatif, l'algorithme de Dijkstra ne donnera pas nécessairement le chemin de plus faible poids.



Comparaison

- Contrairement au cas de Floyd-Warshall, dans l'algorithme de Dijkstra on a besoin d'une origine. De plus il ne calcule que les chemins en partant de cette origine.
- L'algorithme de Dijkstra a une complexité meilleure

Il est clair que l'algorithme passe $|S|$ fois dans la boucle **tant que**, si on stocke les poids des chemins dans un tableau et que l'on fait une recherche linéaire, on a donc une complexité en temps totale $O(|S|^2 + |A|)$ puisque l'on considère chaque arête une fois dans la ligne 8 de l'algorithme.

Cependant on peut faire mieux en considérant une autre structure de données.

Cependant on peut faire mieux en considérant une autre structure de données.

Théorème 2

La complexité temporelle est de :

Cependant on peut faire mieux en considérant une autre structure de données.

Théorème 2

En utilisant une structure de file de priorité pour gérer F

La complexité temporelle est de :

Cependant on peut faire mieux en considérant une autre structure de données.

Théorème 2

En utilisant une structure de file de priorité pour gérer F

La complexité temporelle est de : $O(|A| \log(|S|))$

Ici on considère que le graphe est connexe et donc $|S| = O(|A|)$.

On voit que la taille de la file de priorité est majorée par $|A|$ car on ajoute (ou modifie) les sommets à l'étape 8. Donc, dans notre implémentation de la file de priorité l'insertion et la suppression est en $O(\log |A|) = O(\log |S|)$.

On voit que la taille de la file de priorité est majorée par $|A|$ car on ajoute (ou modifie) les sommets à l'étape 8. Donc, dans notre implémentation de la file de priorité l'insertion et la suppression est en $O(\log |A|) = O(\log |S|)$.

De plus, on passe tous les éléments de la file dans les lignes 5-6, ce qui nécessite $O(|A| \log |S|)$ opérations.

On passe aussi $|A|$ fois dans lignes 7-8-9 car on considère à chaque fois une arête. Le fait de modifier un poids nécessite de déplacer l'élément dans la file de priorité, ce qui se fait en un nombre d'opérations en $O(|\log S|)$. De ce fait, ces trois lignes engendrent $O(|A| \log |S|)$ opérations.

On passe aussi $|A|$ fois dans lignes 7-8-9 car on considère à chaque fois une arête. Le fait de modifier un poids nécessite de déplacer l'élément dans la file de priorité, ce qui se fait en un nombre d'opérations en $O(|\log S|)$. De ce fait, ces trois lignes engendrent $O(|A| \log |S|)$ opérations.

La complexité finale est $O(|A| \log(|S|))$.

Implémentation

On va utiliser une file de priorité f (implémentée par un tas min) pour notre ensemble F. Elle contient des triplets `(x,p,a)` où `x` est un sommet, `p` le poids du chemin de `i` à `x` et `a` l'antécédent de `x` dans ce chemin.

Implémentation

On va utiliser une file de priorité f (implémentée par un tas min) pour notre ensemble F. Elle contient des triplets `(x,p,a)` où `x` est un sommet, `p` le poids du chemin de `i` à `x` et `a` l'antécédent de `x` dans ce chemin.

Afin de reconstruire aussi le chemin, on va, parallèlement à la construction du tableau `poidsChemin` construire un tableau `antecedent` tel que `antecedent(j)` soit le sommet précédent `j` dans le chemin (de plus faible poids) allant de `i` à `j`.

Implémentation

On rappelle les constructeurs des files de priorité

- `creeFileVide : unit -> filePriorite`
  qui créé une file vide.
- `insertFile : (int * int * int) -> filePriorite -> unit`
  qui ajoute un élément (par effets de bord) à la file.
- `defile : filePriorite -> (int*int *int)`
  qui renvoie le sommet de la file (dont le poids est le plus faible) et le supprime de la file (par effets de bord)

Implémentation

```
let dijkstra g  i j =
let n = Array.length g in
let poidsChemin = Array.make n (-1) in
let antecedent = Array.make n (-1) in
poidsChemin.(i) <- 0;
let visit = Array.make n false in
let rec parcours f =
  let (x,p,a) = defile f in
  if not(visit.(x)) then
    (visit.(x) <- true;
     poidsChemin.(x) <- p;
     antecedent.(x) <- a;
     let rec traiteliste x p  l =
         match l with
             |[] -> ()
             |(v,pc):: q -> if not (visit.(v))
              then insertFile (v,p+pc,x) f ; traiteliste x p q
     in
     traiteliste x p g.(x));
  if x <> j then parcours f in
let f = creeFileVide () in
insertFile  (i,0,-1) f;
parcours f;
(poidsChemin,antecedent);;
```

Implémentation

- On ne modifie `poidsChemin.(x)` que quand le sommet `x` sort de F. On n'utilise pas explicitement la fonction `min`. Le minimum est géré par la file de priorité.
- On voit que on commence avec une file qui ne contient que l'origine et on ajoute les sommets dedans. De plus on ajoute sans s'occuper de savoir si cela optimise la distance, de fait si cela ne l'optimise pas cela le placera dans la queue de la file.
- Le tableau `visitsert` à ne pas visiter plusieurs fois un même sommet.

Exercice

Écrire la fonction pour reconstruire le chemin

Exercice

Écrire la fonction pour reconstruire le chemin

```
 let chemin g i j =
 let (poidsChemin,antecedent) = dijkstra g i j in
 let rec aux acc k =
      if k = -1 then acc
      else aux k::acc antecedent.(k)
in aux [] j;;
```