

Paradigmes algorithmiques

Programmation dynamique

Algorithme glouton : construit une solution de manière incrémentale, en optimisant un critère de manière locale.

Diviser pour régner : divise un problème en sous-problèmes **indépendants**, résout chaque sous-problème, et combine les solutions des sous-problèmes pour former une solution du problème initial.

Programmation dynamique : divise un problème en sous-problèmes qui sont non indépendants, et cherche (et stocke) des solutions de sous-problèmes de plus en plus grands.

Programmation dynamique



Historique : paradigme développé par Richard Bellman en 1953.

Technique de conception d'algorithmes très générale et performante.

Permet de résoudre de nombreux **problèmes d'optimisation**.

Exemples d'algorithmes de programmation dynamique : alignement de séquences ADN, algorithme de plus courts chemins (Bellman-Ford), commande Unix diff, etc.

Programmation dynamique

Idée : "recherche exhaustive intelligente" (sous-problèmes + réutilisation de solutions déjà calculées).

Exemple : calcul de F_n : $n^{\text{ème}}$ nombre de Fibonacci

$$\begin{cases} F_1 = F_2 = 1 \\ F_n = F_{n-1} + F_{n-2} \end{cases}$$

Algorithme naïf :

```
Fib(n) :  
  si  $n \leq 2$  alors  $F = 1$   
  sinon  $F = \text{Fib}(n-1) + \text{Fib}(n-2)$   
  retourner  $F$ 
```

Complexité exponentielle : $T(n) = T(n-1) + T(n-2) + \Theta(1) \approx \varphi^n$

Mémoïsation

Exemple :

```
mémo = {}  
Fib(n) :  
  si n est dans mémo, retourner mémo[n]  
  si n ≤ 2 alors F = 1  
  sinon F = Fib(n-1) + Fib(n-2)  
  mémo[n] = F  
  retourner F
```

Fib(k) induit des appels récursifs seulement la première fois qu'elle est appelée.

Ceci peut être fait pour tout algorithme récursif.

Mémoïser = *conserver à la fin de l'exécution d'une fonction le résultat associé aux arguments d'appels, pour ne pas avoir à recalculer ce résultat lors d'un autre appel récursif.*

Programmation dynamique

Idée : **mémoïser** et **réutiliser les solutions de sous-problèmes** qui aident à résoudre le problème.

Complexité temporelle =
nombre de sous-problèmes x (complexité par sous-problème*)

* On ne compte pas les appels récursifs.

Mémoïsation : complexité

Exemple :

```
mémo = {}  
Fib(n) :  
  si n est dans mémo, retourner mémo[n]  
  si n ≤ 2 alors F = 1  
  sinon F = Fib(n-1) + Fib(n-2)  
  mémo[n] = F  
  retourner F
```

Nombre d'appels non "mémoïsés" : n

Coût d'un appel (sans compter les appels récursifs "non mémoïsés") : $\Theta(1)$
(mémo[i] est retourné en $\Theta(1)$ si mémo est un tableau)

Complexité temporelle : $\Theta(n)$

Programmation dynamique (2)

Deuxième manière de voir la programmation dynamique :
approche "du bas vers le haut"

Exemple :

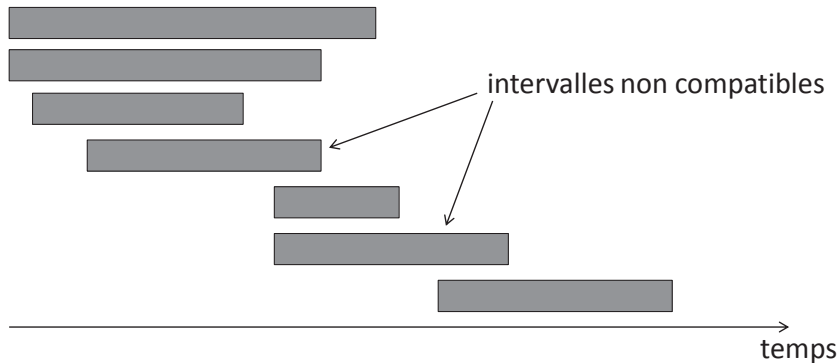
```
Fib = {}  
pour k de 1 à n :  
  si k ≤ 2 alors F = 1  
  sinon F = Fib[k-1] + Fib[k-2]  
  Fib[k] = F  
retourner Fib[n]
```

Cas général :

- Exactement les **mêmes calculs** que dans la **version mémoïsée**. Les **sous-problèmes sont traités dans un ordre topologique**.
- Complexité temporelle : évidente
- Permet souvent de baisser la complexité spatiale.

Exemple : ordonnancement d'intervalles pondérés

- L'intervalle i commence en d_i , se termine en f_i et a une valeur v_i
- Deux intervalles sont compatibles s'ils ne s'intersectent pas.
- But : déterminer un sous-ensemble d'intervalles mutuellement compatibles de valeur maximum.

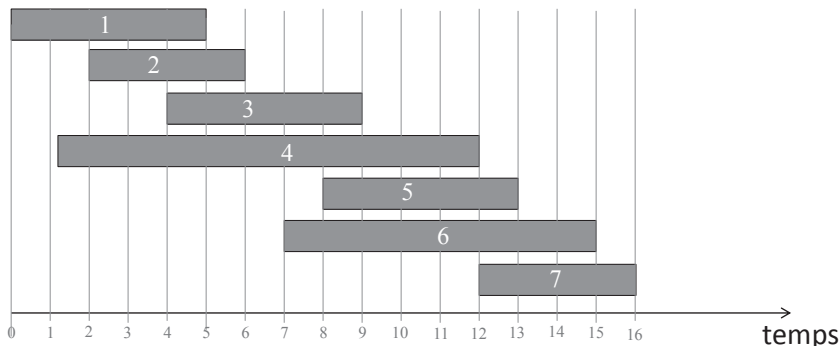


Ordonnancement d'intervalles pondérés

Notation : on numérote les intervalles par dates de fin croissantes : $f_1 \leq f_2 \leq \dots \leq f_n$

Définition : $der(j)$ = plus grand numéro $i < j$ tel que l'intervalle i est compatible avec l'intervalle j .

Exemple : $der(7)=4$; $der(6)=2$; $der(2)=0$.



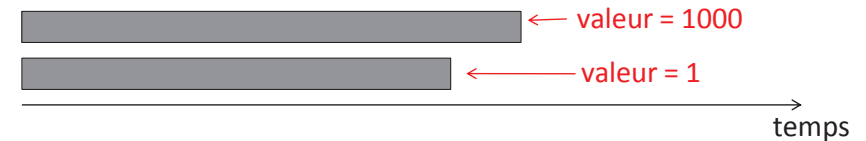
Ordonnancement d'intervalles pondérés

Rappel : Algorithme "celui qui termine le plus tôt d'abord"

- Considérer les intervalles dans l'ordre de leurs dates de fins croissantes.
- Ajouter un intervalle au sous-ensemble des intervalles choisis s'il est compatible avec ces intervalles.

Cet algorithme est optimal si tous les poids sont égaux à 1.

Il peut être mauvais dans la version pondérée :



Ordonnancement d'intervalles pondérés : choix binaire

Notation : $OPT(i)$ = valeur d'une solution optimale en se restreignant aux intervalles 1, ..., i .

Cas 1 : i est choisi dans OPT

- On obtient la valeur de i : v_i
- On ne peut pas choisir les intervalles $\{der(i)+1, der(i)+2, \dots, i-1\}$
- On doit choisir la solution optimale du problème restreint aux intervalles 1, 2, ..., $der(i)$

Cas 2 : i n'est pas dans OPT

- On doit choisir la solution optimale du problème restreint aux intervalles 1, 2, ..., $i-1$

$$OPT(i) = \begin{cases} 0 & \text{si } i=0 \\ \max \{ v_i + OPT(der(i)) , OPT(i-1) \} & \text{sinon} \end{cases}$$

Ordonnancement d'intervalles pondérés : algorithme naïf

Entrée : $n, d[1..n], f[1..n], v[1..n]$

Trier les intervalles de façon à ce que $f[1] \leq f[2] \leq \dots \leq f[n]$.

Calculer $der[1], der[2], \dots, der[n]$

Calculer_OPT(i)

si $i=0$ alors $s = 0$

sinon $s = \max \{v[i] + \text{Calculer_OPT}(der[i]), \text{Calculer_OPT}(i-1)\}$

retourner s

Complexité : exponentielle

Ordonnancement d'intervalles pondérés : mémoïsation

Entrée : $n, d[1..n], f[1..n], v[1..n]$

Trier les intervalles de façon à ce que $f[1] \leq f[2] \leq \dots \leq f[n]$.

Calculer $der[1], der[2], \dots, der[n]$

pour j allant de 1 à n

$mémo[j] = \text{vide}$

$mémo[0] = 0$

Calculer_OPT(i)

si $mémo[i]$ est vide alors

$mémo[i] = \max \{v[i] + \text{Calculer_OPT}(der[i]), \text{Calculer_OPT}(i-1)\}$

retourner $mémo[i]$

Complexité : Initialisation : $\Theta(n \log n)$

Calculer_OPT(n) : $\Theta(n)$

Ordonnancement d'intervalles pondérés : approche "du bas vers le haut"

Entrée : $n, d[1..n], f[1..n], v[1..n]$

Trier les intervalles de façon à ce que $f[1] \leq f[2] \leq \dots \leq f[n]$.

Calculer $der[1], der[2], \dots, der[n]$

$mémo[0] = 0$

pour i allant de 1 à n

$mémo[i] = \max \{v[i] + mémo[der[i]], mémo[i-1]\}$

Complexité : Initialisation : $\Theta(n \log n)$

boucle : $\Theta(n)$

Ordonnancement d'intervalles pondérés : comment retrouver la solution optimale ?

Trouver_solution(i)

si $i = 0$ alors

 retourner \emptyset

si $v[i] + mémo(der[i]) > mémo[i-1]$ alors

 retourner $\{i\} \cup \text{Trouver_solution}(der[i])$

sinon

 retourner $\text{Trouver_solution}(i-1)$

Complexité : $\Theta(n)$

(nombre d'appels récurifs $\leq n$)

Plus courts chemins

Problème : Entrée : un graphe $G=(S,A)$ orienté et valué (c) ;
sommet origine s ; sommet destination v .
Sortie : un **plus court chemin de s à v** dans G .
(ou **$A(s)$** , arborescence des plus courts chemins d'origine s).

On note $d(x)$ la distance de s à x , pour tout $x \in S$.

Propriété : Il existe un plus court chemin entre s et x si et seulement si il n'existe **pas de circuit absorbant** dans un chemin entre s et x dans G .

Supposons qu'il n'y ait pas de circuit absorbant accessible à partir de s .

Comment calculer $A(s)$?

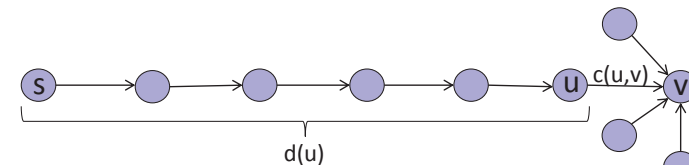
- L'algorithme de Dijkstra ne retourne pas toujours une arborescence des plus courts chemins si les coûts des arcs sont négatifs.
- Augmenter le coût des arcs de façon à avoir seulement des coûts positifs ne marche pas non plus.

Plus courts chemins : programme dynamique

Comment calculer le plus court chemin de s à v ?

On ne connaît pas la réponse ? On **essaie** toutes les solutions (et on prend la meilleure).

Programmation dynamique : récursion + mémoïsation + essais



Quel est le dernier arc du plus court chemin entre s et v ?

On ne sait pas \Rightarrow on les essaie tous.

$$\begin{cases} d(v) = \min_{u \mid (u,v) \in A} \{ d(u) + c(u,v) \} & \text{si } v \neq s \\ d(s) = 0 \end{cases}$$

Plus courts chemins : programme dynamique

$$\text{On a : } \begin{cases} d(s) = 0 \\ d(v) = \min_{u \mid (u,v) \in A} \{ d(u) + c(u,v) \} & \text{si } v \neq s \end{cases}$$

Algorithme récursif :

```

d(s,v) :
  si v=s retourner 0
  d_min = +∞
  pour tout prédécesseur u de v faire
    d_courant = d(s,u) + c(u,v)
    si (d_courant < d_min)
      d_min = d_courant
  retourner d_min
    
```

Plus courts chemins : programme dynamique

Algorithme récursif avec mémoïsation :

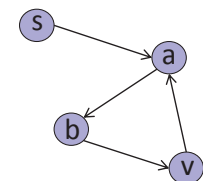
```

pour tout sommet i de S \ {s}
  mémo[i] = vide
  mémo[s] = 0
    
```

```

d(s,v) :
  si mémo[v] est vide
    d_min = +∞
    pour tout prédécesseur u de v faire
      d_courant = d(s,u) + c(u,v)
      si (d_courant < d_min)
        d_min = d_courant
    mémo[v] = d_min
  retourner mémo[v]
    
```

$$\begin{cases} d(s) = 0 \\ d(v) = \min_{u \mid (u,v) \in A} \{ d(u) + c(u,v) \} \end{cases}$$



Présence d'un circuit :
l'algorithme ne termine pas.

Plus courts chemins dans un graphe sans circuit

Cet algorithme est valide s'il n'y a **pas de circuit dans le graphe**.

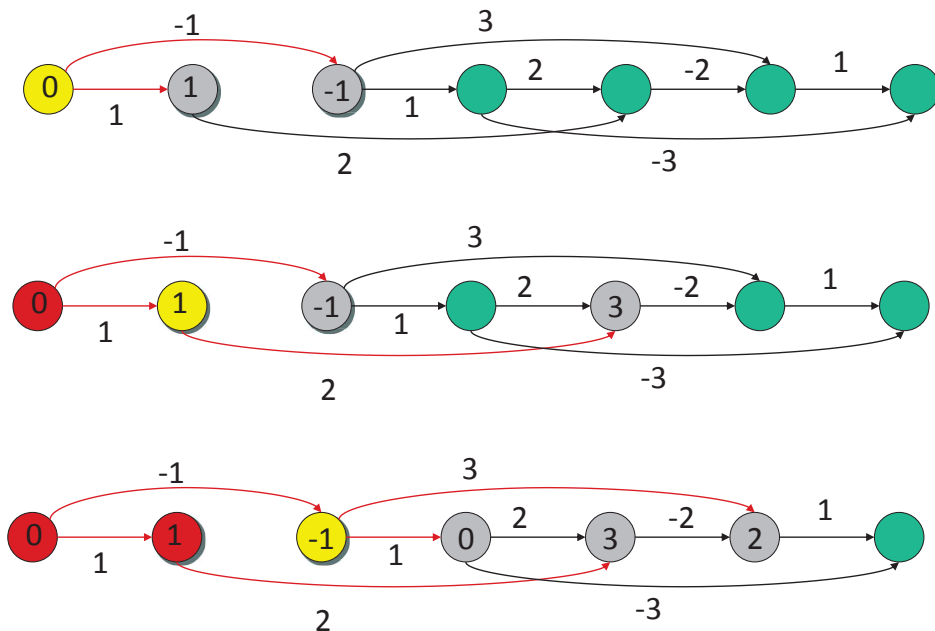
```

d(G,s,v)
  si mémo[v] est vide
    d_min = +∞
    pour tout prédécesseur u de v faire
      d_courant = d(u) + c(u,v)
      si (d_courant < d_min)
        d_min = d_courant
    mémo[v]=d_min
  retourner mémo[v]
    
```

Complexité de l'appel non mémorisé de $d(v)$: $d^+(v) + 1$
 Appels non mémorisés : un par sommet (n sommets)

=> Complexité totale = $O(n+m)$

Exemple :



Plus courts chemins dans un graphe sans circuit

Version ``du bas vers le haut`` de cet algorithme
 (on suppose que s est une racine) :

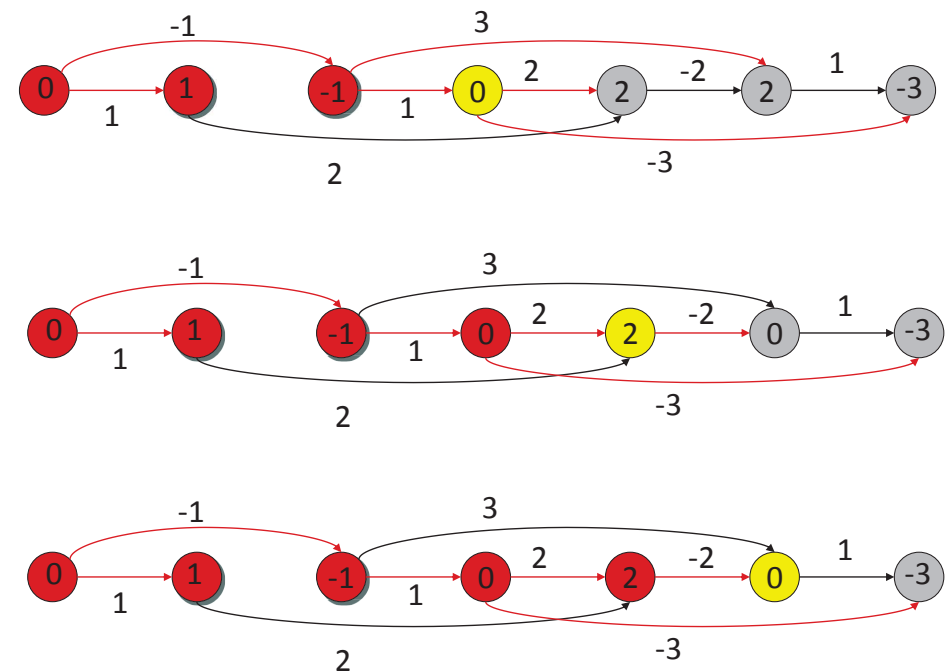
```

soit L=(s_1,...,s_n) une liste topologique des sommets de G telle que s=s_1
pour tout sommet x ≠ s d[x]=+∞ ; d[s]=0
pour k allant de 1 à n
  pour tout successeur y de s_k faire
    si d(y) > d(s_k) + c(s_k,y) alors
      d(y)= d(s_k) + c(s_k,y)
    
```

On obtient $A(s)$, une arborescence des plus courts chemins d'origine s.

C'est **l'algorithme de Bellman**.

Complexité totale = $O(n+m)$



Plus courts chemins dans le cas général

Les **dépendances entre sous-problèmes** doivent être **acycliques**.

Partant d'un graphe avec circuit, il faut trouver un moyen de le rendre acyclique...

Solution : on duplique n fois le graphe (n niveaux). A chaque fois que l'on suit un arc, on va vers un sommet du graphe du niveau suivant.

Exemple : (au tableau)

Cette transformation rend tout graphe acyclique.

Plus courts chemins dans le cas général

Soit $d_k(v)$ le coût d'un **plus court chemin de s à v** qui utilise **au plus k arcs** ($0 \leq k \leq n-1$).

$$\begin{cases} d_k(v) = \min_{u \mid (u,v) \in A} \{ d_{k-1}(u) + c(u,v) \} & \text{si } v \neq s \\ d_k(s) = 0 \end{cases}$$

But : calcul de la valeur $d_{n-1}(v)$ (plus court chemin élémentaire).

Algorithme de Bellman-Ford : algorithme récursif avec mémorisation correspondant à la relation de récurrence ci-dessus.

Complexité de l'appel non mémorisé de $d_k(v)$: $d(v) + 1$

Complexité : **$O(nm)$**

Algorithme de Bellman-Ford

// Initialisation

pour tout sommet x **faire**

si $x=s$ **alors** $d(x) = 0$; $pred=null$

sinon $d(x) = +\infty$; $pred=null$

// Boucle principale

pour $k = 1$ à $n-1$ **faire**

pour chaque arc (x,y) **faire**

si $d(y) > d(x) + c(x,y)$ **alors**

$d(y) = d(x) + c(x,y)$

$pred(y) = x$

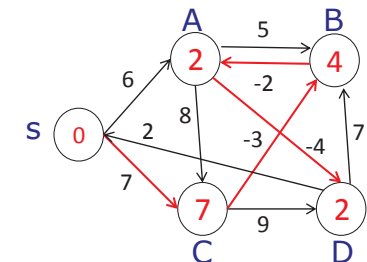
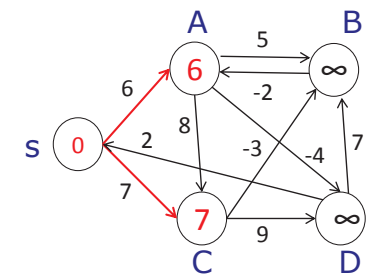
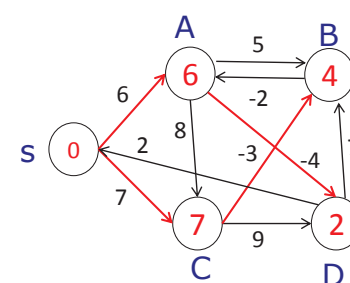
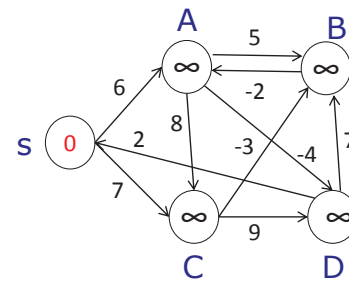
// Détection de circuit absorbant

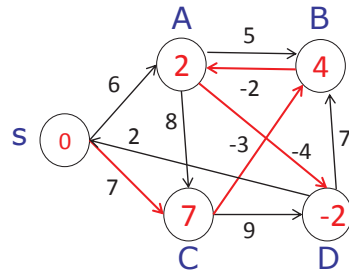
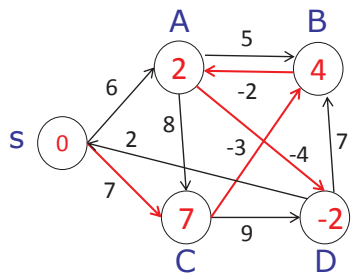
pour chaque arc (x,y) **faire**

si $d(y) > d(x) + c(x,y)$ **alors**

erreur "il n'existe pas de plus court chemin entre s et x"

Exemple





Validité

Propriété 1 : A tout moment après l'étape d'initialisation, si $d(x) \neq \infty$, $d(x)$ est égal à la longueur d'un chemin de s à x .

Preuve (récurrence sur le nombre i de mises à jour de valeurs $d(.)$) :

- Pour $i=0$, la propriété est vérifiée.
- Supposons que cette propriété soit vérifiée jusqu'à la $(i-1)$ -ème mise à jour, et que la i -ème mise à jour consiste à modifier $d(x)$ après l'examen de l'arc (y,x) .
Par hypothèse de récurrence, $d(y)$ est la longueur d'un chemin μ , de s à y . Ainsi $d(x) = d(y) + c(y,x)$ est la longueur du chemin de s à x qui est $\mu.(y,x)$.

Validité

Propriété 2 : (invariant de boucle)

Après i itérations de la boucle "pour" principale : s'il existe un chemin de s à x d'au plus i arcs, alors $d(x)$ est inférieur ou égal à la longueur du plus court chemin ayant au plus i arcs de s à x .

Preuve (récurrence sur i)

- Pour $i=0$, l'invariant est vérifié.
- Supposons qu'il soit vérifié jusqu'au rang $i-1$
 - Soit μ un plus court chemin (pcc) d'au plus i arcs de s à x . Soit y le dernier sommet avant x sur μ : le chemin de s à y est un pcc de s à y d'au plus $(i-1)$ arcs. Par hyp. d'induction, après $(i-1)$ itérations, $d(y)$ est \leq à la longueur de ce chemin.
 - A l'itération i , $d(x)$ est comparé à $d(y) + c(y,x)$ et mise à jour à cette valeur si cela diminue $d(x)$. Donc, à la fin de cette itération, $d(x)$ est \leq à la longueur de μ .

Validité

Propriété 1 : A tout moment après l'étape d'initialisation, si $d(x) \neq \infty$, $d(x)$ est égal à la longueur d'un chemin de s à x .

Propriété 2 :

Après i itérations de la boucle "pour" principale : s'il existe un chemin de s à x d'au plus i arcs, alors $d(x)$ est inférieur ou égal à la longueur d'un plus court chemin ayant au plus i arcs de s à x .

=> à la fin de la boucle principale, pour tout sommet x , s'il existe un chemin de s à x dans G , alors $d(x)$ est égal à la longueur d'un plus court chemin de s à x .

Complexité

```
// Initialisation
// Boucle principale :
pour k = 1 à n-1 faire
    pour chaque arc (x,y) faire
        si  $d(y) > d(x) + c(x,y)$  alors
             $d(y) = d(x) + c(x,y)$ 
             $pred(y) = x$ 
// Détection de circuit absorbant
pour chaque arc (x,y) faire
    si  $d(y) > d(x) + c(x,y)$  alors
        erreur "il n'existe pas de plus court chemin entre s et x"
```

- Complexité : $O(nm)$
- Remarque : si à l'itération i aucune valeur $d(.)$ n'est modifiée, l'algorithme peut s'arrêter.