Programmation dynamique

Paradigmes algorithmiques

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 Bellmann 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.

Pourquoi « programmation dynamique ? »

"An interesting question is, **'Where did the name, dynamic programming, come from?'** The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. (…) The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? **(…). I decided therefore to use the word, 'programming.'** I wanted to get across the idea that this was dynamic, this was multistage, this was **time-varying**—I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is **it's impossible to use the word dynamic, in a pejorative sense**. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. **So I used it as an umbrella for my activities**"

Richard Bellman

Programmation dynamique

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

Exemple : calcul de F_n : $n^{ème}$ 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 ≤ 2 alors F = 1
    sinon F = Fib(n-1) + Fib(n-2)
    retourner F
```

Complexité en nb d'additions : $T(n) = T(n-1) + T(n-2) + \Theta(1) \approx \varphi^n$

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.*

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 (nombre d'additions) : $\Theta(n)$

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.

Programmation dynamique (2)

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

Exemple :

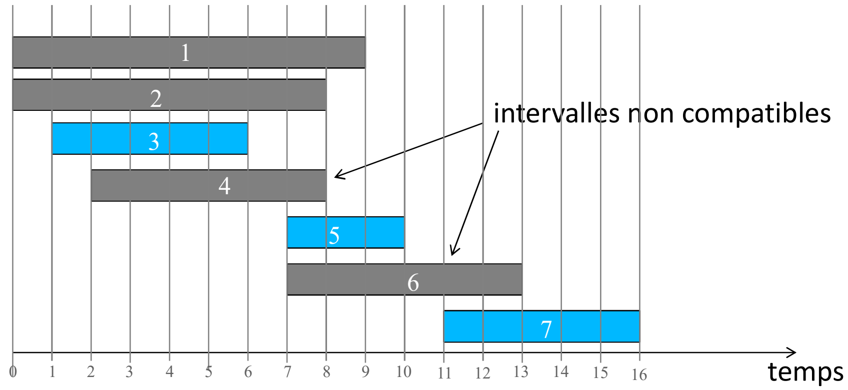
```
Fib = {}
pour k de 1 à n :
    si k ≤ 2 alors F = 1
    sinon F = Fib[k-1] + Fib[k-2]
    Fib[k] = F
retourner Fib[n]
```

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.

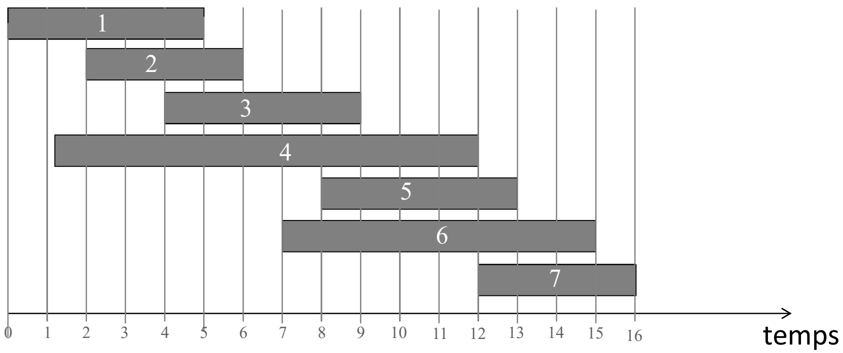


Ordonnancement d’intervalles pondérés

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

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 : 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, …, 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

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

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] ≤ f[2] ≤  … ≤ f[n].
Calculer der[1],der[2],…, der[n]

Calcule_OPT(i)
    si i=0 alors s = 0
    sinon s = max {v[i] + Calcule_OPT(der[i])  , Calcule_OPT(i-1)  }
    retourner s
```

Quelle est la complexité de cet algorithme ?

```
Entrée :  n, d[1..n], f[1..n], v[1..n]
Trier les intervalles de façon à ce que f[1] ≤ f[2] ≤  … ≤ f[n].
Calculer der[1],der[2],…, der[n]
Calcule_OPT(i)
    si i=0 alors s = 0
    sinon s = max {v[i] + Calcule_OPT(der[i])  , Calcule_OPT(i-1)  }
    retourner s
```

A. O(n)
B. O(n log n)
C. O(n^2)
D. O(2^n)

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] ≤ f[2] ≤  … ≤ f[n].
Calculer der[1],der[2],…, der[n]

pour j allant de 1 à n
      mémo[j] = vide
mémo[0] = 0

Calcule_OPT(i)
    si mémo[i] est vide alors
          mémo[i] = max {v[i] + Calcule_OPT(der[i])  , Calcule_OPT(i-1) }
    retourner mémo[i]
```

Quelle est la complexité de cet algorithme ?

```
Entrée :  n, d[1..n], f[1..n], v[1..n]
Trier les intervalles de façon à ce que f[1] ≤ f[2] ≤  … ≤ f[n].
Calculer der[1],der[2],…, der[n],
Initialiser le tableau memo à vide (mémo[0] = 0)
Calcule_OPT(i)
 si mémo[i] est vide alors
         mémo[i] = max {v[i] + Calcule_OPT(der[i])  , Calcule_OPT(i-1) }
retourner mémo[i]
```

A. O(n)
B. O(n log n)
C. O(n^2)
D. O(2^n)

Complexité : Initialisation : Θ(n log n)
Calcule_OPT(n) : Θ(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] ≤ f[2] ≤ … ≤ f[n].
Calculer der[1],der[2],…, 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 : Θ(n log n)
boucle : Θ(n)

Quiz

Soient A et B deux algorithmes de programmation dynamique basés sur la même relation de récurrence. A est un algorithme récursif avec mémoïsation, et B un algorithme itératif.

A et B ont-ils la même complexité temporelle ?

A. Oui
B. Non

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

Rappel :
mémo[i] = max { v[i] + mémo[der[i]] , mémo[i-1] }

```
Trouver_solution(i)
    si i = 0 alors
        retourner ∅
    si v[i] + mémo(der[i]) > mémo[i-1] alors
            retourner {i} ∪ Trouver_solution(der[i])
    sinon
            retourner Trouver_solution(i-1)
```

Complexité : Θ(n) (nombre d'appels récursifs ≤ n)

Quiz

Problème SubsetSum : soit un entier W et n entiers positifs $S = \{a_1, a_2, a_3, \ldots, a_n\}$. Existe-t-il un sous-ensemble de S dont la somme des éléments est W?

Programme dynamique pour SubsetSum : utilise un tableau de booléens X à n lignes et W+1 colonnes t.q. X[i, j] (avec 1 <= i <= n, 0 <= j <= W) est Vrai ssi il existe un sous-ensemble de $\{a_1, a_2, \ldots, a_i\}$ dont la somme des éléments est j.
Quelle égalité est VRAI ($2 \le i \le n$ et $a_i \le j \le W$) ?

A. $X[i, j] = X[i-1, j] \vee X[i, j-a_i]$
B. $X[i, j] = X[i-1, j] \vee X[i-1, j-a_i]$
C. $X[i, j] = X[i-1, j] \wedge X[i, j-a_i]$
D. $X[i, j] = X[i-1, j] \wedge X[i-1, j-a_i]$

Quiz

Un programme dynamique pour résoudre SubsetSum utilise un tableau de booléens X à n lignes et W+1 colonnes t.q. X[i, j] (avec 1 <= i <= n, 0 <= j <= W) est Vrai ssi il existe un sous-ensemble de {a1 ,a2 ,...,ai} dont la somme des éléments est j.

Quelle case du tableau X, si elle est = VRAI indique qu'il existe un sous-ensemble dont la somme est W ?

A. X[1,W]
B. X[n, 0]
C. X[n, W]
D. X[n-1, n]

Résumé

1) Déterminer une sous-structure optimale dont on a besoin pour résoudre le problème
   Fib(n) : $n^{ème}$ nombre de Fibonacci
   OPT(i) : valeur d'une solution optimale en se restreignant aux intervalles 1, ..., i.
   X[i, j] : il existe un sous-ensemble de $\{a_1, a_2, ..., a_i\}$ dont la somme des éléments est j.
2) Caractériser (par une équation) cette sous-structure optimale
   Fib(n) = Fib(n-1)+Fib(n-2)
   OPT(i) = max { v_i + OPT(der(i)) , OPT(i-1)}
   X[i, j] = X[i - 1, j] ou X[i-1, j -a_i]

   Il ne doit pas y avoir de circuit dans le graphe des dépendances entre sous-problèmes.
3) En déduire la valeur d'une solution optimale
   Fib(n) ; OPT(n) ; X[n, W]
4) Ecrire un algorithme de programmation dynamique pour résoudre le problème
   - récursif (avec mémoïsation), ou
   - itératif (approche du bas vers le haut : on considère les sous-problèmes dans un ordre topologique)
5) Retrouver la solution optimale à partir de sa valeur grâce au retour en arrière (backtrack).

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.

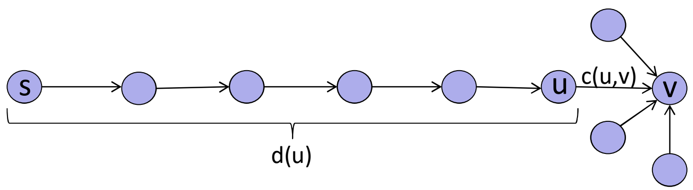
Supposons qu'il n'y ait pas de circuit absorbant accessible à partir de s.
Comment calculer *A(s)* ?

- 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 un plus court chemin de s à v ?

On ne connait 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.

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

Plus courts chemins : programme dynamique

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

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
```

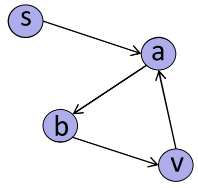
Plus courts chemins : programme dynamique

Algorithme récursif avec mémoïsation :

$d(s) = 0$
$d(v) = \min\limits_{u \mid (u,v) \in A} \{ d(u) + c(u,v) \}$

```
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]
```



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(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]
```

Quelle est la complexité de cet algorithme ?

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

A. $O(n)$
B. $O(n+m)$
C. $O(n^2)$
D. $O(nm)$

Complexité de l'appel non mémoïsé de d(v) : $d^-(v) + 1$
Appels non mémoïsés : un par sommet (n sommets)
=> Complexité totale = $O(n+m)$

Plus courts chemins dans un graphe sans circuit

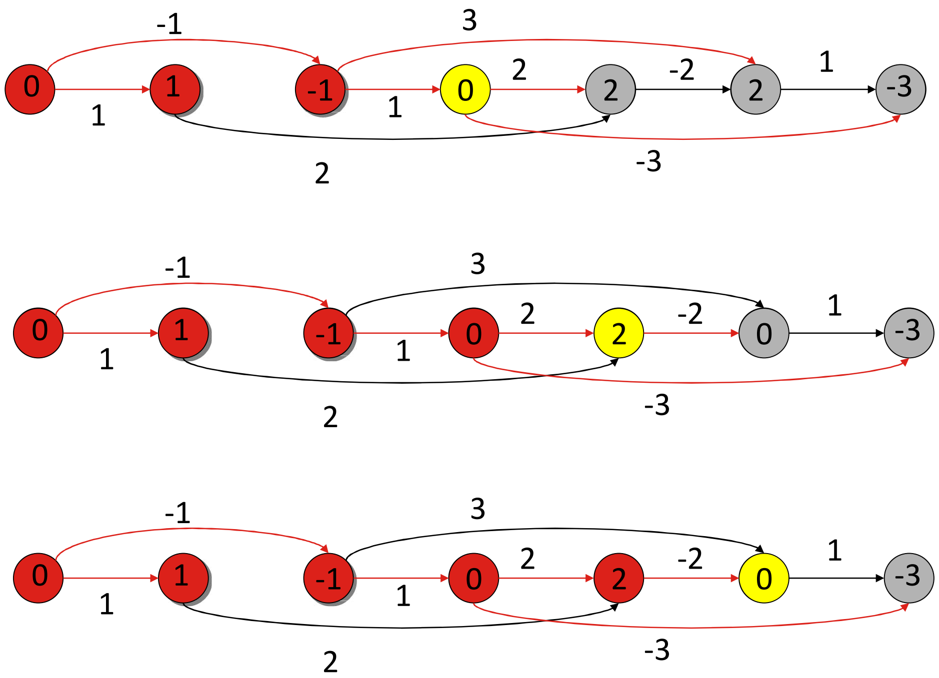
Version ``du bas vers le haut” de cet algorithme
(on suppose que s est une racine) :

soit $L=(s_1,\ldots,s_n)$ une liste topologique des sommets de G telle que $s= s_1$
pour tout sommet $x \neq s$ $d[x]= + \infty$; $d[s]=0$
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
            $d(y)= d(s_k) + c(s_k,y)$

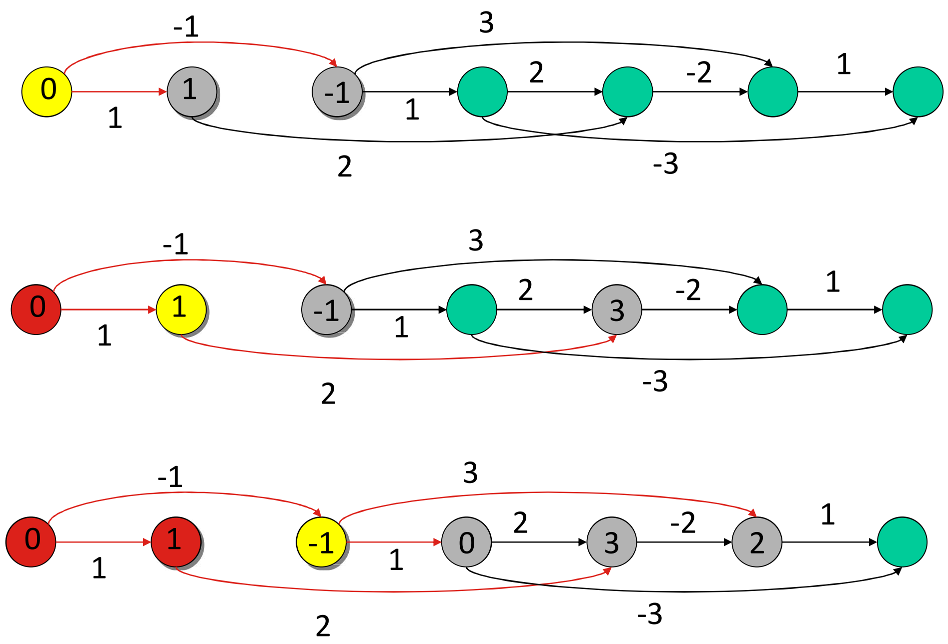
On obtient *A(s)*, une arborescence des plus courts chemins d’origine s.

C’est l’algorithme de Bellman.

Complexité totale = O(n+m)



Exemple :



Plus courts chemins dans le cas général

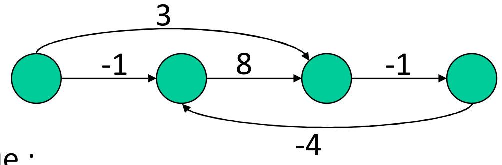
Les dépendances entre sous-problèmes doivent être acycliques.

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.

Cette transformation rend tout graphe acyclique.

Exemple : Graphe avec circuit :



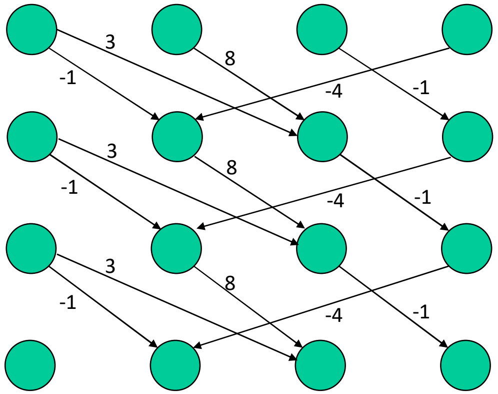
Graphe correspondant acyclique :

niveau 0 :

niveau 1 :

niveau 2 :

niveau 3 :



A chaque chemin dans G correspond un chemin de même coût dans le graphe acyclique correspondant.

Cas général : première version

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

$$\begin{cases} d_0(s) = 0 \text{ , } d_0(x) = +\infty \text{ pour tout } x \neq s \\ d_k(v) = \min\limits_{u \mid (u,v) \in A} \{ d_{k-1}(u) + c(u,v) \} \quad \text{si } k>0 \end{cases}$$

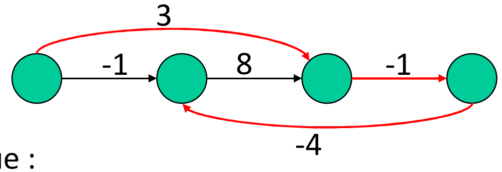
But : calcul de la valeur min $d_i(v)$ (plus court chemin élémentaire).

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

Complexité de l'appel non mémoïsé de $d_k(v)$: $d^-(v) + 1$

Complexité : $O(n^2 + nm)$

Exemple : Graphe avec circuit :



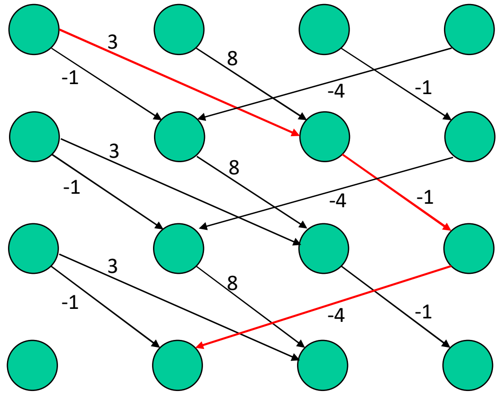
Graphe correspondant acyclique :

niveau 0 :

niveau 1 :

niveau 2 :

niveau 3 :



A chaque chemin élémentaire dans G correspond un chemin de même coût dans le graphe acyclique correspondant.

Cas général : deuxième version

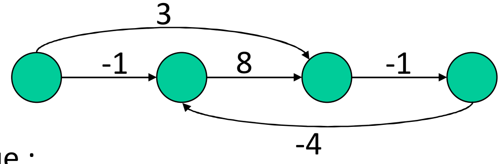
Soit $d_k(v)$ le coût d'un plus court chemin de s à v qui utilise **au plus** k arcs ($0 \leq k \leq n\text{-}1$).

$$\begin{cases} d_0(s) = 0 \text{ , } d_0(x) = +\infty \text{ pour tout } x \neq s \\ d_k(v) = \min \{ d_{k-1}(v) \text{ , } \min\limits_{u \mid (u,v) \in A} \{ d_{k-1}(u) + c(u,v) \} \} \quad \text{si } k>0 \end{cases}$$

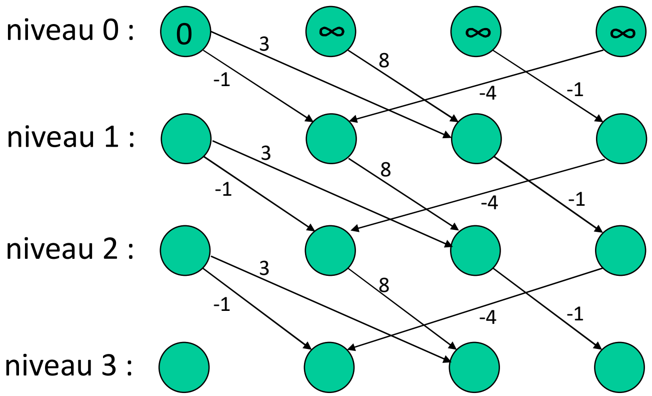
But : calcul de la valeur $d_{n-1}(v)$ (plus court chemin élémentaire).

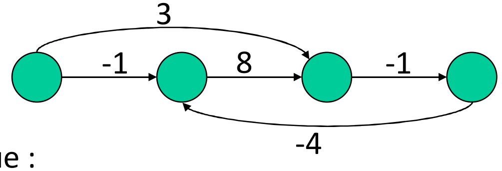
Algorithme de Bellman-Ford : algorithme récursif avec mémoïsation correspondant à la relation de récurrence ci-dessus.

Complexité identique à la première version : $O(n^2 + nm)$

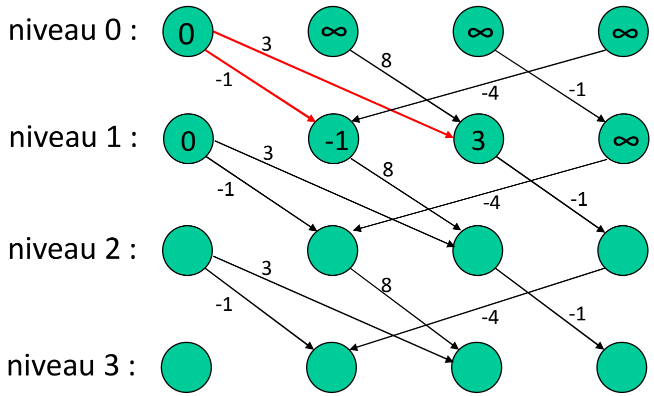
Exemple : Graphe avec circuit : 

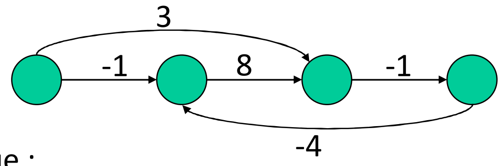
Graphe correspondant acyclique :



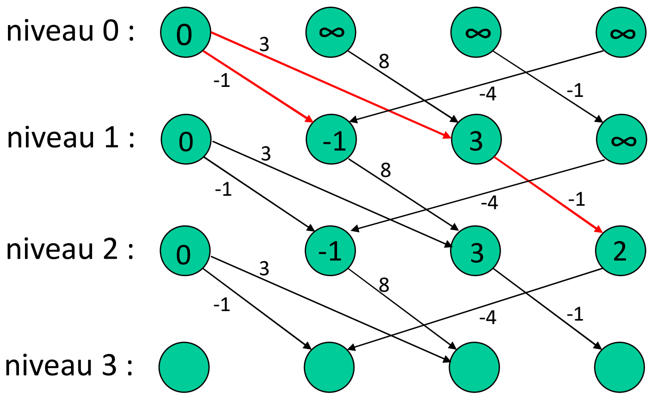
Exemple : Graphe avec circuit : 

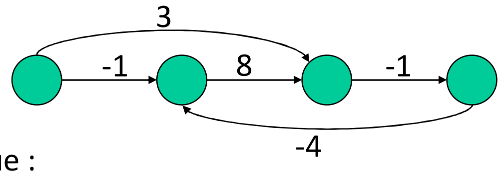
Graphe correspondant acyclique :



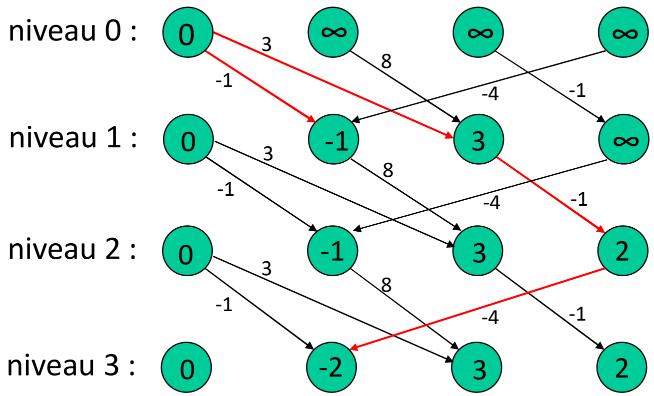
Exemple : Graphe avec circuit : 

Graphe correspondant acyclique :

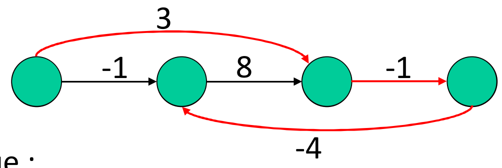


Exemple : Graphe avec circuit : 

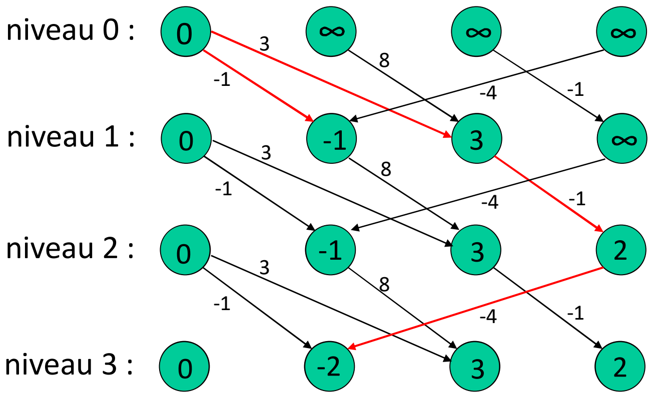
Graphe correspondant acyclique :



Exemple : Graphe avec circuit :



Graphe correspondant acyclique :



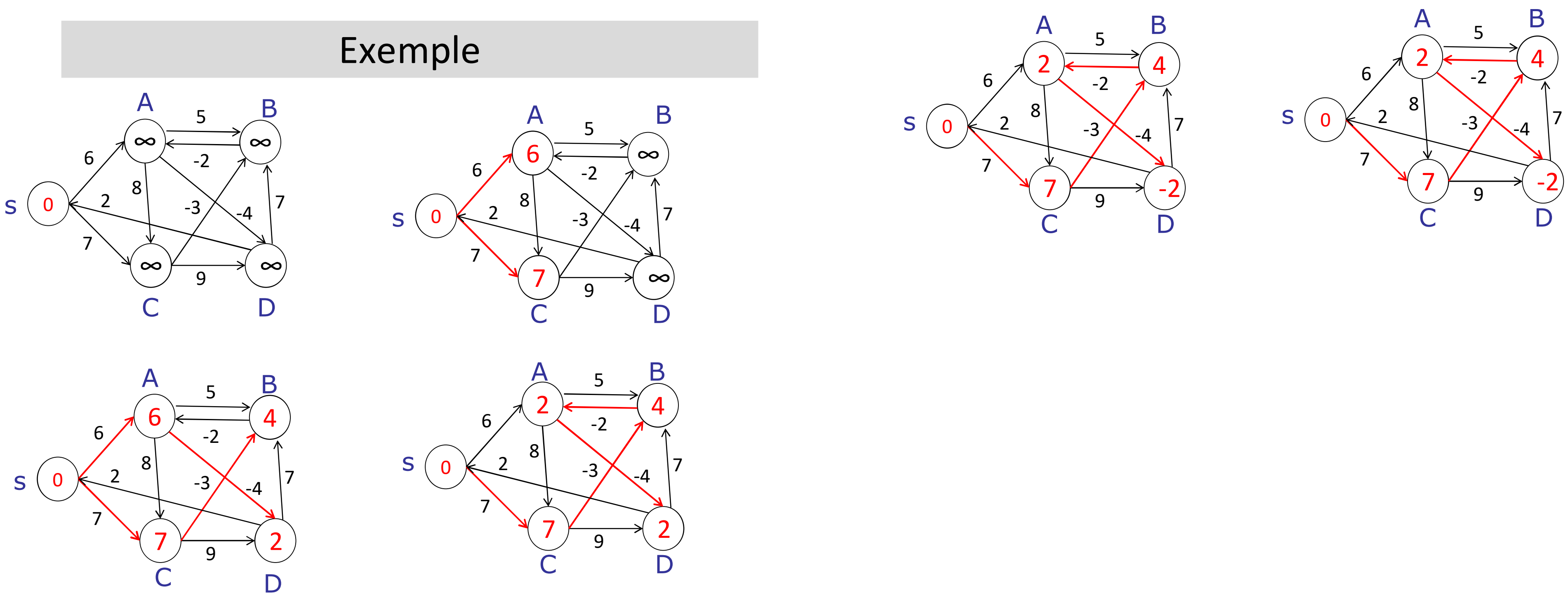
Algorithme de Bellman Ford (troisième version)

// Initialisation
pour tout sommet x **faire**
  si x=s alors d(x) = 0; pred(x)=null
  sinon d(x) = +∞ ; pred(x)=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) ≠ ∞, 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 µ.(y,x).

Validité

Propriété 1 : A tout moment après l'étape d'initialisation, si d(x) ≠ ∞, 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.

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 ≤ à la longueur de ce chemin.

  - A l'itération i, d(x) est comparé à d(y)+c(x,y) et mise à jour à cette valeur si cela diminue d(x). Donc, à la fin de cette itération, d(x) est ≤ à la longueur de µ.

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.