Algorithmique & Programmation (INF 431)

Programmation dynamique

Benjamin Werner   François Pottier

10 avril 2013

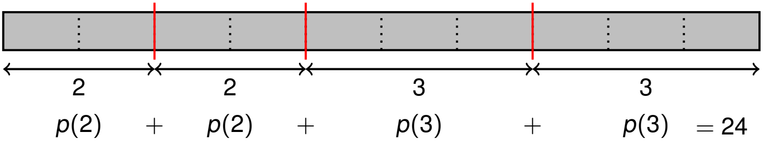
Le problème

Nous avons une barre de métal de longueur n :

et une table du prix de vente des barres suivant leur longueur :

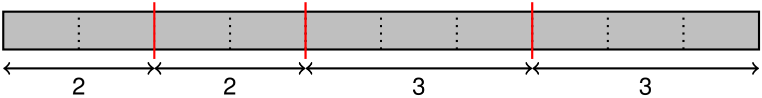
longueur	0	1	2	3	...	i
prix	0	2	5	7	...	$p(i)$

La question est : comment découper ?



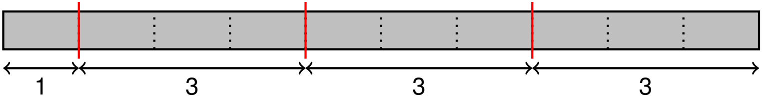
Le problème

Voici donc une découpe possible :



$$p(2) + p(2) + p(3) + p(3) = 24$$

Bien sûr, d'autres découpes sont possibles :



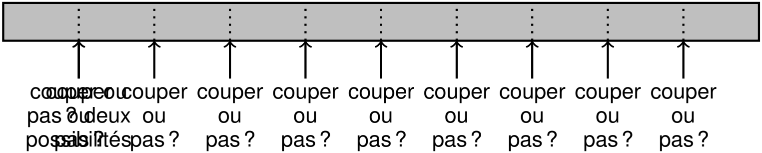
$$p(1) + p(3) + p(3) + p(3) = 23$$

Benjamin Werner, François Pottier

Un problème combinatoire

Le problème est de nature discrète : l'unité de longueur est indivisible.

Nous sommes donc face à un ensemble de choix binaires :



Il y a 2^{n-1} découpages possibles.

Une approche naïve aura une complexité exponentielle.

Découpage en sous-problèmes

Une fois notre barre coupée en deux, nous avons deux barres.

Chacune doit être découpée de façon optimale.

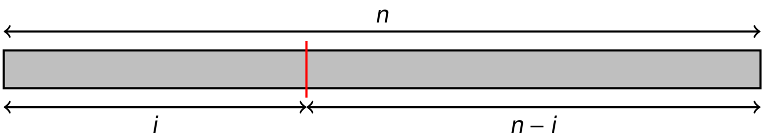
Chacune peut être découpée indépendamment de l'autre.

Nous avons donc deux sous-problèmes analogues au problème initial.

Plus généralement, nous avons une famille de sous-problèmes, indicés par n, dont les solutions optimales sont reliées les unes aux autres.

Soit $M(n)$ le prix optimal que l'on peut tirer d'une barre de longueur n.

Liens entre sous-problèmes



Si on effectue une première découpe en i, alors on obtiendra le prix :

$$M(i) + M(n-i)$$

Ou bien, si on décide de ne plus découper, on obtiendra le prix :

$$p(n)$$

Ce sont les seules possibilités, donc le prix optimal est :

$$M(n) = \max\begin{pmatrix} \max_{0<i<n} \; M(i) + M(n-i) \\ p(n) \end{pmatrix}$$

Mise en équations

Nous avons donc obtenu un système d'équations :

$$M(n) = \max\left(\begin{array}{l} \max_{0<i<n} \; M(i) + M(n-i) \\ p(n) \end{array} \right)$$

Ce système est acyclique : i et $n - i$ sont strictement inférieurs à n.

Mise en équations

On peut raffiner ces équations en exploitant la symétrie et en s'intéressant à la coupure la plus proche de l'extrémité de la barre :

$$M(n) = \max\left(\begin{array}{l} \max_{0 < i \leq n-i} p(i) + M(n-i) \\ p(n) \end{array} \right)$$

Ce raffinement n'a pas un impact profond.

Il ne reste « plus qu'à » calculer...

Lecture récursive des équations

On peut lire l'équation comme une définition de fonction récursive :

```
public int compute (int n)
{
  int max = price[n] ;
  for (int i = 1 ; i <= n - i ; i++)
    max = Math.max (max, price[i] + compute(n-i)) ;
  return max ;
}
```

Cette fonction termine parce que le système est acyclique.

Le problème est-il résolu ?

La complexité est toujours exponentielle

Non, bien sûr.

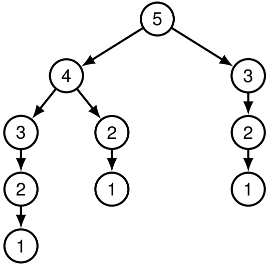
Cette vision récursive reste trop naïve.

Un appel à `compute(10)` provoque en tout 284 appels à `compute`.
Un appel à `compute(20)` provoque 281076 appels !

La complexité de `compute` est exponentielle.

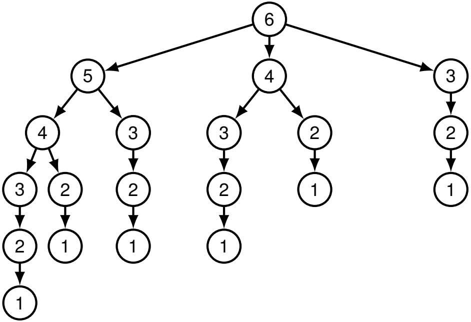
L'arbre des appels récursifs

Voici l'arbre des appels récursifs pour $n = 5$:



L'arbre des appels récursifs

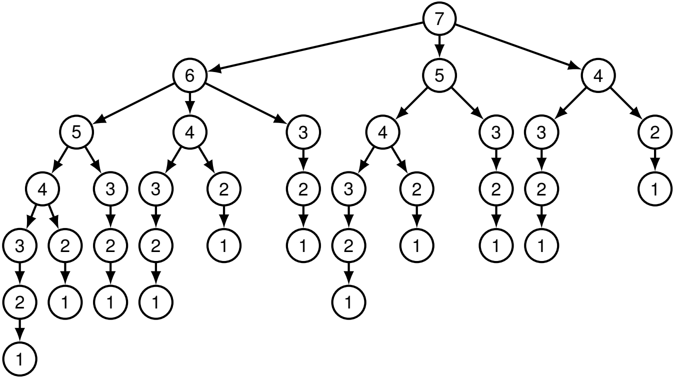
Voici l'arbre des appels récursifs pour $n = 6$:



La complexité est exponentielle car on (re-)calcule toujours la même chose !

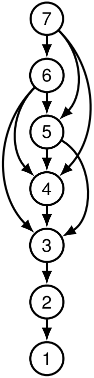
L'arbre des appels récursifs

Voici l'arbre des appels récursifs pour $n = 7$:



Une cure d'amaigrissement s'impose...

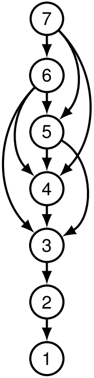
La solution : le partage



Pour éviter cela, il faut partager les calculs redondants.

L'arbre doit devenir un DAG (graphe orienté acyclique) des sous-problèmes.

La solution : le partage



QUIZ : quelle est la taille de ce DAG en fonction de n ?

1. $O(n)$
2. $O(n^2)$ – compter sommets et arêtes !
3. $O(2^n)$

Comment réaliser le partage ?

Ce DAG peut être parcouru :

- de haut en bas : mémoisation ; ou
- de bas en haut : programmation dynamique.

Dans les deux cas, on mémorise la solution de chaque sous-problème. On dépense de l'espace pour économiser du temps.

Voyons cela...

Mémoisation : définition

Voici une définition ré-utilisable de la mémoisation :

```
public abstract class Memoize<A, B> {
  private HashMap<A, B> table = new HashMap<A, B> () ;
  public B get (A a) {
    B b = table.get(a) ;
    if (b == null) {
      b = compute(a) ;
      table.put(a, b) ;
    }
    return b ;
  }
  public abstract B compute (A a) ;
}
```

Le calcul est effectué par `compute`. Le client, ainsi que `compute` elle-même, utilise `get`, qui consulte la table avant d'appeler `compute`.

Mémoisation : utilisation

Et voici une version mémoisante de notre algorithme :

```
public class MemoBarCut extends Memoize<Integer, Integer> {
  private int[] price ;
  @Override public Integer compute (Integer n)
  {
    int max = price[n] ;
    for (int i = 1 ; i <= n - i ; i++)
      // call get, not compute !
      max = Math.max (max, price[i] + get(n-i)) ;
    return max ;
  }
}
```

L'appel récursif est à `get`, surtout pas à `compute`.

Le constructeur qui initialise `price` a été omis.

Mémoisation

Avec la mémoisation,

- on n'a pas étudié dans quel ordre les calculs auront lieu,
- mais le calcul termine car le graphe est acyclique.
- et chaque calcul est effectué au plus une fois.

Une autre technique consiste à fixer à l'avance une stratégie de parcours du DAG... c'est la programmation dynamique.

Programmation dynamique

Un ordre de parcours approprié est « de bas en haut », c'est-à-dire à *n* croissant :

```
public int compute (int N)
{
  int[] optimum = new int [N+1] ;
  for (int n = 0 ; n <= N ; n++) {
    int max = price[n] ;
    for (int i = 1 ; i <= n - i ; i++)
      max = Math.max (max, price[i] + optimum[n-i]) ;
    optimum[n] = max ;
  }
  return optimum[N] ;
}
```

Lorsque nous lisons `optimum[n-i]`, il a déjà été calculé.

Complexité

Les deux versions (mémoisation et programmation dynamique) ont la même complexité en temps : $O(n^2)$ et en espace : $O(n)$.

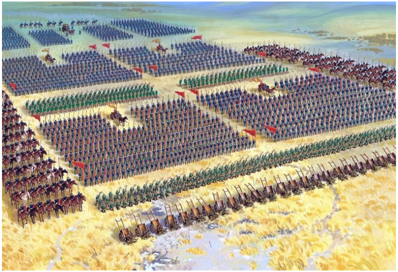
La première correspond à la taille du DAG des sous-problèmes, en comptant sommets et arêtes ; la seconde à sa taille, en comptant les sommets seuls.

Dynamique versus statique

Le mot « dynamique » désigne ce qui est décidé pendant l'exécution.

Dynamique versus statique

Le mot « statique » désigne ce qui est décidé avant l'exécution.



Un contresens historique

L'expression « programmation dynamique » est assez mal choisie.

Cette technique consiste à :

- identifier une famille de sous-problèmes ;
- identifier les liens entre ces sous-problèmes ;
- s'assurer que le graphe des sous-problèmes est acyclique ;
- fixer statiquement un ordre de parcours de ce DAG.

Le problème

On se donne n matrices rectangulaires : $A_0, A_1, \ldots, A_{n-1}$.

On souhaite calculer le produit $A_0 \times A_1 \times \ldots \times A_{n-1}$.

(Cormen et al., §15.2.)

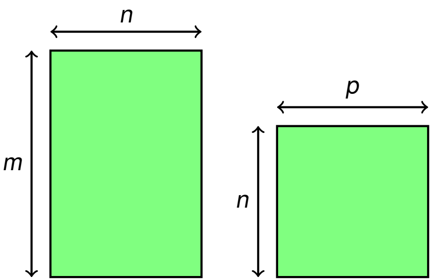
Le problème

La mutiplication de matrices étant associative, il y a de nombreuses façons de parenthéser, donc de calculer.

$$(\ldots((A_0 \times A_1) \times A_2) \times \ldots \times A_{n-1})$$
$$(A_0 \times (A_1 \times \ldots \times (A_{n-2} \times A_{n-1}) \ldots))$$
etc.

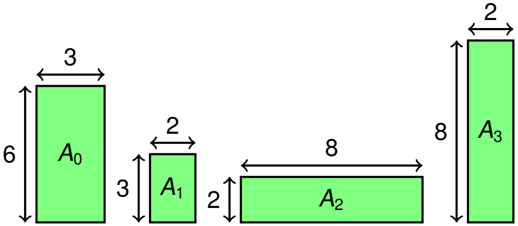
Est-ce que cela fait une différence ?

Une mesure de coût



L'algorithme de multiplication naïf a une complexité $O(m \times n \times p)$.

Une mesure de coût : exemple



$$((A_0 \times (A_1 \times A_2)) \times A_3) \quad 6 \times 3 \times 8 + 3 \times 2 \times 8 + 6 \times 8 \times 2 = 144 + 48 + 96 = \mathbf{288}$$

$$((A_0 \times A_1) \times A_2) \times A_3 \quad 6 \times 3 \times 2 + 6 \times 2 \times 8 + 6 \times 8 \times 2 = 36 + 96 + 96 = \mathbf{228}$$

$$A_0 \times (A_1 \times (A_2 \times A_3)) \quad 6 \times 3 \times 2 + 3 \times 2 \times 2 + 2 \times 8 \times 2 = 36 + 18 + 32 = \mathbf{80}$$

Le problème

On suppose le coût d'une multiplication de matrices donné par la mesure $m \times n \times p$.

On cherche une façon de parenthéser le produit $A_0 \times \ldots \times A_{n-1}$ qui minimise le coût total.

L'approche naïve

Si on essayait naïvement toutes les façons de parenthéser ?

Combien y en a-t-il ?

$$P(n) = \begin{cases} 1 & \text{si } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{si } n \geq 2 \end{cases}$$

$P(n)$ est le n-ième nombre de Catalan. C'est le nombre d'arbres binaires à n feuilles.

On vérifie facilement que $P(n) \geq 2^n$. L'approche naïve est impraticable.

Découpage en sous-problèmes

Le problème est analogue à celui de la découpe de la barre.

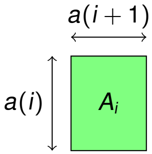
Si $n \geq 2$, le produit $A_0 \times \ldots \times A_{n-1}$ s'écrira $(A_0 \times \ldots \times A_k) \times (A_{k+1} \times \ldots \times A_{n-1})$ pour un certain $i \in [1, k)$.

Une fois k fixé, il reste à déterminer comment calculer $A_0 \times \ldots \times A_k$ et $A_{k+1} \times \ldots \times A_{n-1}$ de façon optimale.

Ce sont deux sous-problèmes analogues au problème initial.

Découpage en sous-problèmes

Pour $0 \leq i < j \leq n$, soit $C(i,j)$ le nombre minimal de multiplications scalaires nécessaires pour calculer le sous-produit $A_i \times \ldots \times A_{j-1}$.



Notons $a(i)$ la hauteur de la matrice A_i.
Notons $a(i+1)$ sa largeur.

Nous avons donc :

$$C(i,j) = \begin{cases} 0 & \text{si } j - i = 1 \\ \min_{i<k<j} \; C(i,k) + C(k,j) + a(i)a(k)a(j) & \text{si } j - i \geq 2 \end{cases}$$

Ordonnancement statique

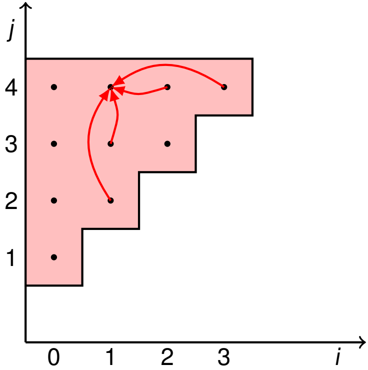
$$C(i,j) = \begin{cases} 0 & \text{si } j-i=1 \\ \min_{i<k<j} \; C(i,k)+C(k,j)+a(i)a(k)a(j) & \text{si } j-i \geq 2 \end{cases}$$

Ce système d'équations est acyclique, car $k-i < j-i$ et $j-k < j-i$.

On calcule les $C(i,j)$ pour $j-i$ croissant.

Le DAG des sous-problèmes

Il est utile de représenter visuellement le DAG des sous-problèmes.



Le calcul en i, j dépend des résultats obtenus plus bas dans la colonne i et plus à droite dans la ligne j.

L'algorithme

```
public static int[][] compute (int[] a)
{
  final int n = a.length - 1 ;
  int[][] c = new int [n][n+1] ;
  for (int i = 0 ; i < n ; i++)          // Base case.
    c[i][i+1] = 0 ;
  for (int jmi = 2 ; jmi <= n ; jmi++) // Inductive case.
    for (int i = 0, j = i + jmi ; j <= n ; i++, j = i + jmi) {
      c[i][j] = Integer.MAX_VALUE ;
      for (int k = i+1 ; k < j ; k++)
        c[i][j] = Math.min(
          c[i][j],
          c[i][k] + c[k][j] + a[i] * a[k] * a[j]
        ) ;
    }
  return c ; // c[0][n] is the cost of the initial problem.
}
```

Complexité

La complexité est $O(n^3)$ pour le temps et $O(n^2)$ pour l'espace.

La première correspond à la taille du DAG des sous-problèmes, en comptant sommets et arêtes ; la seconde à sa taille, en comptant les sommets seuls.

Comment atteindre l'optimal ?

Calculer les $C(i, j)$, c'est bien, mais encore faut-il retrouver quel parenthésage du produit $A_0 \times \ldots \times A_{n-1}$ conduit à la complexité optimale $C(0, n)$.

Exercice : écrire une fonction récursive qui, à l'aide du tableau $C(i, j)$, détermine et affiche le parenthésage optimal.

Le problème

Soit G un graphe orienté à n sommets, numérotés de 0 à $n-1$.

L'arête de i à j est étiquetée par un coût $G(i,j) \in \mathbb{R} \cup \{\infty\}$.

Un coût infini indique que l'arête n'existe pas.

On suppose $G(i,i) = 0$ pour tout i.

On souhaite calculer, pour tous i et j, la distance $d(i,j) \in \mathbb{R} \cup \{-\infty, \infty\}$, c'est-à-dire le coût minimal d'un chemin de i à j.

On ne fixe pas a priori de source ni de cible.

Quels sous-problèmes ? Quelles équations ?

QUIZ : quelle famille de sous-problèmes permet une mise en équations exploitable ?

1. $d(i,j)$, tout simplement ;
2. $d_k(i,j)$, définie comme le coût minimal d'un chemin de i à j qui emprunte des sommets intermédiaires d'indice inférieur à k ;
3. $d_k(i,j)$, définie comme le coût minimal d'un chemin de i à j qui emprunte au plus k arêtes ;
4. $d_k(i,j)$, définie comme le coût minimal d'un chemin de i à j qui passe au moins une fois par le sommet k.

Quels sous-problèmes ? Quelles équations ?

L'algorithme de Floyd et Warshall définit $d_k(i,j)$ comme le coût minimal d'un chemin de i à j qui emprunte des sommets intermédiaires d'indice inférieur à k.

L'algorithme de Bellman et Ford définit $d_k(i,j)$ comme le coût minimal d'un chemin de i à j qui emprunte au plus k arêtes.

Ces définitions vont conduire à des équations acycliques donc exploitables.

L'approche de Floyd et Warshall

L'algorithme de Floyd et Warshall définit $d_k(i, j)$ comme le coût minimal d'un chemin de i à j qui emprunte des sommets intermédiaires d'indice inférieur à k.

Lorsque $k = n$, on obtient $d(i, j)$.

Mise en équations : cas de base

Un chemin qui emprunte des sommets intermédiaires d'indice strictement inférieur à 0 ne peut emprunter aucun sommet intermédiaire.

Donc,

$$d_0(i, j) = G(i, j)$$

Mise en équations : cas inductif

Un chemin optimal de i à j qui emprunte des sommets intermédiaires d'indice strictement inférieur à $k+1$ soit ne passe jamais par le sommet k, soit passe exactement une fois par le sommet k.

Donc,

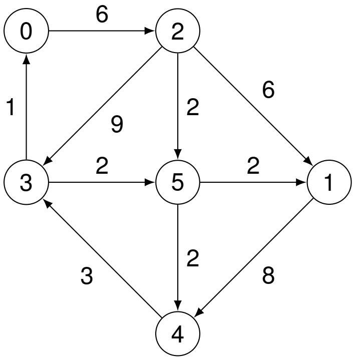
$$d_{k+1}(i,j) = \min\begin{pmatrix} d_k(i,j) \\ d_k(i,k) + d_k(k,j) \end{pmatrix}$$

Est-ce bien vrai ? Oui, si G n'a pas de cycle de coût négatif.

Le système d'équations ainsi obtenu est acyclique.

On peut calculer les $d_k(i,j)$ pour k croissant de 0 à n.

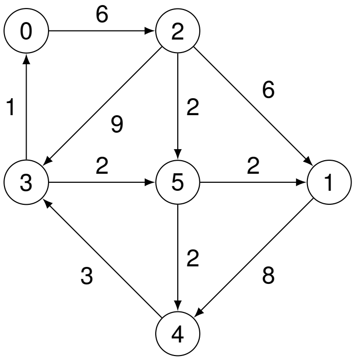
Initialisation : $k = 0$



$d(i,j)$

	0	1	2	3	4	5
0	0	∞	6	∞	∞	∞
1	∞	0	∞	∞	8	∞
2	∞	6	0	9	∞	2
3	1	∞	∞	0	∞	2
4	∞	∞	∞	3	0	∞
5	∞	2	∞	∞	2	0

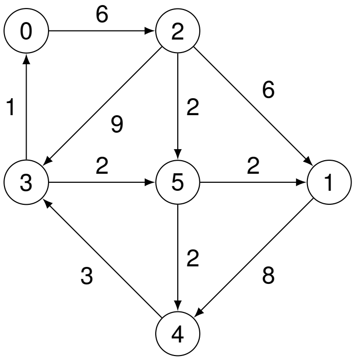
Après l'itération $k = 1$



$d(i, j)$

	0	1	2	3	4	5
0	0	∞	6	∞	∞	∞
1	∞	0	∞	∞	8	∞
2	∞	6	0	9	∞	2
3	1	∞	7	0	∞	2
4	∞	∞	∞	3	0	∞
5	∞	2	∞	∞	2	0

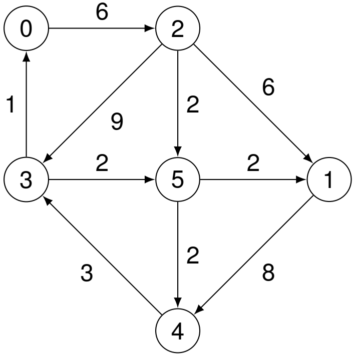
Après l'itération $k = 2$



$d(i,j)$

	0	1	2	3	4	5
0	0	∞	6	∞	∞	∞
1	∞	0	∞	∞	8	∞
2	∞	6	0	9	14	2
3	1	∞	7	0	∞	2
4	∞	∞	∞	3	0	∞
5	∞	2	∞	∞	2	0

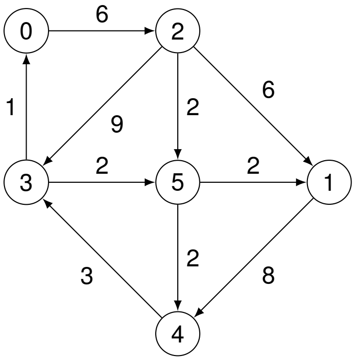
Après l'itération $k = 3$



$d(i, j)$

	0	1	2	3	4	5
0	0	12	6	15	20	8
1	∞	0	∞	∞	8	∞
2	∞	6	0	9	14	2
3	1	13	7	0	21	2
4	∞	∞	∞	3	0	∞
5	∞	2	∞	∞	2	0

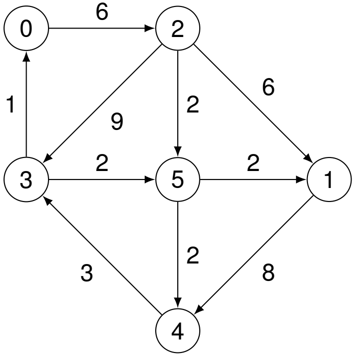
Après l'itération $k = 4$



$d(i, j)$

	0	1	2	3	4	5
0	0	12	6	15	20	8
1	∞	0	∞	∞	8	∞
2	10	6	0	9	14	2
3	1	13	7	0	21	2
4	4	16	10	3	0	5
5	∞	2	∞	∞	2	0

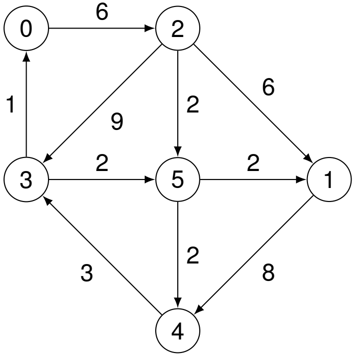
Après l'itération $k = 5$



$d(i,j)$

	0	1	2	3	4	5
0	0	12	6	15	20	8
1	12	0	18	11	8	13
2	10	6	0	9	14	2
3	1	13	7	0	21	2
4	4	16	10	3	0	5
5	6	2	12	5	2	0

Après l'itération $k = 6$



$d(i, j)$

	0	1	2	3	4	5
0	0	10	6	13	10	8
1	12	0	18	11	8	13
2	8	4	0	7	4	2
3	1	4	7	0	4	2
4	4	7	10	3	0	5
5	6	2	12	5	2	0

L'algorithme de Floyd et Warshall

On utilise `Integer.MAX_VALUE` pour représenter ∞.

```
public static void fw (int[][] g)
{
  final int n = g.length ;
  for (int k = 0 ; k < n ; k++)
    for (int i = 0 ; i < n ; i++)
      for (int j = 0 ; j < n ; j++)
        g[i][j] = Math.min(
          g[i][j],
          add(g[i][k], g[k][j]) // addition in ℤ ∪ {∞}
        ) ;
}
```

QUIZ : ce code est-il conforme aux équations ? est-il correct ?

1. oui ; oui ;
2. non ; oui ;
3. non ; non.

Une subtilité

Ce code n'est pas conforme à la lettre aux équations : parce qu'on modifie le tableau `g` en place, on risque de trouver dans `g[i][k]` la valeur $d_{k+1}(i,k)$ au lieu de $d_k(i,k)$.

Cependant on peut démontrer que cela ne change rien, car la différence ne concerne que des chemins qui passent plusieurs fois par le sommet k.

Donc, ce code est correct !

Ne pas mémoriser $d_k(i,j)$ pour toutes les valeurs de k diminue la complexité en espace de $O(n^3)$ à $O(n^2)$.

Mémoriser $d_k(i,j)$ pour une seule valeur de k, au lieu de deux, fait économiser un facteur constant supplémentaire.

La complexité en temps reste $O(n^3)$.

Et le calcul des chemins optimaux ?

Nous avons calculé la distance $d(i, j)$, mais pas comment aller de i à j.

On peut modifier l'algorithme pour mémoriser non seulement la distance optimale, mais comment cette distance a été obtenue.

Donnons-nous un tableau $\pi_k(i, j)$ tel que si $i \neq j$ et si $d_k(i, j) < \infty$, alors $\pi_k(i, j)$ est le dernier sommet avant j sur un chemin optimal de i à j qui emprunte des sommets intermédiaires d'indice inférieur à k.

La valeur de $\pi_k(i, j)$ lorsque $i = j$ ou $d_k(i, j) = \infty$ n'a pas d'importance. Je la noterai $\times$, mais on pourrait employer un entier quelconque.

La ligne i du tableau π représentera une arborescence des plus courts chemins en provenance du sommet i.

Équations à propos de $\pi_k(i,j)$

Cette définition, pour le cas de base, satisfait notre spécification :

$$\begin{array}{ll} \pi_0(i,j) = i & \text{si } i \neq j \text{ et } G(i,j) < \infty \\ \pi_0(i,j) = \times & \text{sinon} \end{array}$$

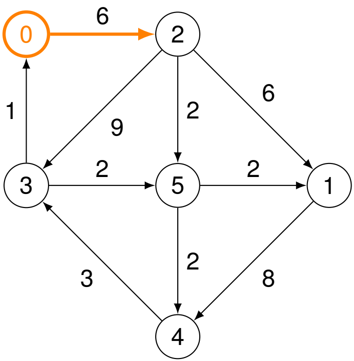
et celle-ci, pour le cas inductif :

$$\begin{array}{ll} \pi_{k+1}(i,j) = \pi_k(i,j) & \text{si } d_{k+1}(i,j) = d_k(i,j) \\ \pi_{k+1}(i,j) = \pi_k(k,j) & \text{si } d_{k+1}(i,j) = d_k(i,k) + d_k(k,j) \end{array}$$

Exercice : modifier le code Java pour calculer $\pi(i,j)$.

Exercice : modifier le code Java pour détecter un cycle de coût négatif.

Initialisation : $k = 0$



Les plus courts chemins à partir du sommet 0 sont affichés en **orange**.

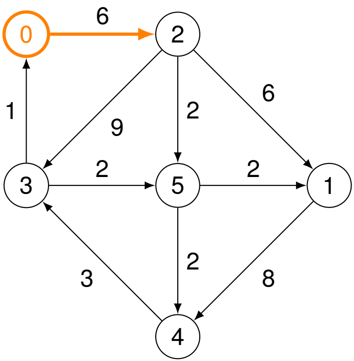
$d(i,j)$

	0	1	2	3	4	5
0	0	∞	6	∞	∞	∞
1	∞	0	∞	∞	8	∞
2	∞	6	0	9	∞	2
3	1	∞	∞	0	∞	2
4	∞	∞	∞	3	0	∞
5	∞	2	∞	∞	2	0

$\pi(i,j)$

	0	1	2	3	4	5
0	$\times$	$\times$	**0**	$\times$	$\times$	$\times$
1	$\times$	$\times$	$\times$	$\times$	1	$\times$
2	$\times$	2	$\times$	2	$\times$	2
3	3	$\times$	$\times$	$\times$	$\times$	3
4	$\times$	$\times$	$\times$	4	$\times$	$\times$
5	$\times$	5	$\times$	$\times$	5	$\times$

Après l'itération $k = 1$



Les plus courts chemins à partir du sommet 0 sont affichés en **orange**.

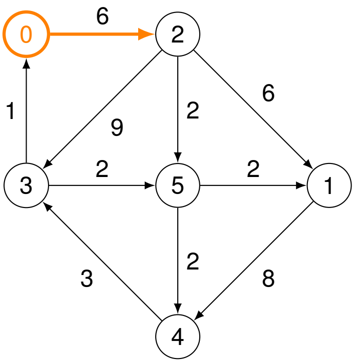
$d(i,j)$

	0	1	2	3	4	5
0	0	∞	6	∞	∞	∞
1	∞	0	∞	∞	8	∞
2	∞	6	0	9	∞	2
3	1	∞	7	0	∞	2
4	∞	∞	∞	3	0	∞
5	∞	2	∞	∞	2	0

$\pi(i,j)$

	0	1	2	3	4	5
0	×	×	**0**	×	×	×
1	×	×	×	×	1	×
2	×	2	×	2	×	2
3	3	×	0	×	×	3
4	×	×	×	4	×	×
5	×	5	×	×	5	×

Après l'itération $k = 2$



Les plus courts chemins à partir du sommet 0 sont affichés en **orange**.

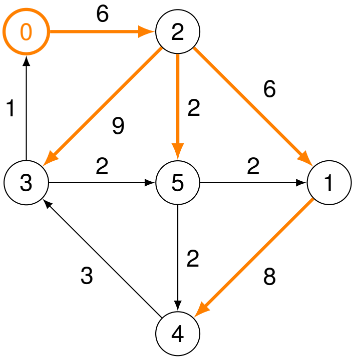
$d(i, j)$

	0	1	2	3	4	5
0	0	∞	6	∞	∞	∞
1	∞	0	∞	∞	8	∞
2	∞	6	0	9	14	2
3	1	∞	7	0	∞	2
4	∞	∞	∞	3	0	∞
5	∞	2	∞	∞	2	0

$\pi(i, j)$

	0	1	2	3	4	5
0	×	×	0	×	×	×
1	×	×	×	×	1	×
2	×	2	×	2	1	2
3	3	×	0	×	×	3
4	×	×	×	4	×	×
5	×	5	×	×	5	×

Après l'itération $k = 3$



Les plus courts chemins à partir du sommet 0 sont affichés en **orange**.

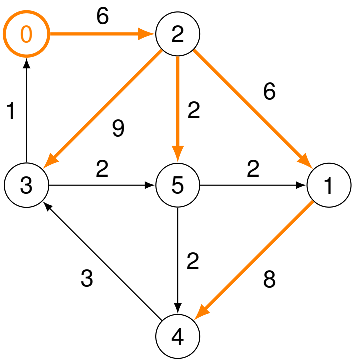
$d(i, j)$

	0	1	2	3	4	5
0	0	12	6	15	20	8
1	∞	0	∞	∞	8	∞
2	∞	6	0	9	14	2
3	1	13	7	0	21	2
4	∞	∞	∞	3	0	∞
5	∞	2	∞	∞	2	0

$\pi(i, j)$

	0	1	2	3	4	5
0	×	2	0	2	1	2
1	×	×	×	×	1	×
2	×	2	×	2	1	2
3	3	2	0	×	1	3
4	×	×	×	4	×	×
5	×	5	×	×	5	×

Après l'itération $k = 4$



Les plus courts chemins à partir du sommet 0 sont affichés en **orange**.

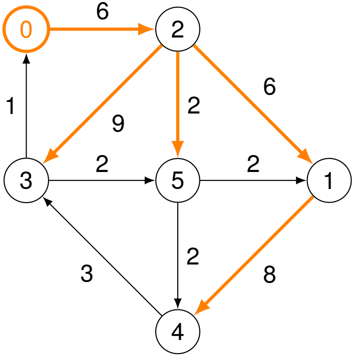
$d(i,j)$

	0	1	2	3	4	5
0	0	12	6	15	20	8
1	∞	0	∞	∞	8	∞
2	10	6	0	9	14	2
3	1	13	7	0	21	2
4	4	16	10	3	0	5
5	∞	2	∞	∞	2	0

$\pi(i,j)$

	0	1	2	3	4	5
0	×	**2**	**0**	**2**	**1**	**2**
1	×	×	×	×	1	×
2	3	2	×	2	1	2
3	3	2	0	×	1	3
4	3	2	0	4	×	3
5	×	5	×	×	5	×

Après l'itération $k = 5$



Les plus courts chemins à partir du sommet 0 sont affichés en **orange**.

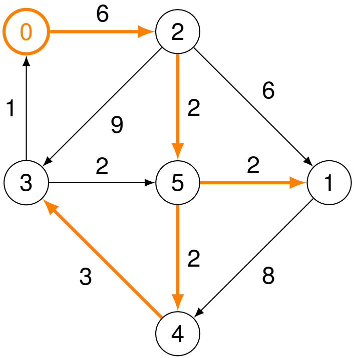
$d(i,j)$

	0	1	2	3	4	5
0	0	12	6	15	20	8
1	12	0	18	11	8	13
2	10	6	0	9	14	2
3	1	13	7	0	21	2
4	4	16	10	3	0	5
5	6	2	12	5	2	0

$\pi(i,j)$

	0	1	2	3	4	5
0	×	2	0	2	1	2
1	3	×	0	4	1	3
2	3	2	×	2	1	2
3	3	2	0	×	1	3
4	3	2	0	4	×	3
5	3	5	0	4	5	×

Après l'itération $k = 6$



Les plus courts chemins à partir du sommet 0 sont affichés en **orange**.

$d(i,j)$

	0	1	2	3	4	5
0	0	10	6	13	10	8
1	12	0	18	11	8	13
2	8	4	0	7	4	2
3	1	4	7	0	4	2
4	4	7	10	3	0	5
5	6	2	12	5	2	0

$\pi(i,j)$

	0	1	2	3	4	5
0	×	**5**	**0**	**4**	**5**	**2**
1	3	×	0	4	1	3
2	3	5	×	4	5	2
3	3	5	0	×	5	3
4	3	5	0	4	×	3
5	3	5	0	4	5	×

L'approche de Bellman et Ford

L'algorithme de Bellman et Ford définit $d_k(i,j)$ comme le coût minimal d'un chemin de i à j qui emprunte au plus k arêtes.

Lorsque $k = n - 1$, on obtient $d(i,j)$.

En effet, en l'absence de cycles de coût négatif, il est inutile pour un chemin optimal d'emprunter plus de $n - 1$ arêtes.

Mise en équations : cas de base

Un chemin qui emprunte au plus 0 arête ne peut mener que de i à i, et a un coût nul.

Donc,

$$\begin{aligned} d_0(i,j) &= 0 && \text{si } i = j \\ d_0(i,j) &= \infty && \text{sinon} \end{aligned}$$

Mise en équations : cas inductif

Un chemin optimal de i à j qui emprunte au plus $k+1$ arêtes soit emprunte au plus k arêtes, soit se compose d'un chemin optimal de k arêtes de i à un certain sommet x et d'une dernière arête de x à j.

Donc,

$$d_{k+1}(i,j) = \min_{0 \leq x < n} \left(\begin{array}{l} d_k(i,j) \\ d_k(i,x) + G(x,j) \end{array} \right)$$

Source fixée

On s'aperçoit que, dans ces équations, i ne varie pas.

On peut donc fixer une source s et rechercher la distance $d_k(j)$ de s vers tous les sommets j.

$$
\begin{array}{ll}
d_0(j) = 0 & \text{si } s = j \\
d_0(j) = \infty & \text{sinon} \\
d_{k+1}(j) = \min_{0 \leq x < n} \left(\begin{array}{l} d_k(j) \\ d_k(x) + G(x, j) \end{array} \right) &
\end{array}
$$

On peut calculer les $d_k(j)$ pour k croissant de 0 à $n - 1$.

On peut également calculer les $\pi_k(j)$.

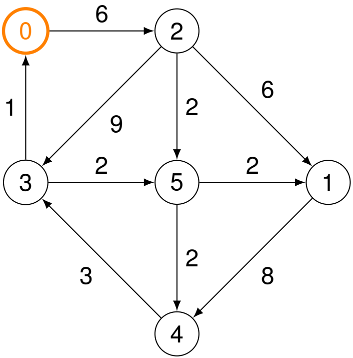
L'algorithme de Bellman et Ford

À nouveau, on modifie le tableau `d` en place. (Qu'en pensez-vous ?)

```
public static int[] bf (int n, Edge[] edges, int source)
{
  int[] d = new int [n] ;          // Initialization.
  for (int j = 0 ; j < n ; j++)
    d[j] = Integer.MAX_VALUE ;
  d[source] = 0 ;
  for (int k = 1 ; k < n ; k++)    // Iteration.
    for (Edge e  : edges)
      d[e.dst] = Math.min(
        d[e.dst],
        add(d[e.src], e.weight) // addition in ℤ ∪ {∞}
      ) ;
  return d ;
}
```

On remplace les boucles sur x et j par une itération sur les arêtes $x \to j$.

Initialisation : $k = 0$



Les plus courts chemins à partir du sommet 0 sont affichés en **orange**.

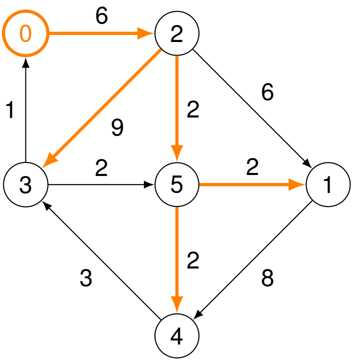
$d(j)$

0	1	2	3	4	5
0	∞	∞	∞	∞	∞

$\pi(j)$

0	1	2	3	4	5
$\times$	$\times$	$\times$	$\times$	$\times$	$\times$

Après l'itération $k = 1$



Les plus courts chemins à partir du sommet 0 sont affichés en **orange**.

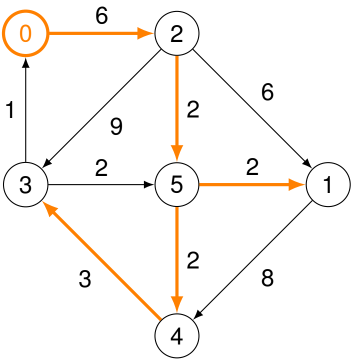
$d(j)$

0	1	2	3	4	5
0	10	6	15	10	8

$\pi(j)$

0	1	2	3	4	5
×	5	0	2	5	2

Après l'itération $k = 2$



$d(j)$

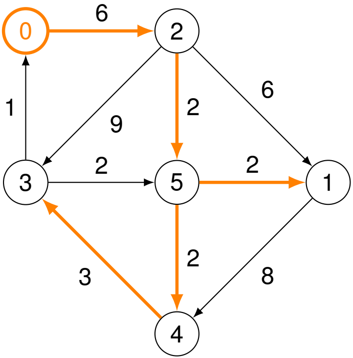
0	1	2	3	4	5
0	10	6	13	10	8

$\pi(j)$

0	1	2	3	4	5
×	**5**	**0**	**4**	**5**	**2**

Les plus courts chemins à partir du sommet 0 sont affichés en **orange**.

Après l'itération $k = 3$



Les plus courts chemins à partir du sommet 0 sont affichés en **orange**.

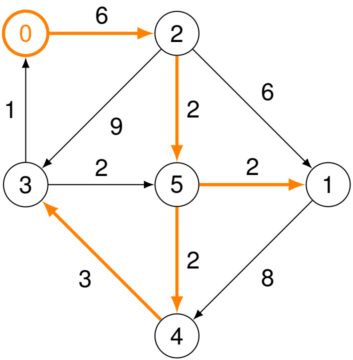
$d(j)$

0	1	2	3	4	5
0	10	6	13	10	8

$\pi(j)$

0	1	2	3	4	5
×	**5**	**0**	**4**	**5**	**2**

Après l'itération $k = 4$



$d(j)$

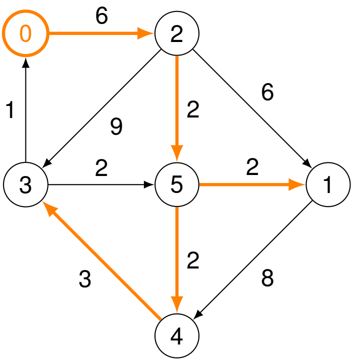
0	1	2	3	4	5
0	10	6	13	10	8

$\pi(j)$

0	1	2	3	4	5
×	**5**	**0**	**4**	**5**	**2**

Les plus courts chemins à partir du sommet 0 sont affichés en **orange**.

Après l'itération $k = 5$



Les plus courts chemins à partir du sommet 0 sont affichés en **orange**.

$d(j)$

0	1	2	3	4	5
0	10	6	13	10	8

$\pi(j)$

0	1	2	3	4	5
×	**5**	**0**	**4**	**5**	**2**

Convergence

Employer un seul tableau `d` accélère la convergence de l'algorithme.

À l'étape k, il a découvert tous les chemins formés d'au plus k arêtes, et certains chemins plus longs.

Le comportement de l'algorithme, et le nombre d'itérations nécessaires, dépendent alors de l'ordre d'examen des arêtes.

À l'extrême, le calcul pourrait être terminé après l'itération $k = 1$!

On peut modifier le code pour détecter que la convergence a eu lieu.

Représentation du graphe

Écrire une boucle sur les arêtes, plutôt que deux boucles imbriquées sur les sommets, permet d'obtenir une complexité en temps $O(nm)$, au lieu de $O(n^3)$.

Cela suppose que le graphe soit représenté par des listes d'adjacence, plutôt que par une matrice d'adjacence.

La complexité en espace est $O(n)$.

Comparaison avec Dijkstra

L'algorithme de Bellman et Ford est proche de celui de Dijkstra. On y retrouve la notion de relaxation : $d(j) \leftarrow \min(d(j), d(x) + G(x, j))$.

Dijkstra, glouton, utilise une file de priorités pour traiter les arêtes « dans le bon ordre » et ne relaxer qu'une fois chaque arête. Il ne tolère pas les coûts négatifs.

Bellman-Ford traite les arêtes dans un ordre arbitraire. Il tolère les coûts négatifs. Pour ces deux raisons, plusieurs itérations peuvent être nécessaires.

La programmation dynamique

Une technique de découpage en sous-problèmes reliés par des équations, avec mémorisation des solutions aux sous-problèmes, et (éventuellement) prédétermination de l'ordre dans lequel examiner les sous-problèmes.

C'est une technique puissante.

Elle permet aussi de mieux comprendre comment certains algorithmes ont pu être conçus (Floyd-W., Bellman-F., etc.).