Programmation dynamique

Informatique tronc commun

Sylvain Pelletier

PSI - LMSC

- La programmation dynamique est une technique permettant de résoudre certaines classes de problèmes de calcul de minimisation.

- Un problème de minimisation consiste à trouver :

$$\min_{x \in E} f(x)$$

avec E un ensemble « complexe » de grand cardinal et f une fonction.

- On ne peut pas balayer simplement les éléments de E.

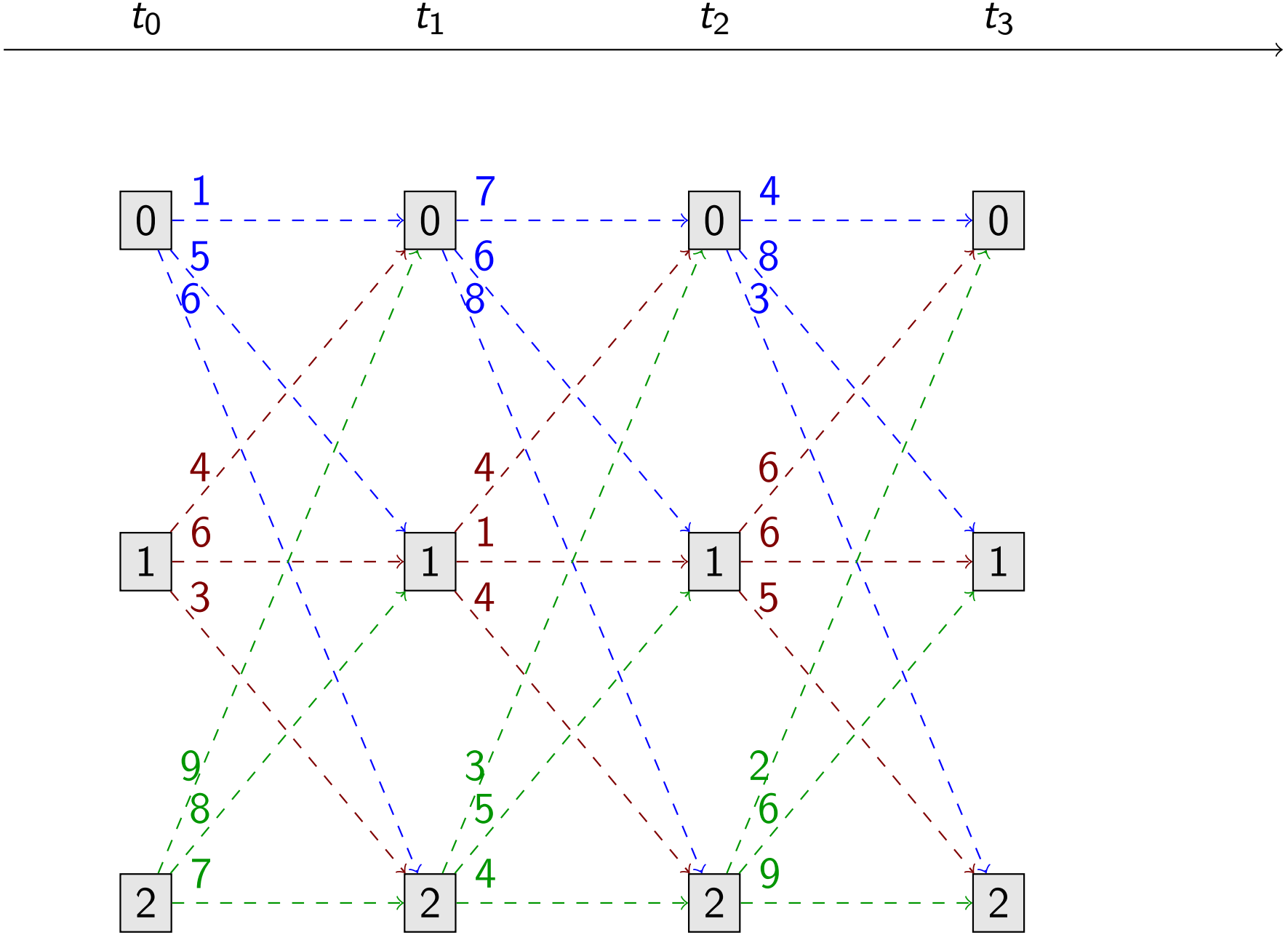
- On va présenter deux exemples où la programmation dynamique s'applique, en essayant de dégager une structure générale.

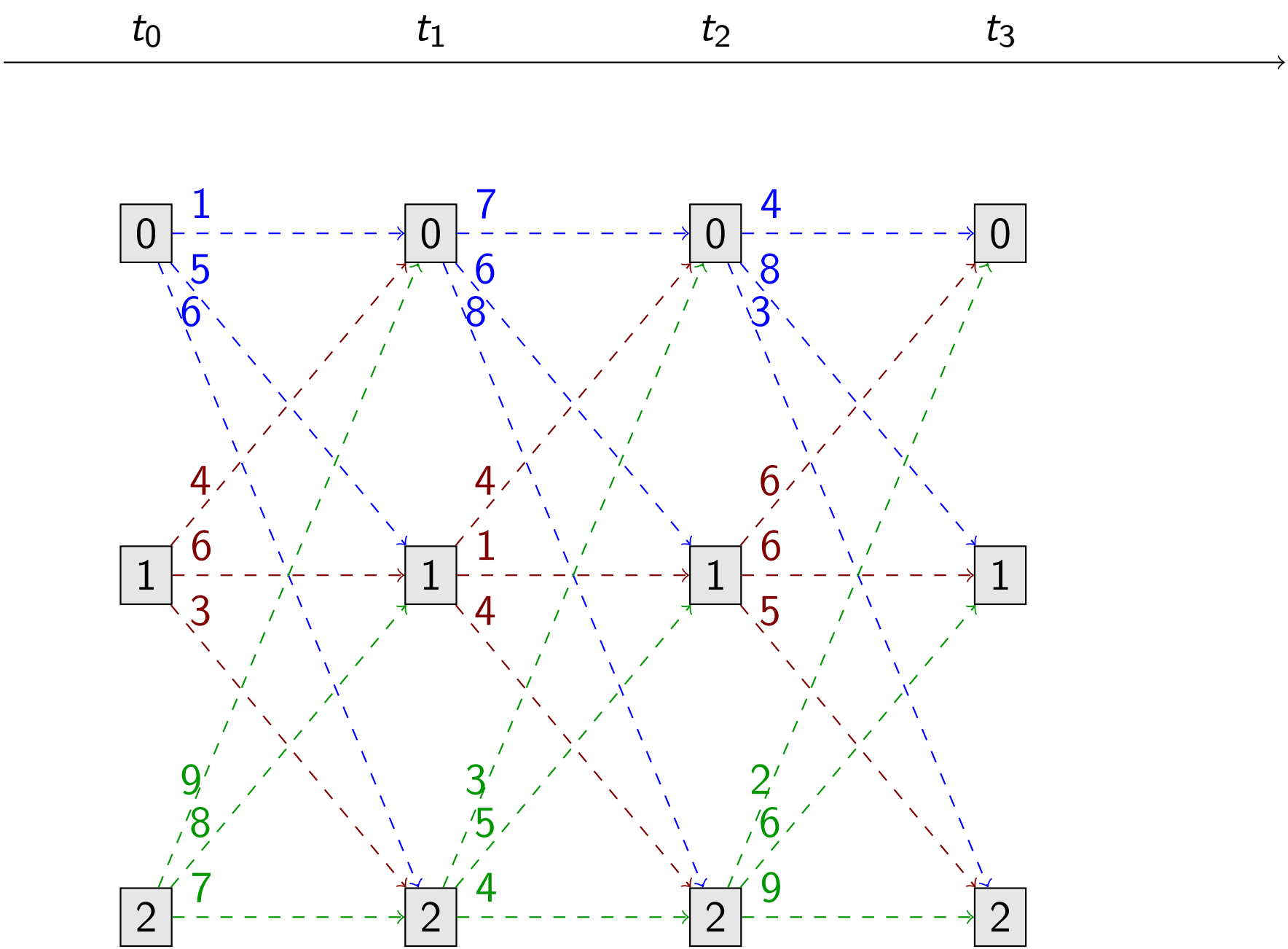
Description du problème

- On imagine un système qui peut être dans p états possibles = un entier de $[\![0, p-1]\!]$.
- On étudie ce système au temps $t_0, \ldots, t_N$. Entre deux temps consécutifs, le système peut changer d'état. On a donc N transitions.
- Le coût de transition dépend du temps et de l'état de départ et d'arrivée.
  On note $f(k, x, y)$ le coût pour passer de l'état x à l'état y entre le temps t_k et le temps t_{k+1}.
- Il est possible que $f(k, x, x) > 0$, autrement dit que rester dans le même état x ait un coût.

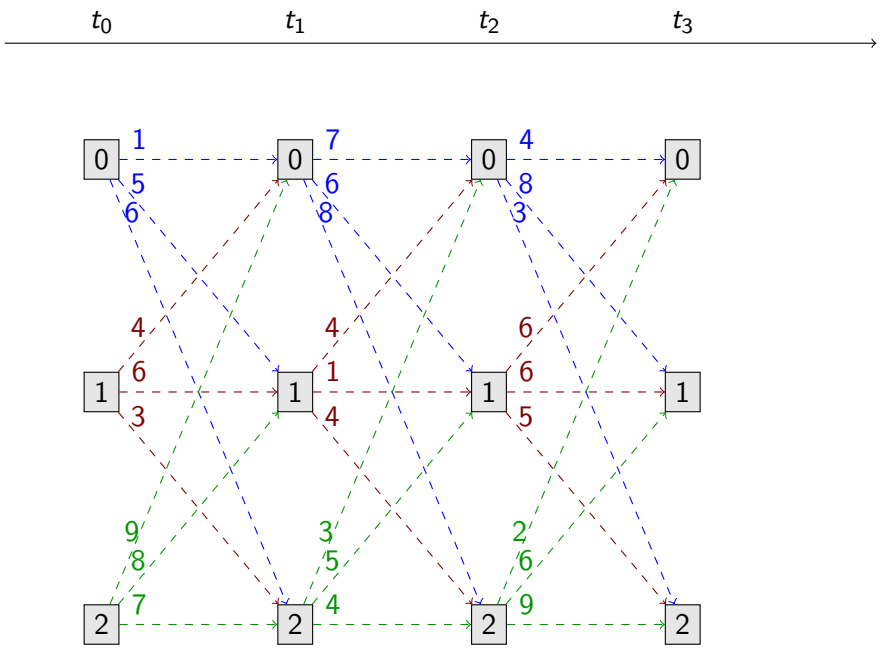
Le problème de minimisation consiste à trouver une suite d'états $x_0, x_1, \ldots, x_N$ qui minimise le coût total :

$$C(x_0, \ldots, x_N) = \sum_{i=0}^{N-1} f(i, x_i, x_{i+1})$$


t_0
t_1
t_2
t_3
0
1
2
1
7
4
5
6
8
3
4
6
1
6
3
4
5
9
8
7
3
5
4
2
6
9



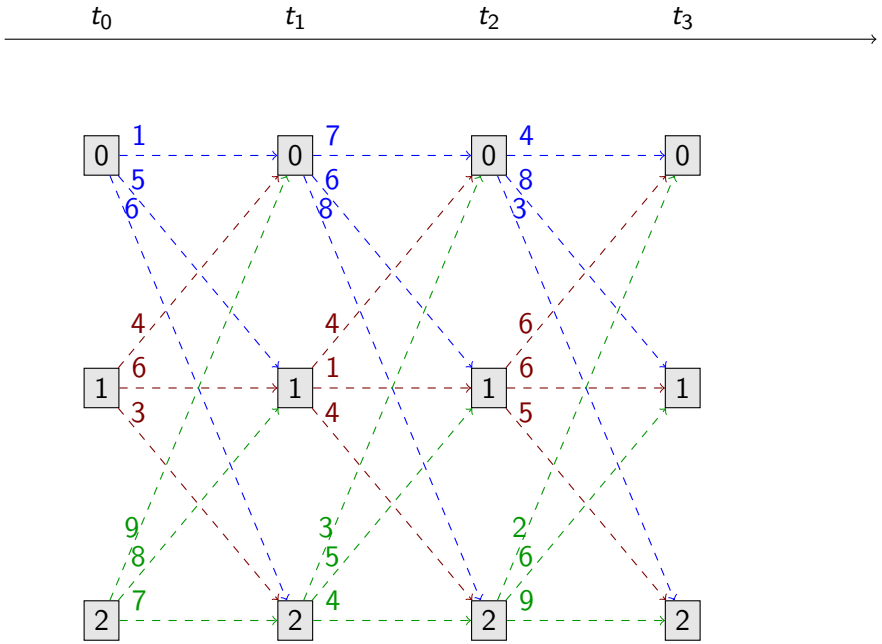
On voit deux chemins minimaux de coût 9 : (1,2,2,0) et (1,2,0,2).



- À chaque transitions on a p^2 flèches , donc Np^2 données.
- On cherche le coût minimal et un chemin de coût minimal dans ce graphe, c'est-à-dire :

$$\min_{(x_0,\ldots,x_n)\in[\![0,p-1]\!]^{N+1}} C(x_0,\ldots,x_n)$$

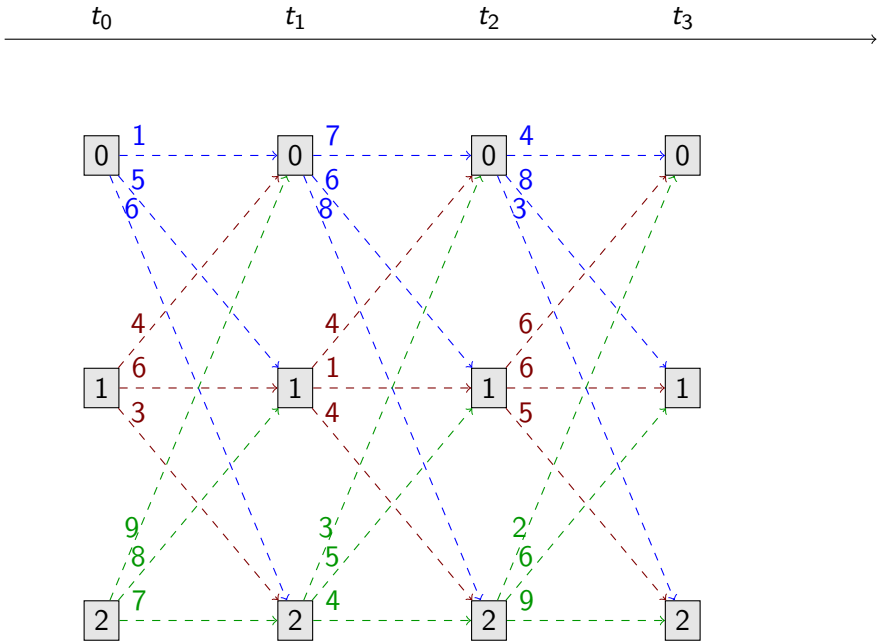
- Il y a p^{N+1} chemins $(x_0,\ldots,x_n)$ possibles, c'est fini mais trop gros pour du temps de calcul raisonnable.
- On cherche une structure de sous-problème optimal pour permettre un calcul plus rapide.



- À chaque transitions on a p^2 flèches , donc Np^2 données.
- On cherche le coût minimal et un chemin de coût minimal dans ce graphe, c'est-à-dire :

$$\min_{(x_0,\dots,x_n)\in[\![0,p-1]\!]^{N+1}} C(x_0,\dots,x_n)$$

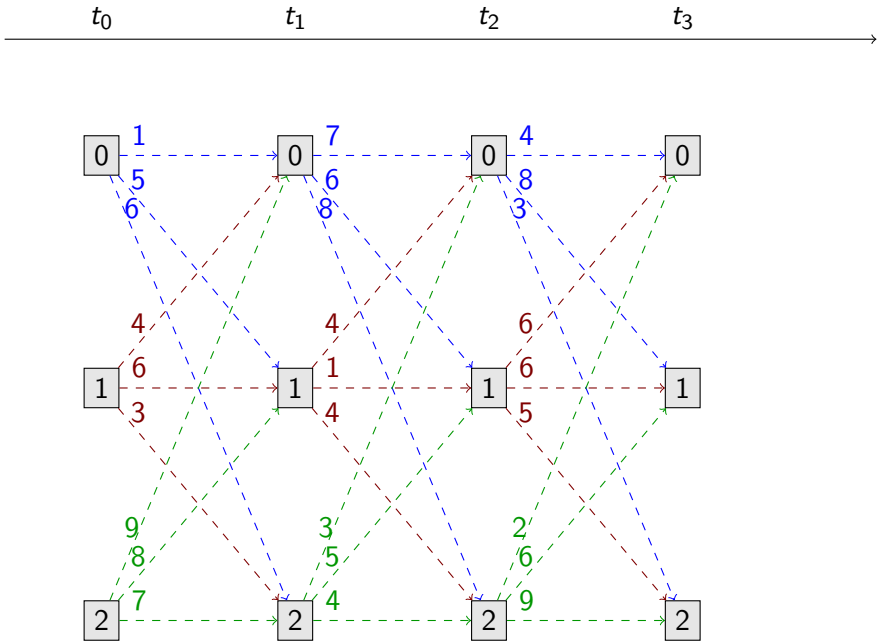
- Il y a p^{N+1} chemins $(x_0,\dots,x_n)$ possibles, c'est fini mais trop gros pour du temps de calcul raisonnable.
- On cherche une structure de sous-problème optimal pour permettre un calcul plus rapide.



- À chaque transitions on a p^2 flèches , donc Np^2 données.
- On cherche le coût minimal et un chemin de coût minimal dans ce graphe, c'est-à-dire :

$$\min_{(x_0,\ldots,x_n)\in[\![0,p-1]\!]^{N+1}} C(x_0,\ldots,x_n)$$

- Il y a p^{N+1} chemins $(x_0,\ldots,x_n)$ possibles, c'est fini mais trop gros pour du temps de calcul raisonnable.
- On cherche une structure de sous-problème optimal pour permettre un calcul plus rapide.



- À chaque transitions on a p^2 flèches , donc Np^2 données.
- On cherche le coût minimal et un chemin de coût minimal dans ce graphe, c'est-à-dire :

$$\min_{(x_0,\ldots,x_n)\in[\![0,p-1]\!]^{N+1}} C(x_0,\ldots,x_n)$$

- Il y a p^{N+1} chemins $(x_0,\ldots,x_n)$ possibles, c'est fini mais trop gros pour du temps de calcul raisonnable.
- On cherche une structure de sous-problème optimal pour permettre un calcul plus rapide.

propriété de sous-problème optimal

- Considérons pour $k \in [\![0, N]\!]$ et $x \in [\![0, p-1]\!]$:

$$\begin{aligned} c_k(x) &= \min_{x_0,\dots,x_{k-1} \in [\![0,p-1]\!]^k} \left(\sum_{i=0}^{k-2} f(i, x_i, x_{i+1}) + f(k-1, x_{k-1}, x) \right) \\ &= \min_{x_0,\dots,x_{k-1} \in [\![0,p-1]\!]^k} C(x_0, \dots, x_{k-1}, x) \end{aligned}$$

- C'est le coût du chemin minimal pour arriver au temps k à l'état x.
- Le coût cherché est $\min_{x \in [\![0,p-1]\!]} c_N(x)$.
- C'est le coût minimal des chemins minimaux de longueur N. Autrement dit, on regarde pour tous les états d'arrivée, lequel est atteint avec un chemin minimal le moins coûteux.

propriété de sous-problème optimal

- Considérons pour $k \in [\![0, N]\!]$ et $x \in [\![0, p-1]\!]$:

$$\begin{aligned} c_k(x) &= \min_{x_0,\dots,x_{k-1} \in [\![0,p-1]\!]^k} \left(\sum_{i=0}^{k-2} f(i, x_i, x_{i+1}) + f(k-1, x_{k-1}, x) \right) \\ &= \min_{x_0,\dots,x_{k-1} \in [\![0,p-1]\!]^k} C(x_0, \dots, x_{k-1}, x) \end{aligned}$$

- C'est le coût du chemin minimal pour arriver au temps k à l'état x.
- Le coût cherché est $\min_{x \in [\![0,p-1]\!]} c_N(x)$.
- C'est le coût minimal des chemins minimaux de longueur N. Autrement dit, on regarde pour tous les états d'arrivée, lequel est atteint avec un chemin minimal le moins coûteux.

propriété de sous-problème optimal

- Considérons pour $k \in [\![0, N]\!]$ et $x \in [\![0, p-1]\!]$:

$$\begin{aligned} c_k(x) &= \min_{x_0,\dots,x_{k-1} \in [\![0,p-1]\!]^k} \left(\sum_{i=0}^{k-2} f(i, x_i, x_{i+1}) + f(k-1, x_{k-1}, x) \right) \\ &= \min_{x_0,\dots,x_{k-1} \in [\![0,p-1]\!]^k} C(x_0, \dots, x_{k-1}, x) \end{aligned}$$

- C'est le coût du chemin minimal pour arriver au temps k à l'état x.
- Le coût cherché est $\min_{x \in [\![0,p-1]\!]} c_N(x)$.
- C'est le coût minimal des chemins minimaux de longueur N. Autrement dit, on regarde pour tous les états d'arrivée, lequel est atteint avec un chemin minimal le moins coûteux.

propriété de sous-problème optimal

- Considérons pour $k \in [\![0, N]\!]$ et $x \in [\![0, p-1]\!]$:

$$\begin{aligned} c_k(x) &= \min_{x_0,\dots,x_{k-1} \in [\![0,p-1]\!]^k} \left(\sum_{i=0}^{k-2} f(i, x_i, x_{i+1}) + f(k-1, x_{k-1}, x) \right) \\ &= \min_{x_0,\dots,x_{k-1} \in [\![0,p-1]\!]^k} C(x_0, \dots, x_{k-1}, x) \end{aligned}$$

- C'est le coût du chemin minimal pour arriver au temps k à l'état x.
- Le coût cherché est $\min_{x \in [\![0,p-1]\!]} c_N(x)$.
- C'est le coût minimal des chemins minimaux de longueur N. Autrement dit, on regarde pour tous les états d'arrivée, lequel est atteint avec un chemin minimal le moins coûteux.

Équation de Bellman

$c_k(x)$ est le coût du chemin minimal pour arriver au temps k à l'état x.
Donc :

- $c_0(x) = 0$ pour tout x (c'est le coût pour partir du point x).
- On a une relation dite équation de Bellman :

$$\forall k \in [\![1, N]\!], \forall x \in [\![0, p-1]\!], \quad c_k(x) = \min_{y \in [\![0,p-1]\!]} (c_{k-1}(y) + f(k-1, y, x))$$

- La programmation dynamique consiste à exploiter ce type de relation de sous-problèmes optimaux.
- On voir que l'on peut calculer les $c_k(x)$ par une boucle `for` en `k` ou par récursivité.
- Au concours : il faut être capable de justifier cette relation et d'écrire le programme correspondant.

Équation de Bellman

$c_k(x)$ est le coût du chemin minimal pour arriver au temps k à l'état x.
Donc :

- $c_0(x) = 0$ pour tout x (c'est le coût pour partir du point x).
- On a une relation dite équation de Bellman :

$$\forall k \in [\![1, N]\!], \forall x \in [\![0, p-1]\!], \quad c_k(x) = \min_{y \in [\![0,p-1]\!]} (c_{k-1}(y) + f(k-1, y, x))$$

- La programmation dynamique consiste à exploiter ce type de relation de sous-problèmes optimaux.
- On voir que l'on peut calculer les $c_k(x)$ par une boucle `for` en `k` ou par récursivité.
- Au concours : il faut être capable de justifier cette relation et d'écrire le programme correspondant.

Équation de Bellman

$c_k(x)$ est le coût du chemin minimal pour arriver au temps k à l'état x.
Donc :

- $c_0(x) = 0$ pour tout x (c'est le coût pour partir du point x).
- On a une relation dite équation de Bellman :

$$\forall k \in [\![1, N]\!], \forall x \in [\![0, p-1]\!], \quad c_k(x) = \min_{y \in [\![0, p-1]\!]} (c_{k-1}(y) + f(k-1, y, x))$$

- La programmation dynamique consiste à exploiter ce type de relation de sous-problèmes optimaux.
- On voir que l'on peut calculer les $c_k(x)$ par une boucle `for` en `k` ou par récursivité.
- Au concours : il faut être capable de justifier cette relation et d'écrire le programme correspondant.

Équation de Bellman

$c_k(x)$ est le coût du chemin minimal pour arriver au temps k à l'état x.
Donc :

- $c_0(x) = 0$ pour tout x (c'est le coût pour partir du point x).
- On a une relation dite équation de Bellman :

$$\forall k \in [\![1, N]\!], \forall x \in [\![0, p-1]\!], \quad c_k(x) = \min_{y \in [\![0,p-1]\!]} (c_{k-1}(y) + f(k-1, y, x))$$

- La programmation dynamique consiste à exploiter ce type de relation de sous-problèmes optimaux.
- On voir que l'on peut calculer les $c_k(x)$ par une boucle `for` en `k` ou par récursivité.
- Au concours : il faut être capable de justifier cette relation et d'écrire le programme correspondant.

Équation de Bellman

$c_k(x)$ est le coût du chemin minimal pour arriver au temps k à l'état x.
Donc :

- $c_0(x) = 0$ pour tout x (c'est le coût pour partir du point x).
- On a une relation dite équation de Bellman :

$$\forall k \in [\![1, N]\!], \forall x \in [\![0, p-1]\!], \quad c_k(x) = \min_{y \in [\![0,p-1]\!]} (c_{k-1}(y) + f(k-1, y, x))$$

- La programmation dynamique consiste à exploiter ce type de relation de sous-problèmes optimaux.
- On voir que l'on peut calculer les $c_k(x)$ par une boucle `for` en `k` ou par récursivité.
- Au concours : il faut être capable de justifier cette relation et d'écrire le programme correspondant.

Justification de l'équation de Bellman

$c_k(x)$ est le coût du chemin minimal pour arriver au temps k à l'état x.
Montrons :

$$c_k(x) = \min_{y \in [\![0, p-1]\!]} (c_{k-1}(y) + f(k-1, y, x))$$

- Il faut justifier :

$$\forall y \in [\![0, p-1]\!], \ c_k(x) \leqslant c_{k-1}(y) + f(k-1, y, x)$$
$$\text{et } \exists y \in [\![0, p-1]\!] \ c_k(x) = c_{k-1}(y) + f(k-1, y, x)$$

- La première ligne signifie : quelque soit l'état y du système au temps $k-1$ (prédécesseur de x), le coût minimal pour arriver à x au temps k est inférieur ou égal au coût pour arriver à y suivi du coût de la transition de y vers x.
- Le deuxième ligne signifie : le chemin minimal pour arriver à x était déjà optimal pour arriver à y.
- Non seulement on obtient ainsi la valeur de $c_k(x)$ mais aussi la valeur d'un meilleur prédécesseur de x au temps $k-1$, c'est l'un des y qui réalise le minimum.

Justification de l'équation de Bellman

$c_k(x)$ est le coût du chemin minimal pour arriver au temps k à l'état x.
Montrons :

$$c_k(x) = \min_{y \in [\![0, p-1]\!]} (c_{k-1}(y) + f(k-1, y, x))$$

- Il faut justifier :

$$\forall y \in [\![0, p-1]\!], \ c_k(x) \leqslant c_{k-1}(y) + f(k-1, y, x)$$
$$\text{et } \exists y \in [\![0, p-1]\!] \, c_k(x) = c_{k-1}(y) + f(k-1, y, x)$$

- La première ligne signifie : quelque soit l'état y du système au temps $k-1$ (prédécesseur de x), le coût minimal pour arriver à x au temps k est inférieur ou égal au coût pour arriver à y suivi du coût de la transition de y vers x.
- Le deuxième ligne signifie : le chemin minimal pour arriver à x était déjà optimal pour arriver à y.
- Non seulement on obtient ainsi la valeur de $c_k(x)$ mais aussi la valeur d'un meilleur prédécesseur de x au temps $k-1$, c'est l'un des y qui réalise le minimum.

Justification de l'équation de Bellman

$c_k(x)$ est le coût du chemin minimal pour arriver au temps k à l'état x.
Montrons :

$$c_k(x) = \min_{y \in [\![0, p-1]\!]} (c_{k-1}(y) + f(k-1, y, x))$$

- Il faut justifier :

$$\forall y \in [\![0, p-1]\!], \ c_k(x) \leqslant c_{k-1}(y) + f(k-1, y, x)$$
$$\text{et } \exists y \in [\![0, p-1]\!] \, c_k(x) = c_{k-1}(y) + f(k-1, y, x)$$

- La première ligne signifie : quelque soit l'état y du système au temps $k-1$ (prédécesseur de x), le coût minimal pour arriver à x au temps k est inférieur ou égal au coût pour arriver à y suivi du coût de la transition de y vers x.
- Le deuxième ligne signifie : le chemin minimal pour arriver à x était déjà optimal pour arriver à y.
- Non seulement on obtient ainsi la valeur de $c_k(x)$ mais aussi la valeur d'un meilleur prédécesseur de x au temps $k-1$, c'est l'un des y qui réalise le minimum.

Justification de l'équation de Bellman

$c_k(x)$ est le coût du chemin minimal pour arriver au temps k à l'état x.
Montrons :

$$c_k(x) = \min_{y \in [\![0, p-1]\!]} (c_{k-1}(y) + f(k-1, y, x))$$

- Il faut justifier :

$$\forall y \in [\![0, p-1]\!], \ c_k(x) \leqslant c_{k-1}(y) + f(k-1, y, x)$$
$$\text{et } \exists y \in [\![0, p-1]\!] \, c_k(x) = c_{k-1}(y) + f(k-1, y, x)$$

- La première ligne signifie : quelque soit l'état y du système au temps $k-1$ (prédécesseur de x), le coût minimal pour arriver à x au temps k est inférieur ou égal au coût pour arriver à y suivi du coût de la transition de y vers x.
- Le deuxième ligne signifie : le chemin minimal pour arriver à x était déjà optimal pour arriver à y.
- Non seulement on obtient ainsi la valeur de $c_k(x)$ mais aussi la valeur d'un meilleur prédécesseur de x au temps $k-1$, c'est l'un des y qui réalise le minimum.

Justification de l'équation de Bellman 1/2

$c_k(x)$ est le coût du chemin minimal pour arriver au temps k à l'état x.
Montrons :

$$\forall y \in [\![0, p-1]\!],\ c_k(x) \leqslant c_{k-1}(y) + f(k-1, y, x)$$

- Soit $y \in [\![0, p-1]\!]$, alors $c_{k-1}(y)$ est le coût d'un chemin minimal $(x_0, \ldots, x_{k-2}, y)$ qui arrive au temps $k-1$ en y.
- Le chemin $(x_0, \ldots, x_{k-2}, y, x)$ arrive donc au temps k en x.
- Son coût est :

$$\begin{aligned}
&\sum_{i=1}^{k-3} f(i, x_i, x_{i+1}) + f(k-2, x_{k-2}, y) + f(k-1, y, x) \\
=&C(x_0, \ldots, x_{k-2}, y) + f(k-1, y, x) \\
=&c_{k-1}(y) + f(k-1, y, x)
\end{aligned}$$

- ce coût est inférieur au minimal donc $c_k(x) \leqslant c_{k-1}(y) + f(k-1, y, x)$

Justification de l'équation de Bellman 1/2

$c_k(x)$ est le coût du chemin minimal pour arriver au temps k à l'état x.
Montrons :

$$\forall y \in [\![0, p-1]\!],\ c_k(x) \leqslant c_{k-1}(y) + f(k-1, y, x)$$

- Soit $y \in [\![0, p-1]\!]$, alors $c_{k-1}(y)$ est le coût d'un chemin minimal $(x_0, \ldots, x_{k-2}, y)$ qui arrive au temps $k-1$ en y.
- Le chemin $(x_0, \ldots, x_{k-2}, y, x)$ arrive donc au temps k en x.
- Son coût est :

$$\begin{aligned}
&\sum_{i=1}^{k-3} f(i, x_i, x_{i+1}) + f(k-2, x_{k-2}, y) + f(k-1, y, x) \\
=&C(x_0, \ldots, x_{k-2}, y) + f(k-1, y, x) \\
=&c_{k-1}(y) + f(k-1, y, x)
\end{aligned}$$

- ce coût est inférieur au minimal donc $c_k(x) \leqslant c_{k-1}(y) + f(k-1, y, x)$

Justification de l'équation de Bellman 1/2

$c_k(x)$ est le coût du chemin minimal pour arriver au temps k à l'état x.
Montrons :

$$\forall y \in [\![0, p-1]\!],\ c_k(x) \leqslant c_{k-1}(y) + f(k-1, y, x)$$

- Soit $y \in [\![0, p-1]\!]$, alors $c_{k-1}(y)$ est le coût d'un chemin minimal $(x_0, \ldots, x_{k-2}, y)$ qui arrive au temps $k-1$ en y.
- Le chemin $(x_0, \ldots, x_{k-2}, y, x)$ arrive donc au temps k en x.
- Son coût est :

$$\begin{aligned}
&\sum_{i=1}^{k-3} f(i, x_i, x_{i+1}) + f(k-2, x_{k-2}, y) + f(k-1, y, x) \\
=&C(x_0, \ldots, x_{k-2}, y) + f(k-1, y, x) \\
=&c_{k-1}(y) + f(k-1, y, x)
\end{aligned}$$

- ce coût est inférieur au minimal donc $c_k(x) \leqslant c_{k-1}(y) + f(k-1, y, x)$

Justification de l'équation de Bellman 1/2

$c_k(x)$ est le coût du chemin minimal pour arriver au temps k à l'état x.
Montrons :

$$\forall y \in [\![0, p-1]\!],\ c_k(x) \leqslant c_{k-1}(y) + f(k-1, y, x)$$

- Soit $y \in [\![0, p-1]\!]$, alors $c_{k-1}(y)$ est le coût d'un chemin minimal $(x_0, \ldots, x_{k-2}, y)$ qui arrive au temps $k-1$ en y.
- Le chemin $(x_0, \ldots, x_{k-2}, y, x)$ arrive donc au temps k en x.
- Son coût est :

$$\begin{aligned}
&\sum_{i=1}^{k-3} f(i, x_i, x_{i+1}) + f(k-2, x_{k-2}, y) + f(k-1, y, x) \\
=&C(x_0, \ldots, x_{k-2}, y) + f(k-1, y, x) \\
=&c_{k-1}(y) + f(k-1, y, x)
\end{aligned}$$

- ce coût est inférieur au minimal donc $c_k(x) \leqslant c_{k-1}(y) + f(k-1, y, x)$

Justification de l'équation de Bellman 2/2

$c_k(x)$ est le coût du chemin minimal pour arriver au temps k à l'état x.
Montrons :

$$\exists y \in [\![0, p-1]\!]\, c_k(x) = c_{k-1}(y) + f(k-1, y, x)$$

- considérons un chemin minimal qui arrive à x au temps k et on pose y le prédécesseur de x. Ce chemin minimal est donc noté $(x_0, \ldots, x_{k-2}, y, x)$.
- Le coût de chemin est donc :

$$\begin{aligned} c_k(x) &= \sum_{i=1}^{k-3} f(i, x_i, x_{i+1}) + f(k-2, x_{k-2}, y) + f(k-1, y, x) \\ &= C(x_0, \ldots, x_{k-2}, y) + f(k-1, y, x) \end{aligned}$$

- Il faut donc montrer que : $C(x_0, \ldots, x_{k-2}, y) = c_{k-1}(y)$, c'est-à-dire que $(x_0, \ldots, x_{k-2}, y)$ est un chemin minimal pour arriver en y au temps $k-1$.

Justification de l'équation de Bellman 2/2

$c_k(x)$ est le coût du chemin minimal pour arriver au temps k à l'état x.
Montrons :

$$\exists y \in [\![0, p-1]\!]\, c_k(x) = c_{k-1}(y) + f(k-1, y, x)$$

- considérons un chemin minimal qui arrive à x au temps k et on pose y le prédécesseur de x. Ce chemin minimal est donc noté $(x_0, \ldots, x_{k-2}, y, x)$.
- Le coût de chemin est donc :

$$\begin{aligned} c_k(x) &= \sum_{i=1}^{k-3} f(i, x_i, x_{i+1}) + f(k-2, x_{k-2}, y) + f(k-1, y, x) \\ &= C(x_0, \ldots, x_{k-2}, y) + f(k-1, y, x) \end{aligned}$$

- Il faut donc montrer que : $C(x_0, \ldots, x_{k-2}, y) = c_{k-1}(y)$, c'est-à-dire que $(x_0, \ldots, x_{k-2}, y)$ est un chemin minimal pour arriver en y au temps $k-1$.

Justification de l'équation de Bellman 2/2

$c_k(x)$ est le coût du chemin minimal pour arriver au temps k à l'état x.
Montrons :

$$\exists y \in [\![0, p-1]\!]\, c_k(x) = c_{k-1}(y) + f(k-1, y, x)$$

- considérons un chemin minimal qui arrive à x au temps k et on pose y le prédécesseur de x. Ce chemin minimal est donc noté $(x_0, \ldots, x_{k-2}, y, x)$.
- Le coût de chemin est donc :

$$\begin{aligned} c_k(x) &= \sum_{i=1}^{k-3} f(i, x_i, x_{i+1}) + f(k-2, x_{k-2}, y) + f(k-1, y, x) \\ &= C(x_0, \ldots, x_{k-2}, y) + f(k-1, y, x) \end{aligned}$$

- Il faut donc montrer que : $C(x_0, \ldots, x_{k-2}, y) = c_{k-1}(y)$, c'est-à-dire que $(x_0, \ldots, x_{k-2}, y)$ est un chemin minimal pour arriver en y au temps $k-1$.

Justification de l'équation de Bellman 2/2

$c_k(x)$ est le coût du chemin minimal pour arriver au temps k à l'état x.

- Un chemin minimal est noté $(x_0, \ldots, x_{k-2}, y, x)$.
- Il faut donc montrer que : $C(x_0, \ldots, x_{k-2}, y) = c_{k-1}(y)$, c'est-à-dire que $(x_0, \ldots, x_{k-2}, y)$ est un chemin minimal pour arriver en y au temps $k-1$.
- Par l'absurde, supposons qu'il existe une chemin $(\tilde{x}_0, \ldots, \tilde{x}_{k-2}, y)$, de coût strictement inférieur. On ajoute alors x et on a un chemin pour arriver à x au temps k.
- Or :

$$\begin{aligned} C(\tilde{x}_0, \ldots, \tilde{x}_{k-2}, y, x) &= C((\tilde{x}_0, \ldots, \tilde{x}_{k-2}, y) + f(k-1, y, x) \\ &< C(x_0, \ldots, x_{k-2}, y) + f(k-1, y, x) = c_k(x) \end{aligned}$$

  impossible car $c_k(x)$ est le coût minimal.
- Ainsi : le chemin minimal pour arriver à x est aussi minimal pour arriver à son prédécesseur y et donc :

$$c_k(x) = c_{k-1}(y) + f(k-1, y, x)$$

Justification de l'équation de Bellman 2/2

$c_k(x)$ est le coût du chemin minimal pour arriver au temps k à l'état x.

- Un chemin minimal est noté $(x_0, \ldots, x_{k-2}, y, x)$.
- Il faut donc montrer que : $C(x_0, \ldots, x_{k-2}, y) = c_{k-1}(y)$, c'est-à-dire que $(x_0, \ldots, x_{k-2}, y)$ est un chemin minimal pour arriver en y au temps $k-1$.
- Par l'absurde, supposons qu'il existe une chemin $(\tilde{x}_0, \ldots, \tilde{x}_{k-2}, y)$, de coût strictement inférieur. On ajoute alors x et on a un chemin pour arriver à x au temps k.
- Or :

$$\begin{aligned} C(\tilde{x}_0, \ldots, \tilde{x}_{k-2}, y, x) &= C((\tilde{x}_0, \ldots, \tilde{x}_{k-2}, y) + f(k-1, y, x) \\ &< C(x_0, \ldots, x_{k-2}, y) + f(k-1, y, x) = c_k(x) \end{aligned}$$

  impossible car $c_k(x)$ est le coût minimal.
- Ainsi : le chemin minimal pour arriver à x est aussi minimal pour arriver à son prédécesseur y et donc :

$$c_k(x) = c_{k-1}(y) + f(k-1, y, x)$$

Justification de l'équation de Bellman 2/2

$c_k(x)$ est le coût du chemin minimal pour arriver au temps k à l'état x.

- Un chemin minimal est noté $(x_0, \ldots, x_{k-2}, y, x)$.
- Il faut donc montrer que : $C(x_0, \ldots, x_{k-2}, y) = c_{k-1}(y)$, c'est-à-dire que $(x_0, \ldots, x_{k-2}, y)$ est un chemin minimal pour arriver en y au temps $k-1$.
- Par l'absurde, supposons qu'il existe une chemin $(\tilde{x}_0, \ldots, \tilde{x}_{k-2}, y)$, de coût strictement inférieur. On ajoute alors x et on a un chemin pour arriver à x au temps k.
- Or :

$$\begin{aligned} C(\tilde{x}_0, \ldots, \tilde{x}_{k-2}, y, x) &= C((\tilde{x}_0, \ldots, \tilde{x}_{k-2}, y) + f(k-1, y, x) \\ &< C(x_0, \ldots, x_{k-2}, y) + f(k-1, y, x) = c_k(x) \end{aligned}$$

  impossible car $c_k(x)$ est le coût minimal.
- Ainsi : le chemin minimal pour arriver à x est aussi minimal pour arriver à son prédécesseur y et donc :

$$c_k(x) = c_{k-1}(y) + f(k-1, y, x)$$

Justification de l'équation de Bellman 2/2

$c_k(x)$ est le coût du chemin minimal pour arriver au temps k à l'état x.

- Un chemin minimal est noté $(x_0, \ldots, x_{k-2}, y, x)$.
- Il faut donc montrer que : $C(x_0, \ldots, x_{k-2}, y) = c_{k-1}(y)$, c'est-à-dire que $(x_0, \ldots, x_{k-2}, y)$ est un chemin minimal pour arriver en y au temps $k-1$.
- Par l'absurde, supposons qu'il existe une chemin $(\tilde{x}_0, \ldots, \tilde{x}_{k-2}, y)$, de coût strictement inférieur. On ajoute alors x et on a un chemin pour arriver à x au temps k.
- Or :

$$\begin{aligned} C\left(\tilde{x}_0, \ldots, \tilde{x}_{k-2}, y, x\right) &= C\left(\left(\tilde{x}_0, \ldots, \tilde{x}_{k-2}, y\right) + f(k-1, y, x)\right. \\ &< C(x_0, \ldots, x_{k-2}, y) + f(k-1, y, x) = c_k(x) \end{aligned}$$

  impossible car $c_k(x)$ est le coût minimal.
- Ainsi : le chemin minimal pour arriver à x est aussi minimal pour arriver à son prédécesseur y et donc :

$$c_k(x) = c_{k-1}(y) + f(k-1, y, x)$$

Justification de l'équation de Bellman 2/2

$c_k(x)$ est le coût du chemin minimal pour arriver au temps k à l'état x.

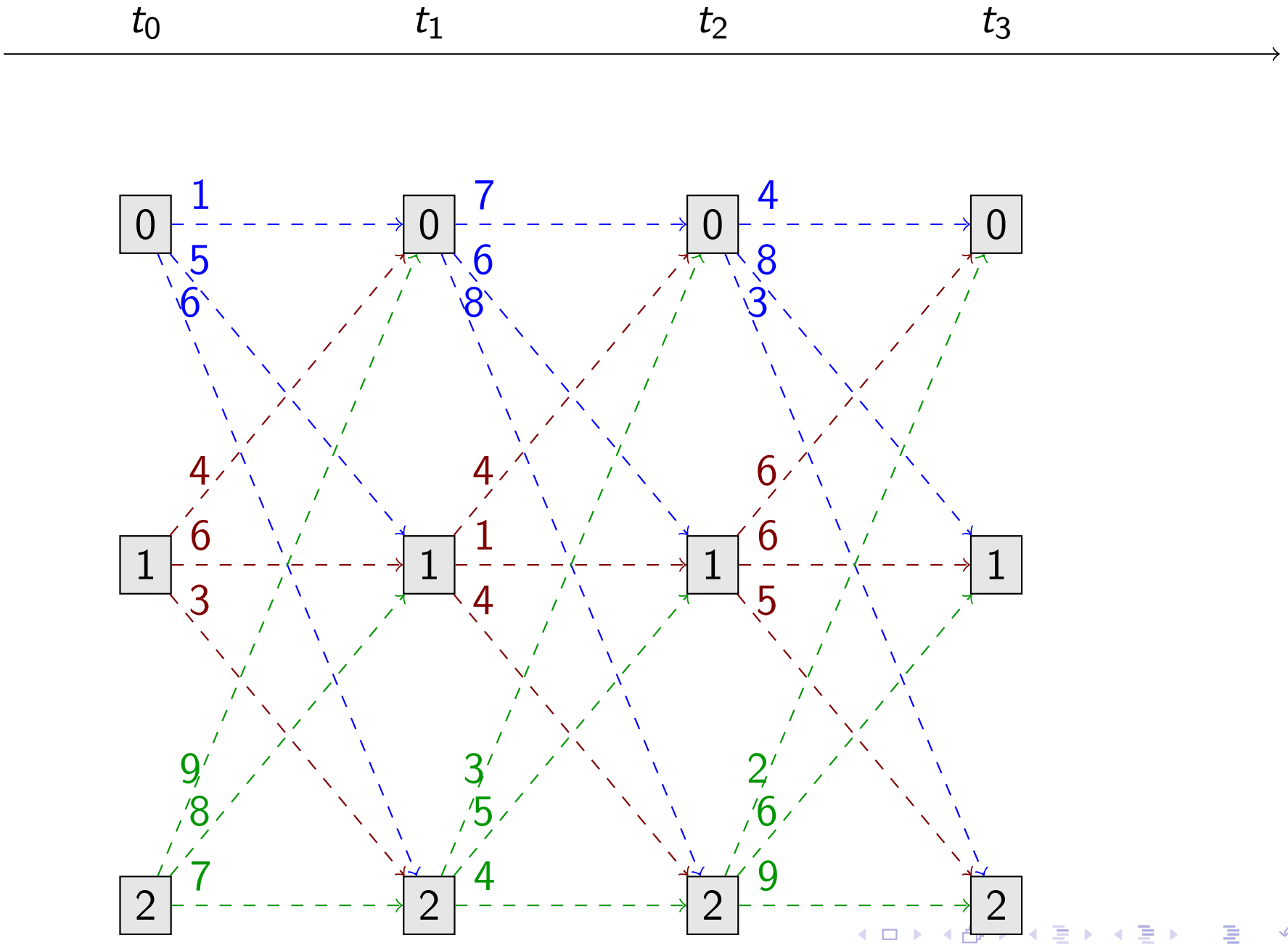
- Un chemin minimal est noté $(x_0, \ldots, x_{k-2}, y, x)$.
- Il faut donc montrer que : $C(x_0, \ldots, x_{k-2}, y) = c_{k-1}(y)$, c'est-à-dire que $(x_0, \ldots, x_{k-2}, y)$ est un chemin minimal pour arriver en y au temps $k-1$.
- Par l'absurde, supposons qu'il existe une chemin $(\tilde{x}_0, \ldots, \tilde{x}_{k-2}, y)$, de coût strictement inférieur. On ajoute alors x et on a un chemin pour arriver à x au temps k.
- Or :

$$\begin{aligned} C\left(\tilde{x}_0, \ldots, \tilde{x}_{k-2}, y, x\right) &= C\left(\left(\tilde{x}_0, \ldots, \tilde{x}_{k-2}, y\right) + f(k-1, y, x)\right. \\ &< C(x_0, \ldots, x_{k-2}, y) + f(k-1, y, x) = c_k(x) \end{aligned}$$

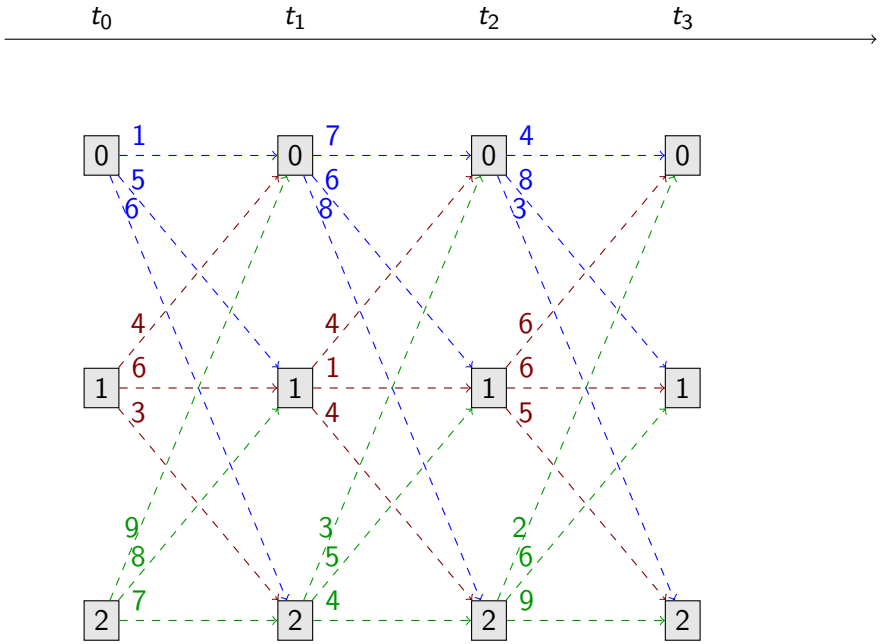
  impossible car $c_k(x)$ est le coût minimal.
- Ainsi : le chemin minimal pour arriver à x est aussi minimal pour arriver à son prédécesseur y et donc :

$$c_k(x) = c_{k-1}(y) + f(k-1, y, x)$$

$c_0(x) = 0$ et pour $k \geqslant 1$ $c_k(x) = \min_{y \in [\![0, p-1]\!]}(c_{k-1}(y) + f(k-1, y, x))$



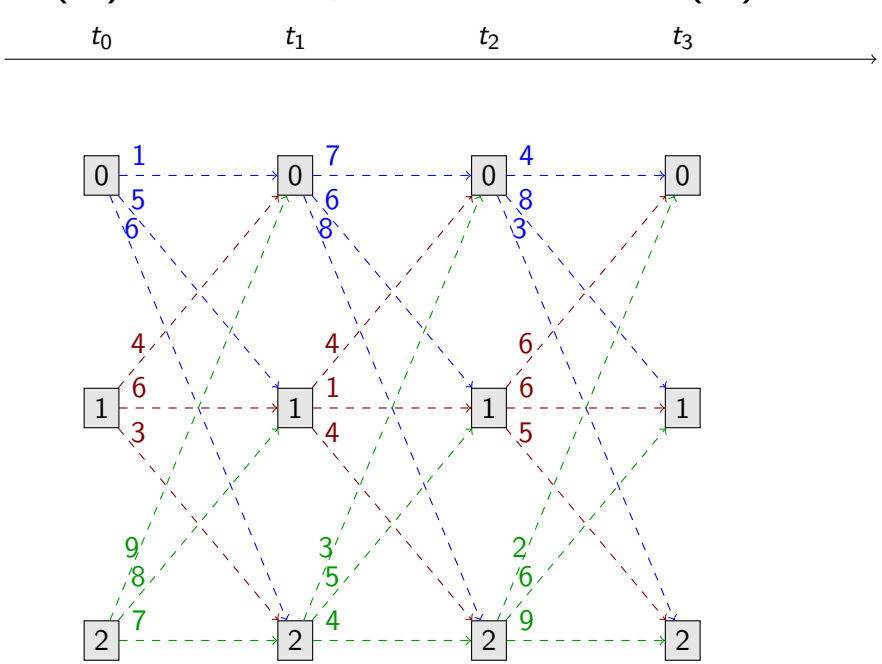
$c_0(x) = 0$ et pour $k \geqslant 1$ $c_k(x) = \min_{y \in [\![0, p-1]\!]}(c_{k-1}(y) + f(k-1, y, x))$



	t_0	t_1	t_2	t_3
état 0	$c_0(0) = 0$	$c_1(0) = 1$ (0)	$c_2(0) = 6$ (2)	$c_3(0) = 9$ (2)
état 1	$c_1(0) = 0$	$c_1(1) = 5$ (0)	$c_2(1) = 6$ (1)	$c_3(1) = 12$ (1)
état 2	$c_2(0) = 0$	$c_1(2) = 3$ (1)	$c_2(2) = 7$ (2)	$c_3(2) = 9(0)$

- On peut donc retrouver le coût du chemin minimal (9) et en remontant les valeur qui réalisent ce minimum, on peut retrouver un chemin minimal (1,2,2,0) ou (1,2,0,2).
- On peut coder de deux manières : par boucle `for` ou par récursivité.

$c_0(x) = 0$ et pour $k \geqslant 1$ $c_k(x) = \min_{y \in [\![0, p-1]\!]}(c_{k-1}(y) + f(k-1, y, x))$

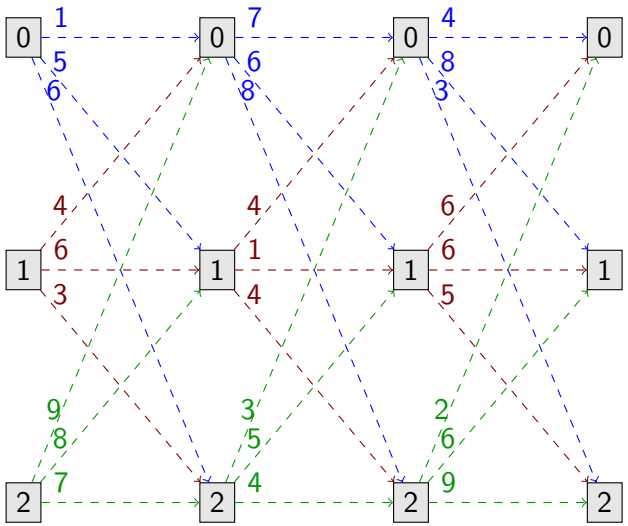


	t_0	t_1	t_2	t_3
état 0	$c_0(0) = 0$	$c_1(0) = 1$ (0)	$c_2(0) = 6$ (2)	$c_3(0) = 9$ (2)
état 1	$c_1(0) = 0$	$c_1(1) = 5$ (0)	$c_2(1) = 6$ (1)	$c_3(1) = 12$ (1)
état 2	$c_2(0) = 0$	$c_1(2) = 3$ (1)	$c_2(2) = 7$ (2)	$c_3(2) = 9$(0)

- On peut donc retrouver le coût du chemin minimal (9) et en remontant les valeur qui réalisent ce minimum, on peut retrouver un chemin minimal (1,2,2,0) ou (1,2,0,2).
- On peut coder de deux manières : par boucle `for` ou par récursivité.

$c_0(x) = 0$ et pour $k \geqslant 1$ $c_k(x) = \min_{y \in [\![0, p-1]\!]}(c_{k-1}(y) + f(k-1, y, x))$

t_0 t_1 t_2 t_3



	t_0	t_1	t_2	t_3
état 0	$c_0(0) = 0$	$c_1(0) = 1$ (0)	$c_2(0) = 6$ (2)	$c_3(0) = 9$ (2)
état 1	$c_1(0) = 0$	$c_1(1) = 5$ (0)	$c_2(1) = 6$ (1)	$c_3(1) = 12$ (1)
état 2	$c_2(0) = 0$	$c_1(2) = 3$ (1)	$c_2(2) = 7$ (2)	$c_3(2) = 9$(0)

- On peut donc retrouver le coût du chemin minimal (9) et en remontant les valeur qui réalisent ce minimum, on peut retrouver un chemin minimal (1,2,2,0) ou (1,2,0,2).
- On peut coder de deux manières : par boucle `for` ou par récursivité.

Calcul de chemin minimal de bas en haut

L'algorithme consiste donc à calculer la valeur de $c_k(x)$ pour chaque temps k et chaque état x, et à conserver le meilleur prédécesseur à x au temps k

- on initialise les valeurs $c_0(x)$ à 0.
- pour chaque temps k et chaque état x on cherche l'état y qui minimise $c_{k-1}(y) + f(k-1, y, x)$.
- La valeur de ce minimum donne la valeur de $c_k(x)$, l'indice de ce minimum (ie le y qui réalise ce minimum) est un des meilleurs prédécesseurs à x au temps k.
- Une fois que tout cela est fait, on calcule le meilleur état d'arrivée, ie le x qui minimise $c_N(x)$.
  Cet état est la dernière étape de notre chemin, ie x_N.
- On considère le prédécesseur de x_N qui est donc x_{N-1}, puis on remonte ainsi, jusqu'à x_0. On peut ainsi reconstruire le chemin $(x_0, \ldots, x_N)$ de coût minimal.

Calcul de chemin minimal de bas en haut

L'algorithme consiste donc à calculer la valeur de $c_k(x)$ pour chaque temps k et chaque état x, et à conserver le meilleur prédécesseur à x au temps k

- on initialise les valeurs $c_0(x)$ à 0.
- pour chaque temps k et chaque état x on cherche l'état y qui minimise $c_{k-1}(y) + f(k-1, y, x)$.
- La valeur de ce minimum donne la valeur de $c_k(x)$, l'indice de ce minimum (ie le y qui réalise ce minimum) est un des meilleurs prédécesseurs à x au temps k.
- Une fois que tout cela est fait, on calcule le meilleur état d'arrivée, ie le x qui minimise $c_N(x)$.
  Cet état est la dernière étape de notre chemin, ie x_N.
- On considère le prédécesseur de x_N qui est donc x_{N-1}, puis on remonte ainsi, jusqu'à x_0. On peut ainsi reconstruire le chemin $(x_0, \ldots, x_N)$ de coût minimal.

Calcul de chemin minimal de bas en haut

L'algorithme consiste donc à calculer la valeur de $c_k(x)$ pour chaque temps k et chaque état x, et à conserver le meilleur prédécesseur à x au temps k

- on initialise les valeurs $c_0(x)$ à 0.
- pour chaque temps k et chaque état x on cherche l'état y qui minimise $c_{k-1}(y) + f(k-1, y, x)$.
- La valeur de ce minimum donne la valeur de $c_k(x)$, l'indice de ce minimum (ie le y qui réalise ce minimum) est un des meilleurs prédécesseurs à x au temps k.
- Une fois que tout cela est fait, on calcule le meilleur état d'arrivée, ie le x qui minimise $c_N(x)$.
  Cet état est la dernière étape de notre chemin, ie x_N.
- On considère le prédécesseur de x_N qui est donc x_{N-1}, puis on remonte ainsi, jusqu'à x_0. On peut ainsi reconstruire le chemin $(x_0, \ldots, x_N)$ de coût minimal.

Calcul de chemin minimal de bas en haut

L'algorithme consiste donc à calculer la valeur de $c_k(x)$ pour chaque temps k et chaque état x, et à conserver le meilleur prédécesseur à x au temps k

- on initialise les valeurs $c_0(x)$ à 0.
- pour chaque temps k et chaque état x on cherche l'état y qui minimise $c_{k-1}(y) + f(k-1, y, x)$.
- La valeur de ce minimum donne la valeur de $c_k(x)$, l'indice de ce minimum (ie le y qui réalise ce minimum) est un des meilleurs prédécesseurs à x au temps k.
- Une fois que tout cela est fait, on calcule le meilleur état d'arrivée, ie le x qui minimise $c_N(x)$.
  Cet état est la dernière étape de notre chemin, ie x_N.
- On considère le prédécesseur de x_N qui est donc x_{N-1}, puis on remonte ainsi, jusqu'à x_0. On peut ainsi reconstruire le chemin $(x_0, \ldots, x_N)$ de coût minimal.

Calcul de chemin minimal de bas en haut

L'algorithme consiste donc à calculer la valeur de $c_k(x)$ pour chaque temps k et chaque état x, et à conserver le meilleur prédécesseur à x au temps k

- on initialise les valeurs $c_0(x)$ à 0.
- pour chaque temps k et chaque état x on cherche l'état y qui minimise $c_{k-1}(y) + f(k-1, y, x)$.
- La valeur de ce minimum donne la valeur de $c_k(x)$, l'indice de ce minimum (ie le y qui réalise ce minimum) est un des meilleurs prédécesseurs à x au temps k.
- Une fois que tout cela est fait, on calcule le meilleur état d'arrivée, ie le x qui minimise $c_N(x)$.
  Cet état est la dernière étape de notre chemin, ie x_N.
- On considère le prédécesseur de x_N qui est donc x_{N-1}, puis on remonte ainsi, jusqu'à x_0. On peut ainsi reconstruire le chemin $(x_0, \ldots, x_N)$ de coût minimal.

Calcul de chemin minimal de bas en haut

L'algorithme consiste donc à calculer la valeur de $c_k(x)$ pour chaque temps k et chaque état x, et à conserver le meilleur prédécesseur à x au temps k

- on initialise les valeurs $c_0(x)$ à 0.
- pour chaque temps k et chaque état x on cherche l'état y qui minimise $c_{k-1}(y) + f(k-1, y, x)$.
- La valeur de ce minimum donne la valeur de $c_k(x)$, l'indice de ce minimum (ie le y qui réalise ce minimum) est un des meilleurs prédécesseurs à x au temps k.
- Une fois que tout cela est fait, on calcule le meilleur état d'arrivée, ie le x qui minimise $c_N(x)$.
  Cet état est la dernière étape de notre chemin, ie x_N.
- On considère le prédécesseur de x_N qui est donc x_{N-1}, puis on remonte ainsi, jusqu'à x_0. On peut ainsi reconstruire le chemin $(x_0, \ldots, x_N)$ de coût minimal.

- On utilise une liste C de longueur $N+1$, de listes de longueur p, de couple `(cout, etat)`.
- Ainsi, `C[k][x]` est donc un couple `(cout, etat)`, avec `cout` un float et `etat` un int. C'est le coût pour arriver en x au temps k ainsi que le prédécesseur de x au temps $k-1$.
- L'instruction :

```
[ C[k-1][y][0] + f(k-1, y, x) for y in range(p)]
```

permet de construire la liste des réels $(c_{k-1}(y) + f(k-1, y, x))$ pour tous les états y.

- C'est sur cette liste que l'on calcule le minimum (valeur de $c_k(x)$) et l'indice du minimum (le meilleur prédécesseur de x).
- On code à part une fonction `minEtindmin` qui prend en entrée une liste de `float` et qui revoir un couple `(valeur, indice)` du minimum de cette liste.
- Le chemin est stockée sous la forme d'une liste $(x_0, \ldots, x_N)$. On construit cette liste en partant de la fin.

- On utilise une liste C de longueur $N+1$, de listes de longueur p, de couple `(cout, etat)`.
- Ainsi, C[k][x] est donc un couple `(cout, etat)`, avec `cout` un float et `etat` un int. C'est le coût pour arriver en x au temps k ainsi que le prédécesseur de x au temps $k-1$.
- L'instruction :

```
[ C[k-1][y][0] + f(k-1, y, x) for y in range(p)]
```

permet de construire la liste des réels $(c_{k-1}(y) + f(k-1, y, x))$ pour tous les états y.

- C'est sur cette liste que l'on calcule le minimum (valeur de $c_k(x)$) et l'indice du minimum (le meilleur prédécesseur de x).
- On code à part une fonction `minEtindmin` qui prend en entrée une liste de `float` et qui revoir un couple `(valeur, indice)` du minimum de cette liste.
- Le chemin est stockée sous la forme d'une liste $(x_0, \ldots, x_N)$. On construit cette liste en partant de la fin.

- On utilise une liste C de longueur $N+1$, de listes de longueur p, de couple `(cout, etat)`.
- Ainsi, C[k][x] est donc un couple `(cout, etat)`, avec `cout` un float et `etat` un int. C'est le coût pour arriver en x au temps k ainsi que le prédécesseur de x au temps $k-1$.
- L'instruction :

```
[ C[k-1][y][0] + f(k-1, y, x) for y in range(p)]
```

  permet de construire la liste des réels $(c_{k-1}(y) + f(k-1, y, x))$ pour tous les états y.
- C'est sur cette liste que l'on calcule le minimum (valeur de $c_k(x)$) et l'indice du minimum (le meilleur prédécesseur de x).
- On code à part une fonction `minEtindmin` qui prend en entrée une liste de `float` et qui revoir un couple `(valeur, indice)` du minimum de cette liste.
- Le chemin est stockée sous la forme d'une liste $(x_0, \ldots, x_N)$. On construit cette liste en partant de la fin.

- On utilise une liste C de longueur $N+1$, de listes de longueur p, de couple `(cout, etat)`.
- Ainsi, C[k][x] est donc un couple `(cout, etat)`, avec `cout` un float et `etat` un int. C'est le coût pour arriver en x au temps k ainsi que le prédécesseur de x au temps $k-1$.
- L'instruction :

```
[ C[k-1][y][0] + f(k-1, y, x) for y in range(p)]
```

  permet de construire la liste des réels $(c_{k-1}(y) + f(k-1, y, x))$ pour tous les états y.
- C'est sur cette liste que l'on calcule le minimum (valeur de $c_k(x)$) et l'indice du minimum (le meilleur prédécesseur de x).
- On code à part une fonction `minEtindmin` qui prend en entrée une liste de `float` et qui revoir un couple `(valeur, indice)` du minimum de cette liste.
- Le chemin est stockée sous la forme d'une liste $(x_0, \ldots, x_N)$. On construit cette liste en partant de la fin.

- On utilise une liste C de longueur $N+1$, de listes de longueur p, de couple `(cout, etat)`.
- Ainsi, C[k][x] est donc un couple `(cout, etat)`, avec `cout` un float et `etat` un int. C'est le coût pour arriver en x au temps k ainsi que le prédécesseur de x au temps $k-1$.
- L'instruction :

```
[ C[k-1][y][0] + f(k-1, y, x) for y in range(p)]
```

  permet de construire la liste des réels $(c_{k-1}(y) + f(k-1, y, x))$ pour tous les états y.
- C'est sur cette liste que l'on calcule le minimum (valeur de $c_k(x)$) et l'indice du minimum (le meilleur prédécesseur de x).
- On code à part une fonction `minEtindmin` qui prend en entrée une liste de `float` et qui revoir un couple `(valeur, indice)` du minimum de cette liste.
- Le chemin est stockée sous la forme d'une liste $(x_0, \ldots, x_N)$. On construit cette liste en partant de la fin.

- On utilise une liste C de longueur $N+1$, de listes de longueur p, de couple `(cout, etat)`.
- Ainsi, C[k][x] est donc un couple `(cout, etat)`, avec `cout` un float et `etat` un int. C'est le coût pour arriver en x au temps k ainsi que le prédécesseur de x au temps $k-1$.
- L'instruction :

```
[ C[k-1][y][0] + f(k-1, y, x) for y in range(p)]
```

  permet de construire la liste des réels $(c_{k-1}(y) + f(k-1, y, x))$ pour tous les états y.
- C'est sur cette liste que l'on calcule le minimum (valeur de $c_k(x)$) et l'indice du minimum (le meilleur prédécesseur de x).
- On code à part une fonction `minEtindmin` qui prend en entrée une liste de `float` et qui revoir un couple `(valeur, indice)` du minimum de cette liste.
- Le chemin est stockée sous la forme d'une liste $(x_0, \ldots, x_N)$. On construit cette liste en partant de la fin.

- Pour construire la liste C :

```
C = [ [ [0, None] for i in range(p)] for j  in range(N+1)]
for k in range(1,N+1):
  for x in range(p):
    C[k][x] = minEtindmin(
        [ C[k-1][y][0] + f(k-1, y, x) for y in range(p)] )
```

- Pour calculer le coût minimal et le point d'arrivée :

```
cout, xN = minEtindmin([ C[N][x][0] for x in range(p)] )
```

- Pour reconstruire le chemin :

```
chemin = [None]*(N+1)
chemin[N] = xN
for i in range(N-1, -1, -1):
  chemin[i] = C[i+1][chemin[i+1]][1]
```

Complexité de l'algorithme :

- Pour chaque k dans $[\![1, N]\!]$, Pour chaque x dans $[\![0, p-1]\!]$,
- on calcule la liste :

```
[ C[k-1][y][0] + f(k-1, y, x) for y in range(p)]
```

  ce qui fait p appel à f et p additions. Donc $2p$ opérations.
- on calcule le minimum, et l'indice du minimum de cette liste de longueur p, ce qui fait p comparaisons (RAPPEL : une comparaison est beaucoup plus rapide qu'une addition).
- il reste un dernier minimum à calculer soit p comparaison.

Cela fait donc $\Theta(2 \times p^2 \times N)$ opérations au lieu de p^{N+1}

Avec la méthode de programmation que l'on a mis en place ici, le coût en mémoire est élevé : on stocke un tableau de $p \times N$ couples. On peut éventuellement, diminuer ce coût en écrasant à chaque itération les valeurs de $c_k(x)$.

Calcul récursif du chemin minimal

- L'équation de Bellman $c_k(x) = \min_{y \in [\![0,p-1]\!]}(c_{k-1}(y) + f(k-1,y,x))$ peut aussi se programmer de manière récursive.
- Cas trivial : $k = 0$ on renvoie [0, None], et si $k > 0$, on calcule la liste des $(c_{k-1}(y) + f(k-1,y,x))$, on renvoie le minimum (le coût) et l'indice du minimum (le prédécesseur).

- Pour calculer $c_k(x)$, on a besoin de tous les $c_{k-1}(y)$, le nombre d'appels à la fonction récursive est trop important
- On refait sans cesse les mêmes calculs ! Il faut garder en mémoire les résultats.

Calcul récursif du chemin minimal

- L'équation de Bellman $c_k(x) = \min_{y \in [\![0, p-1]\!]}(c_{k-1}(y) + f(k-1, y, x))$ peut aussi se programmer de manière récursive.
- Cas trivial : $k = 0$ on renvoie [0, None], et si $k > 0$, on calcule la liste des $(c_{k-1}(y) + f(k-1, y, x))$, on renvoie le minimum (le coût) et l'indice du minimum (le prédécesseur).

- Pour calculer $c_k(x)$, on a besoin de tous les $c_{k-1}(y)$, le nombre d'appels à la fonction récursive est trop important
- On refait sans cesse les mêmes calculs ! Il faut garder en mémoire les résultats.

Calcul récursif du chemin minimal

- L'équation de Bellman $c_k(x) = \min_{y \in [\![0, p-1]\!]}(c_{k-1}(y) + f(k-1, y, x))$ peut aussi se programmer de manière récursive.
- Cas trivial : $k = 0$ on renvoie [0, None], et si $k > 0$, on calcule la liste des $(c_{k-1}(y) + f(k-1, y, x))$, on renvoie le minimum (le coût) et l'indice du minimum (le prédécesseur).

- Pour calculer $c_k(x)$, on a besoin de tous les $c_{k-1}(y)$, le nombre d'appels à la fonction récursive est trop important
- On refait sans cesse les mêmes calculs ! Il faut garder en mémoire les résultats.

Calcul récursif du chemin minimal

- L'équation de Bellman $c_k(x) = \min_{y \in [\![0, p-1]\!]}(c_{k-1}(y) + f(k-1, y, x))$ peut aussi se programmer de manière récursive.
- Cas trivial : $k = 0$ on renvoie [0, None], et si $k > 0$, on calcule la liste des $(c_{k-1}(y) + f(k-1, y, x))$, on renvoie le minimum (le coût) et l'indice du minimum (le prédécesseur).

```
def C(k,x):
  if k == 0 :
    return [0, None]
  return minEtindmin([ C(k-1,y)[0] + f(k-1, y, x)
                                    for y in range(p)])
```

- Pour calculer $c_k(x)$, on a besoin de tous les $c_{k-1}(y)$, le nombre d'appels à la fonction récursive est trop important
- On refait sans cesse les mêmes calculs ! Il faut garder en mémoire les résultats.

Calcul récursif du chemin minimal

- L'équation de Bellman $c_k(x) = \min_{y \in [\![0, p-1]\!]}(c_{k-1}(y) + f(k-1, y, x))$ peut aussi se programmer de manière récursive.
- Cas trivial : $k = 0$ on renvoie [0, None], et si $k > 0$, on calcule la liste des $(c_{k-1}(y) + f(k-1, y, x))$, on renvoie le minimum (le coût) et l'indice du minimum (le prédécesseur).

```
def C(k,x):
  if k == 0 :
    return [0, None]
  return minEtindmin([ C(k-1,y)[0] + f(k-1, y, x)
                                    for y in range(p)])
```

- Pour calculer $c_k(x)$, on a besoin de tous les $c_{k-1}(y)$, le nombre d'appels à la fonction récursive est trop important
- On refait sans cesse les mêmes calculs ! Il faut garder en mémoire les résultats.

Calcul récursif du chemin minimal

- L'équation de Bellman $c_k(x) = \min_{y \in [\![0, p-1]\!]}(c_{k-1}(y) + f(k-1, y, x))$ peut aussi se programmer de manière récursive.
- Cas trivial : $k = 0$ on renvoie [0, None], et si $k > 0$, on calcule la liste des $(c_{k-1}(y) + f(k-1, y, x))$, on renvoie le minimum (le coût) et l'indice du minimum (le prédécesseur).

```
def C(k,x):
  if k == 0 :
    return [0, None]
  return minEtindmin([ C(k-1,y)[0] + f(k-1, y, x)
                                    for y in range(p)])
```

- Pour calculer $c_k(x)$, on a besoin de tous les $c_{k-1}(y)$, le nombre d'appels à la fonction récursive est trop important
- On refait sans cesse les mêmes calculs ! Il faut garder en mémoire les résultats.

Retour sur la récursivité

- Une fonction récursive est une fonction qui s'appelle elle-même.
- Le principe est d'utiliser la récurrence : on sait résoudre le problème lorsque $n = 0$ (cas trivial), pour résoudre le problème pour un n fixé, on appelle la fonction pour $n - 1$.
- Cela donne du code très compact et permet d'écrire facilement du code complexe.
- On prouve la terminaison et la correction d'un programme récursif en écrivant : $\mathcal{P}(n)$:« sur l'entrée n le programme se termine et donne le bon résultat »

```
def fact(n):
  if n == 0:
    return 1
  return fact(n-1)
```

Retour sur la récursivité

- Une fonction récursive est une fonction qui s'appelle elle-même.
- Le principe est d'utiliser la récurrence : on sait résoudre le problème lorsque $n = 0$ (cas trivial), pour résoudre le problème pour un n fixé, on appelle la fonction pour $n - 1$.
- Cela donne du code très compact et permet d'écrire facilement du code complexe.
- On prouve la terminaison et la correction d'un programme récursif en écrivant : $\mathcal{P}(n)$:« sur l'entrée n le programme se termine et donne le bon résultat »

```
def fact(n):
  if n == 0:
    return 1
  return fact(n-1)
```

Retour sur la récursivité

- Une fonction récursive est une fonction qui s'appelle elle-même.
- Le principe est d'utiliser la récurrence : on sait résoudre le problème lorsque $n = 0$ (cas trivial), pour résoudre le problème pour un n fixé, on appelle la fonction pour $n - 1$.
- Cela donne du code très compact et permet d'écrire facilement du code complexe.
- On prouve la terminaison et la correction d'un programme récursif en écrivant : $\mathcal{P}(n)$:« sur l'entrée n le programme se termine et donne le bon résultat »

```
def fact(n):
  if n == 0:
    return 1
  return fact(n-1)
```

Retour sur la récursivité

- Une fonction récursive est une fonction qui s'appelle elle-même.
- Le principe est d'utiliser la récurrence : on sait résoudre le problème lorsque $n = 0$ (cas trivial), pour résoudre le problème pour un n fixé, on appelle la fonction pour $n - 1$.
- Cela donne du code très compact et permet d'écrire facilement du code complexe.
- On prouve la terminaison et la correction d'un programme récursif en écrivant : $\mathcal{P}(n)$:« sur l'entrée n le programme se termine et donne le bon résultat »

```
def fact(n):
  if n == 0:
    return 1
  return fact(n-1)
```

Retour sur la récursivité

- Une fonction récursive est une fonction qui s'appelle elle-même.
- Le principe est d'utiliser la récurrence : on sait résoudre le problème lorsque $n = 0$ (cas trivial), pour résoudre le problème pour un n fixé, on appelle la fonction pour $n - 1$.
- Cela donne du code très compact et permet d'écrire facilement du code complexe.
- On prouve la terminaison et la correction d'un programme récursif en écrivant : $\mathcal{P}(n)$:« sur l'entrée n le programme se termine et donne le bon résultat »

```
def fact(n):
  if n == 0:
    return 1
  return fact(n-1)
```

Exemple pour les tours de Hanoi

```
def hanoiRec(n, D, A, I):
    """
    entrée: n =  int =  nbr de disques à déplacer
            D = int = indice de la tour de départ
            A = int = arrivée
            I = int = intermédiaire
    sortiie : liste de couple de int de la forme (i,j)
            liste des déplacements à effectuer
    """
    if n == 1 :
        return [ [D,A] ]
    return hanoiRec(n-1, D, I, A) + [[D,A]] + hanoiRec(n-1, I, A, D)

def resoud(n):
    """
    entrée: n = int = nbr de disques à déplacer
    sortie: liste de couple de int de la forme (i,j)
            liste des déplacements à effectuer
    """
    return hanoiRec(n, 1, 3, 2)
```

Exemple pour le tri fusion

```
def tri(L):
    """
    entrée: L = list = liste à trier
    sortie: L = la liste triée
    """
    if len(L) <= 1:
        return L

    # tri des deux "moitiés" de liste
    m = len(L) // 2
    L1 = tri(L[:m])
    L2 = tri(L[m:])

    return fusion(L1, L2)
```

Nombres d'appels récursifs

```
def fibbo(n):
    if n == 0:
        return 1
    if n == 1 :
        return 1
    return fibbo(n-1) + fibbo(n-2)
```

- On fait trop d'appels récursifs !
- Il faut garder en mémoire les valeurs fibbo(n), on parle de mémoïsation.
- Un dictionnaire est très utile pour cela : si la valeur existe, on la renvoie, sinon on la calcule et la stocke.
- Cette technique est à connaître !

Nombres d'appels récursifs

```
def fibbo(n):
    if n == 0:
        return 1
    if n == 1 :
        return 1
    return fibbo(n-1) + fibbo(n-2)
```

- On fait trop d'appels récursifs !
- Il faut garder en mémoire les valeurs fibbo(n), on parle de mémoïsation.
- Un dictionnaire est très utile pour cela : si la valeur existe, on la renvoie, sinon on la calcule et la stocke.
- Cette technique est à connaître !

Nombres d'appels récursifs

```
def fibbo(n):
    if n == 0:
        return 1
    if n == 1 :
        return 1
    return fibbo(n-1) + fibbo(n-2)
```

- On fait trop d'appels récursifs !
- Il faut garder en mémoire les valeurs fibbo(n), on parle de mémoïsation.
- Un dictionnaire est très utile pour cela : si la valeur existe, on la renvoie, sinon on la calcule et la stocke.
- Cette technique est à connaître !

Nombres d'appels récursifs

```
def fibbo(n):
    if n == 0:
        return 1
    if n == 1 :
        return 1
    return fibbo(n-1) + fibbo(n-2)
```

- On fait trop d'appels récursifs !
- Il faut garder en mémoire les valeurs fibbo(n), on parle de mémoïsation.
- Un dictionnaire est très utile pour cela : si la valeur existe, on la renvoie, sinon on la calcule et la stocke.
- Cette technique est à connaître !

Nombres d'appels récursifs

```
def fibbo(n):
    if n == 0:
        return 1
    if n == 1 :
        return 1
    return fibbo(n-1) + fibbo(n-2)
```

- On fait trop d'appels récursifs !
- Il faut garder en mémoire les valeurs fibbo(n), on parle de mémoïsation.
- Un dictionnaire est très utile pour cela : si la valeur existe, on la renvoie, sinon on la calcule et la stocke.

```
F = {0 : 1, 1: 1 } # initialisation en dehors de la fonction
def fibbo(n):
    if n not in F :
        F[n] = fibbo(n-1) + fibbo(n-2) # modification du dictionnaire
    return F[n]
```

- Cette technique est à connaître !

Nombres d'appels récursifs

```
def fibbo(n):
    if n == 0:
        return 1
    if n == 1 :
        return 1
    return fibbo(n-1) + fibbo(n-2)
```

- On fait trop d'appels récursifs !
- Il faut garder en mémoire les valeurs fibbo(n), on parle de mémoïsation.
- Un dictionnaire est très utile pour cela : si la valeur existe, on la renvoie, sinon on la calcule et la stocke.

```
F = {0 : 1, 1: 1 } # initialisation en dehors de la fonction
def fibbo(n):
    if n not in F :
        F[n] = fibbo(n-1) + fibbo(n-2) # modification du dictionnaire
    return F[n]
```

- Cette technique est à connaître !

Calcul récursif du chemin minimal

- Équation de Bellman $c_k(x) = \min_{y \in [\![0,p-1]\!]}(c_{k-1}(y) + f(k-1, y, x))$ peut aussi se programmer de manière récursive avec la mémoïsation.
- On utilise donc un dictionnaire C indexé sur les tuples (k, x), le dictionnaire contient le couple `(cout, etat)` .
- On crée une fonction récursive `calculeC` qui modifie le dictionnaire et renvoie le couple C[(k,x)] .
- Il faut déterminer le meilleur état d'arrivée, ie le x qui minimise $c_N(x)$, On appelle cette fonction récursive pour calculer C[(N,x)] pour tous les x.
- Cet état est la dernière étape de notre chemin, ie x_N. Puis de reconstruire le chemin $(x_0, \ldots, x_N)$ de coût minimal.

Calcul récursif du chemin minimal

- Équation de Bellman $c_k(x) = \min_{y \in [\![0,p-1]\!]}(c_{k-1}(y) + f(k-1,y,x))$ peut aussi se programmer de manière récursive avec la mémoïsation.
- On utilise donc un dictionnaire C indexé sur les tuples (k,x), le dictionnaire contient le couple `(cout, etat)` .
- On crée une fonction récursive `calculeC` qui modifie le dictionnaire et renvoie le couple C[(k,x)] .
- Il faut déterminer le meilleur état d'arrivée, ie le x qui minimise $c_N(x)$, On appelle cette fonction récursive pour calculer C[(N,x)] pour tous les x.
- Cet état est la dernière étape de notre chemin, ie x_N. Puis de reconstruire le chemin $(x_0, \ldots, x_N)$ de coût minimal.

Calcul récursif du chemin minimal

- Équation de Bellman $c_k(x) = \min_{y \in [\![0,p-1]\!]}(c_{k-1}(y) + f(k-1,y,x))$ peut aussi se programmer de manière récursive avec la mémoïsation.
- On utilise donc un dictionnaire C indexé sur les tuples (k,x), le dictionnaire contient le couple `(cout, etat)` .
- On crée une fonction récursive `calculeC` qui modifie le dictionnaire et renvoie le couple C[(k,x)] .
- Il faut déterminer le meilleur état d'arrivée, ie le x qui minimise $c_N(x)$, On appelle cette fonction récursive pour calculer C[(N,x)] pour tous les x.
- Cet état est la dernière étape de notre chemin, ie x_N. Puis de reconstruire le chemin $(x_0, \ldots, x_N)$ de coût minimal.

Calcul récursif du chemin minimal

- Équation de Bellman $c_k(x) = \min_{y \in [\![0,p-1]\!]}(c_{k-1}(y) + f(k-1,y,x))$ peut aussi se programmer de manière récursive avec la mémoïsation.
- On utilise donc un dictionnaire C indexé sur les tuples (k,x), le dictionnaire contient le couple `(cout, etat)` .
- On crée une fonction récursive `calculeC` qui modifie le dictionnaire et renvoie le couple C[(k,x)] .
- Il faut déterminer le meilleur état d'arrivée, ie le x qui minimise $c_N(x)$, On appelle cette fonction récursive pour calculer C[(N,x)] pour tous les x.
- Cet état est la dernière étape de notre chemin, ie x_N. Puis de reconstruire le chemin $(x_0, \ldots, x_N)$ de coût minimal.

Calcul récursif du chemin minimal

- Équation de Bellman $c_k(x) = \min_{y \in [\![0,p-1]\!]}(c_{k-1}(y) + f(k-1,y,x))$ peut aussi se programmer de manière récursive avec la mémoïsation.
- On utilise donc un dictionnaire C indexé sur les tuples (k,x), le dictionnaire contient le couple `(cout, etat)` .
- On crée une fonction récursive `calculeC` qui modifie le dictionnaire et renvoie le couple C[(k,x)] .
- Il faut déterminer le meilleur état d'arrivée, ie le x qui minimise $c_N(x)$, On appelle cette fonction récursive pour calculer C[(N,x)] pour tous les x.
- Cet état est la dernière étape de notre chemin, ie x_N. Puis de reconstruire le chemin $(x_0, \ldots, x_N)$ de coût minimal.

Calcul récursif du chemin minimal

- Équation de Bellman $c_k(x) = \min_{y \in [\![0,p-1]\!]}(c_{k-1}(y) + f(k-1,y,x))$ peut aussi se programmer de manière récursive avec la mémoïsation.
- On utilise donc un dictionnaire C indexé sur les tuples (k,x), le dictionnaire contient le couple `(cout, etat)` .
- On crée une fonction récursive `calculeC` qui modifie le dictionnaire et renvoie le couple C[(k,x)] .
- Il faut déterminer le meilleur état d'arrivée, ie le x qui minimise $c_N(x)$, On appelle cette fonction récursive pour calculer C[(N,x)] pour tous les x.
- Cet état est la dernière étape de notre chemin, ie x_N. Puis de reconstruire le chemin $(x_0, \ldots, x_N)$ de coût minimal.

Fonction récursive :

```
C = {}
for x in range(p):
  C[(0,x)] = [0, None]

def calculeC(k,x):
  """
  fonction récursive qui remplit le dictionnaire
  """
  if (k,x) not in C:
    C[(k,x)] = minEtindmin( [ calculeC(k-1,y)[0] + f(k-1, y, x) for y in range(p)] )
  return C[(k,x)]
```

Appel de la fonction récursive et calcul du chemin :

```
# l'appel à la fonction récursive se fait ici:
for y in range(p):
  calculeC(p,y)

cout, xN = minEtindmin([ C[(p,y)][0] for y in range(N)] )

# on reconstruit le chemin complet
chemin = [None]*(N+1)
chemin[N] = xN
for i in range(N-1, -1, -1):
    chemin[i] = C[( i+1, chemin[i+1]) ][1]
```

On peut dégager une structure de problèmes pouvant être résolus par programmation dynamique :

- Il s'agit de problèmes de minimisation : On a un ensemble E fini mais très grand et on cherche la valeur de $\min_{x \in E} f(x)$ et le x qui réalise le minimum. La fonction f est connue.
- On a plusieurs sous-problèmes plus petits.
- Le problème initial est le plus gros des sous-problèmes (le dernier), le plus petit des sous-problème admet une solution triviale.
- *chevauchement des sous-problème* : Ces sous-problèmes sont entremêlés.
- Un sous-problème d'une taille donné se fait facilement en utilisant les solutions des sous problèmes d'une taille plus petite. Cela se traduit par une *équation de Bellman* que l'on peut voir comme une formule de récurrence.

On peut dégager une structure de problèmes pouvant être résolus par programmation dynamique :

- Il s'agit de problèmes de minimisation : On a un ensemble E fini mais très grand et on cherche la valeur de $\min_{x \in E} f(x)$ et le x qui réalise le minimum. La fonction f est connue. Dans notre exemple : le chemin de coût minimal parmi tous les chemins possibles.
- On a plusieurs sous-problèmes plus petits.
- Le problème initial est le plus gros des sous-problèmes (le dernier), le plus petit des sous-problème admet une solution triviale.
- *chevauchement des sous-problème* : Ces sous-problèmes sont entremêlés.
- Un sous-problème d'une taille donné se fait facilement en utilisant les solutions des sous problèmes d'une taille plus petite. Cela se traduit par une *équation de Bellman* que l'on peut voir comme une formule de récurrence.

On peut dégager une structure de problèmes pouvant être résolus par programmation dynamique :

- Il s'agit de problèmes de minimisation : On a un ensemble E fini mais très grand et on cherche la valeur de $\min_{x \in E} f(x)$ et le x qui réalise le minimum. La fonction f est connue.
- On a plusieurs sous-problèmes plus petits.
- Le problème initial est le plus gros des sous-problèmes (le dernier), le plus petit des sous-problème admet une solution triviale.
- *chevauchement des sous-problème* : Ces sous-problèmes sont entremêlés.
- Un sous-problème d'une taille donné se fait facilement en utilisant les solutions des sous problèmes d'une taille plus petite. Cela se traduit par une *équation de Bellman* que l'on peut voir comme une formule de récurrence.

On peut dégager une structure de problèmes pouvant être résolus par programmation dynamique :

- Il s'agit de problèmes de minimisation : On a un ensemble E fini mais très grand et on cherche la valeur de $\min_{x \in E} f(x)$ et le x qui réalise le minimum. La fonction f est connue.
- On a plusieurs sous-problèmes plus petits. Dans notre exemple : $c_k(x)$ le chemin de coût minimal à chacun des temps k.
- Le problème initial est le plus gros des sous-problèmes (le dernier), le plus petit des sous-problème admet une solution triviale.
- *chevauchement des sous-problème* : Ces sous-problèmes sont entremêlés.
- Un sous-problème d'une taille donné se fait facilement en utilisant les solutions des sous problèmes d'une taille plus petite. Cela se traduit par une *équation de Bellman* que l'on peut voir comme une formule de récurrence.

On peut dégager une structure de problèmes pouvant être résolus par programmation dynamique :

- Il s'agit de problèmes de minimisation : On a un ensemble E fini mais très grand et on cherche la valeur de $\min_{x \in E} f(x)$ et le x qui réalise le minimum. La fonction f est connue.
- On a plusieurs sous-problèmes plus petits.
- Le problème initial est le plus gros des sous-problèmes (le dernier), le plus petit des sous-problème admet une solution triviale.
- *chevauchement des sous-problème* : Ces sous-problèmes sont entremêlés.
- Un sous-problème d'une taille donné se fait facilement en utilisant les solutions des sous problèmes d'une taille plus petite. Cela se traduit par une *équation de Bellman* que l'on peut voir comme une formule de récurrence.

On peut dégager une structure de problèmes pouvant être résolus par programmation dynamique :

- Il s'agit de problèmes de minimisation : On a un ensemble E fini mais très grand et on cherche la valeur de $\min_{x \in E} f(x)$ et le x qui réalise le minimum. La fonction f est connue.
- On a plusieurs sous-problèmes plus petits.
- Le problème initial est le plus gros des sous-problèmes (le dernier), le plus petit des sous-problème admet une solution triviale. Dans notre exemple, $c_0(x) = 0$ pour tout x et on cherche $\min_x c_N(x)$.
- *chevauchement des sous-problème* : Ces sous-problèmes sont entremêlés.
- Un sous-problème d'une taille donné se fait facilement en utilisant les solutions des sous problèmes d'une taille plus petite. Cela se traduit par une *équation de Bellman* que l'on peut voir comme une formule de récurrence.

On peut dégager une structure de problèmes pouvant être résolus par programmation dynamique :

- Il s'agit de problèmes de minimisation : On a un ensemble E fini mais très grand et on cherche la valeur de $\min_{x \in E} f(x)$ et le x qui réalise le minimum. La fonction f est connue.
- On a plusieurs sous-problèmes plus petits.
- Le problème initial est le plus gros des sous-problèmes (le dernier), le plus petit des sous-problème admet une solution triviale.
- *chevauchement des sous-problème* : Ces sous-problèmes sont entremêlés.
- Un sous-problème d'une taille donné se fait facilement en utilisant les solutions des sous problèmes d'une taille plus petite. Cela se traduit par une *équation de Bellman* que l'on peut voir comme une formule de récurrence.

On peut dégager une structure de problèmes pouvant être résolus par programmation dynamique :

- Il s'agit de problèmes de minimisation : On a un ensemble E fini mais très grand et on cherche la valeur de $\min_{x \in E} f(x)$ et le x qui réalise le minimum. La fonction f est connue.
- On a plusieurs sous-problèmes plus petits.
- Le problème initial est le plus gros des sous-problèmes (le dernier), le plus petit des sous-problème admet une solution triviale.
- *chevauchement des sous-problème* : Ces sous-problèmes sont entremêlés. Dans notre exemple : un chemin de coût minimal est minimal à toutes les étapes.
- Un sous-problème d'une taille donné se fait facilement en utilisant les solutions des sous problèmes d'une taille plus petite. Cela se traduit par une *équation de Bellman* que l'on peut voir comme une formule de récurrence.

On peut dégager une structure de problèmes pouvant être résolus par programmation dynamique :

- Il s'agit de problèmes de minimisation : On a un ensemble E fini mais très grand et on cherche la valeur de $\min_{x \in E} f(x)$ et le x qui réalise le minimum. La fonction f est connue.
- On a plusieurs sous-problèmes plus petits.
- Le problème initial est le plus gros des sous-problèmes (le dernier), le plus petit des sous-problème admet une solution triviale.
- *chevauchement des sous-problème* : Ces sous-problèmes sont entremêlés.
- Un sous-problème d'une taille donné se fait facilement en utilisant les solutions des sous problèmes d'une taille plus petite. Cela se traduit par une *équation de Bellman* que l'on peut voir comme une formule de récurrence.

On peut dégager une structure de problèmes pouvant être résolus par programmation dynamique :

- Il s'agit de problèmes de minimisation : On a un ensemble E fini mais très grand et on cherche la valeur de $\min_{x \in E} f(x)$ et le x qui réalise le minimum. La fonction f est connue.
- On a plusieurs sous-problèmes plus petits.
- Le problème initial est le plus gros des sous-problèmes (le dernier), le plus petit des sous-problème admet une solution triviale.
- *chevauchement des sous-problème* : Ces sous-problèmes sont entremêlés.
- Un sous-problème d'une taille donné se fait facilement en utilisant les solutions des sous problèmes d'une taille plus petite. Cela se traduit par une *équation de Bellman* que l'on peut voir comme une formule de récurrence.
  Dans notre exemple : $c_k(x) = \min_{y \in [\![0, p-1]\!]}(c_{k-1}(y) + f(k-1, y, x))$.

Programmation dynamique

On peut utiliser la programmation dynamique de deux manières :

- Soit par programmation itérative (approche bas en haut) : on part du sous-problème le plus petit et on utilise une boucle `for` pour résoudre des problèmes de plus en plus gros.
- La programmation itérative est surtout utile lorsque résoudre le sous problème de taille k nécessite la résolution du problème de taille $k-1$
- soit par programmation récursive : on utilise des appels récursifs.
  On part alors du problème le plus gros et on le découpe en sous-problème plus petit, jusqu'au problème trivial.
  Pour éviter les trop nombreux appels récursifs, on utilise la mémoïsation : on stocke les valeurs calculées dans un dictionnaire.
- La programmation récursive est surtout utile lorsqu'il n'est pas nécessaire de résoudre tous les sous-problème.
  Exemple : quand le problème initial est de taille n et nécessite de résoudre les problèmes pour tous les diviseurs de n.

Programmation dynamique

On peut utiliser la programmation dynamique de deux manières :

- Soit par programmation itérative (approche bas en haut) : on part du sous-problème le plus petit et on utilise une boucle `for` pour résoudre des problèmes de plus en plus gros.
- La programmation itérative est surtout utile lorsque résoudre le sous problème de taille k nécessite la résolution du problème de taille $k-1$
- soit par programmation récursive : on utilise des appels récursifs.
  On part alors du problème le plus gros et on le découpe en sous-problème plus petit, jusqu'au problème trivial.
  Pour éviter les trop nombreux appels récursifs, on utilise la mémoïsation : on stocke les valeurs calculées dans un dictionnaire.
- La programmation récursive est surtout utile lorsqu'il n'est pas nécessaire de résoudre tous les sous-problème.
  Exemple : quand le problème initial est de taille n et nécessite de résoudre les problèmes pour tous les diviseurs de n.

Programmation dynamique

On peut utiliser la programmation dynamique de deux manières :

- Soit par programmation itérative (approche bas en haut) : on part du sous-problème le plus petit et on utilise une boucle `for` pour résoudre des problèmes de plus en plus gros.
- La programmation itérative est surtout utile lorsque résoudre le sous problème de taille k nécessite la résolution du problème de taille $k-1$
- soit par programmation récursive : on utilise des appels récursifs.
  On part alors du problème le plus gros et on le découpe en sous-problème plus petit, jusqu'au problème trivial.
  Pour éviter les trop nombreux appels récursifs, on utilise la mémoïsation : on stocke les valeurs calculées dans un dictionnaire.
- La programmation récursive est surtout utile lorsqu'il n'est pas nécessaire de résoudre tous les sous-problème.
  Exemple : quand le problème initial est de taille n et nécessite de résoudre les problèmes pour tous les diviseurs de n.

Programmation dynamique

On peut utiliser la programmation dynamique de deux manières :

- Soit par programmation itérative (approche bas en haut) : on part du sous-problème le plus petit et on utilise une boucle `for` pour résoudre des problèmes de plus en plus gros.
- La programmation itérative est surtout utile lorsque résoudre le sous problème de taille k nécessite la résolution du problème de taille $k-1$
- soit par programmation récursive : on utilise des appels récursifs.
  On part alors du problème le plus gros et on le découpe en sous-problème plus petit, jusqu'au problème trivial.
  Pour éviter les trop nombreux appels récursifs, on utilise la mémoïsation : on stocke les valeurs calculées dans un dictionnaire.
- La programmation récursive est surtout utile lorsqu'il n'est pas nécessaire de résoudre tous les sous-problème.
  Exemple : quand le problème initial est de taille n et nécessite de résoudre les problèmes pour tous les diviseurs de n.

Deuxième exemple : algorithme de Roy-Floyd-Warshall

- On considère un graphe valué dont les sommets sont les entiers $[\![0, n-1]\!]$. Il est représenté par sa matrice d'adjacence W.

$$W_{i,j} = \begin{cases} +\infty & \text{si } i \text{ n'est pas relié à } j \\ w_{i,j} & \text{longueur de l'arc de } i \text{ vers } j \end{cases}$$

- Soit i et j deux sommets. Un chemin qui relie i à j est une suite finie de sommets reliés qui commence à i et finit à j. Autrement dit, c'est une liste

$$p = (i_0, \ldots, i_l) \text{ avec } i_0 = i \text{ et } i_l = j$$

avec $\forall s \in [\![0, l-1]\!], w_{i_s, i_{s+1}} \neq +\infty$ (chaque sommet est bien relié au suivant).

On note de plus $C_{i,j}$ l'ensemble des chemins de i à j.

La longueur du chemin $p = (i_0, \ldots, i_l)$ est : $L(p) = \sum_{s=0}^{l-1} w_{i_s, i_{s+1}}$ C'est la somme des longueurs des arrêtes.

Deuxième exemple : algorithme de Roy-Floyd-Warshall

- On considère un graphe valué dont les sommets sont les entiers $[\![0, n-1]\!]$. Il est représenté par sa matrice d'adjacence W.

$$W_{i,j} = \begin{cases} +\infty & \text{si } i \text{ n'est pas relié à } j \\ w_{i,j} & \text{longueur de l'arc de } i \text{ vers } j \end{cases}$$

- Soit i et j deux sommets. Un chemin qui relie i à j est une suite finie de sommets reliés qui commence à i et finit à j. Autrement dit, c'est une liste

$$p = (i_0, \ldots, i_l) \text{ avec } i_0 = i \text{ et } i_l = j$$

avec $\forall s \in [\![0, l-1]\!], w_{i_s,i_{s+1}} \neq +\infty$ (chaque sommet est bien relié au suivant).

On note de plus $C_{i,j}$ l'ensemble des chemins de i à j.

La longueur du chemin $p = (i_0, \ldots, i_l)$ est : $L(p) = \sum_{s=0}^{l-1} w_{i_s,i_{s+1}}$ C'est la somme des longueurs des arrêtes.

- On considère un graphe valué dont les sommets sont les entiers $[\![0, n-1]\!]$. Il est représenté par sa matrice d'adjacence W.
- Soit i et j deux sommets. Un chemin qui relie i à j est une suite finie de sommets reliés qui commence à i et finit à j. Noté :
$$p = (i_0, \dots, i_l) \text{ avec } i_0 = i \text{ et } i_l = j$$
On note de plus $C_{i,j}$ l'ensemble des chemins de i à j.
La longueur du chemin $p = (i_0, \dots, i_l)$ est : $L(p) = \sum_{s=0}^{l-1} w_{i_s, i_{s+1}}$
- On s'intéresse à la recherche du chemin le plus court entre deux sommets i et j, ie la longueur la plus faible.
On cherche donc : $\mathcal{L}(i,j) = \min_{p \in C_{i,j}} L(p)$
- Comme pour l'algorithme de Dijkstra mais pour tous les points de départ i et d'arrivée j.

- On considère un graphe valué dont les sommets sont les entiers $[\![0, n-1]\!]$. Il est représenté par sa matrice d'adjacence W.
- Soit i et j deux sommets. Un chemin qui relie i à j est une suite finie de sommets reliés qui commence à i et finit à j. Noté :
$$p = (i_0, \dots, i_l) \text{ avec } i_0 = i \text{ et } i_l = j$$
On note de plus $C_{i,j}$ l'ensemble des chemins de i à j.
La longueur du chemin $p = (i_0, \dots, i_l)$ est : $L(p) = \sum_{s=0}^{l-1} w_{i_s, i_{s+1}}$
- On s'intéresse à la recherche du chemin le plus court entre deux sommets i et j, ie la longueur la plus faible.
On cherche donc : $\mathcal{L}(i,j) = \min_{p \in C_{i,j}} L(p)$
- Comme pour l'algorithme de Dijkstra mais pour tous les points de départ i et d'arrivée j.

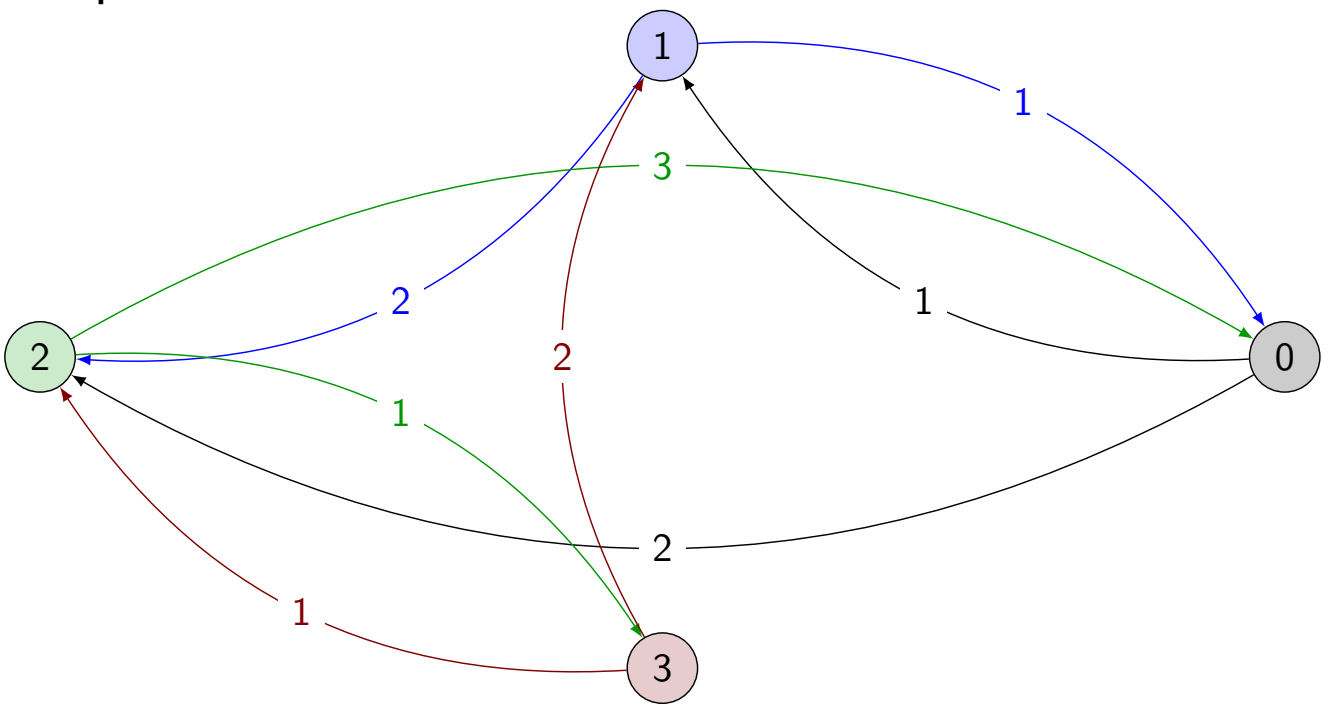
- On considère un graphe valué dont les sommets sont les entiers $[\![0, n-1]\!]$. Il est représenté par sa matrice d'adjacence W.
- Soit i et j deux sommets. Un chemin qui relie i à j est une suite finie de sommets reliés qui commence à i et finit à j. Noté :
$$p = (i_0, \dots, i_l) \text{ avec } i_0 = i \text{ et } i_l = j$$
On note de plus $C_{i,j}$ l'ensemble des chemins de i à j.
La longueur du chemin $p = (i_0, \dots, i_l)$ est : $L(p) = \sum_{s=0}^{l-1} w_{i_s, i_{s+1}}$
- On s'intéresse à la recherche du chemin le plus court entre deux sommets i et j, ie la longueur la plus faible.
On cherche donc : $\mathcal{L}(i,j) = \min_{p \in C_{i,j}} L(p)$
- Comme pour l'algorithme de Dijkstra mais pour tous les points de départ i et d'arrivée j.

- On considère un graphe valué dont les sommets sont les entiers $[\![0, n-1]\!]$. Il est représenté par sa matrice d'adjacence W.
- Soit i et j deux sommets. Un chemin qui relie i à j est une suite finie de sommets reliés qui commence à i et finit à j. Noté :
$$p = (i_0, \dots, i_l) \text{ avec } i_0 = i \text{ et } i_l = j$$
On note de plus $C_{i,j}$ l'ensemble des chemins de i à j.
La longueur du chemin $p = (i_0, \dots, i_l)$ est :$L(p) = \sum_{s=0}^{l-1} w_{i_s, i_{s+1}}$
- On s'intéresse à la recherche du chemin le plus court entre deux sommets i et j, ie la longueur la plus faible.
On cherche donc : $\mathcal{L}(i,j) = \min_{p \in C_{i,j}} L(p)$
- Comme pour l'algorithme de Dijkstra mais pour tous les points de départ i et d'arrivée j.

Matrice d'adjacence :

```
W = array(
    [[INF, 1, 2, INF],
     [1, INF, 2, INF],
     [3, INF, INF,1 ],
     [INF, 2, 1,INF ]])
```

Graphe :



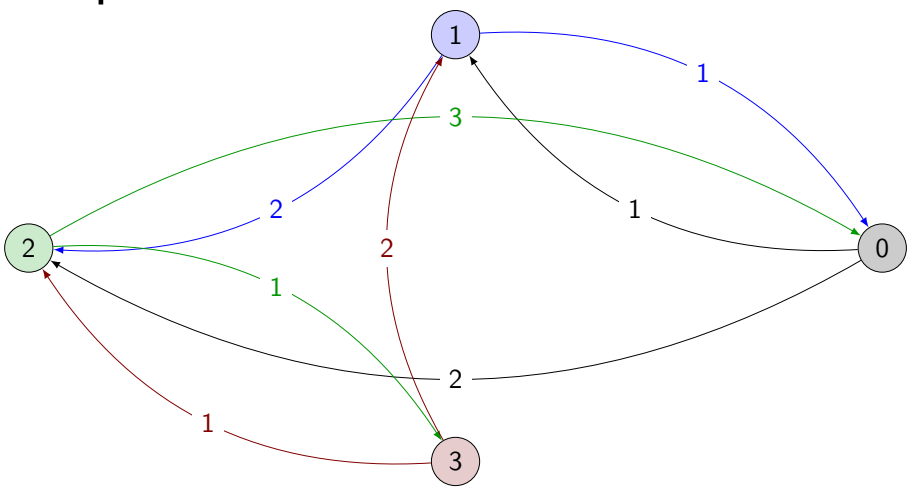
Matrice d'adjacence :

```
W = array(
      [[INF, 1, 2, INF],
       [1, INF, 2, INF],
       [3, INF, INF,1 ],
       [INF, 2, 1,INF ]])
```

Distances minimales :

	0	1	2	3
0	2	1	2	3
1	1	2	2	3
2	3	3	2	1
3	3	2	1	2

Graphe :



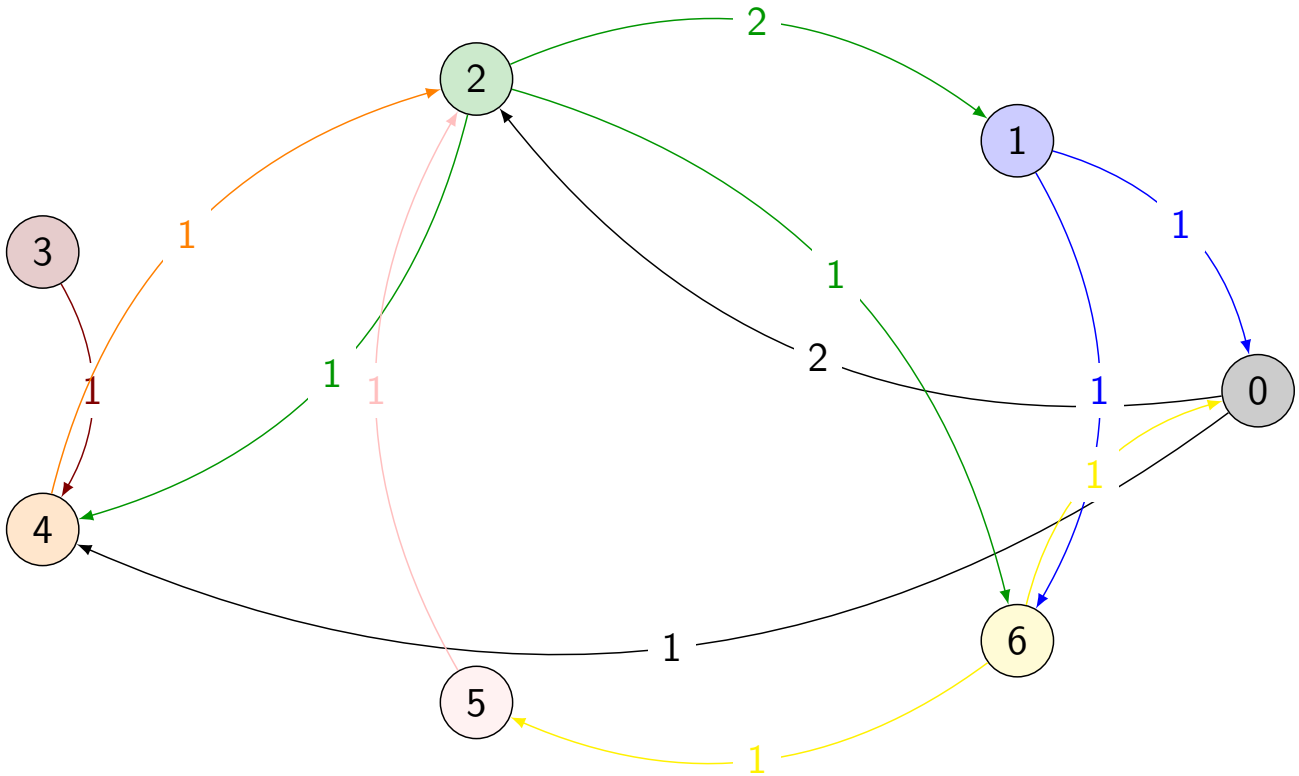
Chemins minimaux :

	0	1	2	3
0	[0, 1, 0]	[0, 1]	[0, 2]	[0, 2, 3]
1	[1, 0]	[1, 0, 1]	[1, 2]	[1, 2, 3]
2	[2, 0]	[2, 3, 1]	[2, 3, 2]	[2, 3]
3	[3, 1, 0]	[3, 1]	[3, 2]	[3, 2, 3]

Matrice d'adjacence :

```
W = array(
    [[INF, INF, 2, INF, 1 , INF, INF],
     [1, INF, INF, INF, INF , INF, 1],
     [INF, 2, INF, INF, 1 , INF, 1],
     [INF, INF, INF, INF, 1 , INF, INF],
     [INF, INF, 1, INF, INF , INF, INF],
     [INF, INF, 1, INF, INF, INF , INF],
     [1, INF, INF, INF, INF, 1 , INF]
     ])
```

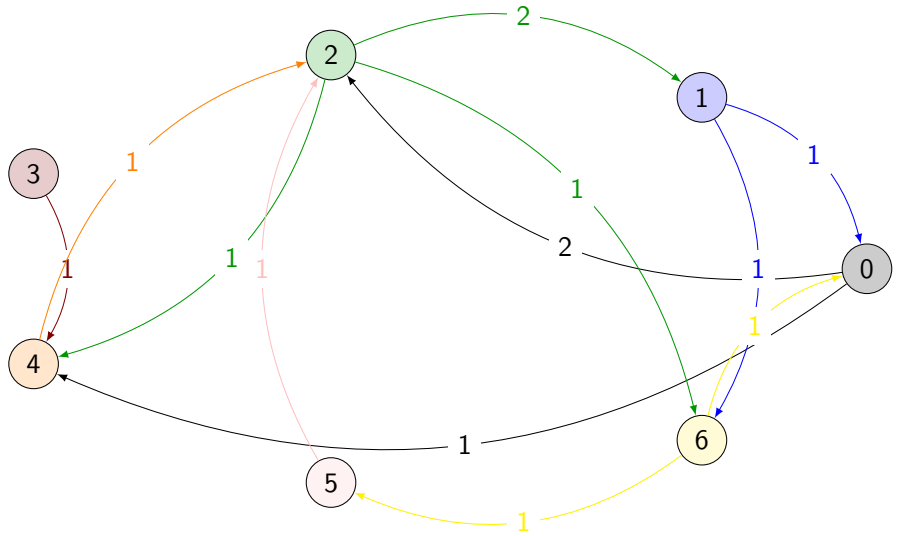
Graphe :



Matrice d'adjacence :

```
W = array(
    [[INF, INF, 2, INF, 1 , INF, INF],
     [1, INF, INF, INF, INF , INF, 1],
     [INF, 2, INF, INF, 1 , INF, 1],
     [INF, INF, INF, INF, 1 , INF, INF],
     [INF, INF, 1, INF, INF , INF, INF],
     [INF, INF, 1, INF, INF, INF , INF],
     [1, INF, INF, INF, INF, 1 , INF]
     ])
```

Graphe :



Distances minimales :

	0	1	2	3	4	5	6
0	4	4	2	1000	1	4	3
1	1	5	3	1000	2	2	1
2	2	2	2	1000	1	2	1
3	4	4	2	1000	1	4	3
4	3	3	1	1000	2	3	2
5	3	3	1	1000	2	3	2
6	1	4	2	1000	2	1	3

Chemins minimaux :

	0	1	2	3	4	5	6
0	[0, 4, 2, 6, 0]	[0, 4, 2, 1]	[0, 4, 2]	[]	[0, 4]	[0, 4, 2, 6, 5]	[0, 4, 2, 6]
1	[1, 0]	[1, 6, 5, 2, 1]	[1, 6, 5, 2]	[]	[1, 0, 4]	[1, 6, 5]	[1, 6]
2	[2, 6, 0]	[2, 1]	[2, 4, 2]	[]	[2, 4]	[2, 6, 5]	[2, 6]
3	[3, 4, 2, 6, 0]	[3, 4, 2, 1]	[3, 4, 2]	[]	[3, 4]	[3, 4, 2, 6, 5]	[3, 4, 2, 6]
4	[4, 2, 6, 0]	[4, 2, 1]	[4, 2]	[]	[4, 2, 4]	[4, 2, 6, 5]	[4, 2, 6]
5	[5, 2, 6, 0]	[5, 2, 1]	[5, 2]	[]	[5, 2, 4]	[5, 2, 6, 5]	[5, 2, 6]
6	[6, 0]	[6, 5, 2, 1]	[6, 5, 2]	[]	[6, 0, 4]	[6, 5]	[6, 5, 2, 6]

- On cherche donc :

$$\mathcal{L}(i,j) = \min_{p \in C_{i,j}} L(p)$$

- On considère un chemin optimal reliant i à j, que l'on note :

$$p = (i_0, \ldots, i_{s-1}, i_s, i_{s+1}, \ldots, i_l)$$

On extrait le sous-chemin $(i_0, \ldots, i_{s-1}, i_s)$, alors ce chemin est optimal pour relier i_0 (ie i) à i_s, car si on pouvait améliorer cette partie du trajet, on pourrait obtenir un meilleur chemin optimal pour relier i à j.

- De même, le sous-chemin $(i_s, i_{s+1}, \ldots, i_l)$ est optimal pour relier i_s à i_l (ie j).
- On voit donc apparaître une structure de sous-problèmes optimaux.

- On cherche donc :

$$\mathcal{L}(i,j) = \min_{p \in C_{i,j}} L(p)$$

- On considère un chemin optimal reliant i à j, que l'on note :

$$p = (i_0, \ldots, i_{s-1}, i_s, i_{s+1}, \ldots, i_l)$$

On extrait le sous-chemin $(i_0, \ldots, i_{s-1}, i_s)$, alors ce chemin est optimal pour relier i_0 (ie i) à i_s, car si on pouvait améliorer cette partie du trajet, on pourrait obtenir un meilleur chemin optimal pour relier i à j.

- De même, le sous-chemin $(i_s, i_{s+1}, \ldots, i_l)$ est optimal pour relier i_s à i_l (ie j).
- On voit donc apparaître une structure de sous-problèmes optimaux.

- On cherche donc :

$$\mathcal{L}(i,j) = \min_{p \in C_{i,j}} L(p)$$

- On considère un chemin optimal reliant i à j, que l'on note :

$$p = (i_0, \ldots, i_{s-1}, i_s, i_{s+1}, \ldots, i_l)$$

  On extrait le sous-chemin $(i_0, \ldots, i_{s-1}, i_s)$, alors ce chemin est optimal pour relier i_0 (ie i) à i_s, car si on pouvait améliorer cette partie du trajet, on pourrait obtenir un meilleur chemin optimal pour relier i à j.
- De même, le sous-chemin $(i_s, i_{s+1}, \ldots, i_l)$ est optimal pour relier i_s à i_l (ie j).
- On voit donc apparaître une structure de sous-problèmes optimaux.

- On cherche donc :

$$\mathcal{L}(i,j) = \min_{p \in C_{i,j}} L(p)$$

- On considère un chemin optimal reliant i à j, que l'on note :

$$p = (i_0, \ldots, i_{s-1}, i_s, i_{s+1}, \ldots, i_l)$$

  On extrait le sous-chemin $(i_0, \ldots, i_{s-1}, i_s)$, alors ce chemin est optimal pour relier i_0 (ie i) à i_s, car si on pouvait améliorer cette partie du trajet, on pourrait obtenir un meilleur chemin optimal pour relier i à j.
- De même, le sous-chemin $(i_s, i_{s+1}, \ldots, i_l)$ est optimal pour relier i_s à i_l (ie j).
- On voit donc apparaître une structure de sous-problèmes optimaux.

- On considère les ensembles $\left(C_{i,j}^k\right)_{(i,j,k)\in[\![0,n-1]\!]^3}$ qui sont définis comme l'ensemble (éventuellement vide) des chemins qui relient les sommets i et j et dont tous les sommets intermédiaires sont dans $[\![0, k-1]\!]$.
- Ainsi, un chemin $p = (i_0, \ldots, i_l)$ est dans $C_{i,j}^k$ lorsque :
$$i = i_0,\ j = i_l,\quad \text{et } \forall s \in [\![1, l-1]\!],\ i_s \in [\![0, k-1]\!]$$
- Par convention : $\left(C_{i,j}^0\right)$ est l'ensemble (éventuellement vide) des arrêtes qui relient i à j.
De plus $\left(C_{i,j}^n\right)$ représente les chemins (quelconques) qui relient i à j.
- On note donc :
$$\mathcal{L}^k(i,j) = \min_{p\in C_{i,j}^k} L(p)$$
Ainsi, $\mathcal{L}^k(i,j)$ est la longueur minimale d'un chemin qui relie i à j en ne passant que par des sommets de $[\![0, k-1]\!]$. Par convention, ce min est égal à $+\infty$ si $C_{i,j}^k$ est vide.
- Ainsi, le problème initial revient à chercher le chemin de longueur minimal dans $\left(C_{i,j}^n\right)$ ie de trouver $\mathcal{L}^n(i,j)$.

- On considère les ensembles $\left(C_{i,j}^k\right)_{(i,j,k)\in[\![0,n-1]\!]^3}$ qui sont définis comme l'ensemble (éventuellement vide) des chemins qui relient les sommets i et j et dont tous les sommets intermédiaires sont dans $[\![0, k-1]\!]$.
- Ainsi, un chemin $p = (i_0, \ldots, i_l)$ est dans $C_{i,j}^k$ lorsque :
$$i = i_0,\ j = i_l,\quad \text{et } \forall s \in [\![1, l-1]\!],\ i_s \in [\![0, k-1]\!]$$
- Par convention : $\left(C_{i,j}^0\right)$ est l'ensemble (éventuellement vide) des arrêtes qui relient i à j.
De plus $\left(C_{i,j}^n\right)$ représente les chemins (quelconques) qui relient i à j.
- On note donc :
$$\mathcal{L}^k(i,j) = \min_{p\in C_{i,j}^k} L(p)$$
Ainsi, $\mathcal{L}^k(i,j)$ est la longueur minimale d'un chemin qui relie i à j en ne passant que par des sommets de $[\![0, k-1]\!]$. Par convention, ce min est égal à $+\infty$ si $C_{i,j}^k$ est vide.
- Ainsi, le problème initial revient à chercher le chemin de longueur minimal dans $\left(C_{i,j}^n\right)$ ie de trouver $\mathcal{L}^n(i,j)$.

- On considère les ensembles $\left(C_{i,j}^k\right)_{(i,j,k)\in[\![0,n-1]\!]^3}$ qui sont définis comme l'ensemble (éventuellement vide) des chemins qui relient les sommets i et j et dont tous les sommets intermédiaires sont dans $[\![0, k-1]\!]$.
- Ainsi, un chemin $p = (i_0, \ldots, i_l)$ est dans $C_{i,j}^k$ lorsque :
$$i = i_0,\ j = i_l,\quad \text{et } \forall s \in [\![1, l-1]\!],\ i_s \in [\![0, k-1]\!]$$
- Par convention : $\left(C_{i,j}^0\right)$ est l'ensemble (éventuellement vide) des arrêtes qui relient i à j.
De plus $\left(C_{i,j}^n\right)$ représente les chemins (quelconques) qui relient i à j.
- On note donc :
$$\mathcal{L}^k(i,j) = \min_{p\in C_{i,j}^k} L(p)$$
Ainsi, $\mathcal{L}^k(i,j)$ est la longueur minimale d'un chemin qui relie i à j en ne passant que par des sommets de $[\![0, k-1]\!]$. Par convention, ce min est égal à $+\infty$ si $C_{i,j}^k$ est vide.
- Ainsi, le problème initial revient à chercher le chemin de longueur minimal dans $\left(C_{i,j}^n\right)$ ie de trouver $\mathcal{L}^n(i,j)$.

- On considère les ensembles $\left(C_{i,j}^k\right)_{(i,j,k)\in[\![0,n-1]\!]^3}$ qui sont définis comme l'ensemble (éventuellement vide) des chemins qui relient les sommets i et j et dont tous les sommets intermédiaires sont dans $[\![0, k-1]\!]$.
- Ainsi, un chemin $p = (i_0, \ldots, i_l)$ est dans $C_{i,j}^k$ lorsque :
$$i = i_0,\ j = i_l,\quad \text{et } \forall s \in [\![1, l-1]\!],\ i_s \in [\![0, k-1]\!]$$
- Par convention : $\left(C_{i,j}^0\right)$ est l'ensemble (éventuellement vide) des arrêtes qui relient i à j.
De plus $\left(C_{i,j}^n\right)$ représente les chemins (quelconques) qui relient i à j.
- On note donc :
$$\mathcal{L}^k(i,j) = \min_{p\in C_{i,j}^k} L(p)$$
Ainsi, $\mathcal{L}^k(i,j)$ est la longueur minimale d'un chemin qui relie i à j en ne passant que par des sommets de $[\![0, k-1]\!]$. Par convention, ce min est égal à $+\infty$ si $C_{i,j}^k$ est vide.
- Ainsi, le problème initial revient à chercher le chemin de longueur minimal dans $\left(C_{i,j}^n\right)$ ie de trouver $\mathcal{L}^n(i,j)$.

- On considère les ensembles $\left(C_{i,j}^k\right)_{(i,j,k)\in[\![0,n-1]\!]^3}$ qui sont définis comme l'ensemble (éventuellement vide) des chemins qui relient les sommets i et j et dont tous les sommets intermédiaires sont dans $[\![0, k-1]\!]$.
- Ainsi, un chemin $p = (i_0, \ldots, i_l)$ est dans $C_{i,j}^k$ lorsque :
$$i = i_0,\ j = i_l,\quad \text{et } \forall s \in [\![1, l-1]\!],\ i_s \in [\![0, k-1]\!]$$
- Par convention : $\left(C_{i,j}^0\right)$ est l'ensemble (éventuellement vide) des arrêtes qui relient i à j.
De plus $\left(C_{i,j}^n\right)$ représente les chemins (quelconques) qui relient i à j.
- On note donc :
$$\mathcal{L}^k(i,j) = \min_{p\in C_{i,j}^k} L(p)$$
Ainsi, $\mathcal{L}^k(i,j)$ est la longueur minimale d'un chemin qui relie i à j en ne passant que par des sommets de $[\![0, k-1]\!]$. Par convention, ce min est égal à $+\infty$ si $C_{i,j}^k$ est vide.
- Ainsi, le problème initial revient à chercher le chemin de longueur minimal dans $\left(C_{i,j}^n\right)$ ie de trouver $\mathcal{L}^n(i,j)$.

Recherche d'une équation de Bellman

Considérons alors $p = (i_0, \ldots, i_l)$ un chemin optimal permettant de relier i à j en ne passant que par des sommets de $[\![0, k-1]\!]$. Ainsi, $p \in C_{i,j}^k$ et la longueur de p est égale à $\mathcal{L}^k(i,j)$.

On a alors deux possibilités :

- Soit $k-1$ n'est pas l'un des sommets intermédiaires
- Soit $k-1$ est l'un des sommets intermédiaires

- Si $k-1$ n'est pas l'un des sommets intermédiaires, p est en fait dans $C_{i,j}^{k-1}$.
- Dans ce cas :

$$\mathcal{L}^k(i,j) = \mathcal{L}^{k-1}(i,j)$$

Recherche d'une équation de Bellman

Considérons alors $p = (i_0, \ldots, i_l)$ un chemin optimal permettant de relier i à j en ne passant que par des sommets de $[\![0, k-1]\!]$. Ainsi, $p \in C_{i,j}^k$ et la longueur de p est égale à $\mathcal{L}^k(i,j)$.

On a alors deux possibilités :

- Soit $k-1$ n'est pas l'un des sommets intermédiaires
- Soit $k-1$ est l'un des sommets intermédiaires

- Si $k-1$ n'est pas l'un des sommets intermédiaires, p est en fait dans $C_{i,j}^{k-1}$.
- Dans ce cas :

$$\mathcal{L}^k(i,j) = \mathcal{L}^{k-1}(i,j)$$

Recherche d'une équation de Bellman

Considérons alors $p = (i_0, \ldots, i_l)$ un chemin optimal permettant de relier i à j en ne passant que par des sommets de $[\![0, k-1]\!]$. Ainsi, $p \in C_{i,j}^k$ et la longueur de p est égale à $\mathcal{L}^k(i,j)$.

On a alors deux possibilités :

- Soit $k-1$ n'est pas l'un des sommets intermédiaires
- Soit $k-1$ est l'un des sommets intermédiaires

- Si $k-1$ n'est pas l'un des sommets intermédiaires, p est en fait dans $C_{i,j}^{k-1}$.
- Dans ce cas :

$$\mathcal{L}^k(i,j) = \mathcal{L}^{k-1}(i,j)$$

Recherche d'une équation de Bellman

Considérons alors $p = (i_0, \ldots, i_l)$ un chemin optimal permettant de relier i à j en ne passant que par des sommets de $[\![0, k-1]\!]$. Ainsi, $p \in C_{i,j}^k$ et la longueur de p est égale à $\mathcal{L}^k(i,j)$.

On a alors deux possibilités :

- Soit $k-1$ n'est pas l'un des sommets intermédiaires
- Soit $k-1$ est l'un des sommets intermédiaires

- Si $k-1$ n'est pas l'un des sommets intermédiaires, p est en fait dans $C_{i,j}^{k-1}$.
- Dans ce cas :

$$\mathcal{L}^k(i,j) = \mathcal{L}^{k-1}(i,j)$$

Considérons alors $p = (i_0, \ldots, i_l)$ un chemin optimal permettant de relier i à j en ne passant que par des sommets de $[\![0, k-1]\!]$. Ainsi, $p \in C_{i,j}^k$ et la longueur de p est égale à $\mathcal{L}^k(i,j)$.

- si $k-1$ est l'un des sommets intermédiaires, alors on peut écrire :

$$p = (i_0, \ldots, i_{s-1}, i_s, i_{s+1}, \ldots, i_l) \text{ avec } i_0 = i,\ i_s = k-1,\ \text{et } i_l = j$$

Le chemin p est ainsi la concaténation de p_1 et p_2, avec :

$$p_1 = (i_0, \ldots, i_{s-1}, i_s)$$
$$\text{et } p_2 = (i_s, i_{s+1}, \ldots, i_l)$$

- p_1 est un chemin qui relie i à $k-1$ et p_2 un chemin qui relie $k-1$ à j.
- p_1 et p_2 sont optimaux et de plus ont leurs sommets intermédiaires dans $[\![0, k-2]\!]$ (ces chemins ne passent pas en cours de route par $k-1$).
  Par définition de la longueur : $L(p) = L(p_1) + L(p_2)$ comme il s'agit de chemin optimaux, cela s'écrit :

$$\mathcal{L}^k(i,j) = \mathcal{L}^{k-1}(i, k-1) + \mathcal{L}^{k-1}(k-1, j)$$

Considérons alors $p = (i_0, \ldots, i_l)$ un chemin optimal permettant de relier i à j en ne passant que par des sommets de $[\![0, k-1]\!]$. Ainsi, $p \in C_{i,j}^k$ et la longueur de p est égale à $\mathcal{L}^k(i,j)$.

- si $k-1$ est l'un des sommets intermédiaires, alors on peut écrire :

$$p = (i_0, \ldots, i_{s-1}, i_s, i_{s+1}, \ldots, i_l) \text{ avec } i_0 = i,\ i_s = k-1,\ \text{et } i_l = j$$

Le chemin p est ainsi la concaténation de p_1 et p_2, avec :

$$p_1 = (i_0, \ldots, i_{s-1}, i_s)$$
$$\text{et } p_2 = (i_s, i_{s+1}, \ldots, i_l)$$

- p_1 est un chemin qui relie i à $k-1$ et p_2 un chemin qui relie $k-1$ à j.
- p_1 et p_2 sont optimaux et de plus ont leurs sommets intermédiaires dans $[\![0, k-2]\!]$ (ces chemins ne passent pas en cours de route par $k-1$).
  Par définition de la longueur : $L(p) = L(p_1) + L(p_2)$ comme il s'agit de chemin optimaux, cela s'écrit :

$$\mathcal{L}^k(i,j) = \mathcal{L}^{k-1}(i, k-1) + \mathcal{L}^{k-1}(k-1, j)$$

Considérons alors $p = (i_0, \ldots, i_l)$ un chemin optimal permettant de relier i à j en ne passant que par des sommets de $[\![0, k-1]\!]$. Ainsi, $p \in C_{i,j}^k$ et la longueur de p est égale à $\mathcal{L}^k(i,j)$.

- si $k-1$ est l'un des sommets intermédiaires, alors on peut écrire :

$$p = (i_0, \ldots, i_{s-1}, i_s, i_{s+1}, \ldots, i_l) \text{ avec } i_0 = i,\ i_s = k-1,\ \text{et } i_l = j$$

Le chemin p est ainsi la concaténation de p_1 et p_2, avec :

$$p_1 = (i_0, \ldots, i_{s-1}, i_s)$$
$$\text{et } p_2 = (i_s, i_{s+1}, \ldots, i_l)$$

- p_1 est un chemin qui relie i à $k-1$ et p_2 un chemin qui relie $k-1$ à j.
- p_1 et p_2 sont optimaux et de plus ont leurs sommets intermédiaires dans $[\![0, k-2]\!]$ (ces chemins ne passent pas en cours de route par $k-1$).
  Par définition de la longueur : $L(p) = L(p_1) + L(p_2)$ comme il s'agit de chemin optimaux, cela s'écrit :

$$\mathcal{L}^k(i,j) = \mathcal{L}^{k-1}(i, k-1) + \mathcal{L}^{k-1}(k-1, j)$$

Considérons alors $p = (i_0, \ldots, i_l)$ un chemin optimal permettant de relier i à j en ne passant que par des sommets de $[\![0, k-1]\!]$. Ainsi, $p \in C_{i,j}^k$ et la longueur de p est égale à $\mathcal{L}^k(i,j)$.

On a alors deux possibilités :

- Soit $k-1$ n'est pas l'un des sommets intermédiaires, et dans ce cas : $\mathcal{L}^k(i,j) = \mathcal{L}^{k-1}(i,j)$
- Soit $k-1$ est l'un des sommets intermédiaires et dans ce cas : $\mathcal{L}^k(i,j) = \mathcal{L}^{k-1}(i,k-1) + \mathcal{L}^{k-1}(k-1,j)$
- Comme l'une et une seules de ces possibilités se produit, cela donne la relation :

$$\mathcal{L}^k(i,j) = \min\left(\mathcal{L}^{k-1}(i,k-1) + \mathcal{L}^{k-1}(k-1,j), \mathcal{L}^{k-1}(i,j)\right)$$

- Précisément, si c'est $\mathcal{L}^{k-1}(i,j)$ qui est inférieur, on n'a aucun intérêt à passer par $k-1$, sinon il faut concaténer un chemin minimal de $C_{i,k-1}^{k-1}$ (qui relie i et $k-1$) et un chemin minimal de $C_{k-1,j}^{k-1}$ (qui relie $k-1$ à j).

Considérons alors $p = (i_0, \ldots, i_l)$ un chemin optimal permettant de relier i à j en ne passant que par des sommets de $[\![0, k-1]\!]$. Ainsi, $p \in C_{i,j}^k$ et la longueur de p est égale à $\mathcal{L}^k(i,j)$.

On a alors deux possibilités :

- Soit $k-1$ n'est pas l'un des sommets intermédiaires, et dans ce cas : $\mathcal{L}^k(i,j) = \mathcal{L}^{k-1}(i,j)$
- Soit $k-1$ est l'un des sommets intermédiaires et dans ce cas : $\mathcal{L}^k(i,j) = \mathcal{L}^{k-1}(i,k-1) + \mathcal{L}^{k-1}(k-1,j)$
- Comme l'une et une seules de ces possibilités se produit, cela donne la relation :

$$\mathcal{L}^k(i,j) = \min\left(\mathcal{L}^{k-1}(i,k-1) + \mathcal{L}^{k-1}(k-1,j), \mathcal{L}^{k-1}(i,j)\right)$$

- Précisément, si c'est $\mathcal{L}^{k-1}(i,j)$ qui est inférieur, on n'a aucun intérêt à passer par $k-1$, sinon il faut concaténer un chemin minimal de $C_{i,k-1}^{k-1}$ (qui relie i et $k-1$) et un chemin minimal de $C_{k-1,j}^{k-1}$ (qui relie $k-1$ à j).

Considérons alors $p = (i_0, \ldots, i_l)$ un chemin optimal permettant de relier i à j en ne passant que par des sommets de $[\![0, k-1]\!]$. Ainsi, $p \in C_{i,j}^k$ et la longueur de p est égale à $\mathcal{L}^k(i,j)$.

On a alors deux possibilités :

- Soit $k-1$ n'est pas l'un des sommets intermédiaires, et dans ce cas : $\mathcal{L}^k(i,j) = \mathcal{L}^{k-1}(i,j)$
- Soit $k-1$ est l'un des sommets intermédiaires et dans ce cas : $\mathcal{L}^k(i,j) = \mathcal{L}^{k-1}(i,k-1) + \mathcal{L}^{k-1}(k-1,j)$
- Comme l'une et une seules de ces possibilités se produit, cela donne la relation :

$$\mathcal{L}^k(i,j) = \min\left(\mathcal{L}^{k-1}(i,k-1) + \mathcal{L}^{k-1}(k-1,j), \mathcal{L}^{k-1}(i,j)\right)$$

- Précisément, si c'est $\mathcal{L}^{k-1}(i,j)$ qui est inférieur, on n'a aucun intérêt à passer par $k-1$, sinon il faut concaténer un chemin minimal de $C_{i,k-1}^{k-1}$ (qui relie i et $k-1$) et un chemin minimal de $C_{k-1,j}^{k-1}$ (qui relie $k-1$ à j).

Considérons alors $p = (i_0, \ldots, i_l)$ un chemin optimal permettant de relier i à j en ne passant que par des sommets de $[\![0, k-1]\!]$. Ainsi, $p \in C_{i,j}^k$ et la longueur de p est égale à $\mathcal{L}^k(i,j)$.

On a alors deux possibilités :

- Soit $k-1$ n'est pas l'un des sommets intermédiaires, et dans ce cas : $\mathcal{L}^k(i,j) = \mathcal{L}^{k-1}(i,j)$
- Soit $k-1$ est l'un des sommets intermédiaires et dans ce cas : $\mathcal{L}^k(i,j) = \mathcal{L}^{k-1}(i,k-1) + \mathcal{L}^{k-1}(k-1,j)$
- Comme l'une et une seules de ces possibilités se produit, cela donne la relation :

$$\mathcal{L}^k(i,j) = \min\left(\mathcal{L}^{k-1}(i,k-1) + \mathcal{L}^{k-1}(k-1,j), \mathcal{L}^{k-1}(i,j)\right)$$

- Précisément, si c'est $\mathcal{L}^{k-1}(i,j)$ qui est inférieur, on n'a aucun intérêt à passer par $k-1$, sinon il faut concaténer un chemin minimal de $C_{i,k-1}^{k-1}$ (qui relie i et $k-1$) et un chemin minimal de $C_{k-1,j}^{k-1}$ (qui relie $k-1$ à j).

Équation de Bellman

- $\mathcal{L}^k(i,j)$ est la longueur du chemin minimal qui relie i à j et dont tous les sommets intermédiaires sont dans $[\![0, k-1]\!]$.
- Si $k = 0$, alors
  - $\mathcal{L}^0(i,j) = +\infty$ si i et j ne sont pas reliés,
  - et $\mathcal{L}^0(i,j) = W_{i,j}$ si i et j sont reliés.
- On a un moyen de calculer les $\mathcal{L}^k(i,j)$

$$\mathcal{L}^k(i,j) = \min\left(\mathcal{L}^{k-1}(i,k-1) + \mathcal{L}^{k-1}(k-1,j), \mathcal{L}^{k-1}(i,j)\right)$$

- On peut calculer les chemins minimaux : si c'est $\mathcal{L}^{k-1}(i,j)$ qui est inférieur, on n'a aucun intérêt à passer par $k-1$, sinon il faut concaténer un chemin minimal de $C^{k-1}_{i,k-1}$ et un chemin minimal de $C^{k-1}_{k-1,j}$.

Équation de Bellman

- $\mathcal{L}^k(i,j)$ est la longueur du chemin minimal qui relie i à j et dont tous les sommets intermédiaires sont dans $[\![0, k-1]\!]$.
- Si $k = 0$, alors
  - $\mathcal{L}^0(i,j) = +\infty$ si i et j ne sont pas reliés,
  - et $\mathcal{L}^0(i,j) = W_{i,j}$ si i et j sont reliés.
- On a un moyen de calculer les $\mathcal{L}^k(i,j)$

$$\mathcal{L}^k(i,j) = \min\left(\mathcal{L}^{k-1}(i,k-1) + \mathcal{L}^{k-1}(k-1,j), \mathcal{L}^{k-1}(i,j)\right)$$

- On peut calculer les chemins minimaux : si c'est $\mathcal{L}^{k-1}(i,j)$ qui est inférieur, on n'a aucun intérêt à passer par $k-1$, sinon il faut concaténer un chemin minimal de $C^{k-1}_{i,k-1}$ et un chemin minimal de $C^{k-1}_{k-1,j}$.

Équation de Bellman

- $\mathcal{L}^k(i,j)$ est la longueur du chemin minimal qui relie i à j et dont tous les sommets intermédiaires sont dans $[\![0, k-1]\!]$.
- Si $k = 0$, alors
  - $\mathcal{L}^0(i,j) = +\infty$ si i et j ne sont pas reliés,
  - et $\mathcal{L}^0(i,j) = W_{i,j}$ si i et j sont reliés.
- On a un moyen de calculer les $\mathcal{L}^k(i,j)$

$$\mathcal{L}^k(i,j) = \min\left(\mathcal{L}^{k-1}(i,k-1) + \mathcal{L}^{k-1}(k-1,j), \mathcal{L}^{k-1}(i,j)\right)$$

- On peut calculer les chemins minimaux : si c'est $\mathcal{L}^{k-1}(i,j)$ qui est inférieur, on n'a aucun intérêt à passer par $k-1$, sinon il faut concaténer un chemin minimal de $C^{k-1}_{i,k-1}$ et un chemin minimal de $C^{k-1}_{k-1,j}$.

Équation de Bellman

- $\mathcal{L}^k(i,j)$ est la longueur du chemin minimal qui relie i à j et dont tous les sommets intermédiaires sont dans $[\![0, k-1]\!]$.
- Si $k = 0$, alors
  - $\mathcal{L}^0(i,j) = +\infty$ si i et j ne sont pas reliés,
  - et $\mathcal{L}^0(i,j) = W_{i,j}$ si i et j sont reliés.
- On a un moyen de calculer les $\mathcal{L}^k(i,j)$

$$\mathcal{L}^k(i,j) = \min\left(\mathcal{L}^{k-1}(i,k-1) + \mathcal{L}^{k-1}(k-1,j), \mathcal{L}^{k-1}(i,j)\right)$$

- On peut calculer les chemins minimaux : si c'est $\mathcal{L}^{k-1}(i,j)$ qui est inférieur, on n'a aucun intérêt à passer par $k-1$, sinon il faut concaténer un chemin minimal de $C^{k-1}_{i,k-1}$ et un chemin minimal de $C^{k-1}_{k-1,j}$.

Équation de Bellman

- $\mathcal{L}^k(i,j)$ est la longueur du chemin minimal qui relie i à j et dont tous les sommets intermédiaires sont dans $[\![0, k-1]\!]$.
- Si $k = 0$, alors
  - $\mathcal{L}^0(i,j) = +\infty$ si i et j ne sont pas reliés,
  - et $\mathcal{L}^0(i,j) = W_{i,j}$ si i et j sont reliés.
- On a un moyen de calculer les $\mathcal{L}^k(i,j)$

$$\mathcal{L}^k(i,j) = \min\left(\mathcal{L}^{k-1}(i,k-1) + \mathcal{L}^{k-1}(k-1,j), \mathcal{L}^{k-1}(i,j)\right)$$

- On peut calculer les chemins minimaux : si c'est $\mathcal{L}^{k-1}(i,j)$ qui est inférieur, on n'a aucun intérêt à passer par $k-1$, sinon il faut concaténer un chemin minimal de $C^{k-1}_{i,k-1}$ et un chemin minimal de $C^{k-1}_{k-1,j}$.

Équation de Bellman

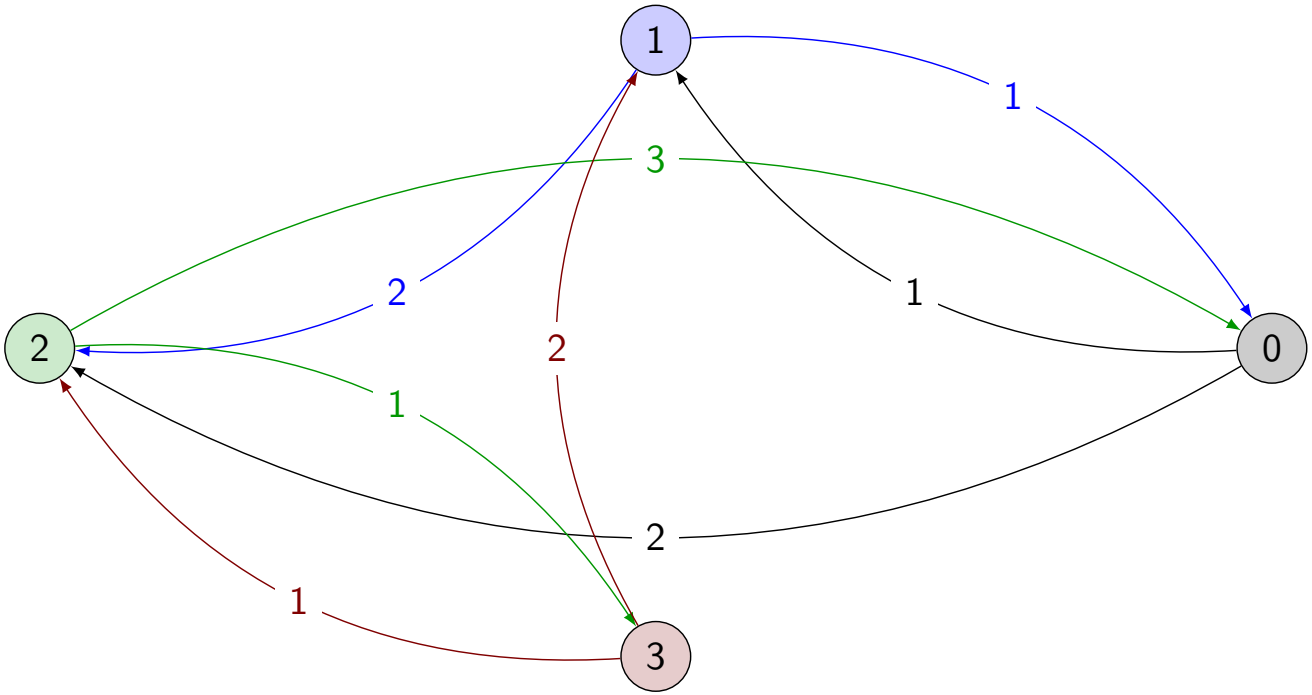
- $\mathcal{L}^k(i,j)$ est la longueur du chemin minimal qui relie i à j et dont tous les sommets intermédiaires sont dans $[\![0, k-1]\!]$.
- Si $k = 0$, alors
  - $\mathcal{L}^0(i,j) = +\infty$ si i et j ne sont pas reliés,
  - et $\mathcal{L}^0(i,j) = W_{i,j}$ si i et j sont reliés.
- On a un moyen de calculer les $\mathcal{L}^k(i,j)$

$$\mathcal{L}^k(i,j) = \min\left(\mathcal{L}^{k-1}(i,k-1) + \mathcal{L}^{k-1}(k-1,j), \mathcal{L}^{k-1}(i,j)\right)$$

- On peut calculer les chemins minimaux : si c'est $\mathcal{L}^{k-1}(i,j)$ qui est inférieur, on n'a aucun intérêt à passer par $k-1$, sinon il faut concaténer un chemin minimal de $C^{k-1}_{i,k-1}$ et un chemin minimal de $C^{k-1}_{k-1,j}$.

Application de la formule de Bellman

$$\mathcal{L}^0(i,j) = W_{i,j} + \text{ ou } +\infty$$
$$\mathcal{L}^k(i,j) = \min\left(\mathcal{L}^{k-1}(i,k-1) + \mathcal{L}^{k-1}(k-1,j), \mathcal{L}^{k-1}(i,j)\right)$$



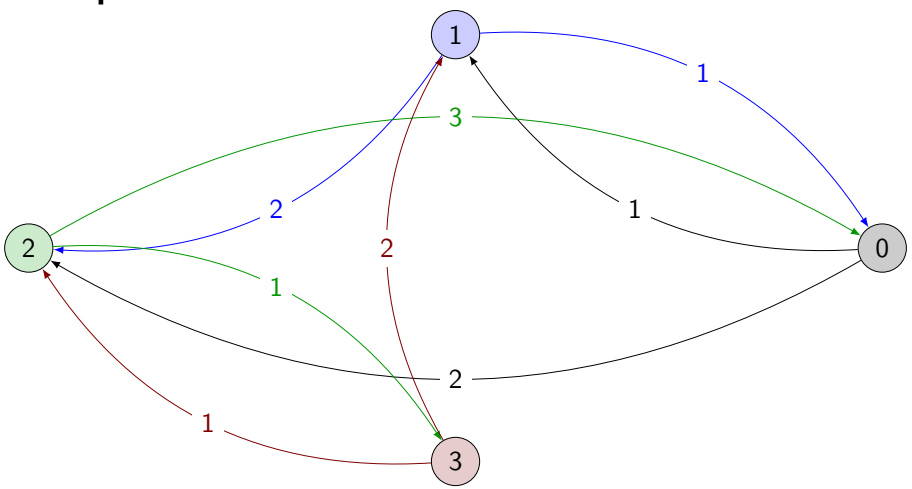
Matrice d'adjacence :

```
W = array(
      [[INF, 1, 2, INF],
       [1, INF, 2, INF],
       [3, INF, INF,1 ],
       [INF, 2, 1,INF ]])
```

Distances minimales :

	0	1	2	3
0	2	1	2	3
1	1	2	2	3
2	3	3	2	1
3	3	2	1	2

Graphe :



Chemins minimaux :

	0	1	2	3
0	[0, 1, 0]	[0, 1]	[0, 2]	[0, 2, 3]
1	[1, 0]	[1, 0, 1]	[1, 2]	[1, 2, 3]
2	[2, 0]	[2, 3, 1]	[2, 3, 2]	[2, 3]
3	[3, 1, 0]	[3, 1]	[3, 2]	[3, 2, 3]

La suite en TP !