Algorithmique et Complexité
Cours 5/7 : Programmation Dynamique

CentraleSupélec – Gif

ST2 – Gif

Plan

Rendu de monnaie

Rendu de monnaie

Rendre 3,57 EUR à l'aide de pièces de 1 et 2 euros et de 1, 2, 5, 10, 20 et 50 centimes.

Rendu de monnaie

Rendu de monnaie

Rendre 3,57 EUR à l'aide de pièces de 1 et 2 euros et de 1, 2, 5, 10, 20 et 50 centimes.

Solutions

Un ensemble de pièces :

- ➜ $2€ + 1€ + 50¢ + 5¢ + 2¢$

Rendu de monnaie

Rendu de monnaie

Rendre 3,57 EUR à l'aide de pièces de 1 et 2 euros et de 1, 2, 5, 10, 20 et 50 centimes.

Solutions

Un ensemble de pièces :

- ✓ $2€ + 1€ + 50¢ + 5¢ + 2¢$
- ➜ $1€ + 1€ + 50¢ + 4 \times 20¢ + 2 \times 10¢ + 3 \times 2¢ + 1¢$

Rendu de monnaie

Rendu de monnaie

Rendre 3,57 EUR à l'aide de pièces de 1 et 2 euros et de 1, 2, 5, 10, 20 et 50 centimes.

Solutions

Un ensemble de pièces :

- ✓ $2€ + 1€ + 50¢ + 5¢ + 2¢$
- ✓ $1€ + 1€ + 50¢ + 4\times20¢ + 2\times10¢ + 3\times2¢ + 1¢$

Rendu de monnaie

Rendu de monnaie

Rendre 3,57 EUR à l'aide de pièces de 1 et 2 euros et de 1, 2, 5, 10, 20 et 50 centimes.

Solutions

Un ensemble de pièces :

- ✓ $2€ + 1€ + 50¢ + 5¢ + 2¢$
- ✓ $1€ + 1€ + 50¢ + 4\times20¢ + 2\times10¢ + 3\times2¢ + 1¢$
- ✓ $357\times1¢$

Rendu de monnaie

Rendu de monnaie

Rendre 3,57 EUR à l'aide de pièces de 1 et 2 euros et de 1, 2, 5, 10, 20 et 50 centimes.

Solutions

Un ensemble de pièces :

- ✓ $2€ + 1€ + 50¢ + 5¢ + 2¢$
- ✓ $1€ + 1€ + 50¢ + 4\times20¢ + 2\times10¢ + 3\times2¢ + 1¢$
- ✓ $357\times1¢$
- ➜ Avec combien de pièces au minimum peut on rendre 3,57€ ?

Problème d'optimisation

Données en entrée

- $S \in \mathbb{N}^{+n}$ un n-uplet de pièces : $S = (200, 100, 50, 20, 10, 5, 2, 1)$
- $total \in \mathbb{N}$ la somme à rendre : $total = 357$

Données en sortie

c le nombre de pièces utilisées pour obtenir la valeur $total$,
tel qu'il existe $L \in \mathbb{N}^n$ un n-uplet vérifiant :

- $total = \sum_{i=0}^{n-1} L_i \times S_i$ *L indique le nombre de chaque pièce*

Problème d'optimisation

Données en entrée

- $S \in \mathbb{N}^{+n}$ un n-uplet de pièces : $S = (200, 100, 50, 20, 10, 5, 2, 1)$
- $total \in \mathbb{N}$ la somme à rendre : $total = 357$

Données en sortie

c le nombre de pièces utilisées pour obtenir la valeur $total$,
tel qu'il existe $L \in \mathbb{N}^n$ un n-uplet vérifiant :

- $total = \sum_{i=0}^{n-1} L_i \times S_i$ *L indique le nombre de chaque pièce*
- $c = \sum_{i=0}^{n-1} L_i$ *c est le coût de la solution L*

Problème d'optimisation

Données en entrée

- $S \in \mathbb{N}^{+n}$ un n-uplet de pièces : $S = (200, 100, 50, 20, 10, 5, 2, 1)$
- $total \in \mathbb{N}$ la somme à rendre : $total = 357$

Données en sortie

c le nombre de pièces utilisées pour obtenir la valeur $total$,
tel qu'il existe $L \in \mathbb{N}^n$ un n-uplet vérifiant :

- $total = \sum_{i=0}^{n-1} L_i \times S_i$ *L indique le nombre de chaque pièce*
- $c = \sum_{i=0}^{n-1} L_i$ *c est le coût de la solution L*
- et c est minimal !

Exemple

$L = (0, 2, 1, 4, 2, 0, 3, 1) \rightarrow c = 13$ ➜ pas minimal !

Quelques observations...

Solution

Pièces de valeur unitaire $\rightarrow$ au moins une solution $\forall total \in \mathbb{N}$

On suppose une infinité de pièces pour chacune des valeurs...

Taille du problème ?

Quelques observations...

Solution

Pièces de valeur unitaire $\rightarrow$ au moins une solution $\forall total \in \mathbb{N}$

On suppose une infinité de pièces pour chacune des valeurs...

Taille du problème

n (le nombre des valeurs de pièces différentes)

➜ La somme à rendre est un **paramètre** du problème

Quelques observations...

Solution

Pièces de valeur unitaire $\rightarrow$ au moins une solution $\forall total \in \mathbb{N}$

On suppose une infinité de pièces pour chacune des valeurs...

Taille du problème

n (le nombre des valeurs de pièces différentes)

➜ La somme à rendre est un **paramètre** du problème

Objectif

✓ Rendre la somme due $\rightarrow$ une solution

✓ Avec un nombre minimum de pièces $\rightarrow$ la solution optimale !

Quelques observations...

Solution

Pièces de valeur unitaire $\rightarrow$ au moins une solution $\forall total \in \mathbb{N}$

On suppose une infinité de pièces pour chacune des valeurs...

Taille du problème

n (le nombre des valeurs de pièces différentes)

➜ La somme à rendre est un **paramètre** du problème

Objectif

✓ Rendre la somme due $\rightarrow$ une solution

✓ Avec un nombre minimum de pièces $\rightarrow$ la solution optimale !

Solution optimale

On cherche le coût de la solution optimale !

Algorithme récursif

Pour rendre 3,57€, je peux rendre :

- 1 pièce de 2 € puis 1,57 ;
- 1 pièce de 1 € puis 2,57 ;
- 1 pièce de 50 ¢ puis 3,07 ;
- *etc. pour chaque valeur de pièce possible*

Algorithme récursif

Pour rendre 3,57€, je peux rendre :

- 1 pièce de 2 € puis 1,57 ;
- 1 pièce de 1 € puis 2,57 ;
- 1 pièce de 50 ¢ puis 3,07 ;
- *etc.* *pour chaque valeur de pièce possible*

➜ La meilleure solution est alors :

$$1 + min(rendu(2,57), rendu(1,57), \ldots)$$

Algorithme récursif

Pour rendre 3,57€, je peux rendre :

- 1 pièce de 2 € puis 1,57 ;
- 1 pièce de 1 € puis 2,57 ;
- 1 pièce de 50 ¢ puis 3,07 ;
- *etc. pour chaque valeur de pièce possible*

➜ La meilleure solution est alors :

$$1 + min(rendu(2,57), rendu(1,57), \ldots)$$

Calcul récursif du coût de la solution optimale

Notons $C(s)$ le nombre minimum de pièces pour obtenir s.

Algorithme récursif

Pour rendre 3,57€, je peux rendre :

- 1 pièce de 2 € puis 1,57 ;
- 1 pièce de 1 € puis 2,57 ;
- 1 pièce de 50 ¢ puis 3,07 ;
- *etc. pour chaque valeur de pièce possible*

➜ La meilleure solution est alors :

$$1 + min(rendu(2,57), rendu(1,57), \ldots)$$

Calcul récursif du coût de la solution optimale

Notons $C(s)$ le nombre minimum de pièces pour obtenir s.

- Cas de base : $C(0) = 0$

Algorithme récursif

Pour rendre 3,57€, je peux rendre :

- 1 pièce de 2 € puis 1,57 ;
- 1 pièce de 1 € puis 2,57 ;
- 1 pièce de 50 ¢ puis 3,07 ;
- *etc. pour chaque valeur de pièce possible*

➜ La meilleure solution est alors :

$$1 + min(rendu(2,57), rendu(1,57), \ldots)$$

Calcul récursif du coût de la solution optimale

Notons $C(s)$ le nombre minimum de pièces pour obtenir s.

- Cas de base : $C(0) = 0$
- Cas général : $C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i)$

Approche récursive

$$\begin{cases} C(0) = 0 \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$

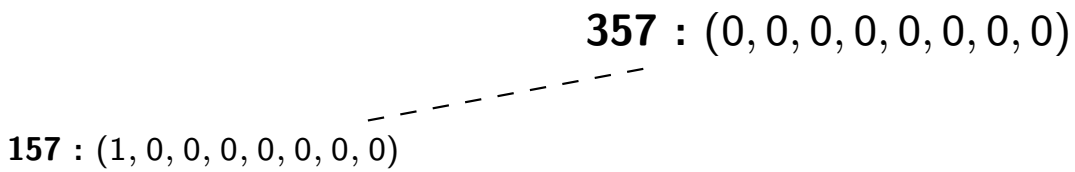
Approche récursive

$$\begin{cases} C(0) = 0 \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$

357 : $(0, 0, 0, 0, 0, 0, 0, 0)$

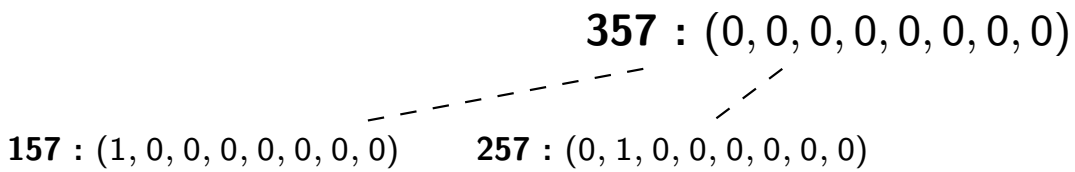
Approche récursive

$$\begin{cases} C(0) = 0 \\ C(s) = 1 + \min_{i \in [0, n-1], S_i \leq s} C(s - S_i) \end{cases}$$



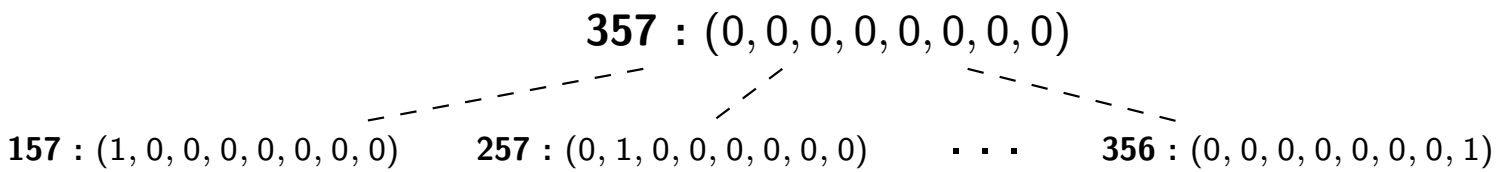
Approche récursive

$$\begin{cases} C(0) = 0 \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$



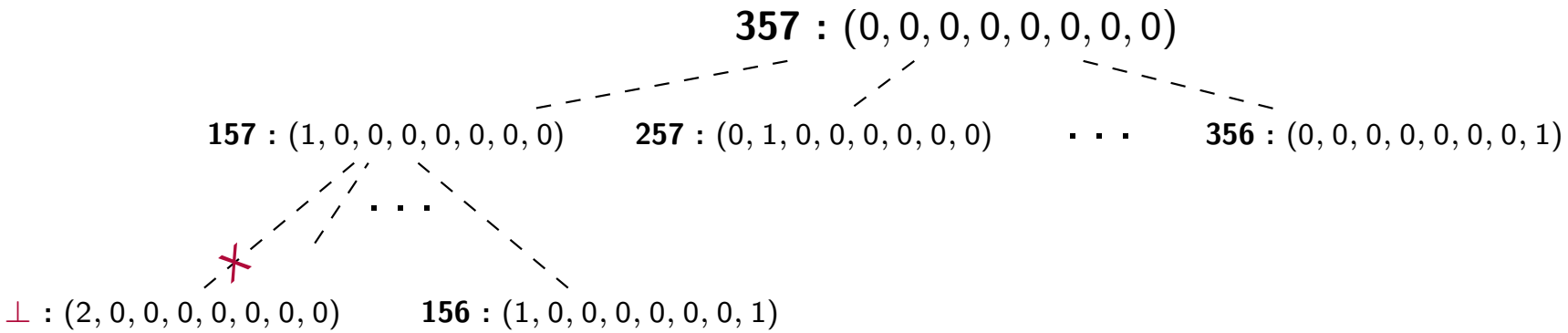
Approche récursive

$$\begin{cases} C(0) = 0 \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$



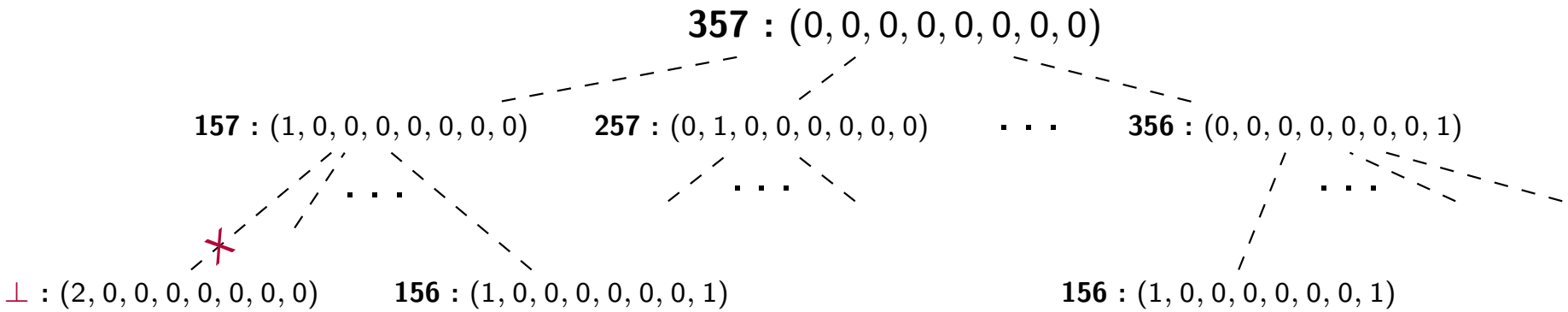
Approche récursive

$$\begin{cases} C(0) = 0 \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$



Approche récursive

$$\begin{cases} C(0) = 0 \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$



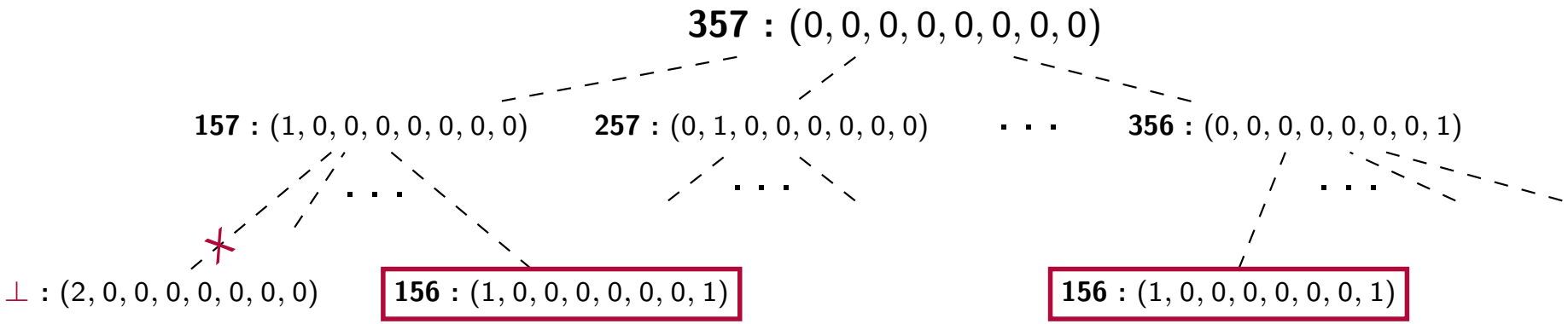
Complexité exponentielle !

- Arbre d'exploration n-aire, non-équilibré, de profondeur entre $[\frac{total}{S_0}, \frac{total}{S_{n-1}}]$
- Complexité supérieure à n^k avec $k = \frac{total}{S_0}$.

mais. . .

Approche récursive

$$\begin{cases} C(0) = 0 \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$



Complexité exponentielle !

- Arbre d'exploration n-aire, non-équilibré, de profondeur entre $[\frac{total}{S_0}, \frac{total}{S_{n-1}}]$
- Complexité supérieure à n^k avec $k = \frac{total}{S_0}$.
- Calculs redondants !

Programmation dynamique

Idée

- ✓ Résoudre des problèmes d'optimisation
- ✓ Dans lesquels il y a une construction récursive de la solution

Programmation dynamique

Idée

- ✓ Résoudre des problèmes d'optimisation
- ✓ Dans lesquels il y a une construction récursive de la solution
- ➜ **Programmation Dynamique**

Principe

- Stocker les solutions intermédiaires pour ne pas les recalculer
- Inventée par Bellman dans les années 50
- S'applique quand la solution optimale du problème est composée des solutions optimales de ses sous-problèmes

Plan

Programmation dynamique

Principe

- Par récurrence : résoudre les plus petits sous-problèmes, continuer avec des problèmes de plus en plus grands, jusqu'à obtenir la solution du problème global.

Programmation dynamique

Principe

- Par récurrence : résoudre les plus petits sous-problèmes, continuer avec des problèmes de plus en plus grands, jusqu'à obtenir la solution du problème global.
- Conserver les solutions des sous-problèmes dans une table

pour éviter les calculs redondants qui rendent la solution récursive inefficace

Programmation dynamique

Principe

- Par récurrence : résoudre les plus petits sous-problèmes, continuer avec des problèmes de plus en plus grands, jusqu'à obtenir la solution du problème global.
- Conserver les solutions des sous-problèmes dans une table

pour éviter les calculs redondants qui rendent la solution récursive inefficace

Exemple : rendu de monnaie

On itère de $s = 0$ à $s = total$

Programmation dynamique

Principe

- Par récurrence : résoudre les plus petits sous-problèmes, continuer avec des problèmes de plus en plus grands, jusqu'à obtenir la solution du problème global.
- Conserver les solutions des sous-problèmes dans une table

pour éviter les calculs redondants qui rendent la solution récursive inefficace

Exemple : rendu de monnaie

On itère de $s = 0$ à $s = total$

- Comme on sait rendre la monnaie pour tout $s' < s$:
  $\rightarrow$ on calcule le coût min de s :

  $C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i)$

Programmation dynamique

Principe

- Par récurrence : résoudre les plus petits sous-problèmes, continuer avec des problèmes de plus en plus grands, jusqu'à obtenir la solution du problème global.
- Conserver les solutions des sous-problèmes dans une table

pour éviter les calculs redondants qui rendent la solution récursive inefficace

Exemple : rendu de monnaie

On itère de $s = 0$ à $s = total$

- Comme on sait rendre la monnaie pour tout $s' < s$:
  $\rightarrow$ on calcule le coût min de s :
  $C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i)$
- On mémorise le résultat dans le tableau
  $\rightarrow$ ultérieurement nécessaire au calcul du coût min des $s + S_i$

Programmation dynamique

Sous-structure optimale

1. Diviser le problème en sous-problèmes.
2. Construire la solution optimale à partir des solutions optimales des sous-problèmes.
3. Déduire une formule de récurrence.

Programmation dynamique

Sous-structure optimale

1. Diviser le problème en sous-problèmes.
2. Construire la solution optimale à partir des solutions optimales des sous-problèmes.
3. Déduire une formule de récurrence.

Exemples d'applications

- Alignement de séquences
- Rendu de monnaie
- Plus court chemin
- Sac à dos

Programmation dynamique vs Diviser pour régner

En commun

Les deux méthodes nécessitent une sous-structure optimale (une formule de récurrence)

Programmation dynamique vs Diviser pour régner

En commun

Les deux méthodes nécessitent une sous-structure optimale (une formule de récurrence)

- Si les sous-problèmes sont indépendants (tous les sous-problèmes sont différents)
  - Programmation dynamique inutile
  - exemple classique : $fact(n+1) = (n+1) \times fact(n)$

Programmation dynamique vs Diviser pour régner

En commun

Les deux méthodes nécessitent une sous-structure optimale (une formule de récurrence)

- Si les sous-problèmes sont indépendants (tous les sous-problèmes sont différents)
  - Programmation dynamique inutile
  - exemple classique : $fact(n+1) = (n+1) \times fact(n)$
- sinon
  - Programmation dynamique plus performante en temps *(en contrepartie, on paie en espace car rien n'est gratuit !)*
  - exemple classique : $fib(n+2) = fib(n+1) + fib(n)$

Programmation dynamique vs Diviser pour régner

- Diviser pour régner (approche récursive)

```
def fib(n):
  if n==1 or n==2:
    return 1
  return fib(n-1)+fib(n-2)
```

  - complexité exponentielle $\mathcal{O}(\phi^n)$ (ϕ le nombre d'or)

Programmation dynamique vs Diviser pour régner

- Diviser pour régner (approche récursive)

```
def fib(n):
  if n==1 or n==2:
    return 1
  return fib(n-1)+fib(n-2)
```

  - complexité exponentielle $\mathcal{O}(\phi^n)$ (ϕ le nombre d'or)

- Programmation dynamique

```
table = {0:0, 1:1}
def fib(n):
  if not n in table:
    table[n] = fib(n-1) + fib(n-2)
  return table[n]
```

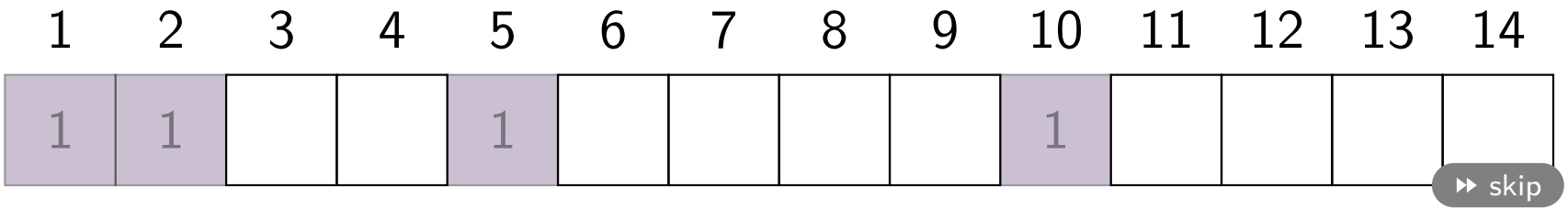
  - complexité linéaire $\mathcal{O}(n)$

Résolution par programmation dynamique (Algorithme 1)

- On réutilise la formule de récurrence précédente tout en mémorisant les résultats intermédiaires :

$$\begin{cases} C(s) = 1 & \text{si } \exists i \in [0, n-1] \text{ tel que } s = S_i \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$

- Soit $S = (10, 5, 2, 1)$ et $total = 14$

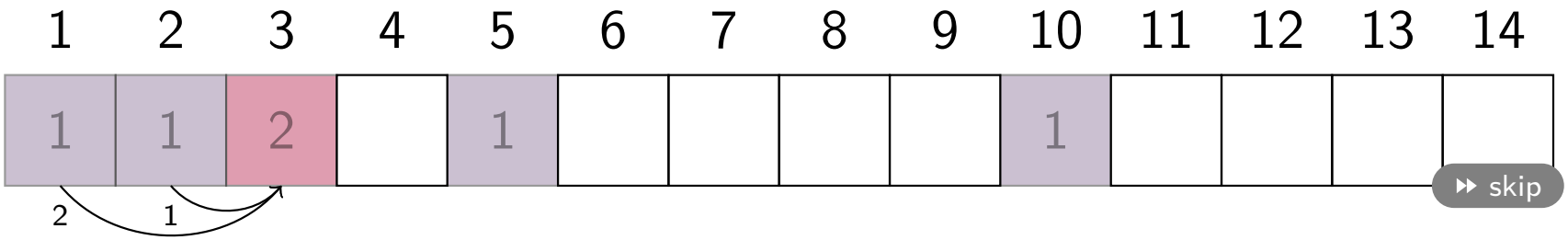


Résolution par programmation dynamique (Algorithme 1)

- On réutilise la formule de récurrence précédente tout en mémorisant les résultats intermédiaires :

$$\begin{cases} C(s) = 1 & \text{si } \exists i \in [0, n-1] \text{ tel que } s = S_i \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$

- Soit $S = (10, 5, 2, 1)$ et $total = 14$

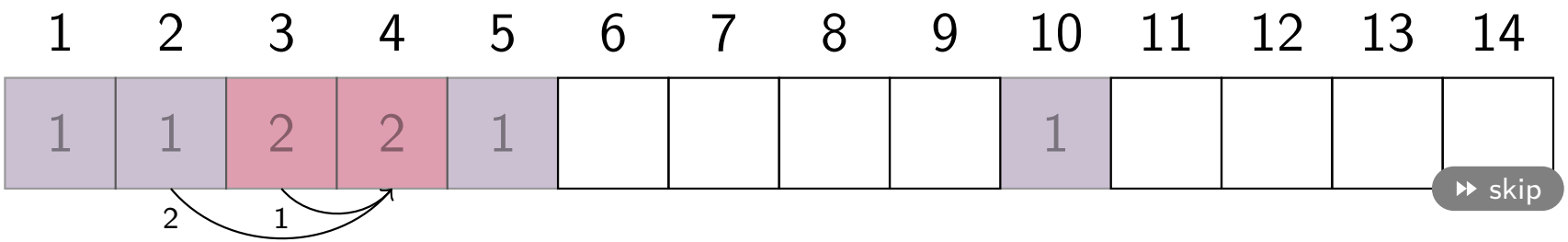


Résolution par programmation dynamique (Algorithme 1)

- On réutilise la formule de récurrence précédente tout en mémorisant les résultats intermédiaires :

$$\begin{cases} C(s) = 1 & \text{si } \exists i \in [0, n-1] \text{ tel que } s = S_i \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$

- Soit $S = (10, 5, 2, 1)$ et $total = 14$

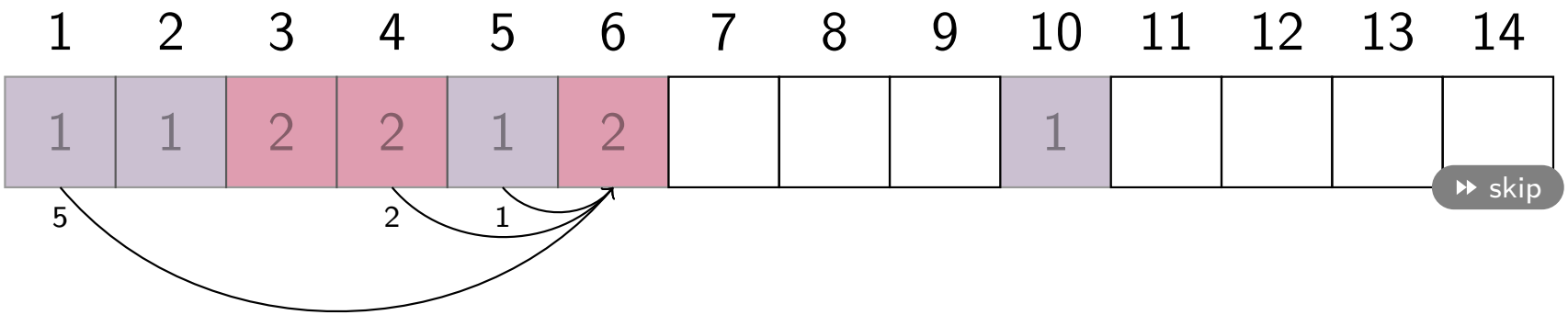


Résolution par programmation dynamique (Algorithme 1)

- On réutilise la formule de récurrence précédente tout en mémorisant les résultats intermédiaires :

$$\begin{cases} C(s) = 1 & \text{si } \exists i \in [0, n-1] \text{ tel que } s = S_i \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$

- Soit $S = (10, 5, 2, 1)$ et $total = 14$

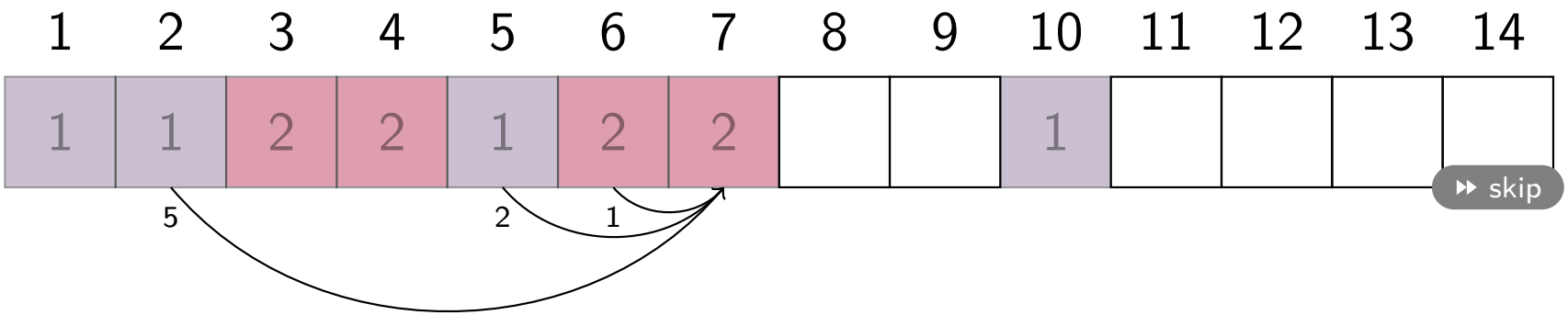


Résolution par programmation dynamique (Algorithme 1)

- On réutilise la formule de récurrence précédente tout en mémorisant les résultats intermédiaires :

$$\begin{cases} C(s) = 1 & \text{si } \exists i \in [0, n-1] \text{ tel que } s = S_i \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$

- Soit $S = (10, 5, 2, 1)$ et $total = 14$

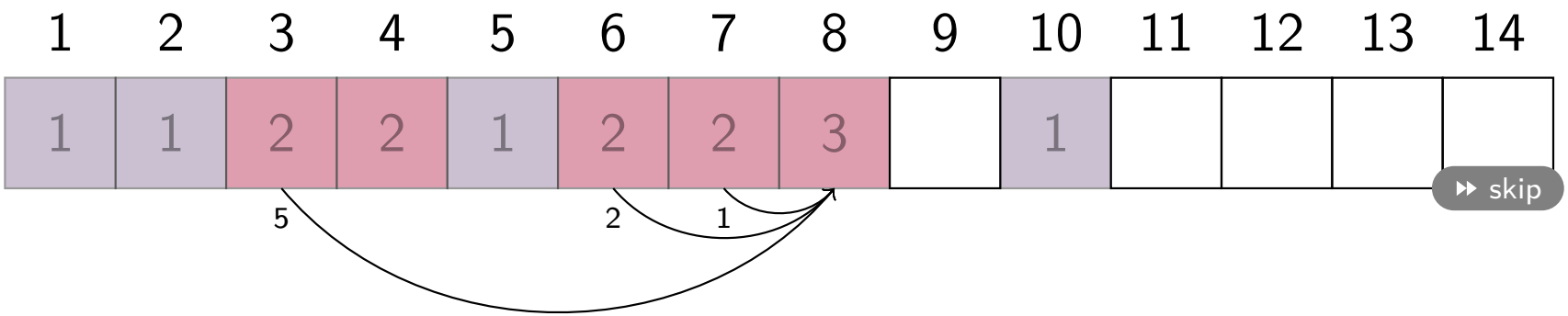


Résolution par programmation dynamique (Algorithme 1)

- On réutilise la formule de récurrence précédente tout en mémorisant les résultats intermédiaires :

$$\begin{cases} C(s) = 1 & \text{si } \exists i \in [0, n-1] \text{ tel que } s = S_i \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$

- Soit $S = (10, 5, 2, 1)$ et $total = 14$

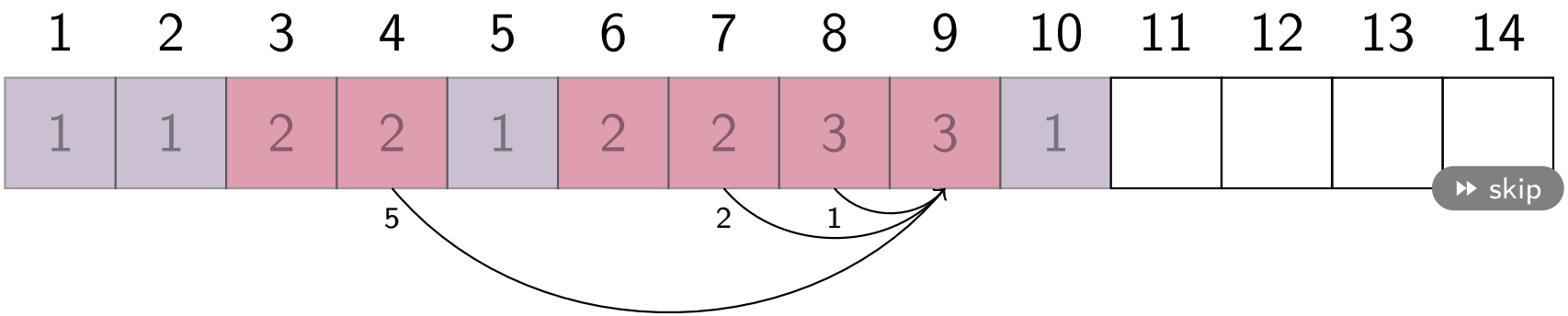


Résolution par programmation dynamique (Algorithme 1)

- On réutilise la formule de récurrence précédente tout en mémorisant les résultats intermédiaires :

$$\begin{cases} C(s) = 1 & \text{si } \exists i \in [0, n-1] \text{ tel que } s = S_i \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$

- Soit $S = (10, 5, 2, 1)$ et $total = 14$

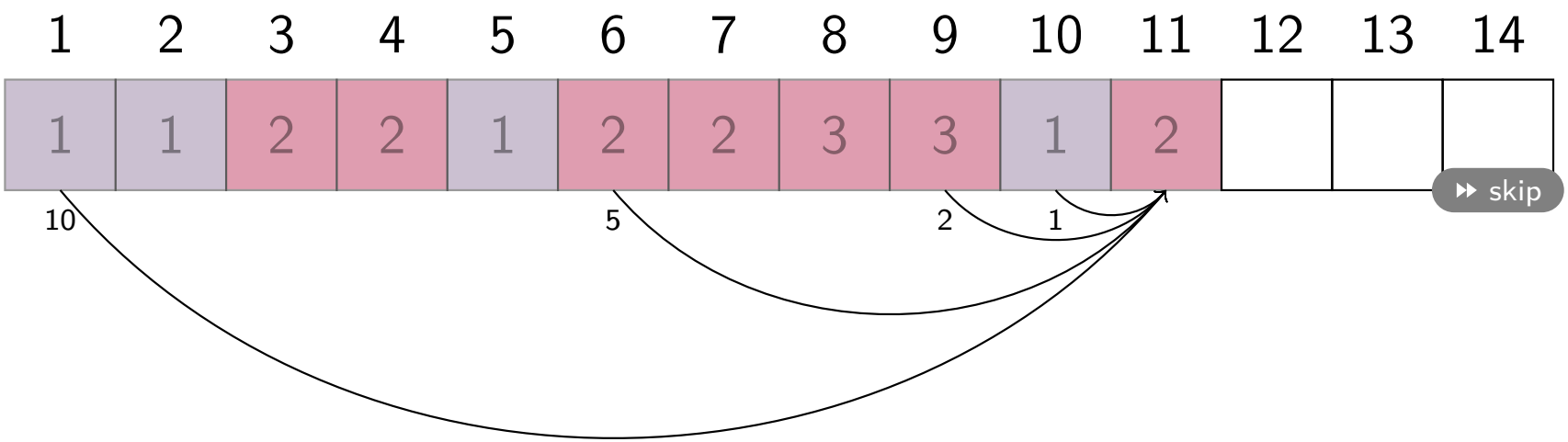


Résolution par programmation dynamique (Algorithme 1)

- On réutilise la formule de récurrence précédente tout en mémorisant les résultats intermédiaires :

$$\begin{cases} C(s) = 1 & \text{si } \exists i \in [0, n-1] \text{ tel que } s = S_i \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$

- Soit $S = (10, 5, 2, 1)$ et $total = 14$

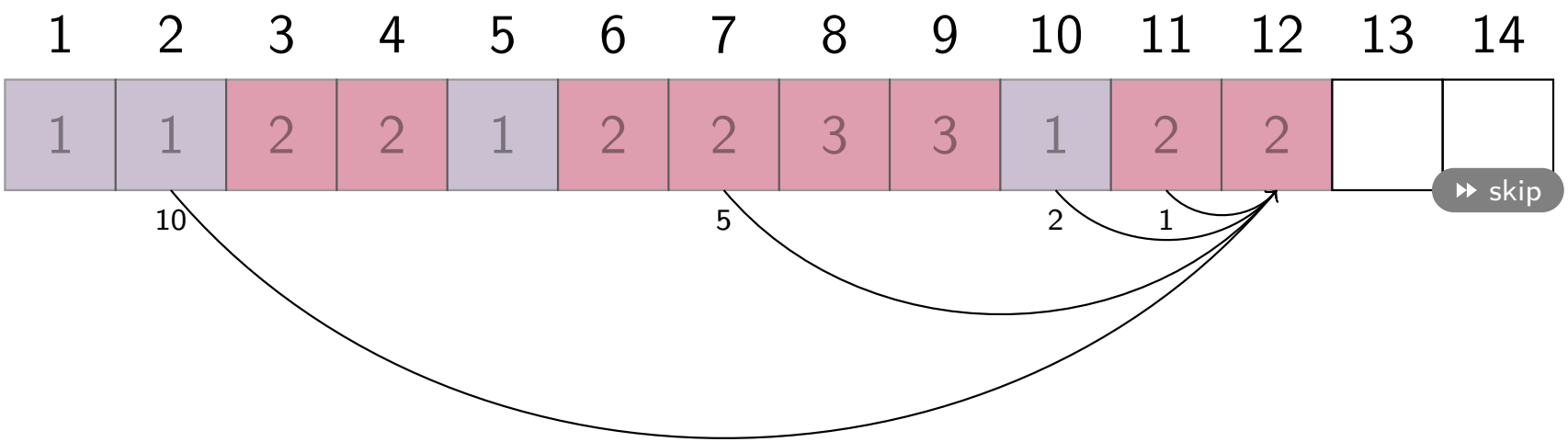


Résolution par programmation dynamique (Algorithme 1)

- On réutilise la formule de récurrence précédente tout en mémorisant les résultats intermédiaires :

$$\begin{cases} C(s) = 1 & \text{si } \exists i \in [0, n-1] \text{ tel que } s = S_i \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$

- Soit $S = (10, 5, 2, 1)$ et $total = 14$

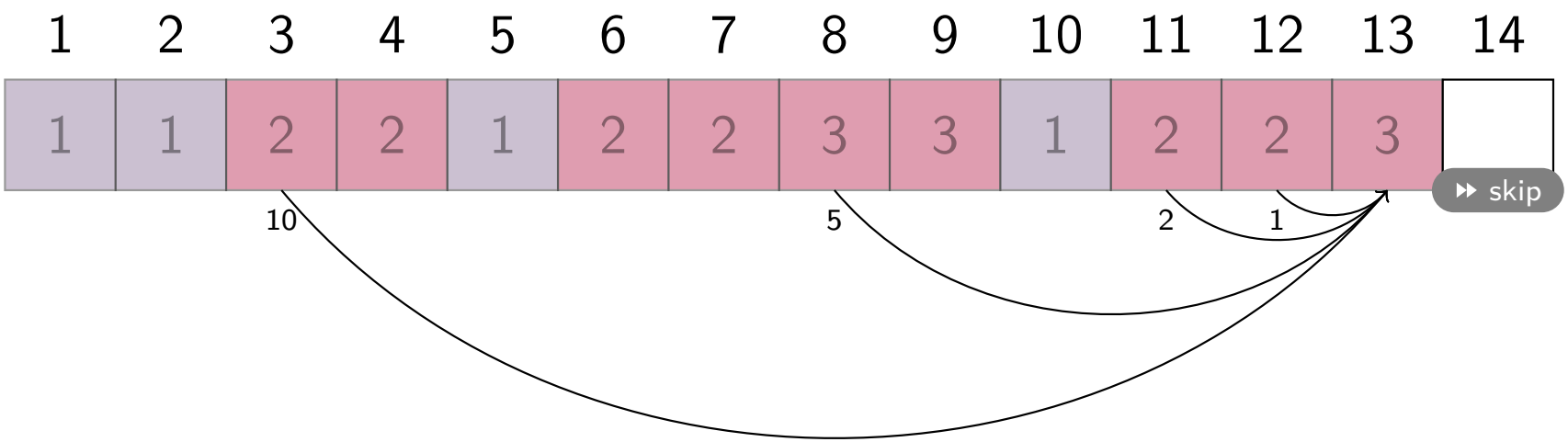


Résolution par programmation dynamique (Algorithme 1)

- On réutilise la formule de récurrence précédente tout en mémorisant les résultats intermédiaires :

$$\begin{cases} C(s) = 1 & \text{si } \exists i \in [0, n-1] \text{ tel que } s = S_i \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$

- Soit $S = (10, 5, 2, 1)$ et $total = 14$

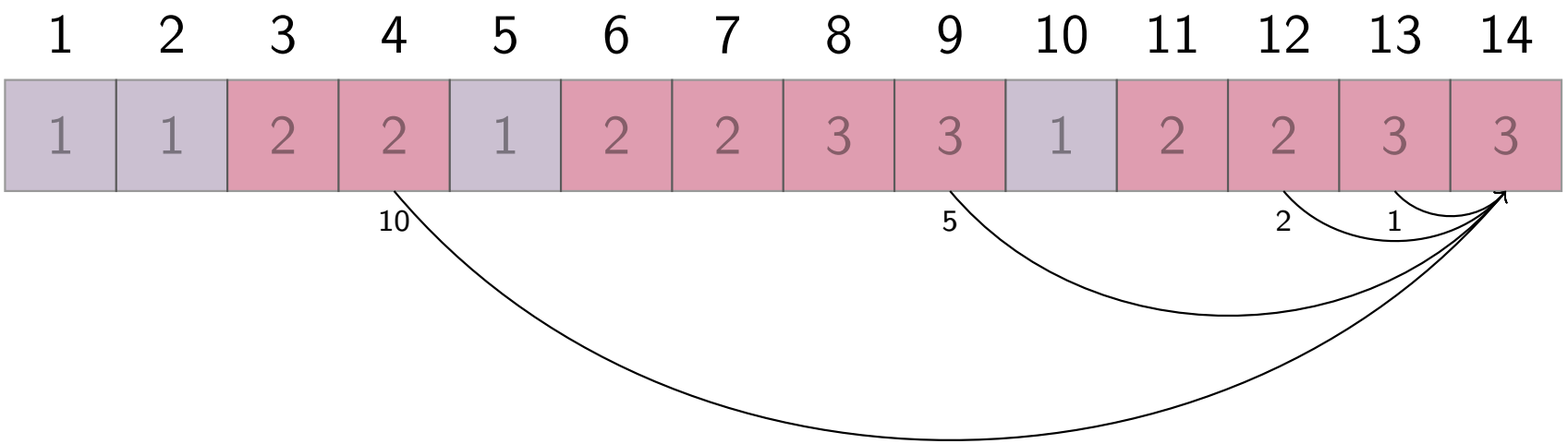


Résolution par programmation dynamique (Algorithme 1)

- On réutilise la formule de récurrence précédente tout en mémorisant les résultats intermédiaires :

$$\begin{cases} C(s) = 1 & \text{si } \exists i \in [0, n-1] \text{ tel que } s = S_i \\ C(s) = 1 + \min_{i \in [0,n-1], S_i \leq s} C(s - S_i) \end{cases}$$

- Soit $S = (10, 5, 2, 1)$ et $total = 14$



➜ temps de calcul $n \times total$

Algorithme 1, version récursive

```
import math
S = (200, 100, 50, 20, 10, 5, 2, 1); n=len(S)
total = 357
C = [math.inf for i in range(total+1)]

def rendu(somme):
 if C[somme]==math.inf:
     if somme in S:
         C[somme]=1
     else:
         best = math.inf
         for i in range(n):
             if S[i]<somme:
                 best = min(best,rendu(somme-S[i]))
         C[somme] = best+1
 return C[somme]

print("Nombre_de_pieces:", rendu(total))
```

Algorithme 1, version itérative

```
import math
S = (200, 100, 50, 20, 10, 5, 2, 1); n=len(S)
total = 357;
C = [0]

for i in range(1,total+1):
    C.append(math.inf)
    for j in range(n):
        if i>=S[j] and 1+C[i-S[j]]<C[i]:
            C[i] = 1+C[i-S[j]]

print("Nombre_de_pieces:", C[total])
```

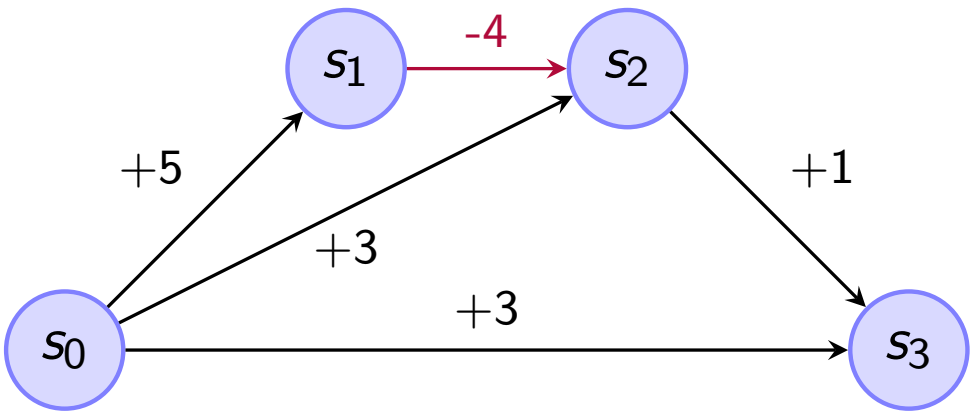
Algorithme 1, version itérative

```
import math
S = (200, 100, 50, 20, 10, 5, 2, 1); n=len(S)
total = 357;
C = [0]

for i in range(1,total+1):
    C.append(math.inf)
    for j in range(n):
        if i>=S[j] and 1+C[i-S[j]]<C[i]:
            C[i] = 1+C[i-S[j]]

print("Nombre_de_pieces:", C[total])
```

➜ Dans les deux cas, nous n'avons pas la solution, juste son coût $C(total)$

Plan

L'algorithme des PCC n'est pas adapté aux poids négatifs

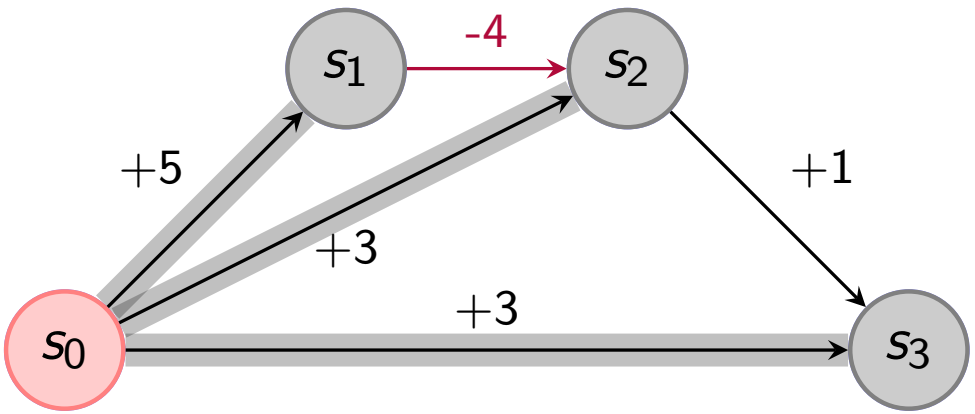


Nœud	Distance	Parent
s_0	0	•
s_1	∞	•
s_2	∞	•
s_3	∞	•

Frontière = $\{s_0\}$

x =

L'algorithme des PCC n'est pas adapté aux poids négatifs

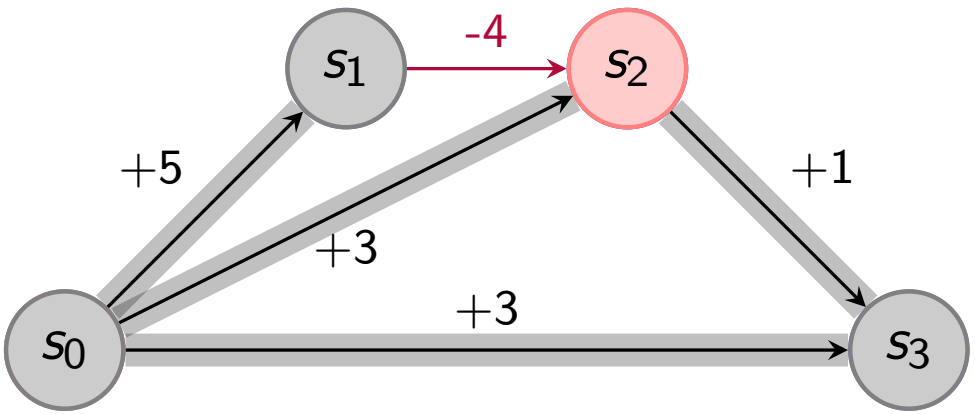


Nœud	Distance	Parent
s_0	0	•
s_1	5	s_0
s_2	3	s_0
s_3	3	s_0

Frontière = $\{s_1, s_2, s_3\}$

x = s_0

L'algorithme des PCC n'est pas adapté aux poids négatifs

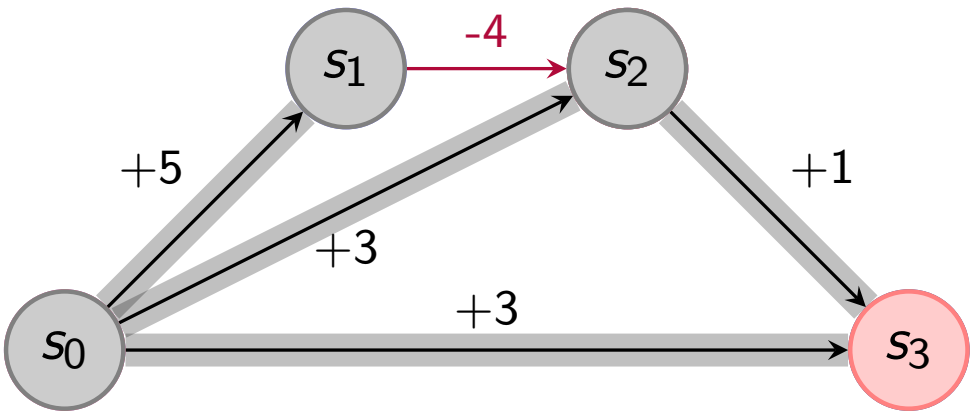


Nœud	Distance	Parent
s_0	0	•
s_1	5	s_0
s_2	3	s_0
s_3	3	s_0

Frontière = $\{s_1, s_3\}$

x = s_2

L'algorithme des PCC n'est pas adapté aux poids négatifs

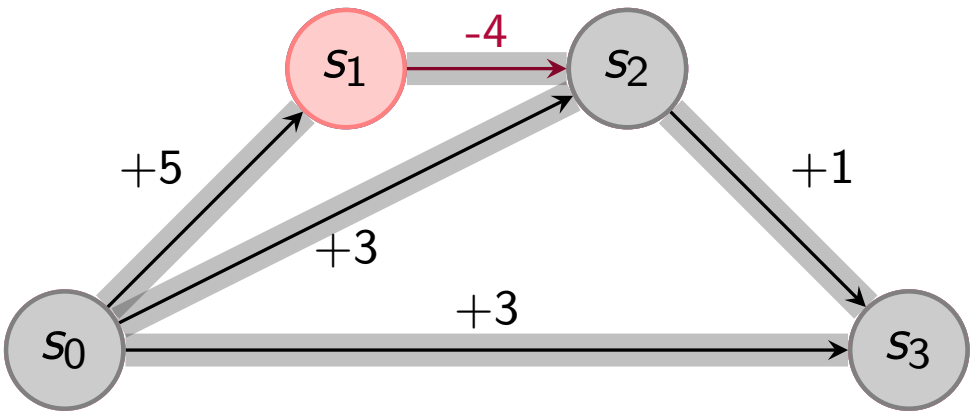


Nœud	Distance	Parent
s_0	0	•
s_1	5	s_0
s_2	3	s_0
s_3	3	s_0

Frontière = $\{s_1\}$

x = s_3

L'algorithme des PCC n'est pas adapté aux poids négatifs

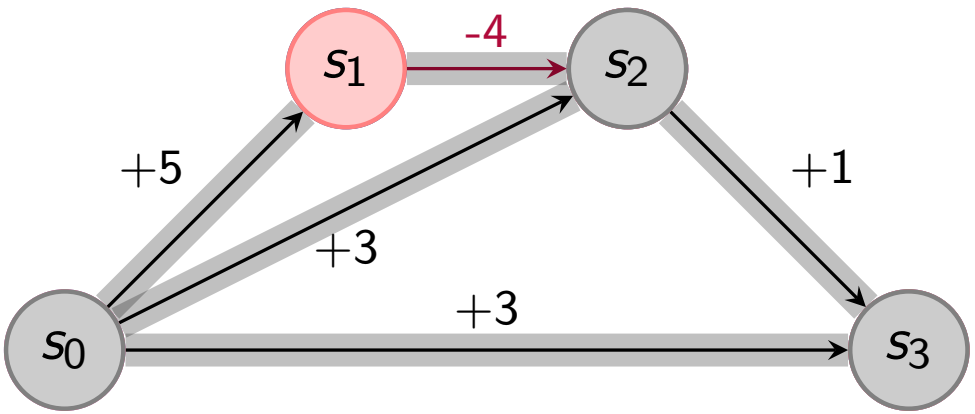


Nœud	Distance	Parent
s_0	0	•
s_1	5	s_0
s_2	1	s_1
s_3	3	s_0

Frontière =

x = s_1

L'algorithme des PCC n'est pas adapté aux poids négatifs



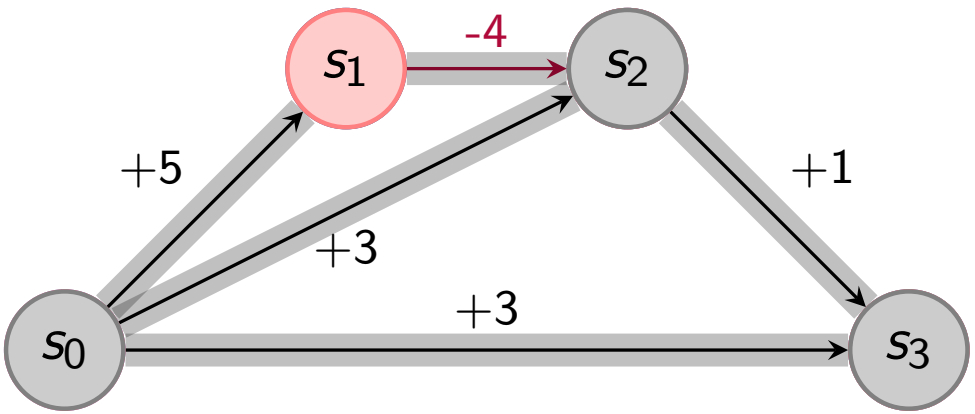
Nœud	Distance	Parent
s_0	0	•
s_1	5	s_0
s_2	1	s_1
s_3	3	s_0

Frontière =

x =

l'algorithme des PCC donne une réponse erronée concernant s_3 !

L'algorithme des PCC n'est pas adapté aux poids négatifs



Nœud	Distance	Parent
s_0	0	•
s_1	5	s_0
s_2	1	s_1
s_3	3	s_0

Frontière =

x =

l'algorithme des PCC donne une réponse erronée concernant s_3 !

Pourquoi voudrait-on calculer un plus court chemin sur un graphe avec des poids négatifs ?

➜ réponse au TD4, exercice 1 (problème des placements)

Algorithme Bellman-Ford (1956, 1958)

Principe

- Basé sur le principe de la programmation dynamique
- Calcule le coût du plus court chemin

  mais on peut retrouver le chemin à partir de la table de mémoïsation. . .

Algorithme Bellman-Ford (1956, 1958)

Principe

- Basé sur le principe de la programmation dynamique
- Calcule le coût du plus court chemin

  mais on peut retrouver le chemin à partir de la table de mémoïsation. . .

Rappel : données du problème

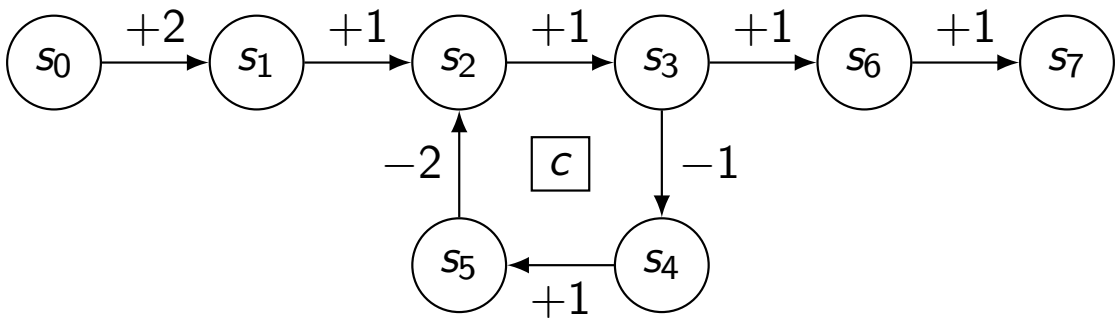
Un graphe orienté pondéré G quelconque, deux sommets s et t

y compris des poids négatifs. . .

➜ Quelle est la longueur du plus court chemin reliant s à t ?

Attention aux circuits absorbants !

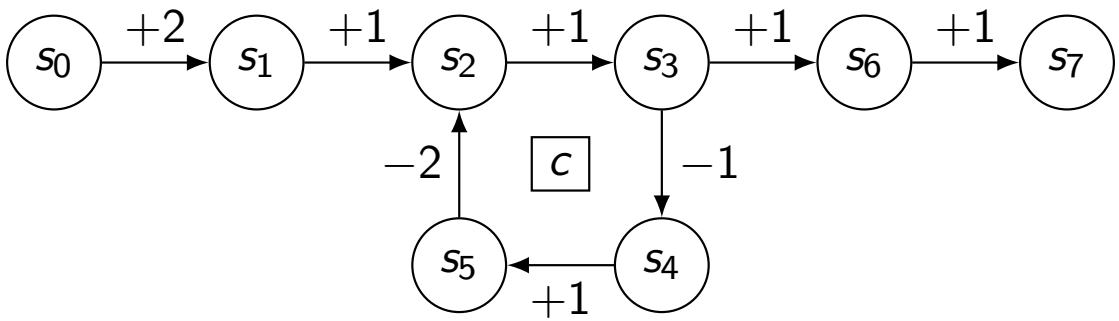
Définition : circuit absorbant



Le circuit c dans cet exemple est *un circuit absorbant*, parce que $\sum_{e=(v,u)\in c} \omega(e) < 0$.

Attention aux circuits absorbants !

Définition : circuit absorbant



Le circuit c dans cet exemple est *un circuit absorbant*, parce que $\sum_{e=(v,u)\in c} \omega(e) < 0$.

Plus court chemin avec poids négatifs

Besoin d'une formulation plus précise :

➜ On cherche le plus court chemin sans circuit !

Algorithme Bellman-Ford (1956, 1958)

Principe

- Basé sur le principe de la programmation dynamique
- Calcule le coût du plus court chemin

  mais on peut retrouver le chemin à partir de la table de mémoïsation. . .

Propriétés

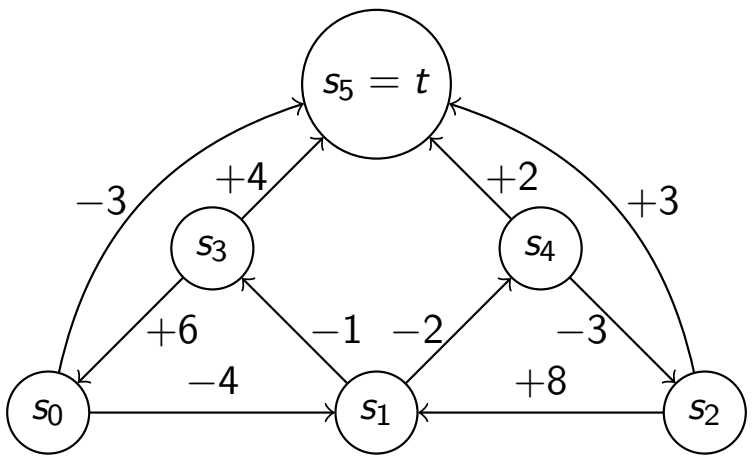
✓ Supporte les poids négatifs (contrairement à Dijkstra)

✓ Détecte s'il y a un circuit absorbant

Principes de l'algorithme Bellman-Ford

Décomposer en sous-problèmes

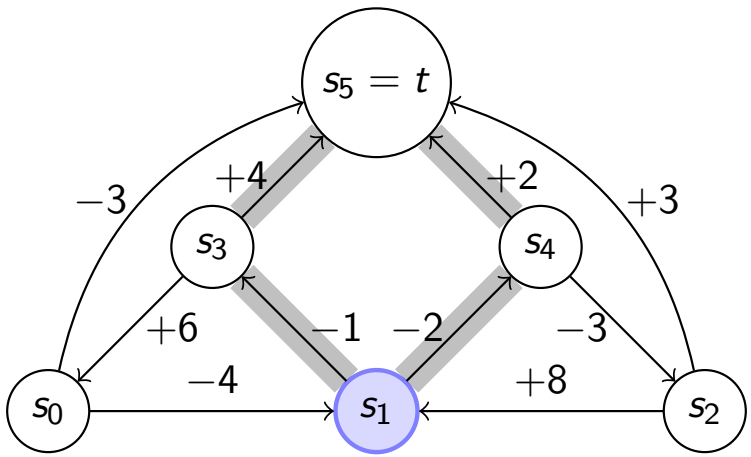
Soit OPT(i, v) la longueur du chemin le plus court vers le nœud cible t depuis un nœud $v, v \neq t$ qui contient au plus i arcs.



Principes de l'algorithme Bellman-Ford

Décomposer en sous-problèmes

Soit OPT(i, v) la longueur du chemin le plus court vers le nœud cible t depuis un nœud $v, v \neq t$ qui contient au plus i arcs.

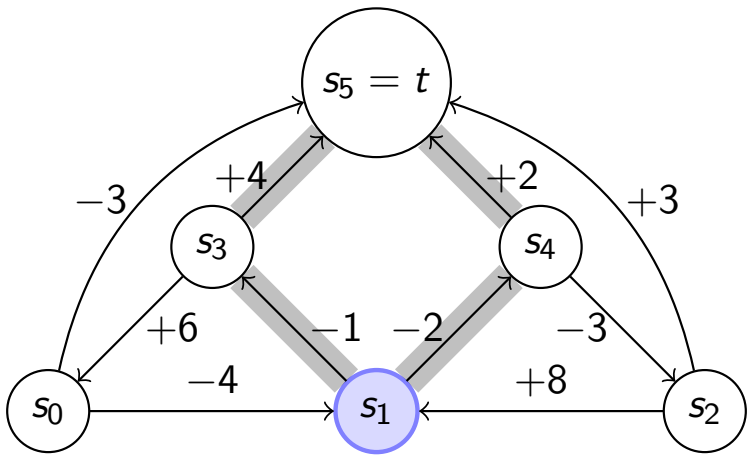


OPT(2,s1)= ?

Principes de l'algorithme Bellman-Ford

Décomposer en sous-problèmes

Soit OPT(i, v) la longueur du chemin le plus court vers le nœud cible t depuis un nœud $v, v \neq t$ qui contient au plus i arcs.



OPT(2,s1)= ?

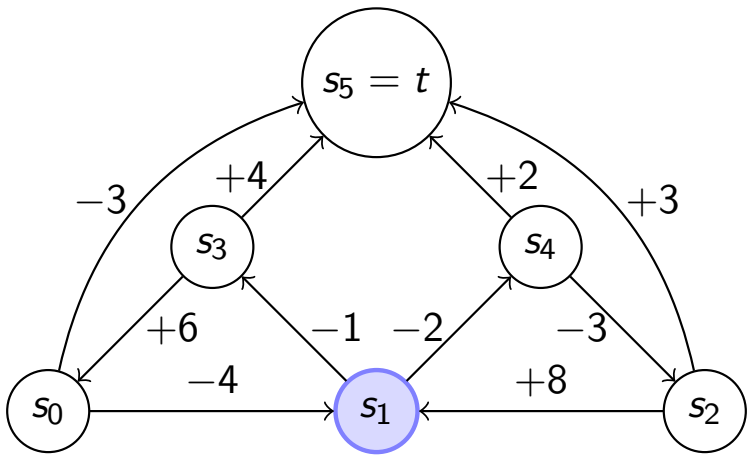
la longueur du plus court chemin en 2 arcs de s1 jusqu'à t (s5)

(en l'occurrence, c'est 0 en passant par s4)

Principes de l'algorithme Bellman-Ford

Décomposer en sous-problèmes

Soit OPT(i, v) la longueur du chemin le plus court vers le nœud cible t depuis un nœud $v, v \neq t$ qui contient au plus i arcs.

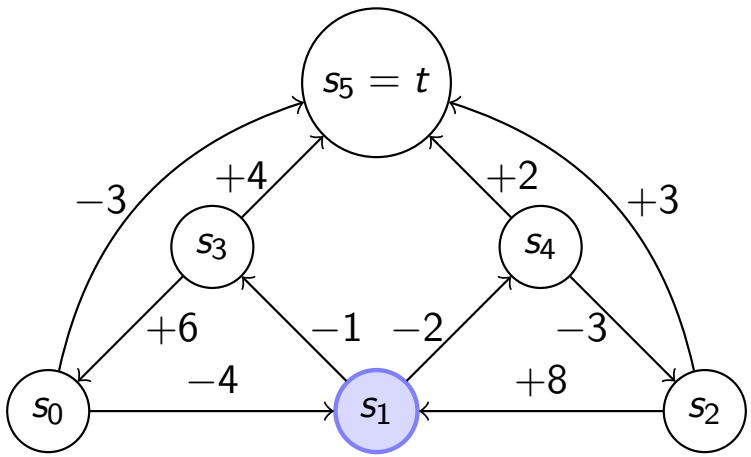


OPT($|V| - 1$,s1)= ?

Principes de l'algorithme Bellman-Ford

Décomposer en sous-problèmes

Soit OPT(i, v) la longueur du chemin le plus court vers le nœud cible t depuis un nœud $v, v \neq t$ qui contient au plus i arcs.



OPT($|V| - 1$,s1)= ?

la longueur du plus court chemin de s1 jusqu'à t (s5)

(en l'occurrence, c'est -2)

Principes de l'algorithme Bellman-Ford

Décomposer en sous-problèmes

Soit $\mathrm{OPT}(i, v)$ la longueur du chemin le plus court vers le nœud cible t depuis un nœud $v, v \neq t$ qui contient au plus i arcs.

Construction récursive d'une solution

- $\mathrm{OPT}(i, v) = \min_{(v,u)\in E}\big(\mathrm{OPT}(i-1, u) + \omega((v, u)))\big)$

  $\rightarrow$ Pour rejoindre t, aller d'abord à u en prenant le plus court chemin en $i-1$ étapes.

Principes de l'algorithme Bellman-Ford

Décomposer en sous-problèmes

Soit $\mathrm{OPT}(i, v)$ la longueur du chemin le plus court vers le nœud cible t depuis un nœud $v, v \neq t$ qui contient au plus i arcs.

Construction récursive d'une solution

- $\mathrm{OPT}(i, v) = \min_{(v,u)\in E}\big(\mathrm{OPT}(i-1, u) + \omega((v, u))\big)$

  $\rightarrow$ Pour rejoindre t, aller d'abord à u en prenant le plus court chemin en $i-1$ étapes.
- Sauf s'il y a déjà un chemin en $i-1$ étapes depuis v qui est plus court que tout le reste !

  $\rightarrow$ Auquel cas $\mathrm{OPT}(i, v) = \mathrm{OPT}(i-1, v)$

Principes de l'algorithme Bellman-Ford

Décomposer en sous-problèmes

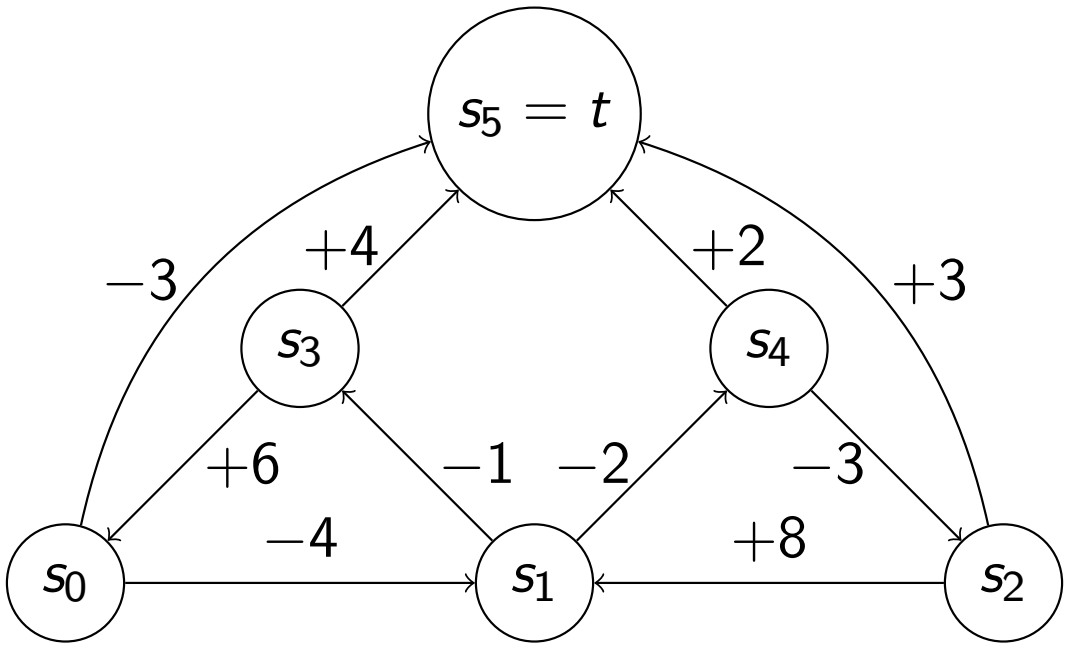
Soit $\mathrm{OPT}(i, v)$ la longueur du chemin le plus court vers le nœud cible t depuis un nœud $v, v \neq t$ qui contient au plus i arcs.

Construction récursive d'une solution

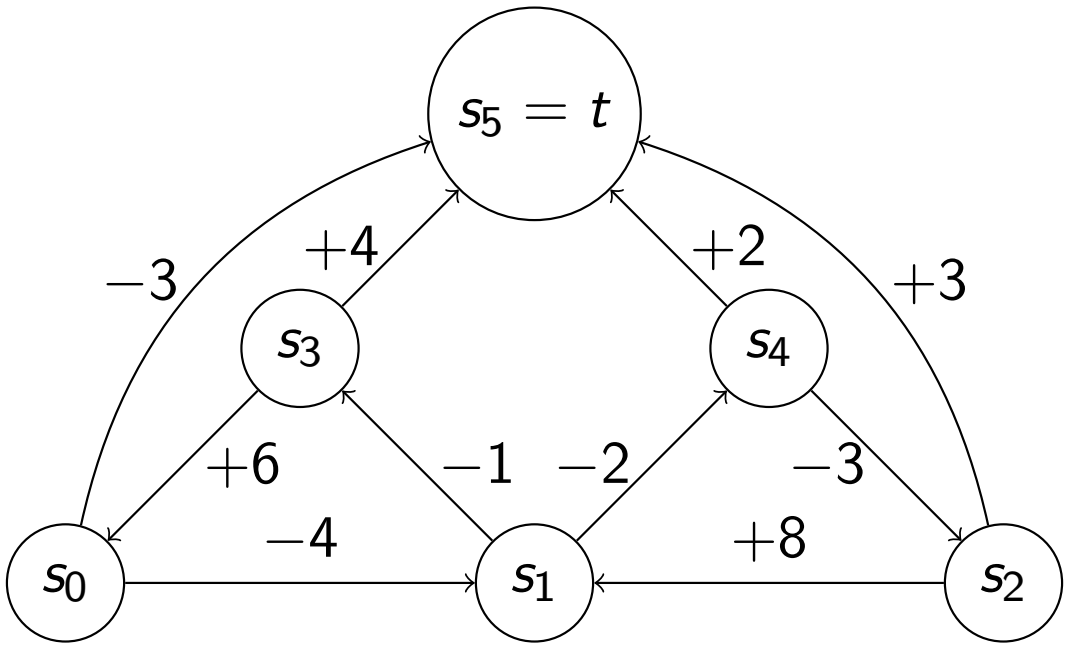
$$\mathrm{OPT}(i, v) = \min\left(\mathrm{OPT}(i-1, v), \min_{u \in V}(\mathrm{OPT}(i-1, u) + \omega((v, u)))\right)$$

Stocker les $\mathrm{OPT}(i, v) \rightarrow$ tableau à 2 dimensions.

Exemple

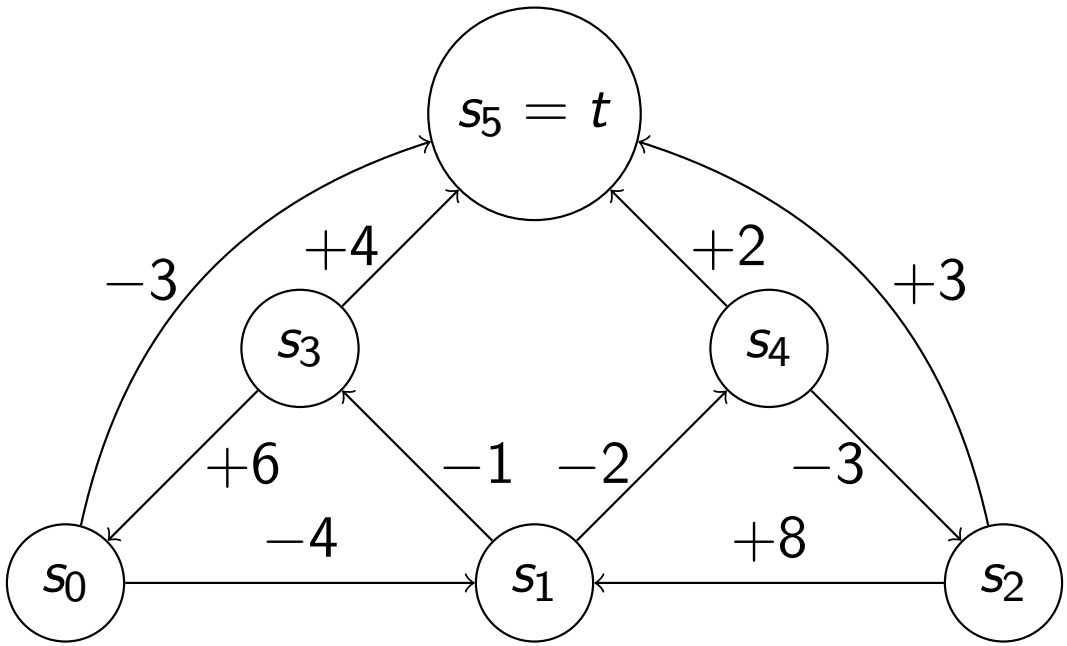


Exemple



	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0

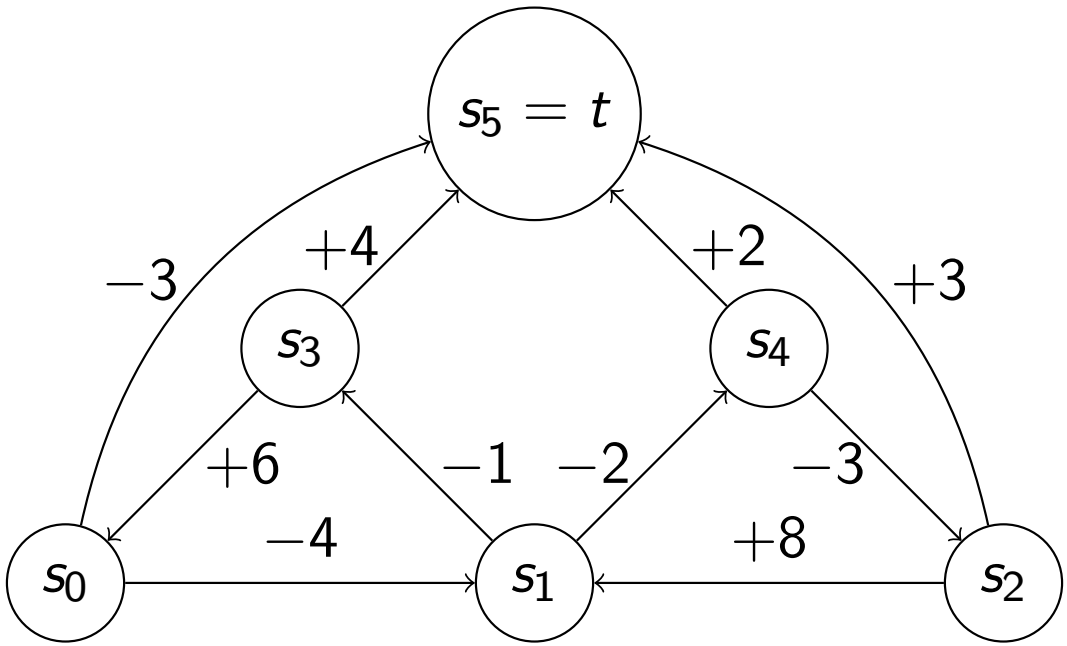
Exemple



	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0
1	-3					

$$s_0 = min(\infty, min(-4 + \infty(by\ s_1), -3 + 0(by\ s_5)))$$

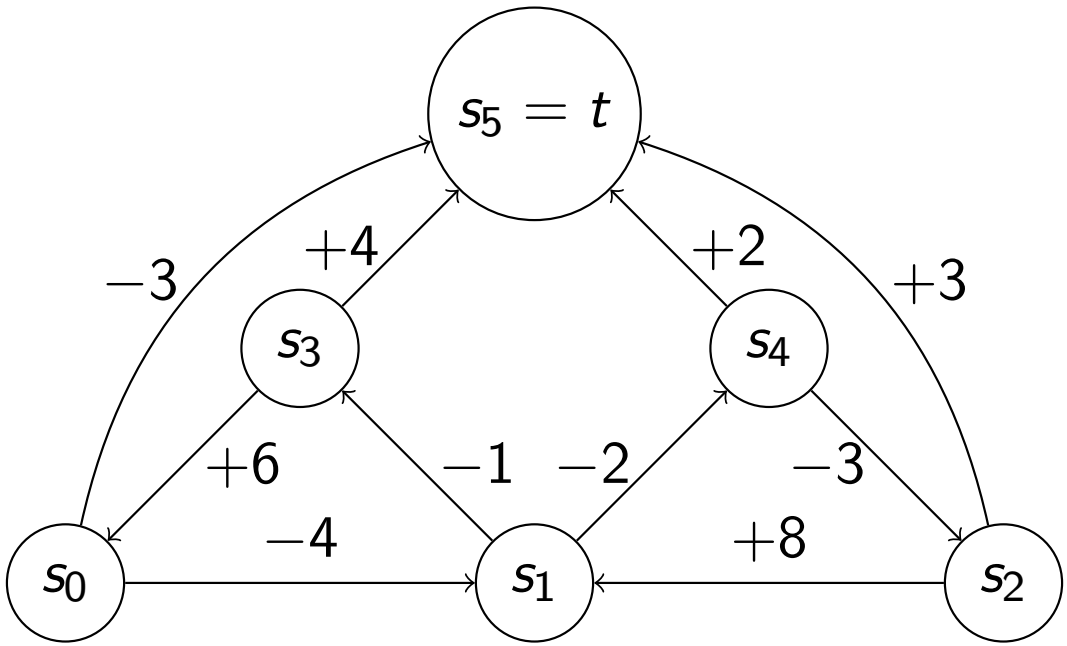
Exemple



	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0
1	-3	∞				

$$s_1 = min(\infty, min(-1 + \infty, -2 + \infty))$$

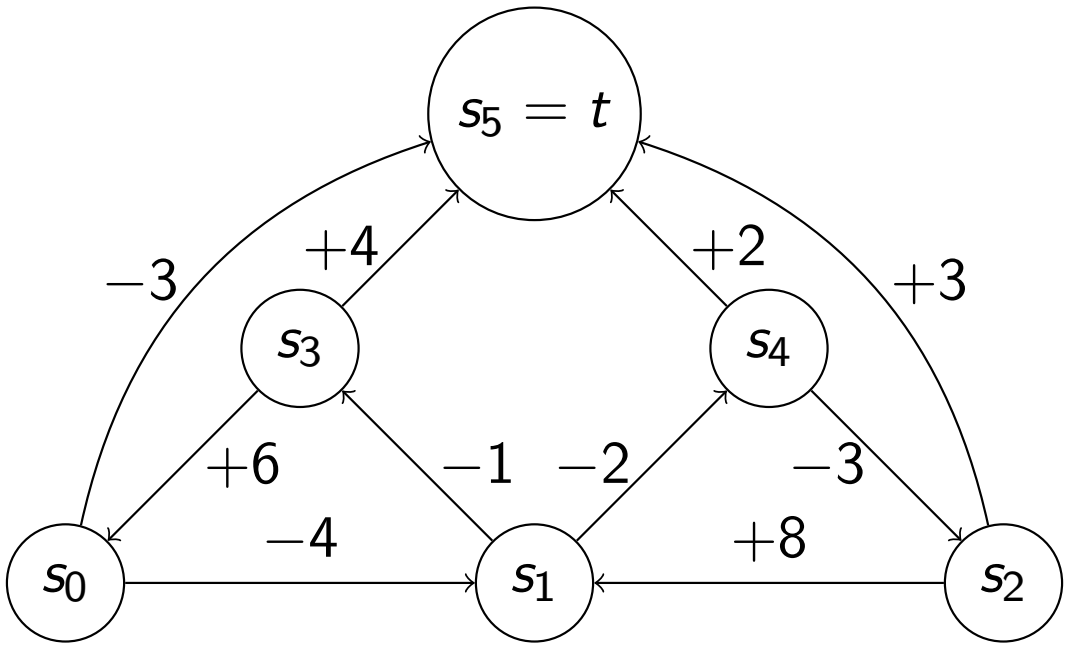
Exemple



	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0
1	-3	∞	3			

$$s_2 = min(\infty, min(8 + \infty, 3 + 0))$$

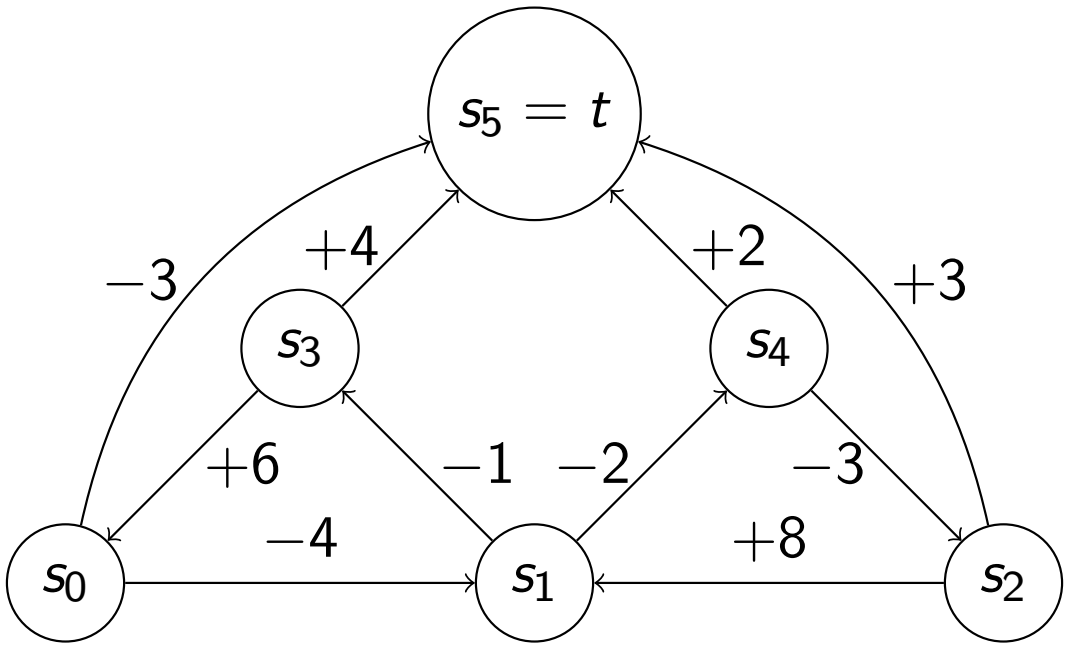
Exemple



	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0
1	-3	∞	3	4	2	0

on finit la ligne sur le même principe

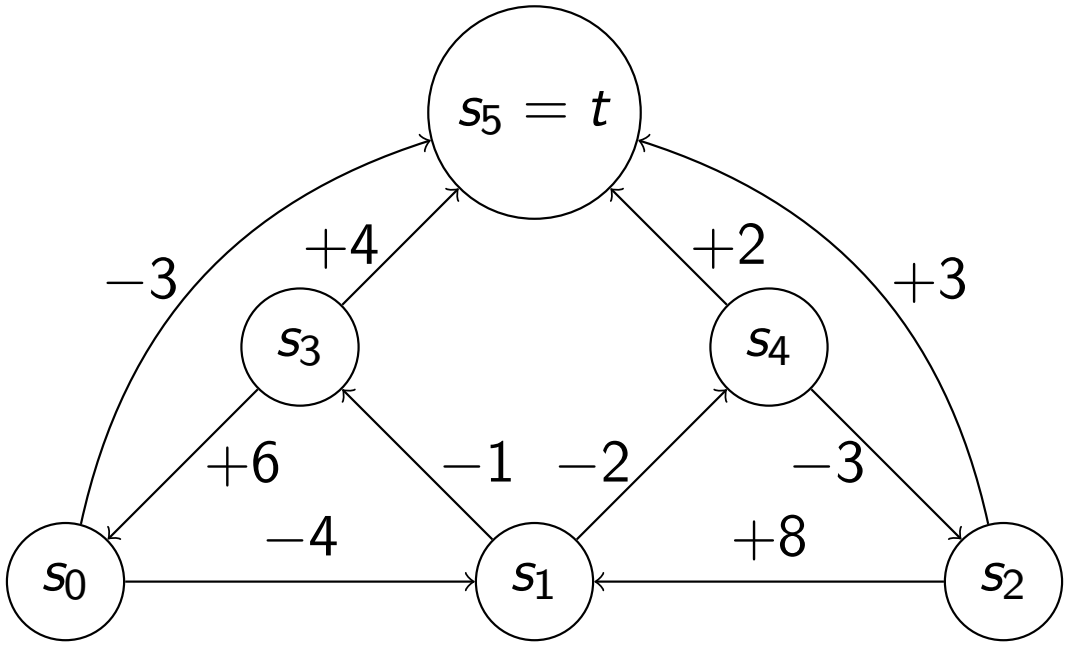
Exemple



	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0
1	-3	∞	3	4	2	0
2	-3	0	3	3	0	0

puis on fait la ligne suivante

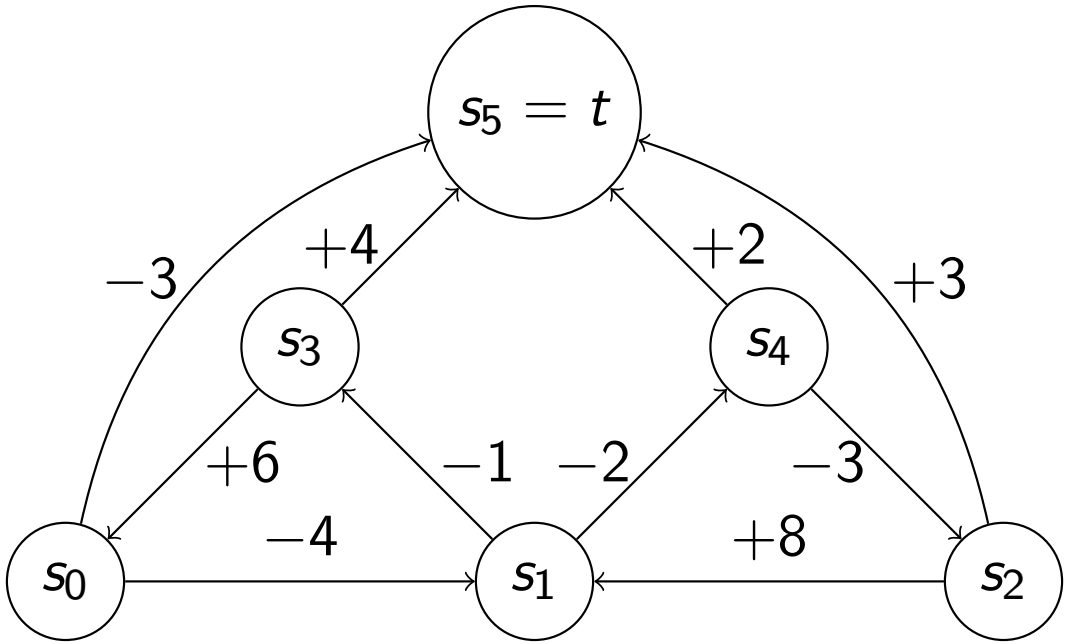
Exemple



	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0
1	-3	∞	3	4	2	0
2	-3	0	3	3	0	0
3	-4	-2	3	3	0	0

et ainsi de suite. . .

Exemple



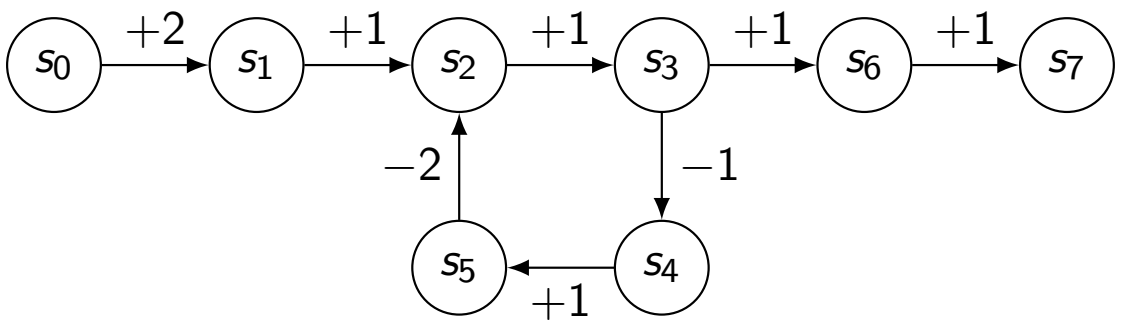
	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0
1	-3	∞	3	4	2	0
2	-3	0	3	3	0	0
3	-4	-2	3	3	0	0
4	-6	-2	3	2	0	0
5	-6	-2	3	0	0	0

Quand s'arrêter ?

Propriété

- Dans un graphe sans circuit, le plus court chemin contient au plus $|V| - 1$ arcs.

➜ C'est vrai aussi dans un graphe quelconque mais sans circuit absorbant.

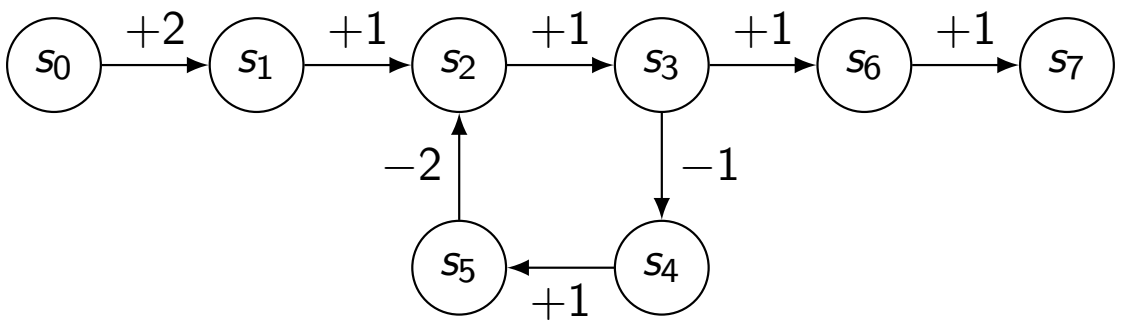


Quand s'arrêter ?

Propriété

- Dans un graphe sans circuit, le plus court chemin contient au plus $|V| - 1$ arcs.

➜ C'est vrai aussi dans un graphe quelconque mais sans circuit absorbant.



➜ On s'arrête lorsqu'on a atteint les chemins comprenant $|V| - 1$ arcs.

Une implémentation de Bellman-Ford (matrice d'adjacence)

```
import math

# graph est une matrice d'adjacence
n = len(graph)

# initialisation du tableau OPT
OPT = [[math.inf for _ in range(n)] for _ in range(n+1)]
OPT[0][5] = 0

# remplissage iteratif du tableau OPT
for i in range(1,n):
    for v in range(n):
        OPT[i][v] = OPT[i-1][v]
        for u in range(n):
            if graph[v][u] != None and \
                    OPT[i][v] > OPT[i-1][u] + graph[v][u]:
                OPT[i][v] = OPT[i-1][u] + graph[v][u]
```

Complexité de l'algorithme Bellman-Ford

Complexité

matrice d'adjacence

- 3 boucles imbriquées de $|V|$ itérations chacune
- un accès en $\mathcal{O}(1)$ à $\omega((v,u))$ à chaque passage dans la boucle intérieure !

Complexité de l'algorithme Bellman-Ford

Complexité **matrice d'adjacence**

- 3 boucles imbriquées de $|V|$ itérations chacune
- un accès en $\mathcal{O}(1)$ à $\omega((v,u))$ à chaque passage dans la boucle intérieure !

➜ La complexité totale de l'algorithme est donc : $\mathcal{O}(|V|^3)$.

Complexité de l'algorithme Bellman-Ford

Complexité **matrice d'adjacence**

- 3 boucles imbriquées de $|V|$ itérations chacune
- un accès en $\mathcal{O}(1)$ à $\omega((v,u))$ à chaque passage dans la boucle intérieure !

➜ La complexité totale de l'algorithme est donc : $\mathcal{O}(|V|^3)$.

Complexité **liste d'adjacence**

- en exercice maison
- 2 boucles : pour chacune des $|V|$ lignes, on itère sur les $|E|$ arêtes

Complexité de l'algorithme Bellman-Ford

Complexité **matrice d'adjacence**

- 3 boucles imbriquées de $|V|$ itérations chacune
- un accès en $\mathcal{O}(1)$ à $\omega((v,u))$ à chaque passage dans la boucle intérieure !

➜ La complexité totale de l'algorithme est donc : $\mathcal{O}(|V|^3)$.

Complexité **liste d'adjacence**

- en exercice maison
- 2 boucles : pour chacune des $|V|$ lignes, on itère sur les $|E|$ arêtes

➜ La complexité totale de l'algorithme est donc : $\mathcal{O}(|V| \times |E|)$.

Détection des circuits absorbants ($\Longleftarrow$)

Propriété (rappel)

- Dans un graphe sans circuit absorbant, le plus court chemin contient au plus $|V| - 1$ arcs.

➜ $\forall v, \text{OPT}(|V| - 1, v)$ contient bien la longueur du plus court chemin

Détection des circuits absorbants ($\Longrightarrow$)

Propriété (rappel)

- Dans un graphe sans circuit absorbant, le plus court chemin contient au plus $|V| - 1$ arcs.

➜ $\forall v, \text{OPT}(|V| - 1, v)$ contient bien la longueur du plus court chemin

Contraposée

Si le chemin le plus court est composé de plus de $|V| - 1$ arcs, alors G contient des circuits absorbants.

Détection des circuits absorbants ($\Longleftarrow$)

Propriété (rappel)

- Dans un graphe sans circuit absorbant, le plus court chemin contient au plus $|V| - 1$ arcs.

➜ $\forall v, \text{OPT}(|V| - 1, v)$ contient bien la longueur du plus court chemin

Contraposée

Si le chemin le plus court est composé de plus de $|V| - 1$ arcs, alors G contient des circuits absorbants.

Corollaire 1 ($\Longrightarrow$)

Si une case de la ligne $|V|$ du tableau est plus petite que la précédente :

$$\exists v. \;\; OPT(|V|, v) < OPT(|V| - 1, v)$$

alors il y a un circuit absorbant.

Détection des circuits absorbants ($\Longleftarrow$)

Propriétés

1. Si G contient un circuit absorbant, alors pour tout noeud v du circuit, on pourra toujours améliorer sa distance.
   - ➜ $\exists v.\ \ \forall n.\ \ \exists m > n\ \ OPT(m, v) < OPT(n, v)$

Détection des circuits absorbants ($\Longleftarrow$)

Propriétés

1. Si G contient un circuit absorbant, alors pour tout noeud v du circuit, on pourra toujours améliorer sa distance.
   - ➜ $\exists v.\ \ \forall n.\ \ \exists m > n\ \ \ OPT(m, v) < OPT(n, v)$
2. Si une ligne du tableau est identique à la suivante, alors toutes les lignes suivantes lui sont identiques aussi
   - conséquence de la formule de récurrence

Détection des circuits absorbants (⟸)

Propriétés

1. Si G contient un circuit absorbant, alors pour tout noeud v du circuit, on pourra toujours améliorer sa distance.
   - ➜ $\exists v.\ \ \forall n.\ \ \exists m > n\ \ OPT(m, v) < OPT(n, v)$
2. Si une ligne du tableau est identique à la suivante, alors toutes les lignes suivantes lui sont identiques aussi
   - conséquence de la formule de récurrence

Corollaire 2 (⟸)

Si G contient un circuit absorbant **alors**

$$\exists v.\ \ OPT(|V|, v) < OPT(|V| - 1, v)$$

Détection des circuits absorbants

Théorème

G contient un circuit absorbant **ssi**

$$\exists v.\ \ OPT(|V|, v) < OPT(|V| - 1, v)$$

Détection des circuits absorbants

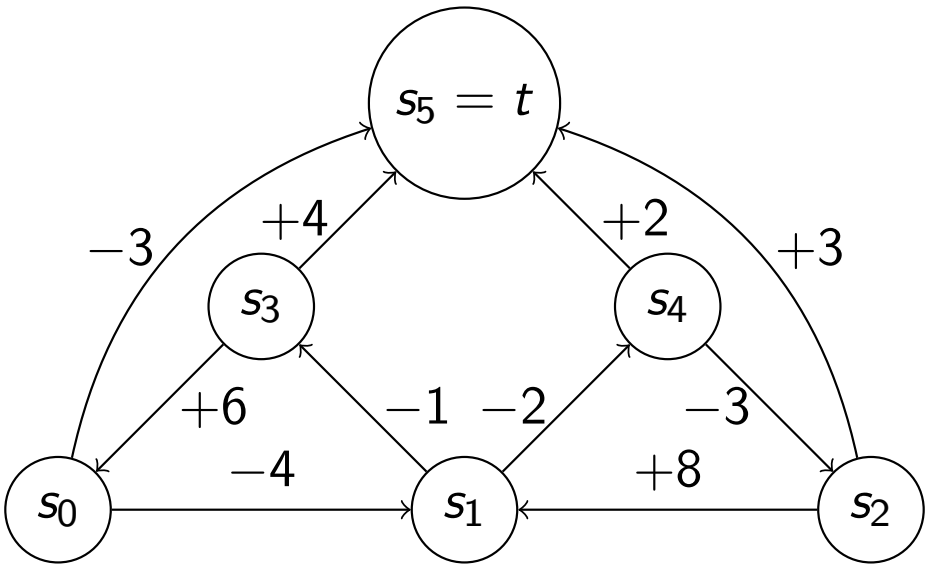
Théorème

G contient un circuit absorbant **ssi**

$$\exists v.\ \ OPT(|V|, v) < OPT(|V| - 1, v)$$

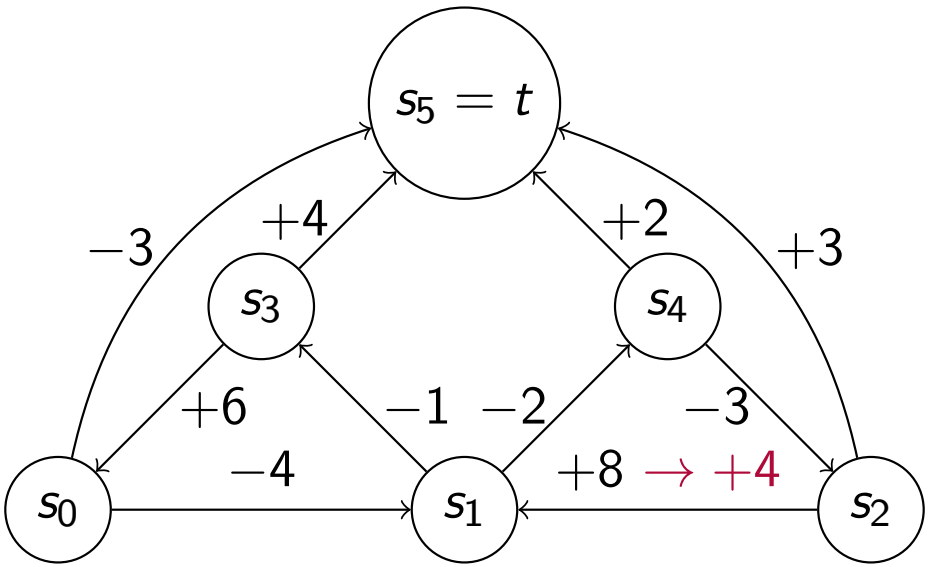
```
# Remplissage iteratif du tableau OPT
for i in range(1,n+1): # on va une ligne plus loin
    for v in range(n):
        OPT[i][v] = OPT[i-1][v]
        for u in range(n):
            if graph[v][u] != None and \
                    OPT[i][v] > OPT[i-1][u] + graph[v][u]:
                OPT[i][v] = OPT[i-1][u] + graph[v][u]

# Detection de circuits absorbants
for v in range(n):
    if OPT[n-1][v] > OPT[n][v]:
      print("ATTENTION:_circuit_absorbant!\n")
      break;
```

Exemple



	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0
1	-3	∞	3	4	2	0
2	-3	0	3	3	0	0
3	-4	-2	3	3	0	0
4	-6	-2	3	2	0	0
5	-6	-2	3	0	0	0
6	-6	-2	3	0	0	0

Exemple



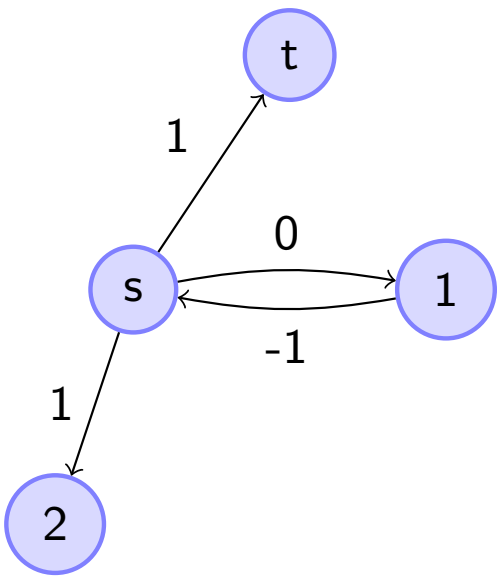
	s_0	s_1	s_2	s_3	s_4	s_5
0	∞	∞	∞	∞	∞	0
1	-3	∞	3	4	2	0
2	-3	0	3	3	0	0
3	-4	-2	3	3	0	0
4	-6	-2	2	2	0	0
5	-6	-2	2	0	-1	0
6	-6	-3	2	0	-1	0

Exemple

Attention

Dès qu'il y a un circuit absorbant, le coût proposé pour $s \to t$ peut être invalide !

...même si $OPT(|V|-1, s) = OPT(|V|, s)$



	s	1	2	t
0	∞	∞	∞	0
1	1	∞	∞	0
2	1	0	∞	0
3	0	0	∞	0
4	0	-1	∞	0

Démo

Bellman-Ford « réparti »

Propriété

Pour déterminer le coût $OPT(i, v)$ du nœud v à l'étape i, on n'a besoin que de :

- $OPT(i-1, v)$ la valeur à l'étape précédente pour v ;
- $OPT(i-1, u)$ la valeur à l'étape précédente pour tous les voisins u de v.

$\rightarrow$ Chaque nœud peut calculer son coût tout seul en communiquant avec ses voisins

. . .sans connaître l'ensemble du graphe !

Bellman-Ford « réparti »

Propriété

Pour déterminer le coût $OPT(i, v)$ du nœud v à l'étape i, on n'a besoin que de :

- $OPT(i-1, v)$ la valeur à l'étape précédente pour v ;
- $OPT(i-1, u)$ la valeur à l'étape précédente pour tous les voisins u de v.

$\rightarrow$ Chaque nœud peut calculer son coût tout seul en communiquant avec ses voisins

. . .sans connaître l'ensemble du graphe !

Application aux réseaux

Problème du routage

Routage dans des réseaux à commutation de paquets

Problème

Déterminer le chemin le plus efficace pour router les messages jusqu'à leurs destinations.

Routage dans des réseaux à commutation de paquets

Problème

Déterminer le chemin le plus efficace pour router les messages jusqu'à leurs destinations.

Critères (poids)

Routes les plus courtes en nombre de liens, latence minimale, ...

Routage dans des réseaux à commutation de paquets

Problème

Déterminer le chemin le plus efficace pour router les messages jusqu'à leurs destinations.

Critères (poids)

Routes les plus courtes en nombre de liens, latence minimale, ...

Spécificité du routage

- Chaque routeur contient une table (destination, routeur_suivant (`Next_Hop`)).
- Calculs faits localement dans les routeurs (sans connaître la configuration du réseau)

Routage dans des réseaux à commutation de paquets

Modélisation des données (réseau)

- routeurs modélisés par les sommets du graphe
- liens entre routeurs modélisés par des arcs
- distances (nombre de liens, délais) modélisées par les poids

Routage dans des réseaux à commutation de paquets

Modélisation des données (réseau)

- routeurs modélisés par les sommets du graphe
- liens entre routeurs modélisés par des arcs
- distances (nombre de liens, délais) modélisées par les poids

Communication

Dès qu'un routeur change sa table de routage, il en informe ses voisins pour qu'ils puissent mettre à jour leurs tables également.

Implémentation

Chaque routeur exécute en boucle :

Implémentation

Chaque routeur exécute en boucle :

1. attend une notification de changement de routage de la part d'un de ses voisins

Implémentation

Chaque routeur exécute en boucle :

1. attend une notification de changement de routage de la part d'un de ses voisins
2. recalcule sa table de routage (Destination finale $p \rightarrow Next\_Hop$)

Implémentation

Chaque routeur exécute en boucle :

1. attend une notification de changement de routage de la part d'un de ses voisins
2. recalcule sa table de routage (Destination finale $p \rightarrow Next\_Hop$)
3. envoie ses nouvelles distances à ses voisins

Implémentation

Chaque routeur exécute en boucle :

1. attend une notification de changement de routage de la part d'un de ses voisins
2. recalcule sa table de routage (Destination finale $p \rightarrow Next\_Hop$)
3. envoie ses nouvelles distances à ses voisins
4. `goto 1`

Implémentation

Chaque routeur exécute en boucle :

1. attend une notification de changement de routage de la part d'un de ses voisins
2. recalcule sa table de routage (Destination finale $p \rightarrow Next\_Hop$)
3. envoie ses nouvelles distances à ses voisins
4. `goto 1`

Chaque routeur v garde localement :

- un tableau M_v où $M_v[p]$ indique la distance du chemin le plus court entre v et p
- un tableau $\texttt{Next\_Hop}_v$ où $\texttt{Next\_Hop}_v[p]$ est le numéro du routeur suivant pour tout envoi vers p

Algorithme implémenté par chaque sommet v

```
Nv = ... # la liste des voisins du noeud courant v

def process(u, Mu) :
    """
    A chaque notification recu depuis un voisin u
    :param Mu: table de routage du voisin u
    """

    # mise a jour de la table de routage locale
    update = False
    for p in V:
        if Mu[p] + Timings[v][u] < Mv[p]:
            Mv[p] = Mu[p] + Timings[v][u]
            NextHop_v[p] = u
            update = True

    # notification des voisins
    if update:
        for u in Nv:
            send_update(u,Mv)
```

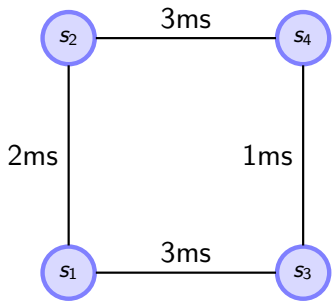
Algorithme Bellman-Ford comme protocole

Distance Vector Protocol

Ce protocole est utilisé dans les réseaux informatiques (*e.g.* sur Internet)

→ *Routing Information Protocol* (RIP)

Exemple



Algorithme Bellman-Ford comme protocole

Distance Vector Protocol

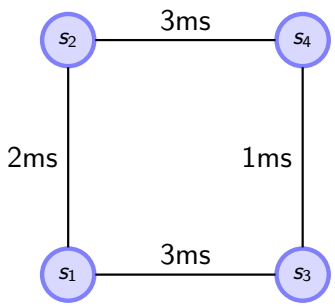
Ce protocole est utilisé dans les réseaux informatiques (*e.g.* sur Internet)

→ *Routing Information Protocol* (RIP)

Exemple

p	s_1	s_4
$M_2(p)$	2	3
$next(p)$	s_1	s_4

p	s_2	s_3
$M_4(p)$	3	1
$next(p)$	s_2	s_3



p	s_2	s_3
$M_1(p)$	2	3
$next(p)$	s_2	s_3

p	s_1	s_4
$M_3(p)$	3	1
$next(p)$	s_1	s_4

Algorithme Bellman-Ford comme protocole

Distance Vector Protocol

Ce protocole est utilisé dans les réseaux informatiques (*e.g.* sur Internet)

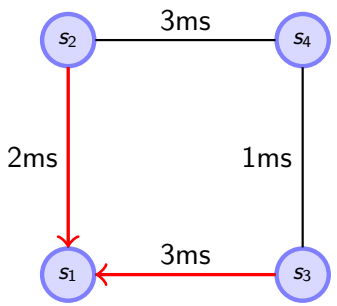
→ *Routing Information Protocol* (RIP)

Exemple

p	s_1	s_4
$M_2(p)$	2	3
$next(p)$	s_1	s_4

p	s_2	s_3
$M_4(p)$	3	1
$next(p)$	s_2	s_3

p	s_2	s_3	s_4
$M_1(p)$	2	3	4
$next(p)$	s_2	s_3	s_3



p	s_1	s_4
$M_3(p)$	3	1
$next(p)$	s_1	s_4

Algorithme Bellman-Ford comme protocole

Distance Vector Protocol

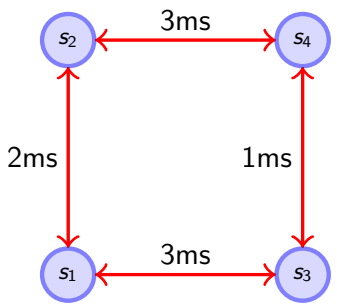
Ce protocole est utilisé dans les réseaux informatiques (*e.g.* sur Internet)

→ *Routing Information Protocol* (RIP)

Exemple

p	s_1	s_3	s_4
$M_2(p)$	2	4	3
$next(p)$	s_1	s_4	s_4

p	s_1	s_2	s_3
$M_4(p)$	4	3	1
$next(p)$	s_3	s_2	s_3



p	s_2	s_3	s_4
$M_1(p)$	2	3	4
$next(p)$	s_2	s_3	s_3

p	s_1	s_2	s_4
$M_3(p)$	3	4	1
$next(p)$	s_1	s_4	s_4

Algorithme Bellman-Ford comme protocole

Distance Vector Protocol

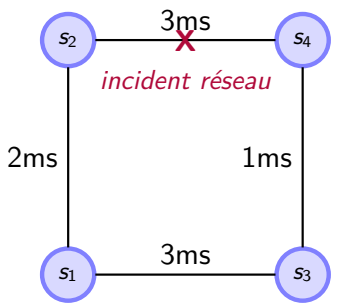
Ce protocole est utilisé dans les réseaux informatiques (*e.g.* sur Internet)

→ *Routing Information Protocol* (RIP)

Exemple

p	s_1	s_3	s_4
$M_2(p)$	2	4	3
$next(p)$	s_1	s_4	s_4

p	s_1	s_2	s_3
$M_4(p)$	4	3	1
$next(p)$	s_3	s_2	s_3



p	s_2	s_3	s_4
$M_1(p)$	2	3	4
$next(p)$	s_2	s_3	s_3

p	s_1	s_2	s_4
$M_3(p)$	3	4	1
$next(p)$	s_1	s_4	s_4

Algorithme Bellman-Ford comme protocole

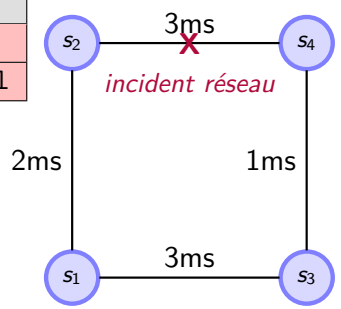
Distance Vector Protocol

Ce protocole est utilisé dans les réseaux informatiques (*e.g.* sur Internet)

→ *Routing Information Protocol* (RIP)

Exemple

p	s_1	s_3	s_4
$M_2(p)$	2	4	∞
$next(p)$	s_1	s_4	`null`



p	s_1	s_2	s_3
$M_4(p)$	4	∞	1
$next(p)$	s_3	`null`	s_3

p	s_2	s_3	s_4
$M_1(p)$	2	3	4
$next(p)$	s_2	s_3	s_3

p	s_1	s_2	s_4
$M_3(p)$	3	4	1
$next(p)$	s_1	s_4	s_4

Algorithme Bellman-Ford comme protocole

Distance Vector Protocol

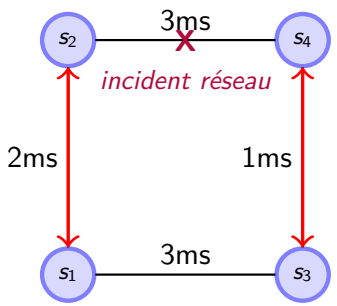
Ce protocole est utilisé dans les réseaux informatiques (*e.g.* sur Internet)

→ *Routing Information Protocol* (RIP)

Exemple

p	s_1	s_3	s_4
$M_2(p)$	2	4	6
$next(p)$	s_1	s_4	s_1

p	s_1	s_2	s_3
$M_4(p)$	4	∞	1
$next(p)$	s_3	`null`	s_3



p	s_2	s_3	s_4
$M_1(p)$	2	3	4
$next(p)$	s_2	s_3	s_3

p	s_1	s_2	s_4
$M_3(p)$	3	5	1
$next(p)$	s_1	s_1	s_4

Algorithme Bellman-Ford comme protocole

Distance Vector Protocol

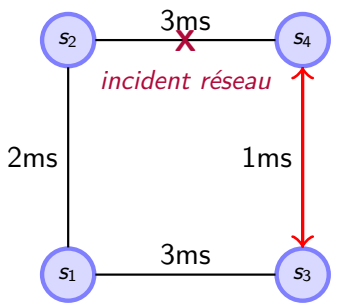
Ce protocole est utilisé dans les réseaux informatiques (*e.g.* sur Internet)

→ *Routing Information Protocol* (RIP)

Exemple

p	s_1	s_3	s_4
$M_2(p)$	2	4	6
$next(p)$	s_1	s_4	s_1

p	s_1	s_2	s_3
$M_4(p)$	4	6	1
$next(p)$	s_3	s_3	s_3



p	s_2	s_3	s_4
$M_1(p)$	2	3	4
$next(p)$	s_2	s_3	s_3

p	s_1	s_2	s_4
$M_3(p)$	3	5	1
$next(p)$	s_1	s_1	s_4

Plan

Les points à retenir

- Méthode de résolution « générale »
- Formule de récurrence (structure sous-optimale)
- Les sous-problèmes ne sont pas indépendants
- Technique de mémoïsation : *diminuer le temps d'exécution en mémorisant les valeurs calculées*
  - ➜ Compromis classique en informatique : temps *vs* mémoire
- Généralement efficace mais pas toujours applicable
- Plus courts chemins
  - ✗ Attention aux poids négatifs !
  - ✗ Attention aux circuits absorbants !
  - ➜ Algorithme de Bellman-Ford : contourne ces deux difficultés
    - Complexité polynomiale ($\mathcal{O}(|V|^3)$ ou $\mathcal{O}(|V| \times |E|)$)
    - Principe utilisé aussi pour le routage de paquets

Plan

Pour aller plus loin

Problème concret

En bio-informatique (l'informatique dédiée à la biologie), l'alignement de séquences permet de rapprocher deux séquences biologiques (ADN, ARN ou protéines), de manière à expliciter les régions similaires.

Exemple

- soit deux séquences de tailles quelconques, la première de taille n et la deuxième de taille m :

$$C\ T\ A\ G\ C\ A\ G\ T\ C\ A$$

$$G\ A\ G\ C\ A\ T\ C\ A\ T\ C\ G$$

Exemple

- soit deux séquences de tailles quelconques, la première de taille n et la deuxième de taille m :

$$C\ T\ A\ G\ C\ A\ G\ T\ C\ A$$
$$G\ A\ G\ C\ A\ T\ C\ A\ T\ C\ G$$

- un alignement :

$$C\ T\ A\ G\ C\ A\ G\ -\ -\ T\ C\ A$$
$$G\ -\ A\ G\ C\ A\ T\ C\ A\ T\ C\ G$$

Exemple

- soit deux séquences de tailles quelconques, la première de taille n et la deuxième de taille m :

$$\begin{array}{cccccccccc} C & T & A & G & C & A & G & T & C & A \\ G & A & G & C & A & T & C & A & T & C & G \end{array}$$

- un alignement :

$$\begin{array}{cccccccccccc} C & T & A & G & C & A & G & - & - & T & C & A \\ G & - & A & G & C & A & T & C & A & T & C & G \end{array}$$

match

Exemple

- soit deux séquences de tailles quelconques, la première de taille n et la deuxième de taille m :

$$\begin{array}{cccccccccc} C & T & A & G & C & A & G & T & C & A \\ G & A & G & C & A & T & C & A & T & C & G \end{array}$$

- un alignement :

$$\begin{array}{cccccccccccc} C & T & A & G & C & A & G & - & - & T & C & A \\ G & - & A & G & C & A & T & C & A & T & C & G \end{array}$$

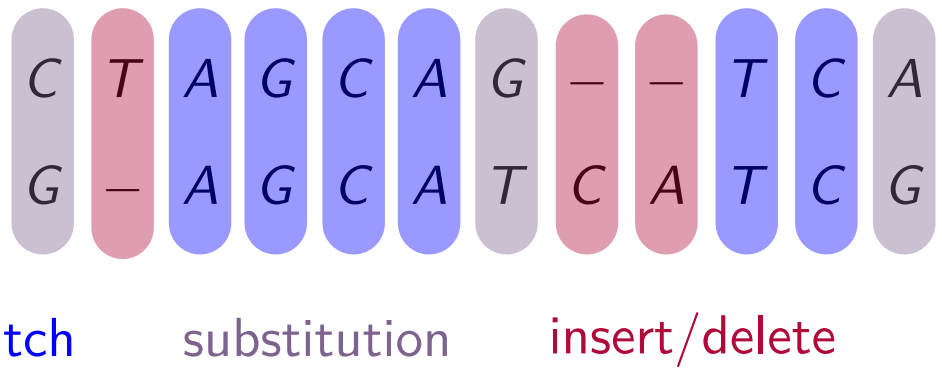
match substitution

Exemple

- soit deux séquences de tailles quelconques, la première de taille n et la deuxième de taille m :

$$C\ T\ A\ G\ C\ A\ G\ T\ C\ A$$
$$G\ A\ G\ C\ A\ T\ C\ A\ T\ C\ G$$

- un alignement :

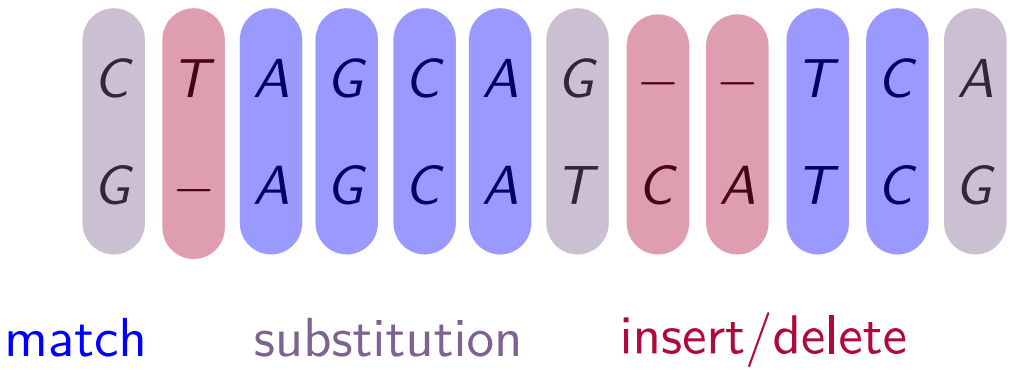


Exemple

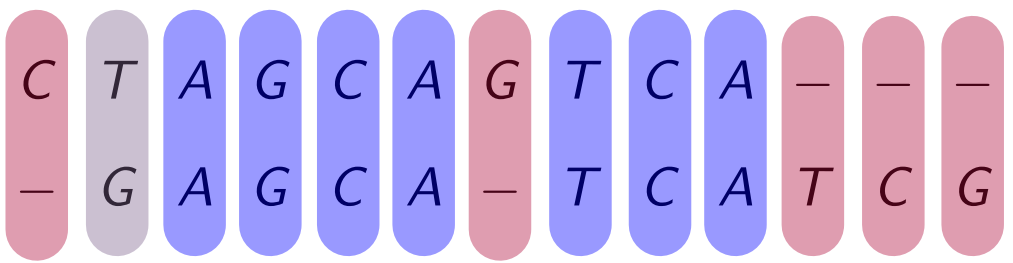
- soit deux séquences de tailles quelconques, la première de taille n et la deuxième de taille m :

$$\begin{array}{cccccccccc} C & T & A & G & C & A & G & T & C & A \\ G & A & G & C & A & T & C & A & T & C & G \end{array}$$

- un alignement :



- un autre alignement :



Quel alignement choisir ?

- à chaque opération élémentaire on associe un score :
  - match : 1
  - substitution : -1
  - insert/delete : -2

Quel alignement choisir ?

- à chaque opération élémentaire on associe un score :
  - match : 1
  - substitution : -1
  - insert/delete : -2

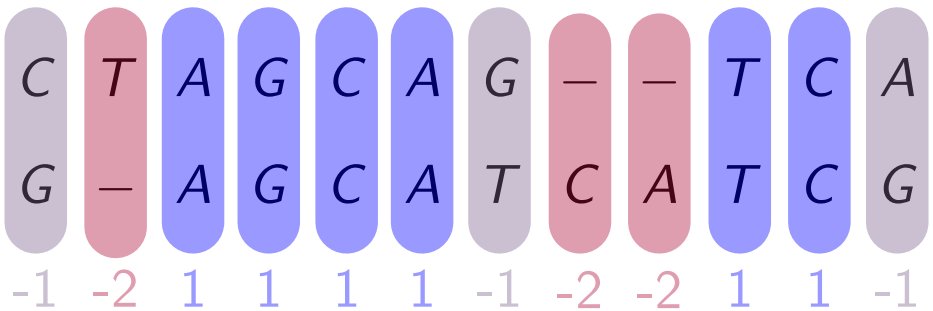
Le score d'un alignement est la somme des scores élémentaires

Quel alignement choisir ?

- à chaque opération élémentaire on associe un score :
  - match : 1
  - substitution : -1
  - insert/delete : -2

Le score d'un alignement est la somme des scores élémentaires

- 1^{er} alignement : -3



Quel alignement choisir ?

- à chaque opération élémentaire on associe un score :
  - match : 1
  - substitution : -1
  - insert/delete : -2

Le score d'un alignement est la somme des scores élémentaires

- 1^{er} alignement : -3

C	T	A	G	C	A	G	–	–	T	C	A
G	–	A	G	C	A	T	C	A	T	C	G
-1	-2	1	1	1	1	-1	-2	-2	1	1	-1

- 2^{eme} alignement : -4

C	T	A	G	C	A	G	T	C	A	–	–	–
–	G	A	G	C	A	–	T	C	A	T	C	G

Problème d'optimisation

Alignement de séquences

Soit :

- 2 séquences
- 3 scores associés aux opérations élémentaires (match, subst, ins/del)

Problème d'optimisation

Alignement de séquences

Soit :

- 2 séquences
- 3 scores associés aux opérations élémentaires (match, subst, ins/del)

Problème : trouver l'alignement de score maximal

Problème d'optimisation

Alignement de séquences

Soit :

- 2 séquences
- 3 scores associés aux opérations élémentaires (match, subst, ins/del)

Problème : trouver l'alignement de score maximal

Complexité exponentielle !

Nombre d'alignements : $\sum_{i=0}^{n} C_{m+i}^{i} \times C_{m}^{n-i} = \mathcal{O}(2^{n+m})$

Approche récursive

$$
\begin{cases}
Align([],[]) & = 0 \\
Align(S[0{:}n],[]) & = n \times score['ins/del'] \\
Align([],T[0{:}m]) & = m \times score['ins/del'] \\
Align(S[0{:}n],T[0{:}m]) & = max \left\{ \right.
\end{cases}
$$

Approche récursive

$$
\begin{cases}
Align([],[]) & = 0 \\
Align(S[0{:}n],[]) & = n \times score['ins/del'] \\
Align([],T[0{:}m]) & = m \times score['ins/del'] \\
Align(S[0{:}n],T[0{:}m]) & = max \begin{cases} \\ \\ \end{cases}
\end{cases}
$$

S :	C	T	A	G	C	A	G	T	C	A
T :	G	A	G	C	A	T	C	A	T	C	G

Approche récursive

$$
\begin{aligned}
Align([],[]) &= 0 \\
Align(S[0{:}n],[]) &= n \times score['ins/del'] \\
Align([],T[0{:}m]) &= m \times score['ins/del'] \\
Align(S[0{:}n],T[0{:}m]) &= max \begin{cases} Align(S[0{:}n],T[0{:}m-1]) + score['ins/del'] \\ \\ \\ \end{cases}
\end{aligned}
$$

S :	C	T	A	G	C	A	G	T	C	A
T :	G	A	G	C	A	T	C	A	T	C	G

C	T	A	G	C	A	G	T	C	A	$-$
G	A	G	C	A	T	C	A	T	C	G

-2

Approche récursive

$$
\begin{aligned}
Align([],[]) &= 0 \\
Align(S[0{:}n],[]) &= n \times score['ins/del'] \\
Align([],T[0{:}m]) &= m \times score['ins/del'] \\
Align(S[0{:}n],T[0{:}m]) &= max \begin{cases} Align(S[0{:}n],T[0{:}m-1]) + score['ins/del'] \\ Align(S[0{:}n-1],T[0{:}m]) + score['ins/del'] \end{cases}
\end{aligned}
$$

S :	C	T	A	G	C	A	G	T	C	A
T :	G	A	G	C	A	T	C	A	T	C	G

C	T	A	G	C	A	G	T	C	A	–
G	A	G	C	A	T	C	A	T	C	G

-2

C	T	A	G	C	A	G	T	C		A
G	A	G	C	A	T	C	A	T	C	G	–

-2

Approche récursive

$$
\begin{aligned}
Align([],[]) &= 0 \\
Align(S[0{:}n],[]) &= n \times score['ins/del'] \\
Align([],T[0{:}m]) &= m \times score['ins/del'] \\
Align(S[0{:}n],T[0{:}m]) &= max \begin{cases} Align(S[0{:}n],T[0{:}m-1]) + score['ins/del'] \\ Align(S[0{:}n-1],T[0{:}m]) + score['ins/del'] \\ Align(S[0{:}n-1],T[0{:}m-1]) + \begin{cases} score['match'] \; si \; S[n]=T[m] \\ score['subst'] \; si \; S[n]\neq T[m] \end{cases} \end{cases}
\end{aligned}
$$

S :	C	T	A	G	C	A	G	T	C	A
T :	G	A	G	C	A	T	C	A	T	C	G

C	T	A	G	C	A	G	T	C	A	–	
G	A	G	C	A	T	C	A	T	C	G	
											-2

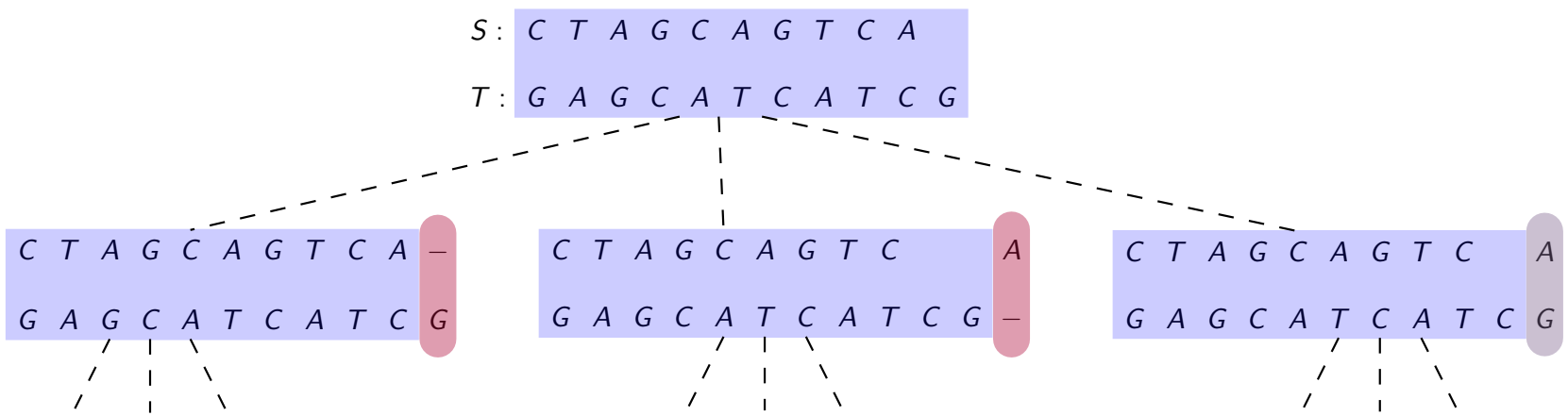
C	T	A	G	C	A	G	T	C		A	
G	A	G	C	A	T	C	A	T	C	G	–
											-2

C	T	A	G	C	A	G	T	C		A	
G	A	G	C	A	T	C	A	T	C	G	
											-1

Approche récursive

Complexité exponentielle !

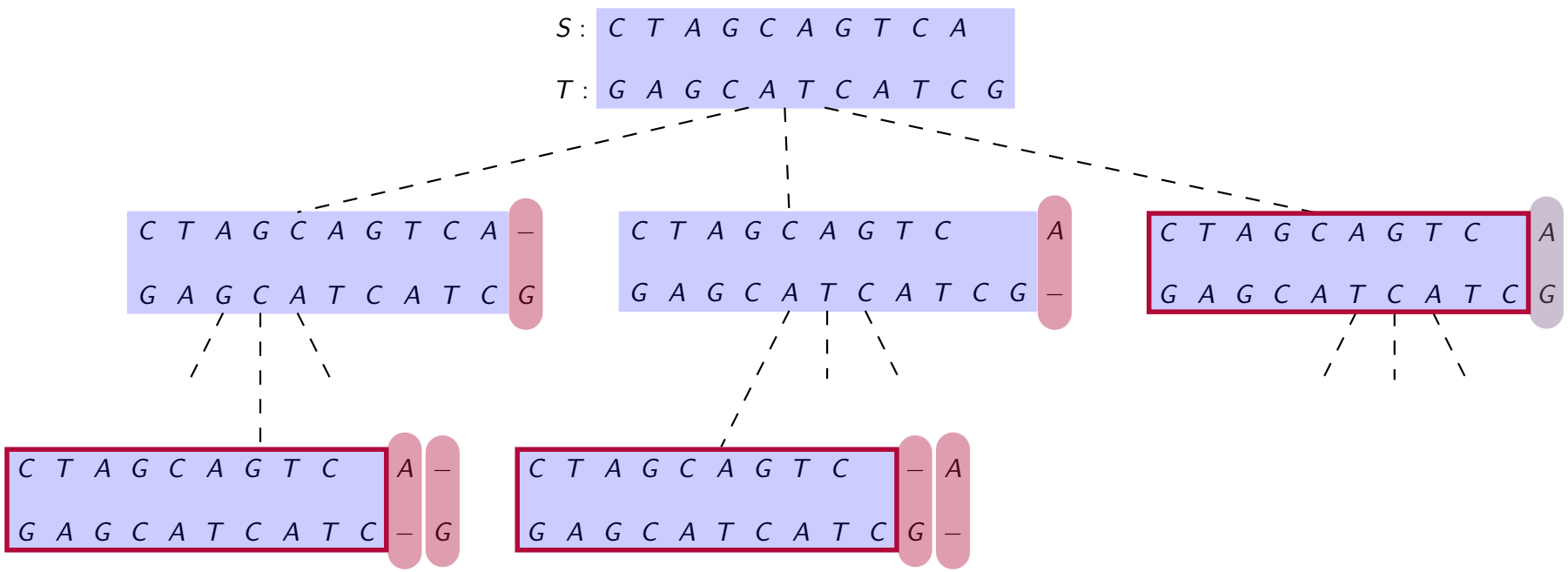
- Arbre d'exploration ternaire de profondeur $n + m$
- Complexité en $\mathcal{O}(3^{n+m})$



Approche récursive

Complexité exponentielle !

- Arbre d'exploration ternaire de profondeur $n + m$
- Complexité en $\mathcal{O}(3^{n+m})$
- Calculs redondants !



Formule de récurrence

Alignement de séquences

Notons $OPT(N, M)$ le score maximal

de l'alignement des **M** premiers nucléotides de la première séquence

avec les **N** premiers nucléotides de la deuxième séquence

- $OPT(0, 0) = 0$
- $OPT(N, M) =$ maximum parmi :

  $OPT(N-1, M) + (-2)$ si insert N^{e}

  $OPT(N, M-1) + (-2)$ si insert M^{e}

  $OPT(N-1, M-1) + (\pm 1)$ selon match ou supp

Algorithme Needleman and Wunsch (1970)

skip

S \ T		G	A	G	C	A	T	C	A	T	C	G
C												
T												
A												
G												
C												
A												
G												
T												
C												
A												

$C\ T\ A\ G\ C\ A\ G\ -\ -\ T\ C\ A$

$G\ -\ A\ G\ C\ A\ T\ C\ A\ T\ C\ G$

Algorithme Needleman and Wunsch (1970)

skip

S \ T		G	A	G	C	A	T	C	A	T	C	G
C												
T												
A												
G												
C												
A												
G												
T												
C												
A												

$C\ T\ A\ G\ C\ A\ G\ -\ -\ T\ C\ A$

$G\ -\ A\ G\ C\ A\ T\ C\ A\ T\ C\ G$

Algorithme Needleman and Wunsch (1970)

skip

S \ T		G	A	G	C	A	T	C	A	T	C	G
C												
T												
A												
G												
C												
A												
G												
T												
C												
A												

C T A G C A G – – T C A

G – A G C A T C A T C G

Algorithme Needleman and Wunsch (1970)

skip

S \ T		G	A	G	C	A	T	C	A	T	C	G
C												
T												
A												
G												
C												
A												
G												
T												
C												
A												

$C\ T\ A\ G\ C\ A\ G\ -\ -\ T\ C\ A$

$G\ -\ A\ G\ C\ A\ T\ C\ A\ T\ C\ G$

Algorithme Needleman and Wunsch (1970)

skip

S \ T		G	A	G	C	A	T	C	A	T	C	G
C												
T												
A												
G												
C												
A												
G												
T												
C												
A												

$C\ T\ A\ G\ C\ A\ G\ -\ -\ T\ C\ A$

$G\ -\ A\ G\ C\ A\ T\ C\ A\ T\ C\ G$

Algorithme Needleman and Wunsch (1970)

skip

S \ T		G	A	G	C	A	T	C	A	T	C	G
C												
T												
A												
G												
C												
A												
G												
T												
C												
A												

$$\begin{array}{cccccccccccc} C & T & A & G & C & A & G & - & - & T & C & A \\ G & - & A & G & C & A & T & C & A & T & C & G \end{array}$$

Algorithme Needleman and Wunsch (1970)

skip

S \ T		G	A	G	C	A	T	C	A	T	C	G
C												
T												
A												
G												
C												
A												
G												
T												
C												
A												

$$\begin{array}{cccccccccccc} C & T & A & G & C & A & G & - & - & T & C & A \\ G & - & A & G & C & A & T & C & A & T & C & G \end{array}$$

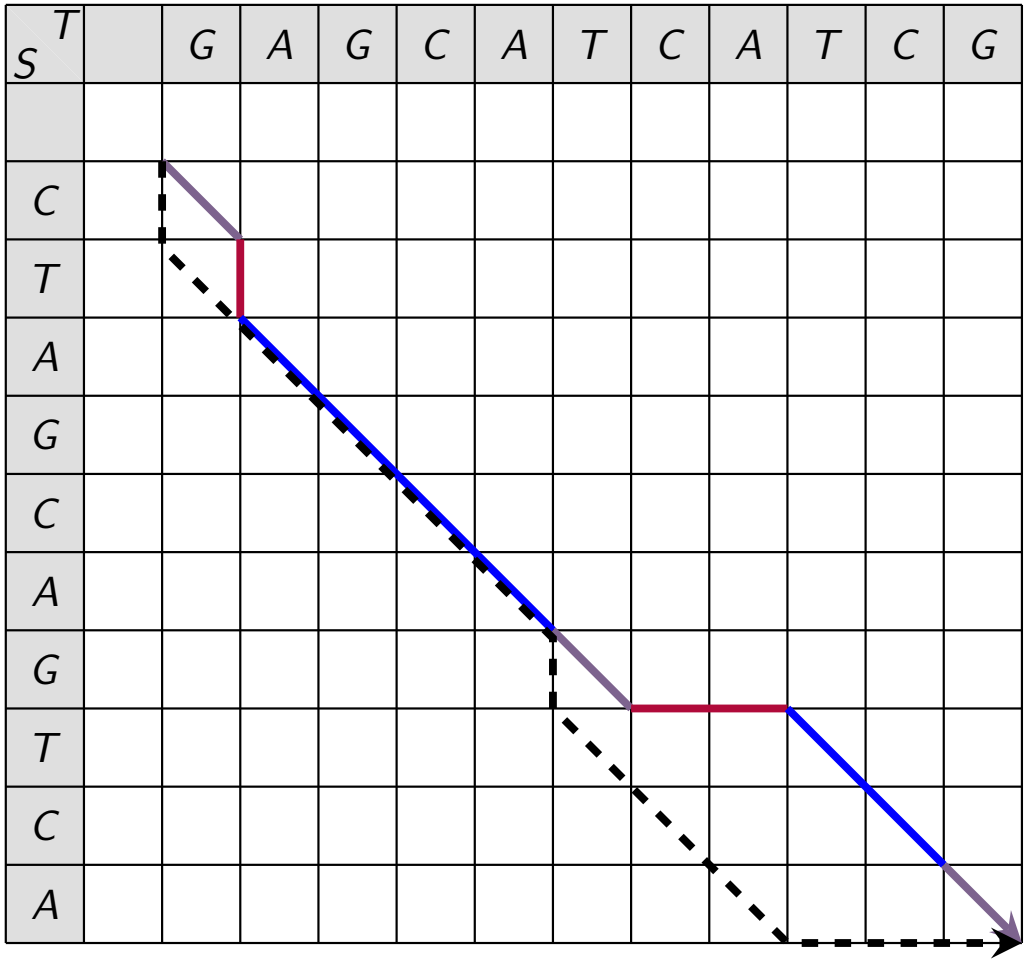
Algorithme Needleman and Wunsch (1970)

skip

S \ T		G	A	G	C	A	T	C	A	T	C	G
C												
T												
A												
G												
C												
A												
G												
T												
C												
A												

C	T	A	G	C	A	G	–	–	T	C	A
G	–	A	G	C	A	T	C	A	T	C	G

Algorithme Needleman and Wunsch (1970)



C	T	A	G	C	A	G	–	–	T	C	A
G	–	A	G	C	A	T	C	A	T	C	G

C T A G C A G T C A – – –

– G A G C A – T C A T C G

Algorithme Needleman and Wunsch (1970)

skip

S \ T		G	A	G	C	A	T	C	A	T	C	G
	0											
C												
T												
A												
G												
C												
A												
G												
T												
C												
A												

Objectif : trouver l'alignement de score maximal

Algorithme Needleman and Wunsch (1970)

skip

S \ T		G	A	G	C	A	T	C	A	T	C	G
	0	−2	−4	−6	−8	−10	−12	−14	−16	−18	−20	−22
C	−2											
T	−4											
A	−6											
G	−8											
C	−10											
A	−12											
G	−14											
T	−16											
C	−18											
A	−20											

Étape 1 : on remplit la première ligne et la première colonne

- ici score['ins/del'] = -2 ($\rightarrow$)

Algorithme Needleman and Wunsch (1970)

skip

S \ T		G	A	G	C	A	T	C	A	T	C	G
	0	−2	−4	−6	−8	−10	−12	−14	−16	−18	−20	−22
C	−2											
T	−4											
A	−6											
G	−8											
C	−10											
A	−12											
G	−14											
T	−16											
C	−18											
A	−20											

S \ T		G
	0	−2
C	−2	

Étape 2 : on remplit chacune des cases en maximisant selon les 3 axes

- ici score['match'] = 1 ($\rightarrow$) et score['subst'] = -1 ($\rightarrow$)

Algorithme Needleman and Wunsch (1970)

skip

S \ T		G	A	G	C	A	T	C	A	T	C	G
	0	−2	−4	−6	−8	−10	−12	−14	−16	−18	−20	−22
C	−2											
T	−4											
A	−6											
G	−8											
C	−10											
A	−12											
G	−14											
T	−16											
C	−18											
A	−20											

S \ T		G
	0	−2
C	−2	−4

Étape 2 : on remplit chacune des cases en maximisant selon les 3 axes

- ici score['match'] = 1 ($\rightarrow$) et score['subst'] = -1 ($\rightarrow$)

Algorithme Needleman and Wunsch (1970)

skip

S \ T		G	A	G	C	A	T	C	A	T	C	G
	0	−2	−4	−6	−8	−10	−12	−14	−16	−18	−20	−22
C	−2											
T	−4											
A	−6											
G	−8											
C	−10											
A	−12											
G	−14											
T	−16											
C	−18											
A	−20											

S \ T		G
	0	−2
C	−2	−1

Étape 2 : on remplit chacune des cases en maximisant selon les 3 axes

- ici score['match'] = 1 ($\rightarrow$) et score['subst'] = -1 ($\rightarrow$)

Algorithme Needleman and Wunsch (1970)

skip

S \ T		G	A	G	C	A	T	C	A	T	C	G
	0	−2	−4	−6	−8	−10	−12	−14	−16	−18	−20	−22
C	−2											
T	−4											
A	−6											
G	−8											
C	−10											
A	−12											
G	−14											
T	−16											
C	−18											
A	−20											

S \ T		G
	0	−2
C	−2	−4

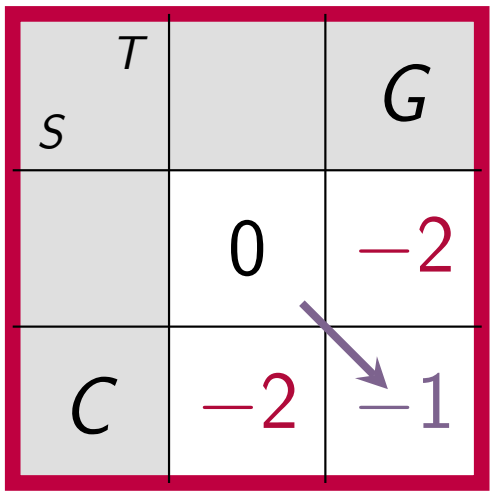
Étape 2 : on remplit chacune des cases en maximisant selon les 3 axes

- ici score['match'] = 1 (→) et score['subst'] = -1 (→)

Algorithme Needleman and Wunsch (1970)

skip

S \ T		G	A	G	C	A	T	C	A	T	C	G
	0	−2	−4	−6	−8	−10	−12	−14	−16	−18	−20	−22
C	−2	−1										
T	−4											
A	−6											
G	−8											
C	−10											
A	−12											
G	−14											
T	−16											
C	−18											
A	−20											



Étape 2 : on remplit chacune des cases en maximisant selon les 3 axes

- ici score['match'] = 1 ($\rightarrow$) et score['subst'] = -1 ($\rightarrow$)

Algorithme Needleman and Wunsch (1970)

skip

S \ T		G	A	G	C	A	T	C	A	T	C	G
	0	−2	−4	−6	−8	−10	−12	−14	−16	−18	−20	−22
C	−2	-1	-3	-5	-5	-7	-9	-11	-13	-15	-17	-19
T	−4	-3	-2	-4	-6	-6	-6	-8	-10	-12	-14	-16
A	−6	-5	-2	-3	-5	-5	-7	-7	-7	-9	-11	-13
G	−8	-5	-4	-1	-3	-6	-6	-8	-8	-8	-10	-10
C	−10	-7	-6	-3	0	-2	-4	-5	-7	-9	-7	-9
A	−12	-9	-6	-5	-2	1	-1	-3	-4	-6	-8	-8
G	−14	-11	-8	-5	-4	-1	0	-2	-4	-5	-7	-7
T	−16	-13	-10	-7	-6	-3	0	-1	-3	-3	-5	-7
C	−18	-15	-12	-9	-6	-5	-2	1	-1	-3	-2	-4
A	−20	-17	-14	-11	-8	-5	-4	-1	2	0	-2	

Étape 2 : on remplit chacune des cases en maximisant selon les 3 axes

- ici score['match'] = 1 ($\rightarrow$) et score['subst'] = -1 ($\rightarrow$)

Algorithme Needleman and Wunsch (1970)

skip

S \ T		G	A	G	C	A	T	C	A	T	C	G
	0	−2	−4	−6	−8	−10	−12	−14	−16	−18	−20	−22
C	−2	-1	-3	-5	-5	-7	-9	-11	-13	-15	-17	-19
T	−4	-3	-2	-4	-6	-6	-6	-8	-10	-12	-14	-16
A	−6	-5	-2	-3	-5	-5	-7	-7	-7	-9	-11	-13
G	−8	-5	-4	-1	-3	-6	-6	-8	-8	-8	-10	-10
C	−10	-7	-6	-3	0	-2	-4	-5	-7	-9	-7	-9
A	−12	-9	-6	-5	-2	1	-1	-3	-4	-6	-8	-8
G	−14	-11	-8	-5	-4	-1	0	-2	-4	-5	-7	-7
T	−16	-13	-10	-7	-6	-3	0	-1	-3	-3	-5	-7
C	−18	-15	-12	-9	-6	-5	-2	1	-1	-3	-2	-4
A	−20	-17	-14	-11	-8	-5	-4	-1	2	0	-2	-3

Étape 3 : au bout du tableau, on obtient le score maximal ainsi que les alignements optimaux

Algorithme Needleman and Wunsch (1970)

S \ T		G	A	G	C	A	T	C	A	T	C	G
	0	−2	−4	−6	−8	−10	−12	−14	−16	−18	−20	−22
C	−2	-1	-3	-5	-5	-7	-9	-11	-13	-15	-17	-19
T	−4	-3	-2	-4	-6	-6	-6	-8	-10	-12	-14	-16
A	−6	-5	-2	-3	-5	-5	-7	-7	-7	-9	-11	-13
G	−8	-5	-4	-1	-3	-6	-6	-8	-8	-8	-10	-10
C	−10	-7	-6	-3	0	-2	-4	-5	-7	-9	-7	-9
A	−12	-9	-6	-5	-2	1	-1	-3	-4	-6	-8	-8
G	−14	-11	-8	-5	-4	-1	0	-2	-4	-5	-7	-7
T	−16	-13	-10	-7	-6	-3	0	-1	-3	-3	-5	-7
C	−18	-15	-12	-9	-6	-5	-2	1	-1	-3	-2	-4
A	−20	-17	-14	-11	-8	-5	-4	-1	2	0	-2	-3

Étape 3 : au bout du tableau, on obtient le score maximal ainsi que les alignements optimaux

Algorithme Needleman and Wunsch (1970)

- Parmi les 3^{n+m} chemins possibles dans la matrice on a trouvé l'alignement optimal en $n \times m$ étapes

Algorithme Needleman and Wunsch (1970)

- Parmi les 3^{n+m} chemins possibles dans la matrice on a trouvé l'alignement optimal en $n \times m$ étapes

Complexité de l'algorithme

- $\mathcal{O}(n \times m)$ en temps : taille de la matrice
- $\mathcal{O}(min(n, m))$ en espace : au lieu de conserver toute la matrice, on ne garde que la ligne courante et la ligne précédente