

M1 Info - Optimisation et Recherche Opérationnelle

# Cours 4 - Programmation dynamique

Le voyageur de commerce

Semestre Automne 2021-2022 - Université Claude Bernard Lyon 1

Christophe Crespelle

`christophe.crespelle@univ-lyon1.fr`



département

**Informatique**

Faculté des Sciences et Technologies

Université Claude Bernard Lyon 1

## Problème du voyageur de commerce

- **Entrée** : un ensemble de villes  $\{c_1, c_2, \dots, c_n\}$  et une fonction de distance  $d(c_i, c_j)$  définie pour tous les couples  $(c_i, c_j)$ .

# Problème du voyageur de commerce

- **Entrée** : un ensemble de villes  $\{c_1, c_2, \dots, c_n\}$  et une fonction de distance  $d(c_i, c_j)$  définie pour tous les couples  $(c_i, c_j)$ .

## Définition (Tour et tour minimum)

Un tour est une permutation  $\pi = (c_{\pi(1)}, c_{\pi(2)}, \dots, c_{\pi(n)})$  des villes.

La longueur d'un tour  $\pi$  est

$$\sum_{i \in \llbracket 1, n-1 \rrbracket} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(n)}, c_{\pi(1)}).$$

Un tour minimum est un tour dont la longueur est minimum.

# Problème du voyageur de commerce

- **Entrée** : un ensemble de villes  $\{c_1, c_2, \dots, c_n\}$  et une fonction de distance  $d(c_i, c_j)$  définie pour tous les couples  $(c_i, c_j)$ .

## Définition (Tour et tour minimum)

Un tour est une permutation  $\pi = (c_{\pi(1)}, c_{\pi(2)}, \dots, c_{\pi(n)})$  des villes.

La longueur d'un tour  $\pi$  est

$$\sum_{i \in \llbracket 1, n-1 \rrbracket} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(n)}, c_{\pi(1)}).$$

Un tour minimum est un tour dont la longueur est minimum.

- **Sortie** : un tour minimum de  $\{c_1, c_2, \dots, c_n\}$ .

# Problème du voyageur de commerce

## Applications classiques :

- en logistique : optimisation des transports

# Problème du voyageur de commerce

## Applications classiques :

- en logistique : optimisation des transports
- en électronique : minimisation des cablages entre composants des circuits intégrés

# Problème du voyageur de commerce

## Applications classiques :

- en logistique : optimisation des transports
- en électronique : minimisation des câblages entre composants des circuits intégrés

## Difficulté de calcul : **NP-complet**

# Problème du voyageur de commerce

## Applications classiques :

- en logistique : optimisation des transports
- en électronique : minimisation des cablages entre composants des circuits intégrés

## Difficulté de calcul : **NP-complet**

On va faire un algorithme exponentiel ( $O^*(2^n)$ ) pour le résoudre, par la programmation dynamique.

# Approche brute force

## **Algo brute force :**

- on essaye un par un tous les tours (= permutations des villes),
- pour chacun, on calcule sa longueur,
- on garde un tour qui realise le minimum de la longueur.

# Approche brute force

## Algo brute force :

- on essaye un par un tous les tours (= permutations des villes),
- pour chacun, on calcule sa longueur,
- on garde un tour qui realise le minimum de la longueur.

## Complexite :

- il y a  $n!$  tour  $\pi$  de  $\{c_1, c_2, \dots, c_n\}$  (nombre de permutations sur  $n$  elements)

# Approche brute force

## Algo brute force :

- on essaye un par un tous les tours (= permutations des villes),
- pour chacun, on calcule sa longueur,
- on garde un tour qui realise le minimum de la longueur.

## Complexite :

- il y a  $n!$  tour  $\pi$  de  $\{c_1, c_2, \dots, c_n\}$  (nombre de permutations sur  $n$  elements)
- calculer la longueur de  $\pi$  prend  $O(n)$

Total :  $O(n \cdot n!)$

# Approche brute force

## Algo brute force :

- on essaye un par un tous les tours (= permutations des villes),
- pour chacun, on calcule sa longueur,
- on garde un tour qui realise le minimum de la longueur.

## Complexite :

- il y a  $n!$  tour  $\pi$  de  $\{c_1, c_2, \dots, c_n\}$  (nombre de permutations sur  $n$  elements)
- calculer la longueur de  $\pi$  prend  $O(n)$

Total :  $O(n \cdot n!)$

On va faire un algo en  $2^n$  par la programmation dynamique.

## Gain de complexite

**Note importante** : taille de  $n!$  comparee a  $2^n$  ?

## Gain de complexite

**Note importante** : taille de  $n!$  comparee a  $2^n$  ?

**Maths** (Stirling)  $n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$

Retenez  $n! = n^n$  : enorme, bien plus gros que  $2^n$

## Gain de complexite

**Note importante** : taille de  $n!$  comparee a  $2^n$  ?

**Maths** (Stirling)  $n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$

Retenez  $n! = n^n$  : enorme, bien plus gros que  $2^n$

**Conclusion** : complexite en  $n!$  est redhibitoire.

On va faire un algo exponentiel, en  $2^n$ , par la programmation dynamique.

# Programmation dynamique (Richard Bellman 1950's)

**Idee generale** : stocker dans une table tous les resultats intermediaires

# Programmation dynamique (Richard Bellman 1950's)

**Idee generale** : stocker dans une table tous les resultats intermediaires

**Conditions de mise en oeuvre** : il faut une "formule de recurrence" qui permette, a partir des resultats intermediaires plus petits, de deduire les resultats intermediaires un peu plus gros

# Programmation dynamique (Richard Bellman 1950's)

**Idee generale** : stocker dans une table tous les resultats intermediaires

**Conditions de mise en oeuvre** : il faut une "formule de recurrence" qui permette, a partir des resultats intermediaires plus petits, de deduire les resultats intermediaires un peu plus gros

- avec peu de temps de calcul (ici, polynomial)

# Programmation dynamique (Richard Bellman 1950's)

**Idee generale** : stocker dans une table tous les resultats intermediaires

**Conditions de mise en oeuvre** : il faut une "formule de recurrence" qui permette, a partir des resultats intermediaires plus petits, de deduire les resultats intermediaires un peu plus gros

- avec peu de temps de calcul (ici, polynomial)
- et surtout, sans stocker trop de resultats intermediaires

# Programmation dynamique (Richard Bellman 1950's)

**Idee generale** : stocker dans une table tous les resultats intermediaires

**Conditions de mise en oeuvre** : il faut une "formule de recurrence" qui permette, a partir des resultats intermediaires plus petits, de deduire les resultats intermediaires un peu plus gros

- avec peu de temps de calcul (ici, polynomial)
- et surtout, sans stocker trop de resultats intermediaires

**gain r/t a brute force** : comme on stocke, on ne perd pas de temps a recalculer les resultats intermediaires

# Programmation dynamique (Richard Bellman 1950's)

**Idee generale** : stocker dans une table tous les resultats intermediaires

**Conditions de mise en oeuvre** : il faut une "formule de recurrence" qui permette, a partir des resultats intermediaires plus petits, de deduire les resultats intermediaires un peu plus gros

- avec peu de temps de calcul (ici, polynomial)
- et surtout, sans stocker trop de resultats intermediaires

**gain r/t a brute force** : comme on stocke, on ne perd pas de temps a recalculer les resultats intermediaires

**contrepartie** : ca prend beaucoup d'espace (en fait on echange de l'espace contre du temps de calcul)

# Programmation dynamique pour le voyageur de commerce

## Remarque

*On peut toujours commencer le tour sur la ville de notre choix : on choisit  $c_1$ .*

# Programmation dynamique pour le voyageur de commerce

## Remarque

*On peut toujours commencer le tour sur la ville de notre choix : on choisit  $c_1$ .*

## Définition

Pour  $S \subseteq \{c_2, \dots, c_n\}$  et  $c_i \in S$ , on note  $OPT[S, c_i]$  la longueur minimum d'un parcours qui :

- commence en  $c_1$
- parcourt les villes de  $S$ , dans un ordre libre
- finit en  $c_i$

# Programmation dynamique pour le voyageur de commerce

## La formule de recurrence :

- si  $|S| = 1$ , c.a.d.  $S = \{c_i\}$ ,  $i \neq 1$ , on a  $OPT[S, c_i] = d(c_1, c_j)$ .

# Programmation dynamique pour le voyageur de commerce

## La formule de recurrence :

- si  $|S| = 1$ , c.a.d.  $S = \{c_i\}$ ,  $i \neq 1$ , on a  $OPT[S, c_i] = d(c_1, c_j)$ .
- si  $|S| > 1$ , alors

$$OPT[S, c_i] = \min_{c_j \in S \setminus \{c_i\}} \{OPT[S \setminus \{c_i\}, c_j] + d(c_j, c_i)\}$$

# Programmation dynamique pour le voyageur de commerce

## La formule de recurrence :

- si  $|S| = 1$ , c.a.d.  $S = \{c_i\}$ ,  $i \neq 1$ , on a  $OPT[S, c_i] = d(c_1, c_j)$ .

- si  $|S| > 1$ , alors

$$OPT[S, c_i] = \min_{c_j \in S \setminus \{c_i\}} \{OPT[S \setminus \{c_i\}, c_j] + d(c_j, c_i)\}$$

**La reponse au probleme :** la longueur d'un tour minimum est

$$OPT = \min_{i \in \llbracket 2, n \rrbracket} \{OPT[\{c_2, \dots, c_n\}, c_i] + d(c_i, c_1)\}.$$

# Algorithme pour le voyageur de commerce

---

## Algorithme 1 : Algorithme pour TSP

---

```
1 pour  $i$  de 2 a  $n$  faire
2   |  $OPT[\{c_i\}, c_i] \leftarrow d(c_1, c_i);$ 
3 fin
4 pour  $j$  de 2 a  $n - 1$  faire
5   | pour tous les  $S \subseteq \{c_2, \dots, c_n\}$  avec  $|S| = j$  faire
6     |   | pour tous les  $c_i \in S$  faire
7       |   |   |  $OPT[S, c_i] = \min_{c_j \in S \setminus \{c_i\}} \{OPT[S \setminus \{c_i\}, c_j] + d(c_j, c_i)\}$ 
8     |   |   | fin
9   |   | fin
10 fin
11 retourner  $\min_{i \in \llbracket 2, n \rrbracket} \{OPT[\{c_2, \dots, c_n\}, c_i] + d(c_i, c_1)\};$ 
```

---

# Analyse de la complexite

## Complexite temporelle :

- pour un  $S$  de taille  $k$ , les lignes 6 a 8 prennent un temps  $O(k^2)$

# Analyse de la complexite

## Complexite temporelle :

- pour un  $S$  de taille  $k$ , les lignes 6 a 8 prennent un temps  $O(k^2)$
- combien d'ensemble  $S$  a considerer ?

# Analyse de la complexite

## Complexite temporelle :

- pour un  $S$  de taille  $k$ , les lignes 6 a 8 prennent un temps  $O(k^2)$
- combien d'ensemble  $S$  a considerer ?  $\longrightarrow 2^{n-1} - (n - 1)$

# Analyse de la complexite

## Complexite temporelle :

- pour un  $S$  de taille  $k$ , les lignes 6 a 8 prennent un temps  $O(k^2)$
- combien d'ensemble  $S$  a considerer ?  $\rightarrow 2^{n-1} - (n - 1)$
- **Total** :  $O(n^2 \cdot 2^n) = O^*(2^n)$  ... beaucoup mieux que  $O(n!)$

# Analyse de la complexite

## Complexite temporelle :

- pour un  $S$  de taille  $k$ , les lignes 6 a 8 prennent un temps  $O(k^2)$
- combien d'ensemble  $S$  a considerer?  $\rightarrow 2^{n-1} - (n - 1)$
- **Total** :  $O(n^2 \cdot 2^n) = O^*(2^n)$  ... beaucoup mieux que  $O(n!)$

## Complexite spatiale :

- le tableau  $OPT[S, c_i]$  a une case pour chaque couple  $(S, c_i)$

# Analyse de la complexite

## Complexite temporelle :

- pour un  $S$  de taille  $k$ , les lignes 6 a 8 prennent un temps  $O(k^2)$
- combien d'ensemble  $S$  a considerer ?  $\longrightarrow 2^{n-1} - (n - 1)$
- **Total** :  $O(n^2 \cdot 2^n) = O^*(2^n)$  ... beaucoup mieux que  $O(n!)$

## Complexite spatiale :

- le tableau  $OPT[S, c_i]$  a une case pour chaque couple  $(S, c_i)$
- mais... quand on en est a  $j$ , on peut ne conserver que les  $S$  de taille  $j$  et  $j - 1$

# Analyse de la complexite

## Complexite temporelle :

- pour un  $S$  de taille  $k$ , les lignes 6 a 8 prennent un temps  $O(k^2)$
- combien d'ensemble  $S$  a considerer ?  $\longrightarrow 2^{n-1} - (n - 1)$
- **Total** :  $O(n^2 \cdot 2^n) = O^*(2^n)$  ... beaucoup mieux que  $O(n!)$

## Complexite spatiale :

- le tableau  $OPT[S, c_i]$  a une case pour chaque couple  $(S, c_i)$
- mais... quand on en est a  $j$ , on peut ne conserver que les  $S$  de taille  $j$  et  $j - 1$
- re-mais... ca fait quand meme  $\Omega(2^n)$  pour  $j = n/2$ .

# Analyse de la complexite

## Complexite temporelle :

- pour un  $S$  de taille  $k$ , les lignes 6 a 8 prennent un temps  $O(k^2)$
- combien d'ensemble  $S$  a considerer ?  $\rightarrow 2^{n-1} - (n - 1)$
- **Total** :  $O(n^2 \cdot 2^n) = O^*(2^n)$  ... beaucoup mieux que  $O(n!)$

## Complexite spatiale :

- le tableau  $OPT[S, c_i]$  a une case pour chaque couple  $(S, c_i)$
- mais... quand on en est a  $j$ , on peut ne conserver que les  $S$  de taille  $j$  et  $j - 1$
- re-mais... ca fait quand meme  $\Omega(2^n)$  pour  $j = n/2$ .
- **au pire de l'algo** : espace  $\Omega(n \cdot 2^n)$ ... c'est la ou le bat blaisse

# Analyse de la complexite

## Complexite temporelle :

- pour un  $S$  de taille  $k$ , les lignes 6 a 8 prennent un temps  $O(k^2)$
- combien d'ensemble  $S$  a considerer ?  $\rightarrow 2^{n-1} - (n - 1)$
- **Total** :  $O(n^2 \cdot 2^n) = O^*(2^n)$  ... beaucoup mieux que  $O(n!)$

## Complexite spatiale :

- le tableau  $OPT[S, c_i]$  a une case pour chaque couple  $(S, c_i)$
- mais... quand on en est a  $j$ , on peut ne conserver que les  $S$  de taille  $j$  et  $j - 1$
- re-mais... ca fait quand meme  $\Omega(2^n)$  pour  $j = n/2$ .
- **au pire de l'algo** : espace  $\Omega(n \cdot 2^n)$ ... c'est la ou le bat blaisse

**Conclusion** : la prog dynamique permet de gagner du temps en consommant de l'espace

# Analyse de la complexite

## Complexite temporelle :

- pour un  $S$  de taille  $k$ , les lignes 6 a 8 prennent un temps  $O(k^2)$
- combien d'ensemble  $S$  a considerer ?  $\rightarrow 2^{n-1} - (n - 1)$
- **Total** :  $O(n^2 \cdot 2^n) = O^*(2^n)$  ... beaucoup mieux que  $O(n!)$

## Complexite spatiale :

- le tableau  $OPT[S, c_i]$  a une case pour chaque couple  $(S, c_i)$
- mais... quand on en est a  $j$ , on peut ne conserver que les  $S$  de taille  $j$  et  $j - 1$
- re-mais... ca fait quand meme  $\Omega(2^n)$  pour  $j = n/2$ .
- **au pire de l'algo** : espace  $\Omega(n \cdot 2^n)$ ... c'est la ou le bat blaisse

**Conclusion** : la prog dynamique permet de gagner du temps en consommant de l'espace

**Limites** : l'espace est aussi une quantite critique dans les ordinateurs (au moins autant que le temps)