M1 Info - Graphes et programmation dynamique

Cours 7 - voyageur de commerce, cycle et chemin hamiltonien

Programmation dynamique

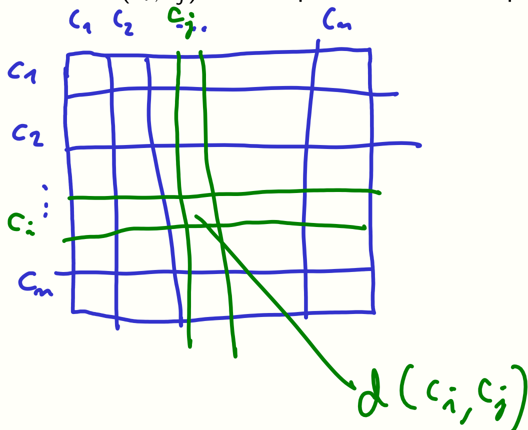
Semestre 1 – Année 2022-2023 – Université Côte D'azur

Christophe Crespelle

christophe.crespelle@univ-cotedazur.fr

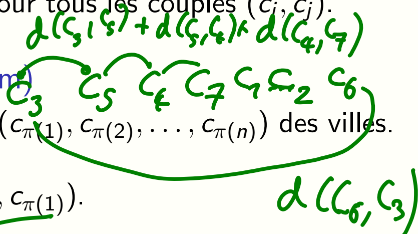
Problème du voyageur de commerce

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



Problème du voyageur de commerce

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



Définition (Tour et tour minimum)

Un tour est une permutation $\pi = (c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(n)})$ des villes.
La longueur d'un tour π est

$$\sum_{i \in [\![1, n-1]\!]} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(n)}, c_{\pi(1)}).$$

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

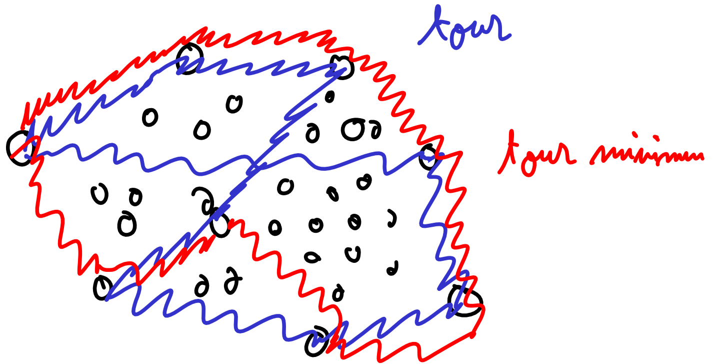
Problème du voyageur de commerce

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

Définition (Tour et tour minimum)

Un tour est une permutation $\pi = (c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(n)})$ des villes.
La longueur d'un tour π est
$\sum_{i \in [\![1, n-1]\!]} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(n)}, c_{\pi(1)})$.
Un tour minimum est un tour dont la longueur est minimum.

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



voyageur de commerce = T. S. P.

travelling salesman problem.

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 electronique : minimisation des cablages entre composants des circuits integres

Problème du voyageur de commerce

Applications classiques :

- en logistique : optimisation des transports
- en electronique : minimisation des cablages entre composants des circuits integres

Difficulte de calcul : **NP-complet**

Problème du voyageur de commerce

Applications classiques :

- en logistique : optimisation des transports
- en electronique : minimisation des cablages entre composants des circuits integres

Difficulte de calcul : **NP-complet**

$O(2^n)$ $\leq K.2^n$

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

$\leq \text{poly}(n) \times 1.5^n$ (n^{100}, 1.99)

$\leq \text{poly}(n) \times 2^n$ (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.

c_i, c_j $O(n)$

$C_{\pi(1)}$ $C_{\pi(2)}$ $C_{\pi(3)}$ . . . $C_{\pi(n)}$

$n \times n-1 \times n-2 \times \cdots \times 1$

de permutations sur n villes : $n!$

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 π de $\{c_1, c_2, \ldots, c_n\}$ (nombre de permutations sur n elements)

Approche brute force

Algo brute force :

$n!$

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

Complexite :

- il y a $n!$ tour π de $\{c_1, c_2, \ldots, c_n\}$ (nombre de permutations sur n elements)
- calculer la longueur de π 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 π de $\{c_1, c_2, \ldots, c_n\}$ (nombre de permutations sur n elements)
- calculer la longueur de π 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 ?

$n \times n-1 \times n-2 \quad 4 \quad 3 \times 2 \times 1$

$2 \times 2 \times 2 \times 2 \times 2 \quad 2 \quad 2$

Gain de complexite

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

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

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

n^n $\sqrt{2\pi n}$ e^n

n^n 2^n $n = 20$

$20^{20} = 2^{20} \times 10^{20}$ $2^{20} = (2^{10})^2 \approx 1M$

10^{20} = 1 000 000 000 00

100 M de M

Pour $n=20$ 20^{20} c'est 100 millions de milliards de fois plus gros que 2^{20}

Gain de complexite

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

Maths (Stirling) $n! \sim \sqrt{2\pi n}(\frac{n}{e})^n$
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 reccurence" 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 reccurence" 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 reccurence" 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 reccurence" 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 reccurence" 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

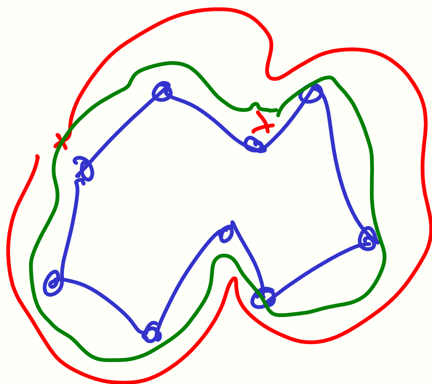
gain r/t a brute force : comme on stocke, on ne perd pas de temps a recalculer les resultats intermediaires

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

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

Programmation dynamique pour le voyageur de commerce

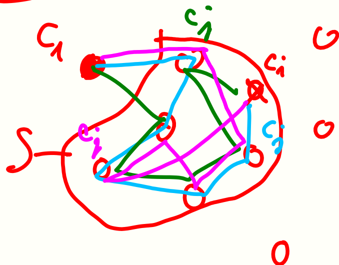
La formule de recurrence :

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

Programmation dynamique pour le voyageur de commerce

La formule de recurrence :

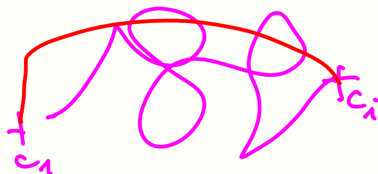
- si $|S| = 1$, c.a.d. $S = \{c_i\}$, $i \neq 1$, on a $OPT[S, c_i] = d(c_1, c_j)$.
- si $|S| > 1$, alors
  $OPT[S, c_i] = \min_{c_j \in S \setminus \{c_i\}} \{OPT[S \setminus \{c_i\}, c_j] + d(c_j, c_i)\}$



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
$OPT = \min_{i \in [\![2,n]\!]} \{OPT[\{c_2, \ldots, c_n\}, c_i] + d(c_i, c_1)\}$

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] ← d(c_1, c_i);
3  fin
4  pour j de 2 a n − 1 faire
5  |   pour tous les S ⊆ {c_2, ..., c_n} avec |S| = j faire
6  |   |   pour tous les c_i ∈ S faire
7  |   |   |   OPT[S, c_i] = min_{c_j ∈ S\{c_i}} {OPT[S \ {c_i}, c_j] + d(c_j, c_i)}
8  |   |   fin
9  |   fin
10 fin
11 retourner min_{i ∈ ⟦2,n⟧} {OPT[{c_2, ..., c_n}, c_i] + d(c_i, c_1)};
```

Algorithme pour le voyageur de commerce

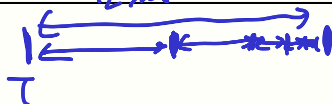
Algorithme 1 : Algorithme pour TSP

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* $S \subseteq \{c_2, \ldots, c_n\}$ *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** $\min_{i \in [\![2,n]\!]} \{OPT[\{c_2, \ldots, c_n\}, c_i] + d(c_i, c_1)\}$;

Algorithme pour le voyageur de commerce

Algorithme 1 : Algorithme pour TSP

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



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 ? $\longrightarrow 2^{n-1} - (n-1)$
- **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 ? $\longrightarrow 2^{n-1} - (n-1)$
- **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)

$$O(n \times 2^n)$$

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 ? $\longrightarrow 2^{n-1} - (n-1)$
- **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

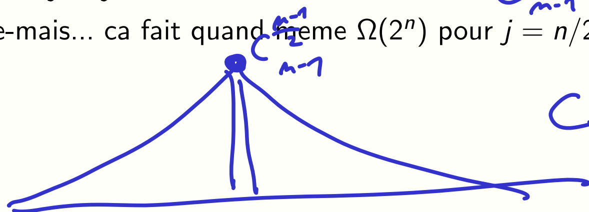
Complexite temporelle :

- 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 $j = n/2$.

C_{n-1}^{k} $\quad k = \frac{n-1}{2}$



$C_{n-1}^{\frac{n-1}{2}} \sim 2^n$

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 ? $\longrightarrow 2^{n-1} - (n-1)$
- **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 $j = n/2$.
- **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 ? $\longrightarrow 2^{n-1} - (n-1)$
- **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 $j = n/2$.
- **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 ? $\longrightarrow 2^{n-1} - (n-1)$
- **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 $j = n/2$.
- **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)

Problème du cycle hamiltonien

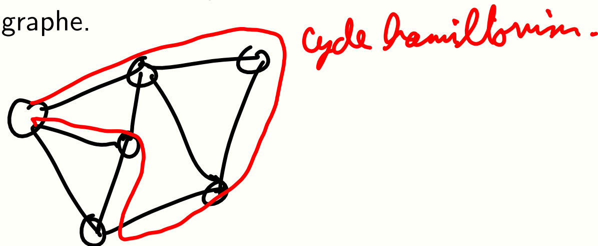
- **Entrée :** un graphe G (non-oriente, sans boucle, sans arete multiple).

Problème du cycle hamiltonien

- **Entrée :** un graphe G (non-oriente, sans boucle, sans arete multiple).

Définition (Cycle hamiltonien)

Un *cycle hamiltonien* est un cycle simple qui contient tous les sommets du graphe.



Problème du cycle hamiltonien

- **Entrée :** un graphe G (non-oriente, sans boucle, sans arete multiple).

Définition (Cycle hamiltonien)

Un *cycle hamiltonien* est un cycle simple qui contient tous les sommets du graphe.

- **Sortie :** OUI si G contient un cycle hamiltonien, NON sinon.

Algorithme pour cycle hamiltonien (reduction a TSP)

Algorithme :

1. Transformer le graphe donne en une instance de TSP.

Algorithme pour cycle hamiltonien (reduction a TSP)

Algorithme :

1. Transformer le graphe donne en une instance de TSP.

2. Resoudre TSP sur cette instance.

Algorithme pour cycle hamiltonien (reduction a TSP)

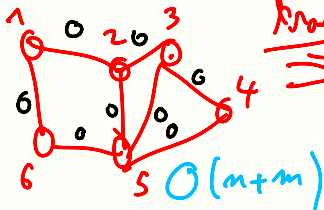
Algorithme :

1. Transformer le graphe donne en une instance de TSP.
2. Resoudre TSP sur cette instance.
3. En fonction de la reponse a TSP, determiner la reponse a `Cycle hamiltonien`.

Algorithme pour cycle hamiltonien (reduction a TSP)

Algorithme :

1. Transformer le graphe donne en une instance de TSP.
   - Comment ?



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

$O(n^2)$

2. Resoudre TSP sur cette instance.
3. En fonction de la reponse a TSP, determiner la reponse a `Cycle hamiltonien`.
   - Comment ?

si OPT $> 0 \Rightarrow$ NON

si OPT $= 0 \Rightarrow$ OUI

Algorithme pour cycle hamiltonien (reduction a TSP)

Algorithme :

1. Transformer le graphe donne en une instance de TSP.
   - Comment ?
2. Resoudre TSP sur cette instance.
3. En fonction de la reponse a TSP, determiner la reponse a `Cycle hamiltonien`.
   - Comment ?

Complexite :

- Transformation en TSP :
- Resolution TSP : $O^*(2^n)$ (avec l'algo donne ici)

Algorithme pour cycle hamiltonien (reduction a TSP)

Algorithme :

1. Transformer le graphe donne en une instance de TSP.
   - Comment ?
2. Resoudre TSP sur cette instance.
3. En fonction de la reponse a TSP, determiner la reponse a `Cycle hamiltonien`.
   - Comment ?

Complexite :

- Transformation en TSP : $O(n^2)$
- Resolution TSP : $O^*(2^n)$ (avec l'algo donne ici)

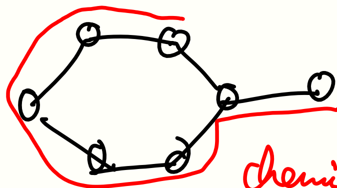
$\Rightarrow O^*(2^n)$

Problème du chemin hamiltonien

- **Entrée :** un graphe G (non-oriente, sans boucle, sans arete multiple).

Définition (Chemin hamiltonien)

Un *chemin hamiltonien* est un chemin simple qui contient tous les sommets du graphe.



pas cycle hamilt.

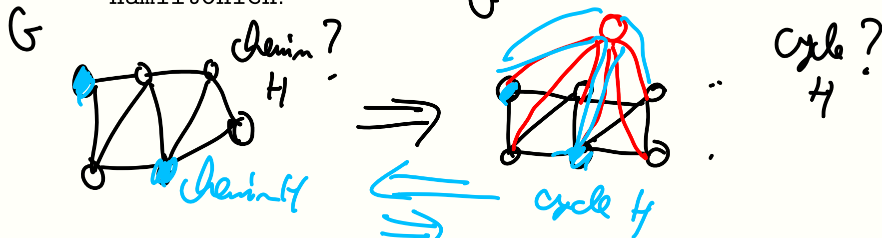
chemin hamiltonien

- **Sortie :** OUI si G contient un chemin hamiltonien, NON sinon.

Algorithme pour chemin hamiltonien (reduction a cycle)

Algorithme :

1. Transformer le graphe donne en une instance de `Cycle hamiltonien`.



2. Resoudre `Cycle hamiltonien` sur cette instance.
3. En fonction de la reponse a `Cycle hamiltonien`, determiner la reponse a `Chemin hamiltonien`.

Algorithme pour chemin hamiltonien (reduction a cycle)

Algorithme :

1. Transformer le graphe donne en une instance de `Cycle hamiltonien`.
   - Comment ?
2. Resoudre `Cycle hamiltonien` sur cette instance.
3. En fonction de la reponse a `Cycle hamiltonien`, determiner la reponse a `Chemin hamiltonien`.
   - Comment ?

Algorithme pour chemin hamiltonien (reduction a cycle)

Algorithme :

1. Transformer le graphe donne en une instance de `Cycle hamiltonien`.
   - Comment ?
2. Resoudre `Cycle hamiltonien` sur cette instance.
3. En fonction de la reponse a `Cycle hamiltonien`, determiner la reponse a `Chemin hamiltonien`.
   - Comment ? OUI → OUI
     NON → NON

Complexite :

- Transformation en `Cycle hamiltonien` : $O(n)$
- Resolution `Cycle hamiltonien` : $O^*(2^n)$ (avec l'algo ici)

Algorithme pour chemin hamiltonien (reduction a cycle)

Algorithme :

1. Transformer le graphe donne en une instance de `Cycle hamiltonien`.
   - Comment ?
2. Resoudre `Cycle hamiltonien` sur cette instance.
3. En fonction de la reponse a `Cycle hamiltonien`, determiner la reponse a `Chemin hamiltonien`.
   - Comment ?

Complexite :

- Transformation en `Cycle hamiltonien` : $O(n)$
- Resolution `Cycle hamiltonien` : $O^*(2^n)$ (avec l'algo ici)