TD AAIA
Programmation dynamique pour le voyageur de commerce

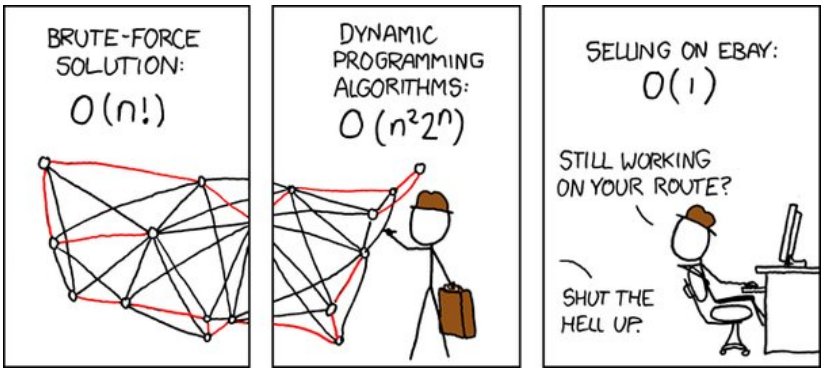

Image provenant de `https://xkcd.com/399/`

Pour ce TP, vous utiliserez le langage C. Pour compiler le programme `src.c` en un code exécutable `nomExec`, vous utiliserez la commande :

```
gcc -O3 src.c -o nomExec
```

(l'option `-O3` demande au compilateur d'optimiser le code).

Attention : Vous devez répondre (avant dimanche 24 mai, minuit) à un questionnaire disponible sur Moodle.

1 Définition du problème du voyageur de commerce

Un circuit hamiltonien est un circuit passant par chaque sommet d'un graphe exactement une fois. Le problème du voyageur de commerce consiste à chercher un circuit hamiltonien de longueur minimale, la longueur d'un circuit étant définie par la somme des longueurs de ses arcs.

Pour ce TP, nous supposons que le graphe est complet. En effet, s'il n'est pas complet, nous pouvons facilement le transformer en un graphe complet admettant la même solution : il suffit de fixer la longueur de chaque arc ajouté à une valeur très grande (par exemple, $n \cdot k$ où n est le nombre de sommets et k le coût maximal d'un arc).

Le problème du voyageur de commerce est $\mathcal{NP}$-difficile.

Questions (réponses à donner sur Moodle) :

Q1 : Les assertions suivantes sont-elles vraies ou fausses ? (voir les diapos 89 à 95 du cours)
- — Un problème appartient à la classe $\mathcal{NP}$ s'il n'est pas possible de le résoudre en temps polynomial.
- — Un problème appartient à la classe $\mathcal{NP}$ s'il existe un algorithme permettant de le résoudre en temps polynomial sur une machine de Turing non déterministe.
- — Les problèmes $\mathcal{NP}$-complets sont les problèmes les plus difficiles de la classe $\mathcal{NP}$.
- — Tout problème $\mathcal{NP}$-complet appartient à la classe $\mathcal{NP}$.
- — Tout problème $\mathcal{NP}$-difficile appartient à la classe $\mathcal{NP}$.

Q2 : Combien existe-t-il de circuits hamiltoniens différents dans un graphe complet comportant n sommets ?

2 Algorithme de Held et Karp

Nous notons S l'ensemble des sommets, et nous supposons que les sommets sont numérotés de 0 à $n-1$, de sorte que $S = [0, n-1]$. Nous supposons également que le tour commence et termine au sommet 0. Enfin, nous notons $cout[i][j]$ la longueur de l'arc (i, j).

Held et Karp ont proposé en 1962 un algorithme utilisant la programmation dynamique pour résoudre le problème du voyageur de commerce. Comme nous l'avons vu en cours (voir les diapos 55 à 57), l'idée de la programmation dynamique est de décomposer le problème en sous-problèmes, puis de définir la solution des sous-problèmes à l'aide d'équations récursives (appelées équations de Bellman).

Décomposition en sous-problèmes : Pour le voyageur de commerce, on associe un sous-problème à chaque couple (i, E) tel que i est un sommet de S et E est un sous-ensemble de sommets ne comportant ni 0 ni i (autrement dit, $E \subseteq S \setminus \{0, i\}$). Ce sous-problème est noté $D(i, E)$, et la solution de ce sous-problème est la longueur du plus court chemin allant de i jusque 0 en passant par chaque sommet de E exactement une fois.

Par exemple, $D(3, \{1, 4, 5\})$ est égal à la longueur du plus court chemin partant de 3, passant par les sommets 1, 4 et 5 (dans n'importe quel ordre), puis terminant sur 0.

Questions (réponses à donner sur Moodle) :

Q3 : Combien y-a-t-il de sous-problèmes différents possibles dans le cas d'un graphe comportant n sommets ?

Q4 : Quel est le sous-problème donnant la longueur du plus court circuit hamiltonien partant de 0 et revenant sur 0 ?

Equations de Bellman : La propriété d'optimalité des sous-chemins vue en cours nous permet de définir $D(i, E)$ récursivement :

— si $E = \emptyset$, alors $D(i, E) = cout[i][0]$;

— si $E \neq \emptyset$, alors $D(i, E) = \min_{j \in E}(cout[i][j] + D(j, E \setminus \{j\})$.

3 Implémentation naïve

Nous pouvons écrire une première version naïve du calcul de D en suivant très exactement la définition récursive des équations de Bellman.

```
1 Fonction calculeD(i, E)
      Entrée      : Un sommet i ∈ S et un sous-ensemble de sommets E ⊆ S \ {0, i}
      Postcondition : Retourne la longueur du plus court chemin allant de i jusque 0 en passant par chaque
                      sommet de E exactement une fois
2     si E = ∅ alors retourne cout[i][0];
3     min ← ∞
4     pour chaque sommet j ∈ E faire
5         d ← calculeD(j, E \ {j})
6         si cout[i][j] + d < min alors min ← cout[i][j] + d;
7     retourne min
```

La difficulté essentielle pour implémenter cet algorithme réside dans le choix d'une structure de données permettant de manipuler efficacement des ensembles. Nous vous proposons pour cela d'utiliser des vecteurs de bits : pour représenter un ensemble E dont les éléments sont compris entre 1 et n, il suffit d'utiliser un vecteur de n bits tel que le $j^{\text{ème}}$ bit est égal à 1 si et seulement si $j \in E$. En supposant que le plus grand élément n'aura jamais une valeur supérieure à 32, chaque vecteur de bit est un entier (les entiers sont codés sur 32 bits en C).

Vous trouverez sur Moodle et sur `http://perso.citi-lab.fr/csolnon/TSPnaif.c` les fonctions de base permettant de créer et manipuler des ensembles représentés par des vecteurs de 32 bits (des entiers). Vous y trouverez également une implémentation de l'algorithme *calculeD*. Compilez le programme `TSPnaif.c` avec l'option de compilation `-O3`, et exécutez-le en faisant varier le nombre de sommets.

Questions (réponses à donner sur Moodle) :

Q5 : Combien y-a-t-il d'appels récursifs à la fonction *calculeD* quand $n = 8$?

Q6 : Combien y-a-t-il d'appels récursifs à la fonction *calculeD* quand $n = 10$?

Q7 : Combien y-a-t-il d'appels récursifs à la fonction *calculeD* quand $n = 12$?

Q8 : Combien y-a-t-il d'appels récursifs par rapport au nombre de sous-problèmes différents ?

4 Implémentation avec mémoïsation

Comme nous l'avons vu en cours, un algorithme utilisant un principe de programmation dynamique doit être implémenté de façon à ne pas résoudre plusieurs fois le même sous-problème. Le programme `TSPnaif.c` appelle plusieurs fois la fonction *calculeD* avec les mêmes valeurs passées en paramètres, et il est donc particulièrement inefficace. Pour éviter ces calculs inutiles, nous avons vu en cours qu'il existe deux solutions.

La première solution consiste à utiliser un principe de mémoïsation. L'idée est d'utiliser un tableau $memD$ tel que pour tout sommet $i \in S$ et pour tout ensemble $E \subseteq S \setminus \{0, i\}$, $memD[i][E]$ contienne la valeur de $D(i, E)$. Avant de commencer la résolution, toutes les valeurs de ce tableau sont initialisées à 0 pour indiquer que la valeur n'a pas encore été calculée (si tous les coûts sont positifs, $D(i, E)$ est nécessairement supérieur à 0). A chaque appel à la fonction *calculeD*, si $memD[i][E]$ contient une valeur positive, alors la fonction retourne cette valeur, sinon la fonction calcule la valeur de $D(i, E)$, la mémorise dans $memD[i][E]$, puis la retourne. Ce principe est très facile à mettre en œuvre, et c'est celui que nous vous proposons de programmer en modifiant le code de la fonction *calculeD*.

Questions (réponses à donner sur Moodle) :

Q9 : Quelle est la longueur du plus court circuit hamiltonien quand $n = 20$?

5 Implémentation itérative

Cette dernière partie est facultative. Elle est cependant intéressante pour ceux désirant comparer une implémentation récursive et une implémentation itérative...

La seconde solution pour éviter les calculs inutiles consiste à remplir le tableau *memD* itérativement de la façon suivante :

```
1 pour chaque sous-ensemble E ⊂ S \ {0} faire
2     pour chaque sommet i ∈ S \ E faire
3         si E = ∅ alors memD[i][E] ← cout[i][0];
4         sinon
5             memD[i][E] ← ∞
6             pour chaque sommet j ∈ E faire
7                 si cout[i][j] + memD[j][E \ {j}] < memD[i][E] alors
8                     memD[i][E] ← cout[i][j] + memD[j][E \ {j}]
```

Pour implémenter cet algorithme, la difficulté essentielle consiste à choisir l'ordre dans lequel les sous-ensembles de S sont énumérés (ligne 1) : il s'agit de garantir, au moment de calculer $memD[i][E]$, d'avoir déjà calculé et mémorisé toutes les valeurs $memD[j][E \setminus \{j\}]$, pour tout $j \in E$.

Votre travail :

- Soient E et E' deux ensembles tels que $E' \subset E$, et soient i et i' les valeurs en base 10 des vecteurs de bits représentant E et E'. Que peut-on dire de i' par rapport à i ?
- En déduire une façon d'itérer sur les sous-ensembles de S garantissant qu'au moment où on considère un ensemble E, on a déjà vu tous les sous-ensembles de E.
- Implémentez l'algorithme itératif et exécutez le en faisant varier le nombre de sommets. Comparez les temps d'exécution de cette version avec ceux de la version récursive avec mémoïsation.
- Modifiez votre programme afin de pouvoir afficher l'ordre des sommets du plus court circuit hamiltonien.