TD AAIA : Programmation dynamique pour le voyageur de commerce

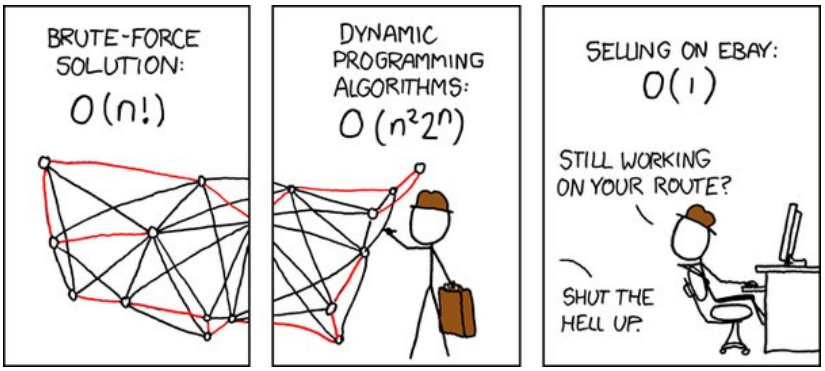

Image empruntée à http ://phoxis.org

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. Ce problème est $\mathcal{NP}$-difficile et dans ce TP nous allons utiliser un principe de programmation dynamique pour le résoudre.

Nous allons supposer 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 nk, où n est le nombre de sommets et k la longueur de l'arc le plus long du graphe). Nous noterons S l'ensemble des sommets, et nous supposerons que les sommets sont numérotés de 0 à $n-1$, de sorte que $S = [0, n-1]$. Nous supposerons également que le tour commence et termine au sommet 0. Enfin, nous noterons $cout[i][j]$ la longueur de l'arc (i, j).

1 Résolution par programmation dynamique : Version naïve

Held et Karp ont proposé en 1962 un algorithme utilisant un principe de programmation dynamique pour résoudre le problème du voyageur de commerce. Comme pour les algorithmes de calcul de plus courts chemins étudiés en cours, cet algorithme exploite la propriété d'optimalité des sous-chemins pour décomposer le problème en sous-problèmes. Plus précisément, l'algorithme se déduit des constatations suivantes :

- Un circuit hamiltonien est composé d'un arc $(0, i)$ suivi d'un chemin c allant de i jusque 0 en passant par chaque sommet de $S \setminus \{0, i\}$ exactement une fois. Si ce circuit est optimal alors c est le plus court chemin allant de i jusque 0 en passant par chaque sommet de $S \setminus \{0, i\}$.
- Étant donnés un sommet $i \in S$ et un ensemble de sommets $E \subseteq S \setminus \{0, i\}$, notons $D(i, E)$ la longueur du plus court chemin allant de i jusque 0 en passant par chaque sommet de E exactement une fois.
- La propriété d'optimalité des sous-chemins nous permet de définir $D(i, E)$ récursivement :
  - si $E = \emptyset$, alors $D(i, E) = cout[i][0]$;
  - si $E \neq \emptyset$, alors $D(i, E) = \min_{j \in E}(cout[i][j] + D(j, E \setminus \{j\}))$.
- La longueur du plus court circuit hamiltonien est $D(0, S \setminus \{0\})$.

Nous pouvons écrire une première version naïve du calcul de D en suivant très exactement sa définition récursive.

```
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://liris.cnrs.fr/christine.solnon/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*.

Votre travail :

- Compilez le programme (en utilisant l'option de compilation `-O3` pour optimiser le code exécutable), et exécutez-le en faisant varier le nombre de sommets. Quelle est la plus grande valeur pour laquelle l'exécution se termine en moins d'une minute ?
- Modifiez le code afin de compter le nombre d'appels récursifs à la fonction *calculeD*. Comment évolue ce nombre en fonction du nombre n de sommets ? Comparez ce nombre avec $n^2 2^n$ et $n!$.

2 Résolution par programmation dynamique : Version 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 recalculer plusieurs fois les mêmes choses. 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 $D(i, E)$. Initialement, ce tableau est vide. A chaque appel à la fonction *calculeD*, si $memD[i][E]$ contient une valeur, 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.

Votre travail :

- Modifiez le code de la fonction *calculeD* de façon à utiliser ce principe de mémoïsation.
- Exécutez le nouveau programme en faisant varier le nombre de sommets. Quelle est la plus grande valeur pour laquelle l'exécution se termine en moins d'une minute ? Comptez le nombre d'appels récursifs à la fonction *calculeD*. Comment évolue ce nombre en fonction du nombre n de sommets ? Comparez ce nombre avec $n^2 2^n$, $n!$, et le nombre d'appels récursifs dans la version naïve de l'exercice précédent.

3 Résolution par programmation dynamique : Version 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 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. Quelle est la plus grande valeur pour laquelle l'exécution se termine en moins d'une minute ? 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.