# TP AAIA numéro 3 : le voyageur de commerce (suite)

L'objectif de ce TP est de résoudre le problème du voyageur de commerce en utilisant un principe de résolution par séparation et évaluation (*Branch & Bound*). Rappelons que nous supposons que le graphe $G = (S, A)$ est complet, que les sommets sont numérotés de 0 à $n - 1$, et que le tour commence et termine au sommet 0. Enfin, nous notons $cout[i][j]$ la distance entre les sommets $i$ et $j$.

## 1 Enumérer tous les circuits hamiltoniens

Quand le graphe est complet, nous pouvons facilement énumérer tous les circuits hamiltoniens (partant de 0 et arrivant sur 0) à l'aide de la procédure récursive suivante :

1 **Procédure** *permut*($vus, nonVus$)
  **Entrée** : Une liste ordonnée de sommets $vus$ (déjà visités)
    Un ensemble de sommets $nonVus$ (restant à visiter)
  **Postcondition** : Affiche toutes les listes commençant par $vus$ et se terminant par une permutation de $nonVus$
2  **si** $nonVus$ *est vide* **alors** Afficher les éléments de $vus$, dans l'ordre de la liste;
3  **pour** *chaque sommet* $s_j \in nonVus$ **faire**
4  Ajouter $s_j$ à la fin de la liste $vus$ et enlever $s_j$ de l'ensemble $nonVus$
5  permut($vus, nonVus$)
6  Retirer $s_j$ de la fin de la liste $vus$ et remettre $s_j$ dans l'ensemble $nonVus$

Au premier appel, la liste $vus$ doit contenir le sommet de départ (0), tandis que l'ensemble $nonVus$ doit contenir tous les autres sommets de $S$.

**Votre travail :** Vous devez implémenter la procédure `permut` (implémentant l'algorithme *permut* décrit ci-dessus) :

```
void permut(int vus[], int nbVus, int nonVus[], int nbNonVus)
```

telle que :
- le tableau `vus[0..nbVus-1]` contient les sommets de la liste $vus$ ;
- le tableau `nonVus[0..nbNonVus-1]` contient les sommets de l'ensemble $nonVus$.

Cette procédure affiche toutes les séquences de sommets commençant par `vus[0..nbVus-1]`, et se terminant par n'importe quelle permutation de `nonVus[0..nbNonVus-1]`.

**Exemple d'exécution :** Si l'entier $nbSommets$ saisi en entrée est 4, alors le programme affichera les 6 permutations suivantes (l'ordre dans lequel les permutations sont affichées peut varier) :

```
0 1 3 2
0 1 2 3
0 2 1 3
0 2 3 1
0 3 1 2
0 3 2 1
```

## 2 Calcul des longueurs de tous les circuits hamiltoniens

Il s'agit maintenant de compléter la procédure `permut` de l'exercice 1 pour afficher la longueur de chaque circuit hamiltonien. La longueur de chaque circuit sera calculée incrémentalement. Pour cela, vous ajouterez à la procédure `permut` un paramètre contenant la longueur du chemin correspondant à `vus[0..nbVus]`.

**Votre travail :** Vous devez implémenter la procédure `permut` :

```
void permut(int* vus, int nbVus, int* nonVus, int nbNonVus, int longueur)
```

telle que :

— le tableau `vus[0..nbVus-1]` contient les sommets de la liste $vus$,
— le tableau `nonVus[0..nbNonVus-1]` contient les sommets de l'ensemble $nonVus$,
— la variable `longueur` contient la longueur du chemin correspondant à `vus[0..nbVus-1]`.
Cette procédure affiche la longueur de chaque séquence de sommets commençant par `vus[0.. nbVus-1]`, et se terminant par n'importe quelle permutation de `nonVus[0..nbNonVus-1]`.

**Exemple d'exécution :** Si l'entier $nbSommets$ saisi en entrée est 4, alors le programme affichera les longueurs des 6 permutations possibles (l'ordre d'affichage des longueurs peut varier) :

```
72
93
119
104
111
95
```

## 3 Calcul de la longueur du plus court circuit hamiltonien

Il s'agit de reprendre le code de la procédure `permut` de l'exercice 2, pour calculer la longueur du plus court circuit hamiltonien. Pour cela, vous ajouterez à `permut` un paramètre `borne` contenant la longueur du plus court circuit trouvé depuis le début de la recherche, et `permut` retournera en sortie la longueur du plus court circuit trouvé à la fin de son exécution.

**Votre travail :** Vous devez implémenter la fonction `permut` :

```
int permut(int* vus, int nbVus, int* nonVus, int nbNonVus, int longueur, int pcc)
```

telle que :
— le tableau `vus[0..nbVus-1]` contient les sommets de la liste $vus$,
— le tableau `nonVus[0..nbNonVus-1]` contient les sommets de l'ensemble $nonVus$,
— la variable `longueur` contient la longueur du chemin correspondant à `vus[0..nbVus-1]`,
— la variable `pcc` contient la longueur du plus court circuit trouvé depuis le début.
Soit `C` l'ensemble des circuits commençant par `Vus[0..nbVus-1]` et visitant ensuite chaque sommet de `nonVus[0..nbNonVu` puis retournant en 0. S'il existe un circuit de `C` de longueur inférieure à `pcc`, alors la fonction `permut` retourne la longueur du plus petit circuit de `C`, sinon elle retourne `pcc`.

**Exemple d'exécution :** Les temps CPU (en secondes) sont donnés à titre indicatif, pour un processeur 2,6 GHz Intel Core i5.

| Entrée | Sortie | Temps CPU |
|---|---|---|
| 4 | 72 | 0.00 |
| 6 | 91 | 0.00 |
| 8 | 123 | 0.00 |
| 10 | 134 | 0.01 |
| 12 | 161 | 0.94 |
| 14 | 182 | 148.94 |

## 4 Résolution par "séparation et évaluation" (*branch and bound*)

Nous ne pouvons pas espérer énumérer tous les circuits hamiltoniens en un temps raisonnable dès lors que le graphe comporte plus de 15 sommets. Rappelons qu'il existe $14!$ (soit près de 8 milliards) circuits différents commençant par 0 quand le nombre de sommets est égal à 15. Cette explosion combinatoire est inévitable dans la mesure où le problème est $\mathcal{NP}$-difficile. Cependant, nous pouvons reculer le moment de l'explosion en coupant les branches de l'arbre de recherche au plus tôt. L'idée est d'appeler une fonction d'évaluation (appelée *bound*) au début de chaque appel récursif. Cette fonction *bound* calcule une borne inférieure du coût du plus court chemin allant du dernier sommet de $vus$ (i.e., `vus[nbVus-1]`) jusqu'au premier sommet de $vus$ (i.e., `0`), et passant par chaque sommet de $nonVus$ exactement une fois. Si la longueur du chemin défini par `vus[0..nbVus-1]` ajoutée à cette borne est supérieure à la longueur du plus court circuit trouvé jusqu'ici (i.e., `pcc`), alors nous pouvons en déduire qu'il n'existe pas de

solution commençant par $vus$, et il n'est pas nécessaire d'appeler *enum* récursivement (nous disons dans ce cas que la branche est coupée).

La fonction d'évaluation la plus simple que nous puissions imaginer est : $(|nonVus| + 1) * dMin$, où $dMin$ est la plus petite longueur d'un arc du graphe. Cette fonction a l'avantage d'être calculable en temps constant à chaque appel récursif (en supposant que $dMin$ est évalué une fois pour toute au début de la recherche.

**Votre travail :** Vous devez implémenter la procédure `permut` :

```
int permut(int* vus, int nbVus, int* nonVus, int nbNonVus, int longueur,
           int pcc, int dMin)
```

telle que :

- le tableau `vus[0..nbVus-1]` contient les sommets de la liste $vus$,
- le tableau `nonVus[0..nbNonVus-1]` contient les sommets de l'ensemble $nonVus$,
- la variable `longueur` contient la longueur du chemin correspondant à `vus[0..nbVus-1]`,
- la variable `pcc` contient la longueur du plus court circuit trouvé depuis le début,
- la variable `dMin` contient la longueur du plus petit arc du graphe.

La postrelation de cette fonction est la même que pour l'exercice 3, mais vous ajouterez une étape d'évaluation pour réduire l'espace de recherche.

**Exemple d'exécution :** Les temps CPU (en secondes) sont donnés à titre indicatif, pour un processeur 2,6 GHz Intel Core i5.

| Entrée | Sortie | Temps CPU |
|---|---|---|
| 4 | 72 | 0.00 |
| 6 | 91 | 0.00 |
| 8 | 123 | 0.00 |
| 10 | 134 | 0.00 |
| 12 | 161 | 0.00 |
| 14 | 182 | 0.02 |
| 16 | 198 | 0.10 |
| 18 | 230 | 2.16 |
| 20 | 261 | 107.28 |

# 5 Implémentation d'une fonction d'évaluation plus sophistiquée

Une fonction d'évaluation plus évoluée (qui calcule une borne plus proche de la solution optimale, mais avec une complexité plus élevée) consiste à rechercher, pour chaque sommet de $nonVus$, la longueur de l'arête la plus courte permettant de le relier au circuit. Plus précisément :

- soit $l$ la longueur du plus petit arc partant du dernier sommet visité (i.e., `vus[nbVus-1]`) et arrivant sur un des sommets non visités (i.e., de `nonVus[0..nbNonVus-1]`);
- $\forall i \in [0, nbNonVus - 1]$, soit $l_i$ la longueur du plus petit arc partant de `nonVus[i]` et arrivant soit sur 0, soit sur un des sommets de `nonVus[0..nbNonVus-1]`;

une borne inférieure de la longueur du plus court chemin partant de `vus[nbVus-1]`, passant par chaque sommet de `nonVus[0..nbNonVus-1]`, et se terminant sur `0` est : $l + \sum_{i=0}^{nbNonVus-1} l_i$

**Votre travail :** Vous devez implémenter la procédure `permut` :

```
int permut(int* vus, int nbVus, int* nonVus, int nbNonVus,int longueur, int pcc)
```

telle que :

- le tableau `vus[0..nbVus-1]` contient les sommets de la liste $vus$,
- le tableau `nonVus[0..nbNonVus-1]` contient les sommets de l'ensemble $nonVus$,
- la variable `longueur` contient la longueur du chemin correspondant à `vus[0..nbVus-1]`,
- la variable `pcc` contient la longueur du plus court circuit trouvé depuis le début.

La postrelation de cette fonction est la même que pour l'exercice 3, mais vous ajouterez une étape d'évaluation pour réduire l'espace de recherche.

**Exemple d'exécution :** Les temps CPU (en secondes) sont donnés à titre indicatif, pour un processeur 2,6 GHz Intel Core i5.

| Entrée | Sortie | Temps CPU |
|---|---|---|
| 4 | 72 | 0.00 |
| 6 | 91 | 0.00 |
| 8 | 123 | 0.00 |
| 10 | 134 | 0.00 |
| 12 | 161 | 0.00 |
| 14 | 182 | 0.00 |
| 16 | 198 | 0.00 |
| 18 | 230 | 0.02 |
| 20 | 261 | 0.35 |
| 22 | 281 | 0.26 |
| 24 | 299 | 1.09 |
| 26 | 313 | 12.48 |
| 28 | 326 | 10.16 |
| 30 | 349 | 46.25 |
| 32 | 361 | 61.24 |

# 6 Ajout d'une heuristique d'ordre

Une autre façon d'améliorer les performances d'un algorithme procédant par séparation et évaluation, consiste à choisir l'ordre dans lequel les sommets sont énumérés. L'objectif est de trouver le plus rapidement possible le circuit le plus court afin de pouvoir couper plus rapidement les autres branches. La règle utilisée pour choisir l'ordre des sommets est appelée une *heuristique d'ordre*. Pour le voyageur de commerce, une heuristique d'ordre qui donne généralement d'assez bons résultats consiste à visiter en premier les sommets les plus proches du dernier sommet visité (`vus[nbVus-1]`). Ainsi, à chaque appel récursif, il s'agit de trier le tableau `nonVus[0..nbNonVus-1]` de telle sorte que

$$\forall i \in [1, nbNonVus - 1], cout[vus[nbVus - 1]][nonVus[i - 1]] \leq cout[vus[nbVus - 1]][nonVus[i]]$$

**Votre travail :** Vous devez implémenter la procédure `permut` :

```
int permut(int* vus, int nbVus, int* nonVus, int nbNonVus, int longueur, int pcc)
```

telle que :

- le tableau `vus[0..nbVus-1]` contient les sommets de la liste $vus$,
- le tableau `nonVus[0..nbNonVus-1]` contient les sommets de l'ensemble $nonVus$,
- la variable `longueur` contient la longueur du chemin correspondant à `vus[0..nbVus-1]`,
- la variable `pcc` contient la longueur du plus court circuit trouvé depuis le début.

La postrelation de cette fonction est la même que pour l'exercice 5, et vous utiliserez la même fonction d'évaluation que pour l'exercice 5, mais vous ajouterez l'heuristique d'ordre pour énumérer les sommets de `nonVus` en commençant par ceux qui sont le plus proches de `vus[nbVus-1]`.

**Exemple d'exécution :** Les temps CPU (en secondes) sont donnés à titre indicatif, pour un processeur 2,6 GHz Intel Core i5.

| Entrée | Sortie | Temps CPU |
|---|---|---|
| 4 | 72 | 0.00 |
| 6 | 91 | 0.00 |
| 8 | 123 | 0.00 |
| 10 | 134 | 0.00 |
| 12 | 161 | 0.00 |
| 14 | 182 | 0.00 |
| 16 | 198 | 0.00 |
| 18 | 230 | 0.00 |
| 20 | 261 | 0.06 |
| 22 | 281 | 0.05 |
| 24 | 299 | 0.28 |
| 26 | 313 | 3.33 |
| 28 | 326 | 4.61 |
| 30 | 349 | 9.79 |
| 32 | 361 | 13.21 |