TP AAIA : Branch & Bound pour le voyageur de commerce

Pour ce TP, vous utiliserez le langage C. Pour compiler le programme `src.c` en un code exécutable `nomExec`, vous utiliserez la commande suivante (l'option `-O3` demande au compilateur d'optimiser le code) :

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

Lors du TD précédent, vous avez utilisé un principe de programmation dynamique pour résoudre le problème du voyageur de commerce. L'objectif du TP d'aujourd'hui est de résoudre ce problème en utilisant un principe de résolution différent appelé *Branch & Bound*.

Rappelons que nous supposons que le graphe $G = (S, A)$ est complet, que les sommets sont numérotés de 0 à $n-1$, que le tour commence et termine au sommet 0, et que 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 : Récupérez sur Moodle le fichier `exo1.c` et complétez la procédure `permut` implémentant l'algorithme *permut* décrit ci-dessus. Cette procédure a la signature suivante :

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

telle que :
- le tableau `vus[0..nbVus-1]` contient les sommets déjà 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 : Récupérez sur Moodle le fichier `exo2.c` et complétez la procédure `permut` implémentant l'algorithme *permut* décrit ci-dessus. Cette procédure a la signature suivante :

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

telle que `longueur` contienne la longueur du chemin correspondant à `vus[0..nbVus-1]`. Cette procédure doit afficher 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) :

```
1792
3373
2112
2274
1991
1778
```

Votre travail (suite) : Ajoutez une variable globale `pcc`, et modifiez la procédure `permut` afin qu'elle maintienne dans `pcc` la longueur du plus court circuit hamiltonien : `pcc` est initialisée à $+\infty$ et, à chaque fois qu'une permutation p est calculée, `pcc` est mise à jour si la longueur de p est inférieure à `pcc`.

Exemple d'exécution : Les temps sont donnés à titre indicatif, pour un processeur 2,2 GHz Intel Core i7.

n	`pcc`	Temps CPU
8	1607	0.00s
10	1220	0.00s
12	1520	0.63s
14	1685	99.53s

Comparez ces temps d'exécution à ceux du programme utilisant un principe de programmation dynamique.

3 Résolution par "séparation et évaluation" (*branch & 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 d'évaluation calcule une borne inférieure de la longueur du plus court chemin allant du dernier sommet visité (i.e., `vus[nbVus-1]`) jusqu'au premier sommet visité (i.e., `0`), et passant par chaque sommet non visité (i.e., `nonVus[0..nbNonVus-1]`) 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 améliorante commençant par `vus[0..nbVus-1]`, et il n'est pas nécessaire d'appeler `permut` récursivement (nous disons dans ce cas que la branche est coupée).

La fonction d'évaluation la plus simple que nous puissions imaginer est : `(nbNonVus+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 : Implémentez une fonction `bound(i,nonVus,nbNonVus)` retournant une borne inférieure de la longueur du plus court chemin allant de `i` à `0` en passant par chaque sommet de `nonVus[0..nbNonVus-1]`. Modifiez la procédure `permut` de l'exercice précédent de telle sorte qu'elle appelle cette fonction `bound` avant chaque appel récursif pour éviter les appels récursifs inutiles.

Exemple d'exécution : Les temps sont donnés à titre indicatif, pour un processeur 2,2 GHz Intel Core i7.

n	`pcc`	Temps CPU
12	1520	0.00s
14	1685	0.04s
16	1631	0.12s
18	1741	1.91s
20	1632	5.99s
22	1721	82.99s

4 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 non visité, la longueur de l'arc le plus court 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]`);

— $\forall i \in [0, \texttt{nbNonVus} - 1]$, 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]` (autre que i);

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 : $l + \sum_{i=0}^{\texttt{nbNonVus}-1} l_i$

Votre travail : Modifiez la fonction `bound` de l'exercice précédent pour calculer la borne décrite ci-dessus.

Exemple d'exécution : Les temps sont donnés à titre indicatif, pour un processeur 2,2 GHz Intel Core i7.

n	`pcc`	Temps CPU
16	1631	0.01s
18	1741	0.11s
20	1632	0.19s
22	1721	1.57s
24	1811	9.52s
26	1328	8.94s
28	1602	311.98s

5 Ajout d'une heuristique d'ordre

Une autre façon d'améliorer les performances 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 de 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

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

Votre travail : Modifiez la procédure `permut` de l'exercice précédent en ajoutant l'heuristique d'ordre pour énumérer les sommets de `nonVus` (pour commencer par ceux qui sont les plus proches de `vus[nbVus-1]`).

Exemple d'exécution : Les temps sont donnés à titre indicatif, pour un processeur 2,2 GHz Intel Core i7.

n	`pcc`	Temps CPU
18	1741	0.02s
20	1632	0.08s
22	1721	1.57s
24	1811	3.57s
26	1328	0.3s
28	1602	33.97s
30	1573	351.05s

6 Limited Discrepancy Search (LDS)

Cette dernière partie du TP est facultative.

Quand le graphe est trop grand, il n'est plus possible de trouver le circuit optimal en un temps raisonnable. Dans ce cas, on peut utiliser une méta-heuristique permettant de calculer rapidement (avec une complexité en temps polynomiale) une bonne solution. Une méta-heuristique facile à implémenter est la recherche à divergence limitée (LDS), introduite par Harvey et Ginsberg dans [HG95].

A chaque appel récursif, LDS utilise une heuristique d'ordre h pour classer toutes les décisions possibles, de la meilleure à la plus mauvaise. Pour le voyageur de commerce, nous pouvons utiliser l'heuristique d'ordre introduite à l'exercice précédent pour trier les sommets non visités dans un tableau tab tel que $tab[0]$ soit le sommet non visité le plus proche du dernier sommet visité, et $tab[nbNonVus - 1]$ soit le sommet non visité le plus éloigné du dernier sommet visité.

A chaque appel récursif, on choisit un sommet s de *tab* qui est mis à la fin de vus, et la divergence de ce choix est égale à l'indice de s dans *tab*. La divergence d'un appel récursif est égale à la somme des divergences des choix pour tous les sommets de vus. L'idée de LDS est de limiter cette divergence a une valeur d_{max} donnée [1]. Ce principe est décrit dans l'algorithme suivant :

1 **Procédure** $LDS(vus, nonVus, d)$
  Entrée : Une liste ordonnée de sommets vus (déjà visités)
    Un ensemble de sommets $nonVus$ (restant à visiter)
    La divergence courante d (somme des divergences des choix des sommets de vus)
2 **si** $nonVus$ *est vide* **alors** mettre à jour *pcc* si la longueur de vus est inférieure à *pcc*;
3 Soit l la longueur du chemin correspondant aux sommets déjà visités
4 **si** $d \leq d_{max}$ *et* $l + bound(vus[nbVus - 1], nonVus, nbNonVus) < pcc$ **alors**
5 Trier les sommets de *nonVus* selon l'heuristique d'ordre h dans un table tab
6 **pour** i *variant de 0 à* $nbNonVus - 1$ **faire**
7 Ajouter $tab[i]$ à la fin de la liste vus et enlever $tab[i]$ de l'ensemble $nonVus$
8 LDS$(vus, nonVus, d + i)$
9 Retirer $tab[i]$ de la fin de la liste vus et remettre $tab[i]$ dans l'ensemble $nonVus$

Au premier appel, LDS est appelée avec $d = 0$.

Votre travail : Implémentez l'algorithme LDS, en repartant du code de l'exercice précédent. Exécutez le programme avec différentes valeurs de d_{max} et observez l'impact de la valeur de d_{max} sur le temps d'exécution et la valeur de *pcc*. Vous pouvez aussi trier les sommets de *nonVus* par rapport à la valeur retournée par la fonction `bound` définie à la question 4. Comparez la qualité des solutions trouvées et le temps mis pour les trouver avec la version précédente.

Références

[BP00] J. C. Beck and L. Perron. Discrepancy-bounded depth first search. In *Second International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, pages 8–10, 2000.

[HG95] W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 1995.

1. Nous considérons ici une variante de LDS (introduite par Beck et Perron dans [BP00]) qui permet de ne pas revisiter plusieurs fois un même état.