TP AAIA : Branch & Bound pour le voyageur de commerce

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

```
gcc -O3 -Wall branch.c -o branch -lm
```

Le programme a un paramètre n en entrée, correspondant au nombre de sommets du graphe, et la valeur de ce paramètre est passée en ligne de commande. Par exemple, pour exécuter `branch` avec $n = 4$, vous taperez

```
./branch 4
```

et pour exécuter `branch` avec $n \in \{4, 6, 8, 10\}$, vous taperez

```
for i in 4 6 8 10; do ./branch $i; done
```

1 Enumérer tous les circuits hamiltoniens

Lors des TD précédents, vous avez utilisé un principe de programmation dynamique puis un principe de recherche locale 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*. Pour cela, la première étape consiste à énumérer tous les circuits hamiltoniens (partant de 0, visitant tous les sommets sauf 0, et retournant sur 0). Quand le graphe est complet, cette énumération peut être faite à l'aide de la procédure récursive suivante :

1 **Procédure** *permut*($visited, notVisited$)
  Entrée : $visited$ = la liste ordonnée de sommets déjà visités (commençant par "0")
    $notVisited$ = l'ensemble des sommets restant à visiter
  Postcondition : Affiche tous les circuits commençant par $visited$ et se terminant par une permutation de $notVisited$ suivie de "0"
2 **si** $notVisited$ *est vide* **alors** Afficher les éléments de $visited$, dans l'ordre de la liste, suivi de "0";
3 **pour** *chaque sommet* $i \in notVisited$ **faire**
4   ajouter i à la fin de la liste $visited$ et enlever i de l'ensemble $notVisited$
5   permut($visited, notVisited$)
6   retirer i de la fin de la liste $visited$ et remettre i dans l'ensemble $notVisited$

Au premier appel, la liste $visited$ doit contenir le sommet de départ (0), tandis que l'ensemble $notVisited$ doit contenir tous les autres sommets du graphe.

Votre travail : Récupérez sur Moodle le fichier `branch.c` et complétez la procédure `permut` implémentant l'algorithme décrit ci-dessus. Comme pour le TD de la semaine dernière, le programme génère un script Python qui vous permettra de visualiser les circuits construits (sous réserve d'appeler la fonction `print(visited, nbVisited)` à chaque fois qu'un circuit complet a été construit dans `visited`).

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

```
$ ./branch 4
0 1 3 2 0
0 1 2 3 0
0 2 1 3 0
0 2 3 1 0
0 3 1 2 0
0 3 2 1 0
n=4 nbCalls=16 time=0.000s
```

2 Calcul des longueurs de tous les circuits hamiltoniens

Votre travail : Complétez la procédure `permut` pour afficher la longueur de chaque circuit hamiltonien. Cette longueur doit être calculée incrémentalement. Pour cela, ajoutez à la procédure `permut` un paramètre contenant la longueur du chemin correspondant à `visited[0..nbVisited-1]`.

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

```
$ ./branch 4
31319
32786
34415
31319
34415
32786
n=4 nbCalls=16 time=0.000s
```

3 Recherche du plus court circuit hamiltonien

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

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

n	best	nbCalls	time
8	53591	13 700	0.000s
10	66393	986 410	0.007s
12	67063	108 505 112	0.633s
14	69382	16 926 797 486	100.462s

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

4 Propagation de contraintes

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. Pour cela, nous allons exploiter une propriété que nous avons démontrée la semaine dernière dans le cas où le graphe est euclidien : étant donné un circuit hamiltonien $C = \langle v_0, v_1, \ldots, v_{n-1}, v_n \rangle$ avec $v_n = v_0$, s'il existe deux indices i et j (avec $0 \leq i < j \leq n$) tels que les arcs (v_i, v_{i+1}) et (v_j, v_{j+1}) se croisent, alors le circuit obtenu en échangeant ces arcs avec les arcs (v_i, v_j) et (v_{i+1}, v_{j+1}) est de longueur inférieure à celle de C.

Votre travail :

- Montrez qu'une conséquence de cette propriété est que la solution optimale ne peut pas contenir deux arêtes qui se croisent.
- Modifiez votre code afin de ne pas appeler récursivement `permut` lorsque le dernier arc ajouté (joignant `visited[nbVisited-1]` à un sommet de `notVisited`) croise un arc du chemin des sommets déjà visités (*i.e.*, `(visited[i],visited[i+1])` avec `i < nbVisited-1`). Pour tester si deux arcs (v_i, v_{i+1}) et (v_j, v_{j+1}) se croisent, il suffit de comparer la somme des coûts de ces deux arcs avec la somme des coûts des arcs (v_i, v_j) et (v_{i+1}, v_{j+1}) : si elle est supérieure, alors les arcs se croisent.

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

n	best	nbCalls	time
14	69382	97 343	0.009s
16	70310	596 119	0.053s
18	75456	3 767 726	0.356s
20	81292	19 821 721	2.151s
22	82447	107 963 329	13.105s
24	83193	638 366 435	84.749s

5 Résolution par *branch & bound* (séparation et évaluation)

Pour reculer le moment de l'explosion combinatoire, nous pouvons appeler une fonction d'évaluation (appelée *bound*) avant chaque appel récursif à `permut`. Cette fonction d'évaluation calcule une borne inférieure de la longueur du plus court chemin allant du dernier sommet visité jusqu'à `0` en passant par chaque sommet non visité exactement une fois. Si la longueur du chemin correspondant aux sommets déjà visités ajoutée à cette borne est supérieure ou égale à la longueur du plus court circuit trouvé jusqu'ici (i.e., `best`), alors nous pouvons en déduire qu'il n'existe pas de solution améliorante commençant par ce chemin, et il n'est pas nécessaire d'appeler `permut` récursivement.

Votre travail : Implémentez une première fonction *bound* retournant la somme, pour chaque sommet non visité, de 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 et arrivant sur un des sommets non visités ;
— pour chaque sommet i non visité, soit l_i la longueur du plus petit arc partant de i et arrivant soit sur 0, soit sur un des sommets non visités (autre que i).

La fonction *bound* retourne $l + \sum_i l_i$.

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

n	`best`	`nbCalls`	`time`
20	81292	429 737	0.321s
22	82447	2 003 996	1.688s
24	83193	10 759 285	9.301s
26	85449	20 492 536	19.334s
28	87005	65 891 592	73.845s

6 Implémentation d'une fonction *bound* 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 calculer le coût c de l'arbre couvrant minimal du sous-graphe induit par les sommets non visités, et à ajouter à c :

— le coût du plus petit arc reliant le dernier sommet visité et arrivant sur un des sommets non visités,
— et le coût du plus petit arc reliant un des sommets non visités à 0.

Votre travail : Impémentez cette fonction *bound*. Pour calculer l'arbre couvrant minimal, vous pouvez utiliser l'algorithme de Prim vu en cours (cf diapo 38). Vous ferez une recherche séquentielle pour sélectionner le sommet s_i minimisant c (ligne 8) [1].

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

n	`best`	`nbCalls`	`time`
22	82447	325 750	0.196s
24	83193	2 215 815	1.404s
26	85449	7 950 442	5.039s
28	87005	20 148 442	14.724s
30	89288	111 920 536	84.797s

7 Ajout d'une heuristique d'ordre

Une autre façon d'améliorer les performances consiste à choisir l'ordre dans lequel les sommets sont énumérés (à la ligne 3 de l'algorithme décrit à l'exercice 1). 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*. Une heuristique d'ordre qui donne généralement de bons résultats consiste à visiter en premier les sommets les plus proches du dernier sommet visité.

1. Sachant que le graphe est complet, nous avons $p = n * (n-1)/2$ et, dans ce cas, la complexité en temps de Prim n'est pas améliorée par l'utilisation d'une file de priorité.

Votre travail : Modifiez la fonction `permut` en triant les sommets restant à visiter par ordre croissant de coût de l'arc partant du dernier sommet visité, et en choisissant ces sommets dans cet ordre. Attention : ce tri doit être fait dans un autre tableau que `notVisited` afin de ne pas changer l'ordre défini lors des appels précédents. Vous pourrez utiliser la fonction `qsort_r` (déjà utilisée pour le TP1).

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

n	best	nbCalls	time
28	87005	5 268	0.083s
30	89288	18 228	0.198s
32	95293	58 312	0.925s
34	96116	59 572	1.109s
36	97027	95 642	1.973s
38	102757	494 645	11.172s
40	106431	1 394 961	32.887s

8 Pour aller plus loin...

Ce que vous avez vu dans ce TP est à la base d'algorithmes plus sophistiqués qui sont capables de résoudre des instances avec beaucoup plus de sommets à visiter. Pour améliorer les performances, on peut notamment utiliser la borne décrite par Held et Karp dans [HK71] (consistant à calculer plusieurs arbres couvrants minimaux en faisant varier les coûts sur les arêtes) : en utilisant cette borne, la solution optimale pour $n = 50$ sommets est trouvée en 3s, par exemple. L'approche la plus efficace pour les graphes symétriques est Concorde [2], qui combine une approche similaire à ILS (vu la semaine dernière) avec une approche similaire à celle que nous avons implémentée cette semaine mais utilisant la programmation linéaire pour calculer efficacement des bornes : cette approche est capable de résoudre des instances ayant des milliers de sommets.

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, comme vous l'avez vu la semaine dernière avec ILS. Une méta-heuristique facile à implémenter dès lors que vous avez implémenté une approche par *Branch & Bound* est *Limited Discrepancy Search (LDS)* de Harvey et Ginsberg [HG95]. À 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 le plus proche du dernier sommet visité, et $tab[nbNotVisited - 1]$ soit le sommet le plus éloigné. À chaque appel récursif, on choisit un sommet s de *tab* qui est mis à la fin de *visited*, 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 *visited*. L'idée de LDS est de limiter cette divergence a une valeur d_{max} donnée [3]. Ce principe est décrit dans l'algorithme suivant :

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

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

2. `https://www.math.uwaterloo.ca/tsp/concorde.html`
3. 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.

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.

[HK71] Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees : Part II. *Math. Program.*, 1(1) :6–25, 1971.