TP4 AAIA : Programmation dynamique et recherche locale 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 : `gcc -O3 src.c -o nomExec`. Et bien sûr, il est conseillé d'écrire un Makefile (en vous inspirant de celui du TP précédent, par exemple).

Lors du TP3, vous avez résolu le problème du voyageur de commerce (consistant à rechercher le plus court circuit hamiltonien d'un graphe) en utilisant un principe de séparation et évaluation (*Branch & Bound*). Cette approche est exacte et trouve la solution optimale. En revanche, sa complexité en temps est exponentielle. Nous allons implémenter aujourd'hui deux nouvelles approches pour résoudre ce problème :

— une approche exacte, basée sur un principe de programmation dynamique, ayant une complexité en temps inférieure à celle de l'algorithme par séparation et évaluation (même si cette complexité reste exponentielle) ;
— une approche basée sur de la recherche locale, permettant de calculer rapidement des solutions approchées même pour des graphes comportant des milliers de sommets.

1 Résolution du TSP par programmation dynamique

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, 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 :

— Combien y-a-t-il de sous-problèmes différents possibles dans le cas d'un graphe comportant n sommets ?
— 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 $D(i, E) = \min_{j \in E}(cout[i][j] + D(j, E \setminus \{j\})$.

L'algorithme *calculeD* décrit à la page suivante se fonde sur ces équations pour calculer D. 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 le fichier `TSPnaif.c` contenant 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*.

Question : Observez l'évolution du nombre d'appels récursifs quand n prend pour valeurs 8, 10, puis 12. Ce nombre d'appels récursifs est-il du même ordre de grandeur que votre évaluation théorique du nombre de sous-problèmes ?

```
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
```

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.

Votre travail : Modifiez le code de la fonction *calculeD* afin qu'il fasse de la mémoïsation. Comparez les performances de cette version avec celles de l'implémentation ne faisant pas de mémoïsation.

Implémentation itérative. Cette 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 $j \in E$. Afin de vous aider à déterminer cet ordre, répondez aux questions suivantes :

— 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.

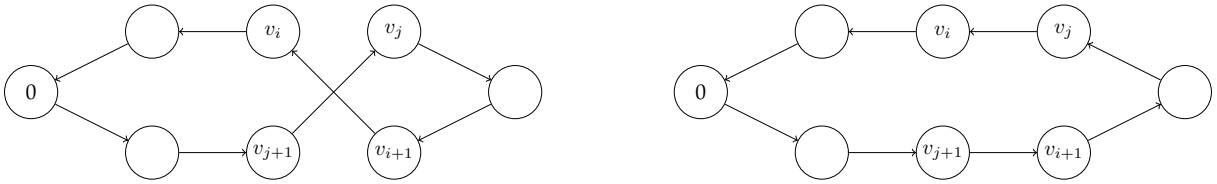
2 Résolution du TSP par recherche locale

Le fichier `tsp.c` (sur Moodle) contient un code C permettant de construire des circuits hamiltoniens aléatoirement. Afin de faciliter la mise au point de vos programmes, ce code génère également un script Python pour visualiser les circuits construits. Après chaque exécution de ce code, le script généré est dans le fichier `script.py`, et vous pourrez visualiser les différents circuits construits avec la commande `Python3 script.py`.

Compilez `tsp.c`, exécutez le, et visualisez les circuits construits.

2.1 Amélioration d'un circuit par recherche locale gloutonne

Nous avons vu la semaine dernière que lorsque deux arcs (v_i, v_{i+1}) et (v_j, v_{j+1}) se croisent, alors on peut obtenir un circuit de longueur inférieure en les échangeant avec les arcs (v_i, v_j) et (v_{i+1}, v_{j+1}), comme cela est illustré ci-dessous.



Nous avons également vu que la longueur du nouveau circuit peut être obtenue à partir de la longueur du circuit initial en temps constant.

Nous en déduisons l'algorithme de recherche locale suivant [1] :

Fonction *GreedyLS*
  Entrée : un circuit hamiltonien C et une fonction coût
  Sortie : un circuit hamiltonien dont la longueur ne peut pas être diminuée en échangeant 2 de ses arcs
  tant que *C contient 2 arcs dont l'échange permet de réduire le coût* **faire**
    Chercher les 2 arcs dont l'échange permet de réduire au maximum le coût
    $C \leftarrow$ circuit obtenu en échangeant ces deux arcs
  retourner C

Cet algorithme est dit glouton car à chaque itération la longueur du circuit diminue. L'algorithme s'arrête lorsque C ne contient plus d'arcs dont l'échange permet de réduire le coût.

Question : La solution retournée par *GreedyLS* est-elle optimale ?

Votre travail : Implémentez la fonction `greedyLS` du fichier `tsp.c`. Vérifiez que les circuits retournés par GreedyLS ne contiennent plus d'arcs se croisant à l'aide du script Python.
Exemple d'exécution (sur un processeur 2,2 GHz Intel Core i7) :

```
Number of vertices: 200
Number of random tour constructions: 5
Trial 0: Initial tour length = 101725; Tour length after GreedyLS = 11921; Time = 0.016s
Trial 1: Initial tour length = 103886; Tour length after GreedyLS = 11916; Time = 0.012s
Trial 2: Initial tour length = 100491; Tour length after GreedyLS = 11592; Time = 0.013s
Trial 3: Initial tour length = 106315; Tour length after GreedyLS = 11941; Time = 0.014s
Trial 4: Initial tour length = 101433; Tour length after GreedyLS = 11850; Time = 0.013s
Best solution found: 11592
```

2.2 Iterated Local Search (ILS)

On constate que quand on exécute *GreedyLS* au départ de différents circuits, on obtient des solutions de longueurs différentes. On peut répéter *GreedyLS* au départ d'un très grand nombre de circuits et espérer obtenir comme cela une bonne solution, mais cela n'est pas très efficace. *Iterated Local Search* (ILS) est une approche introduite en 2003 par Lourenço et Stützle qui consiste à répéter *GreedyLS*, mais au lieu de repartir à chaque fois d'un circuit généré

1. Cet algorithme a été introduit en 1958 par G. A. Croes, et la fonction de voisinage consistant à échanger deux arêtes est appelée 2-opt.

complètement aléatoirement, on repart d'un circuit obtenu en perturbant le meilleur circuit connu comme décrit dans l'algorithme suivant.

```
1  Fonction ILS
2      Entrée : un graphe pondéré, deux entiers positifs k et l
3      Sortie : un circuit hamiltonien
4      C* ← circuit hamiltonien généré aléatoirement
5      C* ← GreedyLS(C*)
6      pour i variant de 1 à k faire
7          C ← C*
8          pour j variant de 1 à l faire
9              Echanger deux sommets de C choisis aléatoirement
10         C ← GreedyLS(C)
11         si longueur de C < longueur de C* alors C* ← C;
12     retourner C*
```

ILS a deux paramètres : k qui permet de contrôler le nombre d'appels à *GreedyLS* et donc le temps d'exécution, et l qui permet de contrôler l'intensité de la perturbation, une perturbation consistant à échanger k couples de sommets.

Implémentez ILS, et étudiez l'impact de k et l sur le temps d'exécution et la qualité des solutions trouvées.
Exemple d'exécution (sur un processeur 2,2 GHz Intel Core i7) :

```
Number of iterations of ILS (k): 1000
Perturbation strength (l): 20
Number of vertices: 200
Initial tour length = 101725; Tour length after GreedyLS = 11921; Time = 0.016s
New best found at iteration 0; Tour length = 11724; Time = 0.019s
New best found at iteration 1; Tour length = 11406; Time = 0.023s
New best found at iteration 3; Tour length = 11380; Time = 0.029s
New best found at iteration 6; Tour length = 11357; Time = 0.039s
New best found at iteration 12; Tour length = 11346; Time = 0.060s
New best found at iteration 24; Tour length = 11320; Time = 0.099s
New best found at iteration 34; Tour length = 11270; Time = 0.129s
New best found at iteration 38; Tour length = 11228; Time = 0.142s
New best found at iteration 118; Tour length = 11213; Time = 0.391s
New best found at iteration 131; Tour length = 11196; Time = 0.430s
New best found at iteration 134; Tour length = 11189; Time = 0.439s
New best found at iteration 181; Tour length = 11156; Time = 0.577s
New best found at iteration 214; Tour length = 11155; Time = 0.680s
New best found at iteration 215; Tour length = 11075; Time = 0.683s
New best found at iteration 266; Tour length = 11070; Time = 0.843s
New best found at iteration 274; Tour length = 11063; Time = 0.866s
New best found at iteration 288; Tour length = 11032; Time = 0.907s
New best found at iteration 308; Tour length = 11012; Time = 0.963s
New best found at iteration 389; Tour length = 10994; Time = 1.216s
New best found at iteration 696; Tour length = 10981; Time = 2.114s
New best found at iteration 818; Tour length = 10946; Time = 2.478s
```