TD AAIA
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`

La semaine dernière, vous avez résolu le problème du voyageur de commerce (consistant à rechercher le plus court circuit visitant tous les sommets d'un graphe) en utilisant la programmation dynamique. Cette approche est exacte (elle trouve la solution optimale), mais sa complexité en temps et en espace est exponentielle. Elle ne peut donc pas être utilisée quand les graphes sont trop gros. Nous allons implémenter aujourd'hui une approche basée sur de la recherche locale, permettant de calculer très rapidement des solutions approchées.

1 Construction aléatoire et visualisation de circuits

Le fichier `tsp.c` (sur Moodle) contient un code C permettant de construire des circuits 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 Amélioration d'un circuit par recherche locale gloutonne

Un graphe euclidien est un graphe dont les coûts des arêtes correspondent à des distances euclidiennes : chaque sommet i a des coordonnées (x_i, y_i), et le coût d'une arête (i, j) est $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.

Considérons un circuit hamiltonien $C = \langle v_0, v_1, \ldots, v_{n-1}, v_n \rangle$ avec $v_n = v_0$. Montrez que si le graphe est euclidien et qu'il existe deux indices i et j (avec $0 \leq i < j \leq n$) tels que les arêtes (v_i, v_{i+1}) et (v_j, v_{j+1}) se croisent, alors le circuit obtenu en échangeant ces arêtes avec les arêtes (v_i, v_j) et (v_{i+1}, v_{j+1}) est de longueur inférieure à C. Montrez que la longueur de ce nouveau circuit peut être calculée en temps constant.

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

Fonction *GreedyLS*
  Entrée : un circuit hamiltonien $C = \langle v_0, v_1, \ldots, v_{n-1}, v_n \rangle$ avec $v_n = v_0$ et une fonction coût
  Sortie : un circuit hamiltonien n'ayant pas d'arc se croisant et de longueur totale inférieure ou égale à C
  tant que C *contient des arêtes se croisant* **faire**
    Chercher les indices i et j tels que :
    – $0 \leq i < j \leq n$;
    – (v_i, v_{i+1}) et (v_j, v_{j+1}) se croisent ;
    – coût(v_i, v_{i+1}) + coût(v_j, v_{j+1}) − coût(v_i, v_j) − coût(v_{i+1}, v_{j+1}) est maximal.
    $C \leftarrow$ circuit obtenu en remplaçant les arêtes (v_i, v_{i+1}) et (v_j, v_{j+1}) par (v_i, v_j) et (v_{i+1}, v_{j+1})
  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'arêtes se croisant. Dans ce cas, a-t-on trouvé la solution optimale ? Si oui, alors prouvez-le ; si non, alors donnez un contre-exemple.

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
```

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.

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

Fonction *ILS*
  Entrée : un graphe pondéré, deux entiers positifs k et l
  Sortie : un circuit hamiltonien
  $C^* \leftarrow$ circuit hamiltonien généré aléatoirement
  $C^* \leftarrow$ GreedyLS(C^*)
  pour i *variant de 1 à* k **faire**
    $C \leftarrow C^*$
    pour j *variant de 1 à* l **faire**
      Echanger deux sommets de C choisis aléatoirement
    $C \leftarrow$ GreedyLS(C)
    si *longueur de* $C <$ *longueur de* C^* **alors** $C^* \leftarrow C$;
  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
```