Implémentation d'algorithmes de recherche opérationnelle

Dominique Orban

Laboratoire n^{o}1 (MTH6412B)

Jean-Noël Weller (1843975) & Juliette Tibayrenc (1800292)

27 septembre 2016

Table des matières

1 Introduction

Le problème du voyageur de commerce étant un problème « difficile », il est nécessaire d'organiser correctement notre code. Cela commence par représenter les données utilisées de manière simple et robuste. Dans notre cas, il s'agit de représenter des graphes, des arêtes pondérées, des nœuds, et de savoir créer le graphe depuis des fichiers textes, contenant les données.

2 La classe Edge

2.1 Attributs

Nous avons décidé de donner en attribut à une arête un tuple ___nodes contenant les identifiants des deux nœuds reliés par ladite arête. Nous y avons adjoint un champ ___weight pour le poids. Une exception est renvoyée, si les deux nœuds sont identiques, car nous partons du principe qu'un nœud ne peut pas pointer sur lui-même (à moins bien sûr que le poids associé à l'arête ne soit nul). Une arête a aussi un identifiant associé (un nombre entier, assigné grâce à un compteur associé à la classe).

2.2 Méthodes

Les méthodes implémentées pour la classe Edge sont des méthodes d'accès aux différents attributs (noeuds, poids, identifiant), une méthode d'accès au compteur ___edge_count associé à la classe et une surcharge du print.

2.3 Tests unitaires

Nous avons écrit et effectué quelques tests unitaires visant à vérifier le compteur de nœuds ainsi que la levée d'exception lorsqu'une arête pointe sur elle-même.

3 Modifications de la classe Graph

3.1 Attributs

Plutôt que de représenter un graphe comme une liste de nœuds et une liste d'arêtes, nous avons choisi, comme conseillé, de représenter un graphe comme un dictionnaire de dictionnaire d'arêtes, ayant comme clés respectives les différents nœuds du graphe : {nœud1 : {nœud2 : arête12, nœud3 : arête13, ...}, nœud4...}. L'avantage d'une telle méthode est la facilité de l'accès aux différents nœuds et arêtes du graphe, ainsi que de la construction ou modification de celui-ci, après ajout progressif des nœuds puis des arêtes.

3.2 Méthodes

Nous avons d'abord défini des méthodes d'ajout de nœuds au graphe, puis d'ajout d'arête (l'arête n'est ajoutée que si les deux nœuds sont bel et bien présents dans le graphe).

3.3 Tests unitaires

Nous avons effectué quelques tests unitaires visant à vérifier l'impossibilité d'ajouter une arête à un graphe sans avoir ajouté les nœuds, et le bon fonctionnement des accesseurs aux nœuds et aux arêtes sur un graphe vide. Il n'est pas très pertinent de vérifier le nombre de nœuds et d'arêtes du graphe, du fait du conflit entre les compteurs de nœuds et d'arêtes des classes Node et Edge et du graphe attribut de la classe de test : dans chaque méthode de test, le graphe est initialement vide tandis que les nœuds créés dans les méthodes précédentes continuent à être pris en compte...Ceci n'est pas un souci dans le cadre du problème du voyageur de commerce (un graphe, tous les nœuds existants inclus dedans), mais une autre utilisation de cette implémentation d'un graphe appellerait sans doute à deux compteurs associés à la classe Graph, plutôt qu'à un compteur pour la classe Node et un pour la classe Edge.

Nous avons ensuite défini les accesseurs à différents paramètres du graphe, en tirant profit de la structure en dictionnaire de sa matrice d'adjacence : nom, nœuds, nombre de nœuds, arêtes et nombre d'arêtes. On remarquera que seul l'accès aux arêtes est plus « laborieux » avec notre représentation qu'avec directement deux listes d'adjacence. Nous avons tiré parti de la variable de classe Node.___node_count pour l'implémentation de get_nb_nodes(). Nous avons supposé :

1. qu'il n'y a qu'un graphe,
2. que tous les nœuds créés appartiennent à ce graphe,

ce qui est vrai en pratique dans tous les exemples d'application. Nous avons fait de même pour la méthode get_nb_edges().

Enfin, nous avons modifié la surcharge du print implémentée. Sont affichées successivement les listes des nœuds et des arêtes.

4 Modifications de la fonction read_edges()

La méthode initiale read_edges() de read_stsp() ne lit pas les poids des arêtes du graphe analysé. En effet, elle se contente de mettre dans un tuple de taille deux les arêtes existantes sous la forme (nœud1, nœud2), selon les caractéristiques du fichier considéré. Il nous suffit de rajouter le poids lu dans le flux de charactères à l'indice courant j, `data[j]`.

5 Programme principal : main

Le programme principal reprend la structure de celui ébauché dans la classe read_stsp. Nous lisons un fichier sous le format stsp avec notre nouvelle fonction et créons le graphe correspondant avec les fonctions expliquées précédemment, avant de l'afficher. Le programme principal peut être exécuté comme suit :

```
python ./main.py [fichier]
```

Par exemple :

```
python ./main.py instances/stsp/bays29.tsp
```

6 Conclusion

Dans ce premier lab, nous avons réalisé l'implémentation des classes principales dont nous aurons besoin pour représenter les données du problème du voyageur de commerce. Nous pouvons à présent lire les fichiers de données au format tsp fournis dans le dossier stsp et construire le graphe correspondant à chacun de ces fichiers. Nous espérons que la représentation choisie et son implémentation nous faciliteront la résolution (approchée) du problème dans la suite du projet.