Implémentation d'algorithmes de recherche opérationnelle

Laboratoire n^{o}2 (MTH6412B) – Dominique Orban

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

17 octobre 2016

1 Introduction

Lors du premier labo, nous avons mis en place la structure de base nécessaire au bon démarrage du projet. Dans ce nouveau labo, après être revenu sur quelques détails du premier labo, nous avons implémenté l'algorithme de Kruskal, afin, à partir d'un graphe, de trouver un arbre de recouvrement minimal.

2 Modifications sur le labo 1

2.1 Identifiants des nœuds

Précédemment, nous repérions les nœuds par leurs identifiants et pas par leur adresse mémoire. Ceci n'était pas particulièrement intéressant, car cela ne permettait pas facilement, à partir de l'identifiant, d'utiliser l'objet concerné. Nous avons donc remédié à ce problème et ne manipulons dorénavant que des objets nodes (et edges) directement.

2.2 Compteurs de nœuds et d'arêtes

Nous avons en outre cessé d'utiliser les attributs globaux de classe, comptant le nombre de noeuds et d'arêtes créées. Au lieu de cela, nous utilisons des compteurs de noeud et d'arêtes dans la classe Graph, pour chaque graphe créé. Notons que ces deux compteurs ont changé d'implémentation, mais que celui du nombre d'arêtes, pour éviter un parcours coûteux à chaque appel, se base sur les identifiants des arêtes et n'est donc pas robuste hors du contexte de ce problème particulier et de la façon dont sont ajoutées les arêtes lors de la lecture d'un fichier tsp. Nous sommes conscients de ce manque de robustesse et implémenterons une solution différente par la suite.

2.3 Représentation des graphes dans la sortie par défaut

Nous avons effectué de petites modifications dans la représentation des graphes, afin d'éviter des lourdeurs inutiles et des répétitions dans leur affichage texte.

2.4 Tests unitaires

Nous avons complété les tests unitaires pour la classe Graph, en ajoutant des tests pour les compteurs de nœuds et d'arêtes.

3 Classe DisjointSet

3.1 Attributs

DisjointSet possède deux attributs : node et parent. Node contient le nœud courant du DisjointSet considéré, et parent contient le nœud dont node est le fils. Si $parent = None$, le nœud est isolé.

3.2 Méthodes

DisjointSet contient les méthodes suivantes :

- Des propriétés pour accéder et modifier ses attributs. Parent est le seul à être modifiable, et ce seulement s'il est vide. Les cas contraires soulèvent des exceptions.
- find_root : permet de remonter les nœuds des DisjointSet jusqu'à trouver la racine (le DisjointSet qui ne possède pas de parent).
- union_set : met les deux ensembles disjoints dans la même composant connexe s'ils ne sont pas déjà connexes, c'est-à-dire procède à l'union des ensembles si leurs racines sont distinctes. Il renvoie True si l'union a été possible et False sinon.
- ___repr___ : affiche le nœud et son parent.

3.3 Tests unitaires

Les tests vérifient qu'on ne peut modifier ni le nœud ni le parent une fois définis. Ils vérifient que find_root renvoie le nœud courant s'il n'a pas de parent, et qu'il renvoie la bonne racine si le nœud courant possède un parent. D'autres tests vérifient que union_set renvoie False si les deux DisjointSet ne sont pas disjoints, et True s'ils le sont.

4 Modifications de la classe Graph

4.1 Algorithme de Kruskal

Nous avons implémenté l'algorithme de Kruskal comme une méthode de Graph. Elle renvoie un objet de classe Graph contenant l'arbre de recouvrement minimal trouvé. Précisons que la méthode implémentée ne fonctionnera que sur les graphes qui admettent effectivement un arbre de recouvrement minimal, et ne renverra pas d'erreur *a priori* dans le cas contraire.

Nous gérons les DisjointSet dans un dictionnaire indexé par les nœuds et avec les DisjoitSet comme valeur. L'intérêt est de pouvoir parcourir facilement les DisjointSet à partir des nœuds.

Pour trier les arêtes, nous avons complété la classe Edge en y implémentant les méthodes ___gt___ et ___ge___ et simplement utilisé la méthode sort pour les listes Python.

Nous effectuons ensuite un parcours sur cette liste triée d'arêtes, puis effectuons l'union des DisjointSet associés aux nœuds composant l'arête : dans le cas où l'union renvoie True, nous complétons l'arbre minimal. L'algorithme s'arrête ou bien quand il n'y a plus d'arêtes à parcourir, ou bien quand il y a autant de nœuds dans l'abre minimal que dans le graphe initial.

4.2 Représentation des graphes avec Matplotlib

Nous avons également adapté la méthode plot_graph() du module read_stsp pour pouvoir visualiser une représentation basique (poids des arêtes non affichés, pour l'instant) de nos graphes. La représentation n'interrompt pas l'exécution du programme qui a appelé la méthode si ce programme est exécuté depuis la CLI par défaut de Python ou ipython. Si le programme est appelé depuis le terminal, son exécution ne sera pas interrompue non plus mais la fenêtre sera fermée automatiquement à la fin de l'exécution. Ce comportement permettant d'éviter la multiplication du nombre de fenêtres ouvertes lors de l'exécution du script (non inclus) que nous utilisons pour tester `main` sur tous les exemples fournis, il n'est pas considéré comme un problème.

4.3 Calcul du poids d'un graphe

Pour disposer de cette information intéressante, nous avons ajouté une méthode permettant de calculer le poids total des arêtes d'un graphe. Cette méthode dispose de son test unitaire associé.

5 Programme principal : main

Pour le programme principal, nous avons simplement repris le programme utilisé pour le lab précédent et remplacé l'affichage en console du graphe lu par sa représentation avec Matplotlib, suivie par l'exécution de l'algorithme de Kruskal et l'affichage de l'arbre minimum résultant. Comme précédemment, on peut l'exécuter depuis le terminal comme suit :

```
python ./main stsp/bayg29.tsp
```

(tout autre fichier tsp symétrique peut bien sûr être substitué à bayg29.tsp).

6 Conclusion

Pour ce deuxième laboratoire, nous avons retravaillé les structures de données manipulées afin de mieux les adapter au travail à réaliser et de réduire la complexité algorithmique de certaines méthodes. Nous avons également suivi les conseils donnés et créé la classe DisjointSet, qui nous a énormément facilité le travail lors l'implémentation de l'algorithme de Kruskal lui-même. Le travail réalisé n'est cependant pas encore optimal et nous comptons pour un prochain rendu intégrer notamment l'utilisation de Logging, ce qui devrait beaucoup nous aider pour la suite.