Implémentation d'algorithmes de recherche opérationnelle

Laboratoire n^{o}3 (MTH6412B) – Dominique Orban

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

14 novembre 2016

1 Introduction

Dans cette étape, nous avons amélioré et mesuré (de manière informelle) les performances de l'algorithme de Kruskal, et implémenté l'algorithme de Prim. D'un point de vue plus général, nous avons commencé à utiliser du « Logging » dans notre code pour simplifier la recherche d'erreurs.

2 Partie théorique

2.1 Première majoration du rang d'un sommet

On souhaite montrer que : $\forall s \in S\,, \mathrm{rg}(s) \leq |S| - 1$. On procède par récurrence multiple sur le nombre de sommets de l'arbre considéré.

Inititalisation Si $|S| = 1$. Alors le rang de l'unique sommet de S est par définition 0, et $0 \leq 1 - 1$.

Hérédité Soit $n \in \mathbb{N}^*$. Supposons que pour tout arbre S tel que $|S| \leq n$, $\forall s \in S\,, \mathrm{rg}(s) \leq |S| - 1$. Soit T_{n+1} arbre de taille $n + 1$.

On appelle T_1 et T_2 les deux arbres (non vides) desquels on a formé T_{n+1}. On a $|T_1|\,, |T_2| \leq n$ donc par hypothèse de récurrence : $\forall t \in T_1 \cup T_2\,, \mathrm{rg}_{T_{1ou2}}(t) \leq n - 1$. Soient t_1 et t_2 les racines respectives de T_1 et T_2.

- ou bien : $\mathrm{rg}_{T_1}(t_1) \neq \mathrm{rg}_{T_2}(t_2)$. Alors l'union se fait sans modification du rang et le rang de chaque sommet dans T_{n+1} est le même que dans T_1 et T_2, donc l'hérédité est vérifiée.
- ou bien : $\mathrm{rg}_{T_1}(t_1) = \mathrm{rg}_{T_2}(t_2)$. Alors l'union se fait avec augmentation de 1 du rang de la nouvelle racine, au rang alors inférieur ou égal à $n - 1 + 1 = n$, et les autres rangs sont inchangés. Tous ces nouveaux rangs sont inférieurs ou égax à n, et l'hérédité est vérifiée.

2.2 Seconde majoration du rang d'un sommet

On souhaite montrer que : $\forall s \in S\,, \mathrm{rg}(s) \leq \lfloor \log_2(|S|) \rfloor$. On procède par récurrence multiple sur le nombre de sommets de l'arbre considéré.

Initialisation Si $|S| = 1$. Alors le rang de l'unique sommet de S est par définition 0, et $0 \leq \lfloor \log_2(1) \rfloor$.

Hérédité Soit $n \in \mathbb{N}^*$. Supposons que pour tout arbre S tel que $|S| \leq n$, $\forall s \in S\,, \mathrm{rg}(s) \leq \lfloor \log_2(|S|) \rfloor$. Soit T_{n+1} arbre de taille $n + 1$.

On appelle T_1 et T_2 les deux arbres (non vides) desquels on a formé T_{n+1}. On a $|T_1|\,, |T_2| \leq n$ donc par hypothèse de récurrence : $\forall t \in T_i\,, \mathrm{rg}_{T_i}(t) \leq \lfloor \log_2(|T_i|) \rfloor$, pour $i \in \{1\,, 2\}$. Soient t_1 et t_2 les racines respectives de T_1 et T_2.

- ou bien : $\mathrm{rg}_{T_1}(t_1) \neq \mathrm{rg}_{T_2}(t_2)$. Alors l'union se fait sans modification du rang et le rang de chaque sommet dans T_{n+1} est le même que dans T_1 et T_2, donc l'hérédité est vérifiée (car $\lfloor \log_2(n+1) \rfloor = \lfloor \log_2(|T_1| + |T_2|) \rfloor \geq \max(\lfloor \log_2(|T_1|) \rfloor\,, \lfloor \log_2(|T_2|) \rfloor)$).

- ou bien : $\mathrm{rg}_{T_1}(t_1) = \mathrm{rg}_{T_2}(t_2)$. On choisit comme nouvelle racine celle dont l'arbre associé possède le plus petit nombre de noeuds. On peut supposer, quitte à échanger les indices, que c'est t_1 qui a été choisi comme nouvelle racine. Pour tous les nœuds descendants, l'hérédité est vérifiée par le même argument que dans le cas précédent. Il suffit de vérifier la propriété pour la nouvelle racine.

$$\begin{aligned}\mathrm{rg}_{T_{n+1}}(t_1) &= \mathrm{rg}_{T_1}(t_1) + 1 \\ &\leq \lfloor \log_2(|T_1|) \rfloor + 1, \text{ par hypothèse de récurrence} \\ &= \lfloor \log_2(2|T_1|) \rfloor, \text{ car } \log_2(2) = 1 \\ &\leq \lfloor \log_2(|T_{n+1}|) \rfloor, \text{ car } |T_{n+1}| = |T_1| + |T_2| \text{ et } |T_1| \leq |T_2|\end{aligned}$$

3 Amélioration de l'algorithme de Kruskal

Nous avons commencé par implémenter l'union par le rang et la compression d'arbre.

Pour cela, nous avons modifié la classe `DisjointSet` en lui ajoutant l'attribut `rank`. Nous aurions pu placer cet attribut dans la classe `Node`, mais avons préféré garder le plus possible ce qui est utilisé dans l'algorithme de Kruskal dans la classe `DisjointSet`. Ensuite, nous avons implémenté en une seule méthode l'union par le rang avec compression des chemins (`rank_compressed_union`).

Nous avons enfin écrit une nouvelle version de l'algorithme de Kruskal, `kruskal_pp`, dans la classe `Graph`, qui utilise l'union par le rang et la compression des chemins. On remarque, sans chercher à prendre des mesures formelles, que cette nouvelle version apporte sur nos machines une amélioration du temps d'exécution de l'algorithme par rapport à l'ancienne version (entre quelques millisecondes et quelques dizaines de millisecondes, selon la machine et le nombre de sommets du graphe d'origine). Ceci découle bien sûr de la majoration démontrée précédemment sur le rang d'un sommet, puisque la remontée sur plusieurs nœuds vers la racine n'est plus nécessaire.

N.B. : Pour harmoniser, nous utilisons tout de même `find_root` malgré la compression des chemins, ce qui ne change pas la complexité puisqu'il ne s'agit que d'un appel récursif au pire.

4 Implémentation de l'algorithme de Prim

Notre implémentation de l'algorithme de Prim a besoin d'un élément qu'ont nos instances de `DisjointSet` et que n'ont pas nos instances de `Node` : un parent. Ceci mis à part, les méthodes spécifiques à `DisjointSet` (union, compression…) ne sont pas utilisées dans l'algorithme de Prim. Nous avons donc décidé d'intégrer directement l'attribut supplémentaire dont nous avons besoin, `key`, dans la classe `Node`. Nous l'initialisons dès la création de l'objet à `sys.maxsize` (préférable à `sys.maxint` pour ne pas en prendre l'habitude et avoir des problèmes plus tard en Python 3). On implémente aussi `__lt__` et `__le__` pour pouvoir faire des comparaisons entre nœuds basées sur les clefs.

Par ailleurs, on a besoin pour l'algorithme d'une file de priorité. On utilise pour cela le module `heapq`, qui est fait pour cela. La https ://docs.python.org/2/library/heapq.html nous montre que nous n'avons besoin que de deux fonctions, `heappush` et `heappop`.

L'implémentation de l'algorithme est basée sur le livre de Cormen. L'arbre est construit au fur et à mesure et retourné à la fin. On constate que, bien que l'arbre retourné comporte le bon nombre d'arêtes, les arêtes sélectionnées ne sont pas toujours les mêmes, et le poids n'est donc pas toujours le même. Il peut s'agir d'un problème dans notre implémentation que nous n'avons pas trouvé, ou alors des variations naturelles conséquentes à la présence de plusieurs arêtes de même poids dans les graphes fournis en exemple.

5 Programme principal : main

Le programme rendu exécute l'algorithme de Kruskal amélioré suivi de l'algorithme de Prim. Les arbres retournés sont affichés dans le terminal. En décommentant les lignes appropriées, on peut, si on le souhaite, exécuter l'ancien algorithme de Kruskal ou afficher une représentation graphique des arbres retournés.

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 Autres changements

Nous avons (enfin) commencé à logger avec le module `logging`. Quelques exemples des messages utilisés ont été laissés dans les fichiers de code. Nous n'avons pas inclus le fichier de logs dans le travail remis.

Les anciens accesseurs des attributs de classe ont été remplacés par une formulation plus idiomatique qui utilise des décorateurs.

7 Conclusion

Dans ce lab, nous avons pu à la fois améliorer l'algorithme de Kruskal implémenté au dernier laboratoire, et implémenter un nouvel algorithme, celui de Prim. Nous avons aussi nettoyé un peu notre code en supprimant les différences préexistantes entre les attributs et méthodes associées, et nous avons amélioré notre efficacité grâce au logging.