LINMA1691 : Théorie des graphes
Devoir 2 : Arbres sous-tendants

Comme vu au cours, le problème d'arbre sous-tendant a de nombreuses applications. Dans ce devoir, nous en explorons et implémentons deux en **Python**. Plus précisément, il vous est demandé d'implémenter efficacement les fonctions *prim_ mst* et *min_ cut* du fichier *template.py*. Vous ne pouvez pas importer d'autres librairies que celles déjà présentes.

1 Promesses électorales

Contexte

En tant que nouveau bourgmestre de votre commune, vous désirez satisfaire votre promesse de renouveler les routes du village. Chaque route renouvelée apporte un "score de satisfaction" à la population en fonction de sa position et fréquentation mais est bloquée pendant la durée des travaux ! Pour éviter de mécontenter vos électeurs, vous voulez donc vous assurer qu'il est toujours possible à partir de chaque carrefour d'atteindre n'importe quel autre carrefour. Quel est le plus gros score de satisfaction (défini comme la somme des scores de satisfaction des routes renouvelées) que vous pouvez obtenir ?

Input de la fonction

L'input vous est donné sous la forme d'arguments d'une fonction à compléter : *prim_ mst(N, roads)*

N est le nombre de carrefours, indexés de 0 à N-1. *roads* est une liste de dimension M décrivant le réseau. Chaque élément du tableau est un tuple (u, v, s) représentant une route à double sens entre les carrefours u et v de satisfaction s. Il est possible que vous ayez à transformer cet argument en une structure plus adaptée à ce qui est demandé comme une liste d'adjacence par exemple (*i.e.* une liste de taille N contenant à l'indice n une liste des voisins de n). Il est garanti que $4 \leq N, M \leq 10^3$.

Output de la fonction

Il vous est demandé de compléter la fonction et de retourner le plus gros score de satisfaction possible en utilisant l'algorithme de **Prim** (décrit dans les notes de cours).

La réelle difficulté d'implémentation de cet algorithme se situe dans les structures de données à utiliser afin d'atteindre une complexité de calcul raisonnable[1]. En effet, l'algorithme de Prim demande d'entrelacer des ajouts d'arêtes pondérées à un ensemble et des extractions d'arêtes de poids minimal de cet ensemble. Une implémentation "naïve" donnerait de très mauvais résultats. Nous vous demandons d'utiliser ici un tas binaire.

Un tas binaire est une façon d'implémenter les files de priorités. Ce sont des files où les éléments sont affublés d'un nombre, et dont les opérations de base sont "ajouter un élément" et "extraire l'élément minimum". Les

[1] voir `https://en.wikipedia.org/wiki/Prim's_algorithm#Time_complexity`

tas binaires permettent d'effectuer efficacement ces opérations, avec une complexité $O(\log n)$ pour chacune d'entre-elles avec n le nombre d'éléments présents dans la file. Il existe de nombreuses références pour vous informer sur leur fonctionnement interne [2].

Avec cette structure de données, vous devriez obtenir une complexité asymptotique en $\mathcal{O}(|E| \log |V|)$. Vous ne devez pas implémenter cette structure de données, vous pouvez utiliser l'implémentation Python *heapq* déjà importée dans le template fourni.

Exemple d'input:

$(4, [(0, 1, 3), (0, 2, 2), (1, 2, 1), (1, 3, 4), (2, 3, 5)])$

Réponse : 8

2 Algorithme de Karger

Contexte

Il est parfois intéressant d'utiliser des algorithmes probabilistes pour résoudre certains problèmes. Le tri rapide[3] est par exemple un algorithme de tri probabiliste très efficace. Nous allons implémenter dans cet exercice un algorithme probabiliste pour estimer la coupe minimum d'un graphe. Une coupe d'un graphe est un ensemble d'arêtes déconnectant le graphe si elles sont toutes enlevées. La coupe minimum d'un graphe est donc la coupe ayant le plus petit nombre d'arêtes.

L'algorithme de Karger[4] est un algorithme probabiliste estimant la coupe minimum d'un graphe. Il consiste à prendre à chaque étape une arête uniformément aléatoirement et la contracter[5]. On répète la procédure jusqu'à ce qu'il ne reste plus que 2 noeuds dans le graphe et l'estimation de la coupe minimum est le nombre d'arêtes entre ces noeuds. Cet algorithme peut sembler très simple, mais il s'avère qu'il fonctionne bien en pratique. On peut montrer que la probabilité de succès de l'algorithme de Karger est plus grande ou égale à $\binom{N}{2}^{-1}$. Une minute de réflexion permet de voir que le sous-graphe formé de l'ensemble des arêtes contractées lors de ce processus forme une forêt à deux arbres. Le nombre d'arêtes du graphe séparant les arbres est l'estimation de la coupe minimum recherchée. Il s'agit donc de choisir des arêtes uniformément au hasard tout en évitant les cycles jusqu'à ce qu'elles forment une forêt à deux arbres, et ensuite de compter la coupe obtenue entre les deux arbres. Nous vous demandons d'implémenter cet algorithme.

Afin de vous aider dans cette tâche, une classe Union-Find (ou ensembles disjoints[6]) est disponible dans le template. Celle-ci vous permet d'atteindre une complexité en $\mathcal{O}(|E| \log |V|)$ dans votre implémentation de Karger.

NB pour les plus curieux : cet algorithme est équivalent à l'application de l'algorithme de Kruskal au graphe dont on a pondéré les arêtes de manière aléatoire, à un détail près : Il suffit en effet d'arrêter l'algorithme de Kruskal quand il ne reste plus que deux arbres à relier (soit à l'avant-dernière étape) et de compter les arêtes entre ceux-ci.

Input de la fonction

L'input vous est donné sous la forme d'arguments d'une fonction à compléter : *min_ cut(N, edges).*

N est le nombre de noeuds du graphe, indexés entre 0 et *N-1*. *edges* est une liste de dimension *M* décrivant

[2] `https://www.geeksforgeeks.org/binary-heap/` et `https://visualgo.net/en/heap` pour sa visualisation
[3] `https://en.wikipedia.org/wiki/Quicksort`
[4] `https://en.wikipedia.org/wiki/Karger%27s_algorithm`
[5] Contracter une arête revient à la retirer et fusionner les deux noeuds qu'elle relie
[6] Voir notes de cours p.26-27

les arêtes du graphe. Chaque élément de la liste est un tuple (u, v) correspondant à une arête entre u et v.

Exemple d'input :

$(8, [(0,1),(0,2),(0,3),(1,2),(1,3),(2,3),(4,5),(4,6),(4,7),(5,6),(5,7),(6,7),(0,4),(1,5)])$

Coupe minimale : 2

Output de la fonction

Il vous est demandé d'implémenter ainsi un algorithme donnant la coupe minimum avec une probabilité plus grande que $0.9999 =: 1 - \alpha$.

Hint : Vous allez devoir lancer un nombre suffisant de fois (notons le k) votre algorithme qui a une probabilité de succès $p \geq \binom{N}{2}^{-1}$. On veut que au cours de ces k "lancers", au moins un donne la bonne réponse avec une probabilité supérieure ou égale à $1-\alpha$: $\mathbb{P}(\#success \geq 1) \geq 1-\alpha$. si l'on prend l'événement complémentaire, on veut : $1 - \mathbb{P}(\#success = 0) \geq 1 - \alpha$. Nos k "lancers" étant indépendants, on peut calculer facilement $\mathbb{P}(\#success = 0) = (1-p)^k$ et de là en déduire k.

Consignes

Vous devez soumettre chaque corps de fonction sur l'activité Inginious[7] du même nom. **Attention** à garder l'indentation de votre fichier pour votre soumission.

Le langage de programmation est **Python 3** (version 3.5).

Deadline : vendredi 11 novembre 22h.

[7] `https://inginious.info.ucl.ac.be`