ENSAE 1A – Projet de programmation
Optimisation d'un réseau de livraison

Responsable du cours: Patrick Loiseau

patrick.loiseau@inria.fr

Version du 16 mars 2023

1 Instructions et recommandations générales

1.1 Objectif

Le projet de programmation a pour objectif global de vous aider à acquérir une expérience de programmation **par la pratique**, via la résolution d'un problème particulier (décrit dans la section suivante). Plus spécifiquement, il est attendu au cours du projet que vous acquériez un certain nombre de compétences (liste non nécessairement exhaustive) :

- Apprendre à structurer du code pour un projet de moyenne envergure
- Apprendre à chercher en ligne les informations, les documentations, et les librairies utiles
- Apprendre à analyser un problème et à le traduire en terme algorithmique
- Apprendre à analyser la complexité d'un algorithme
- Apprendre à manipuler quelques algorithmes et structures de données classiques
- Apprendre à debugger, tester, et optimiser du code
- Se familiariser avec les bonnes pratiques de code et avec un environnement d'édition.

Le projet est fait en Python (obligatoirement). On utilisera la notion de classe, avec une structure minimale imposée (voir plus loin). Pour l'ensemble du projet, on utilisera le service vscode-python de la plateforme SSP Cloud (hébergée par l'Insee) qui offre un environnement Visual Studio code avec la version 3.10 de python. Pour y accéder, allez sur https://datalab.sspcloud.fr/catalog/ide et sélectionnez `Vscode-python`. Il n'est pas autorisé d'utiliser des notebook jupyter, il est obligatoire de faire les TP et le projet dans des fichiers python d'extension `.py`.

Note : Le service sur sspcloud permet d'accéder à un environnement uniforme avec des logiciels à jour de n'importe où, y compris à la maison. Attention toutefois à bien enregistrer votre travail car un service inutilisé peut être "tué" à tout moment.

1.2 Évaluation et rendus

Le projet doit être fait en binôme ; les deux membres d'un même binôme doivent appartenir au même groupe de TP. Vous devez vous inscrire avec votre binôme, **avant le début de la séance 2**, sur ce formulaire :

https://forms.office.com/e/uRELb0abdQ.

Le projet contient 6 séances de TP. Les trois premières séances de TP seront relativement guidées, les deux suivantes plus ouvertes, la dernière servira à l'évaluation finale. On demande 2 rendus :

Un rendu intermédiaire, dû le mercredi 15/03 (23h59) : Après la 3ème séance de TP, on demande le rendu de l'ensemble du code écrit jusque-là, testé et documenté. Le code doit être déposé dans un dépôt de code github, seul le lien doit être envoyé au chargé de TP (voir section 1.4).

Un rendu final, dû le jeudi 06/04 (23h59) : Une semaine avant la dernière séance de TP, on demande le rendu final de l'ensemble du code du projet, documenté, accompagné d'un rapport d'environ 4-6 pages au format PDF. Le rapport doit décrire succinctement les choix algorithmiques et les résultats et justifier la complexité des algorithmes utilisés. Comme pour le rendu intermédiaire, le code lui-même doit être déposé dans un dépôt github dont le lien doit être donné dans le rapport PDF.

Lors de la dernière séance de TP, le 13/04, chaque chargé de TP passera voir tous les binômes de son groupe pour une séance de questions destinée à évaluer la compréhension des deux membres du binôme. Il ne sera pas demandé de faire une présentation du projet (pas de slide), mais il faudra pouvoir faire une démo si demandé.

La présence aux TPs est obligatoire et sera vérifiée au début de chaque séance. La notation inclura 5 points (sur 20) pour la présence et le rendu intermédiaire. Nous serons particulièrement attentifs au fait que le code soit testé et bien documenté. Le reste (15 points) sera attribué sur la base du rendu final et de la séance de questions lors du dernier TP. Nous tiendrons compte d'un certain nombre de critères : qualité du code, clarté de la documentation, pertinence des choix algorithmiques, analyse des performances algorithmiques, respect des bonnes pratiques, etc.

Note générale : Nous donnons une base de code (classe Graph, etc.) et un certain nombre d'instructions sur la structure du code et les spécifications. Il est crucial de bien respecter ces instructions et de ne pas modifier les noms des fonctions données en les implémentant.

1.3 Ressources utiles

Le projet suppose une connaissance du language Python acquise au premier semestre. Toutefois, en cas de besoin, il existe de nombreux cours et livres sur la programmation en Python, tels que *Introduction to programming in Python* de R. Sedgewick, K Wayne, et R. Dondero. Pour ce qui est des bonnes pratiques, il existe de nombreuses ressources. Citons en particulier :

— Les documents de la communauté Python, e.g., le PEP8 (qui contient une section sur le nommage) et le PEP257 (sur l'écriture de documentation)
— Le Hitchhiker's Guide to Python
— Les supports de Lino Galiana (enseignant du cours Python pour la data science de deuxième année) sur les bonnes pratiques et la qualité du code
— Les pages de Xavier Dupré (ancien responsable du projet et enseignant du cours du premier semestre) donnant des conseils divers

Il pourra être utile de s'y référer au long du projet. Enfin, les élèves pourront être amenés à vouloir consulter des livres d'algorithmique ; il en existe là aussi de nombreux (e.g., *Introduction to Algorithms* de Cormen et al. ou (plus court !) *Algorithms* de Dasgupta, Papadimitriou, et Vazirani).

1.4 Contact des chargés de TP

— Groupe 1 : Lucas Baudin, `lucas.baudin@ensae.fr`
— Groupe 2 : Patrick Loiseau, `patrick.loiseau@inria.fr`
— Groupe 3 : Ziyad Benomar, `ziyad.benomar@ensae.fr`
— Groupe 4 : Quentin Jacquet, `quentin.jacquet@edf.fr`
— Groupe 5 : Salim Chouaki, `salim.chouaki@inria.fr`

2 Description du problème : optimisation d'un réseau de livraison

On considère un réseau routier constitué de villes et de routes entre les villes. L'objectif du projet va être de construire un réseau de livraison pouvant couvrir un ensemble de trajets entre deux villes avec des camions. La difficulté est que chaque route a une limite inférieure de puissance, c'est-à-dire à dire qu'un camion ne peut emprunter cette route que si sa puissance est supérieure ou égale à la limite inférieure de puissance de la route. Il faudra donc déterminer pour chaque couple de villes si un camion d'une puissance donnée peut trouver un chemin possible entre ces deux villes ; puis chercher à optimiser la flotte de camions qu'on achète en fonction des trajets à couvrir.

On représente le réseau routier par un graphe non orienté $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. L'ensemble des sommets $\mathcal{V}$ correspond à l'ensemble de villes. L'ensemble des arêtes $\mathcal{E}$ correspond à l'ensemble des routes existantes entre deux villes. Chaque arête $e \in \mathcal{E}$ est associée à une valeur (ou poids) $\bar{p}_e \geq 0$ qu'on appelle puissance minimale de l'arête e, qui correspond à la puissance minimale d'un camion pour qu'il puisse passer sur la route e. Chaque arête $e \in \mathcal{E}$ est également associée à une deuxième valeur $d_e > 0$ correspondant à la distance entre les deux villes sommets sur l'arête e.

On considère un ensemble $\mathcal{T}$ de trajets, où chaque trajet t est un couple de deux villes distinctes, i.e., $t = (v, v')$ où $v \in \mathcal{V}, v' \in \mathcal{V}, v \neq v'$. L'ensemble $\mathcal{T}$ représente l'ensemble des couples de villes (v, v') pour lesquelles on souhaite avoir un camion qui puisse faire le transport entre v et v'. Notez que le graphe n'est pas orienté et on ne distingue pas la direction du trajet. À chaque trajet t est également associé un profit (ou utilité) $u_t \geq 0$, qui sera acquis par la compagnie de livraison si le trajet t est couvert.

Enfin, le transport est fait par des camions. Chaque camion (noté K) a une puissance p et coûte un prix c. Le transport sur un trajet $t = (v, v') \in \mathcal{T}$ sera possible si et seulement si on peut trouver dans le graphe $\mathcal{G}$ un chemin allant de v à v' sur lequel la puissance minimale de chaque arête est inférieure où égale à p (i.e., le camion a une puissance suffisante pour passer partout sur le chemin). On dit alors que le trajet t peut être couvert par le camion K considéré.

Le projet contient deux parties :

Partie 1 : calcul de la puissance minimale pour un trajet. Dans cette partie, qui couvre environ les trois premières séances, on s'intéresse au problème de calculer la puissance minimale nécessaire d'un camion pour un trajet $t \in \mathcal{T}$ (ainsi que le chemin associé).

Partie 2 : optimisation de l'acquisition de camions. Dans cette deuxième partie, on cherche à calculer quels camions acheter pour optimiser le profit des trajets couverts.

Notation On notera $V = |\mathcal{V}|$ le nombre de sommets du graphe $\mathcal{G}$ (aussi souvent appelé n) ; $E = |\mathcal{E}|$ le nombre d'arêtes du graphe $\mathcal{G}$ (aussi souvent appelé m) ; et $T = |\mathcal{T}|$ le nombre de trajets que l'entreprise souhaite couvrir.

Toutes les ressources associées au projet pour les étudiants (base de code, fichiers d'input, tests préliminaires, etc.) se trouve sur le dépôt github (publique) suivant :

https://github.com/patrickloiseau/ensae-prog23

qui contient également une explication des fichiers fournis.

Séance 1 : 10/2

Dans cette première séance, on va prendre en main la structure générale du code et implémenter quelques premiers algorithmes pour calculer si un camion peut couvrir un trajet (et la puissance minimale d'un camion pour un trajet donné).

Vous devez programmer vous-mêmes les algorithmes de graphe dans la classe `Graph` fournie dans le dépôt git du projet. En particulier, il n'est pas autorisé d'utiliser des paquets comme `networkx` pour la partie algorithmique. Vous pouvez l'utiliser, exclusivement pour la visualisation.

Une base de code, des fichiers de tests pour une partie des questions et des fichiers de données sont disponibles. Pour exécuter les tests, vous pouvez ou bien le faire dans l'onglet « Test » de VS Code, ou bien les exécuter en ligne de commande, par exemple pour la première question :

```
python tests/test_s1q1_graph_loading.py
```

Note : Il est utile de regarder les tests en codant, pour être sûr qu'une fonction retourne le résultat au format attendu par le test. Nous ne fournissons que des tests très minimalistes dont le but est de vous aiguiller vers une bonne structure de tests pour démarrer. Il est essentiel que vous implémentiez d'autres tests par vous-même pour compléter.

Dans cette première séance, on s'intéressera uniquement aux fichiers d'entrés `network.x.in` où `x` est dans $\{00, 01, 02, 03, 04\}$, qui sont des petits fichiers destinés à pouvoir tester que les fonctions implémentés retournent les résultats corrects.

Les fichiers python à compléter dans un premier temps sont `main.py` et `graph.py`. Vous pouvez tester une partie de vos fonctions en appelant les fonctions dans `main.py`.

Question 1. Compléter la définition de la classe Graph en implémentant en particulier la méthode `add_edge` et écrivez une fonction `graph_from_file` qui permet de charger un fichier. Pour commencer, on s'intéresse seulement aux fichiers `network.x.in` où `x` est dans $\{00, 01, 02, 03\}$. Le fichier a le format suivant (comme dans le dossier `inputs`) :

- la première ligne est composée de deux entiers séparés par un espace : le nombre de sommets et le nombre d'arêtes ;
- les lignes suivantes représentent chacune une arête et sont composées de 3 entiers : les 2 numéros des sommets (à partir de 1) suivi de la puissance de l'arête.

Par exemple, le graphe :

$$1 — 2 — 3$$

sera représenté par le fichier (avec des puissances de 10 pour chaque arête) :

```
3 2
1 2 10
2 3 10
```

Mettre par défaut la distance sur chaque arête égale à 1 lorsque celle-ci n'est pas fournie.

Question 2. Écrire une méthode `connected_components_set` qui trouve les composantes connectées du graphe $\mathcal{G}$ et analyser la complexité de l'algorithme en fonction de V et E. La méthode `connected_components_set` doit retourner un set de frozenset (ceci est apparent si on regarde le test correspondant). Dans le code fourni, on suggère de commencer par implémenter une méthode `connected_components` qui retourne une liste de listes (une par composante connectée) puis d'appeler la fonction `connected_components_set` ; mais ce n'est pas obligatoire. Vous pouvez aussi modifier directement la méthode `connected_components_set`. L'important est de passer le test unitaire associé.

Indice : On pourra implémenter un parcours en profondeur du graphe en utilisant une fonction récursive et on pourra se limiter ici à une méthode qui fonctionne pour les petits graphes.

Question 3. Écrire une fonction `get_path_with_power` qui prend en entrée une puissance de camion p et un trajet t et qui décide si un camion de puissance p peut couvrir le trajet t (et retourne, si c'est possible, un chemin admissible pour le trajet t). En pratique, on retournera le chemin si c'est possible et `None` sinon.

Analyser la complexité de l'algorithme.

Question 4. Modifier la fonction de lecture des fichiers `graph_from_file` pour lire la distance d'une arête qui est optionnelle. Ainsi la ligne `1 2 10 5` représente une arête qui va de 1 à 2, avec puissance minimale 10 et distance 5. La ligne `1 2 10` représente toujours une arête qui va de 1 à 2, avec puissance minimale 10 et distance 1 (la valeur par défaut). Tester sur le fichier `network.04.in`.

Question 5 (bonus)**.** Modifier la fonction de la question 3 pour qu'elle retourne, lorsque le trajet peut être couvert par le camion, le plus court (au sens de la distance d) chemin admissible pour le trajet t.

Analyser la complexité de l'algorithme ci-dessus.

Question 6. Écrire une fonction `min_power` qui calcule, pour un trajet t donné, la puissance minimale d'un camion pouvant couvrir ce trajet. La fonction devra retourner le chemin, et la puissance minimale.

Analyser la complexité de l'algorithme ci-dessus.

Indice : On pourra penser à la recherche binaire.

Question 7 (bonus)**.** Implémenter une représentation graphique du graphe, du trajet, et du chemin trouvé. On pourra utiliser une bibliothèque pour cela, par exemple `graphviz`, vous pouvez l'installer sur le SSP Cloud avec le script `install_graphviz.sh`, à exécuter dans un terminal.

Question 8. Tester les algorithmes sur les graphes fournis. Implémenter de nouveaux tests des fonctions développées.

Question 9 (bonus, difficile)**.** Télécharger une carte de France. La convertir en un graphe avec des villes et des routes où les puissances minimales sont $\bar{p}_1$ pour les routes départementales et inférieures, $\bar{p}_2 > \bar{p}_1$ pour les routes nationales, et $\bar{p}_3 > \bar{p}_2$ pour les autoroutes. Pour la distance, on pourra prendre soit la distance en km soit le temps de trajet. Pour l'ensemble des trajets $\mathcal{T}$, on prendra l'ensemble des couples de deux villes de plus 100,000 habitants. À chaque trajet, on associera un profit égal au minimum du nombre d'habitants entre les deux villes.

Indice : On pourra utiliser la bibliothèque OSMnx à l'adresse suivante
https://osmnx.readthedocs.io/en/stable/

Si vous êtes arrivé à ce point et que tout fonctionne bien, vous pouvez commencer à tester sur des graphes plus gros (`network.x.in` avec `x` ≥ 1), ainsi qu'en répétant un grand nombre de fois la requête de la question 6. Pour cela, on fournit des ensembles de trajets `routes.x.in` avec `x` ≥ 1 associés à chaque graphe correspondant. La structure des fichiers `routes.x.in` est la suivante :

— la première ligne contient un entier qui correspond au nombres de trajets dans l'ensemble
— les lignes suivantes contiennent chacune un trajet sous la forme
`ville1 ville2 utilité`

Préparation de la séance 2

En préparation de la séance 2, les élèves doivent :

1. Créer un dépôt git du projet pour le binôme avec exactement la même structure que le git de base https://github.com/patrickloiseau/ensae-prog23, et se familiariser avec l'utilisation de git dans vscode.
2. Se familiariser avec le terminal et les commandes de base.
3. Réviser la notion de classes en Python
4. Terminer au minimum la question 1 de la séance 1 (obligatoire) et faire tourner le test associé (et se familiariser avec `unittest` si ce n'est pas acquis). Il est très très fortement recommandé d'avoir abordé aussi la question 2 et d'en profiter pour réviser la notion de graphe si elle n'est pas acquise. Il est également conseillé d'étudier les autres questions de la séance 1 pour avancer ensuite plus efficacement.

Chacun de ces points est court mais nécessaire pour pouvoir avancer efficacement dans les prochaines séances et acquérir une expérience suffisante de programmation. Pour vous aider, les instructions précises de chaque point sont détaillées ci-dessous—lisez les très attentivement.

2.1 Créer un dépôt git sur GitHub avec les fichiers du TP

GitHub est une plateforme de dépôt de code permettant le contrôle de version, c'est-à-dire qu'on peut tracer les modifications et les mises à jour. C'est un standard en programmation, il faut savoir mettre son code sur GitHub, et prendre l'habitude de mettre à jour le dépôt quand on fait des modifications. C'est d'autant plus important quand on code sur une plateforme cloud éphémère comme SSP Cloud.

Faites les différentes étapes ci-dessous :

Créer votre dépôt git Vous devez créer votre propre dépôt git sur GitHub **avec la même structure** que le git de base https://github.com/patrickloiseau/ensae-prog23 (c'est à dire les même répertoire et les mêmes fichiers aux mêmes endroits). (Plus généralement d'ailleurs, rappelons : il ne faut **jamais renommer un fichier ou répertoire** tel que `graph.py`.)

Si vous avez déjà créé votre dépôt git pour le projet, vérifier bien qu'il contient la même structure que le git de base. Notez aussi que nous avons ajouté de la documentation dans le fichier `graph.py`, vous pourrez donc vouloir le remplacer, mais faites bien attention à ne pas perdre le code que vous avez ajouté.

Si vous ne l'avez pas encore créé, une solution facile est la suivante :

— Connectez vous sur https://github.com/ (créez d'abord un compte si vous n'en avez pas)
— Cliquez sur le "+" en haut à droite puis sur "import repository"
— Dans le champs "Your old repository's clone URL" indiquez https ://github.com/patrickloiseau/ensae-prog23
— Dans le champs "repository name", mettez "ensae-prog23". Il est inutile de mettre votre nom dans le nom du dépôt puisque le chemin du dépôt sera déjà associé au nom de votre compte github.
— Cliquez sur "Begin Import"

Quand c'est fini, vous verrez "Importing complete! Your new repository `votre_login_github`/ensae-prog23 is ready." où `votre_login_github` est votre login github et vous pourrez y accéder à l'adresse https ://github.com/votre_login_github/ensae-prog23.

Cloner le dépôt git dans VSCode Maintenant le dépôt git créé ci-dessus sera le lieu sûr de stockage de la dernière version de votre code à tout moment.

Si vous utilisez SSP Cloud, à chaque début de session de codage (i.e., nouveau démarrage d'un service vs-code-python), dans VSCode, vous devrez clone le dépôt git. Cela copiera sur le cloud le

contenu du dépôt et le rendra directement accessible depuis VSCode. Pour cela, tapez "Shift+Ctrl+P" ou "Shift+Command+P" sur macOS, puis tapez "clone". Vous verrez apparaitre l'option "Git : clone", cliquez dessus. Cela vous demande l'adresse du dépôt git, donnez l'adresse du dépôt créé ci-dessus et cliquez sur entrée. Cela vous donne enfin une liste de répertoires où mettre le clone du dépôt, choisissez "work" (ce qui mettra le clone dans `/home/onixya/work/`) puis cliquez sur OK. Ensuite, suivez les instructions pour que VSCode se connecte à votre compte github. Vous devriez arrivez dans un nouvel onglet sur github jusqu'à l'étape "Congratulations, you're all set ! Your device is now connected." Vous pouvez fermer l'onglet. VSCode est en train de cloner le dépôt. Lorsque que c'est fini, cliquez sur "Open". C'est terminé. Vous devriez voir dans le panneau de gauche tous les fichiers de votre dépôt avec la bonne arborescence.

Note : si vous utilisez votre ordinateur personnel, vous n'avez pas à cloner le dépôt à chaque fois. Vous pouvez le faire une fois pour toute au début. Ensuite, vous pouvez simplement ouvrir dans VSCode le dossier dans lequel se trouve la copie locale du dépôt sans avoir à re-cloner.

Se familiariser avec l'utilisation de git dans VSCode Une fois le dépôt git accessible dans VSCode, il est possible de le mettre à jour directement depuis VSCode après avoir fait des modifications. La documentation officielle se trouve ici, mais c'est très simple. Il suffit de cliquer sur Commit dans l'onglet Source control à gauche et de se laisser guider. Essayez de faire des petites modifications et de les commiter (puis faire pull/push) sur le git pour vous familiariser.

ATTENTION : On rappelle encore une fois que si vous travaillez sur SSP Cloud, il est impératif de commiter et pusher ses modifications à la fin de chaque séance car sinon elles sont perdues.

2.2 Résumé sur l'utilisation du terminal

Le terminal est une interface en ligne de commande (par opposition aux interfaces graphiques) qui permet d'interagir avec le système d'exploitation [1] en écrivant des instructions. Dans ce TP, on va l'utiliser essentiellement pour exécuter du code python, mais il est très pratique de connaitre les commandes pour se déplacer d'un répertoire à un autre, ou lister les fichier d'un répertoire.

Depuis VSCode, pous pouvez ouvrir un terminal en utilisant le menu en haut à gauche, puis en sélectionnant « Terminal » et « New Terminal ». Sur la partie inférieure de votre écran, vous obtenez l'invite de commandes qui est de la forme :

```
onyxia@vscode-python-xxxx:~/work$
```

où `~/work` est le dossier actuel, `onyxia` est le nom d'utilisateur sur la machine virtuelle et `vscode-python-xxxx` est le nom de la machine virtuelle.

Une première commande : `pwd` La commande `pwd` permet d'afficher le chemin du dossier dans lequel on se trouve. Si vous tapez la commande et validez avec la touche Entrée, vous obtenez un résultat de la forme :

```
/home/onyxia/work
```

En effet, ce chemin est équivalent à `~/work`, le `~` étant un raccourci pour le dossier de l'utilisateur, ici `/home/onyxia`.

1. Ici, il s'agit d'un système d'exploitation de type GNU/Linux (en fait la distribution Ubuntu) exécuté sur une machine virtuelle sur la plateforme SSP Cloud.

Lister les fichiers du dossier La commande `ls` (pour *list*) permet d'afficher les fichiers et dossiers du dossier courant. Si vous avez cloné votre dépôt git dans le dossier `work`, vous devez obtenir quelque chose comme :

```
onyxia@vscode-python-xxxx:~/work$ ls
ensae-prog23 lost+found
onyxia@vscode-python-xxxx:~/work$
```

On voit notre dossier `ensae-prog23` (et le dossier `lost+found` qui est un dossier spécial dont on ne traitera pas ici).

Changer de dossier avec `cd` La commande `cd` permet de changer de dossier (c'est un acronyme de ***c**hange **d**irectory*). Par exemple, si vous avez cloné votre dépôt dans le dossier `work` sous le nom `ensae-prog23`, vous pouvez déplacer l'invite de commande dans ce répertoire avec :

```
onyxia@vscode-python-xxxx:~/work$ cd ensae-prog23
onyxia@vscode-python-xxxx:~/work/ensae-prog23$
```

Notez que le dossier de la deuxième ligne a changé.

Le dossier parent s'écrit `..`, ainsi la commande `cd ..` permet de retourner au dossier `~/work`.

Exécuter du code python La commande `python` permet d'exécuter un fichier `.py`.

```
onyxia@vscode-python-xxxx:~/work$ cd ensae-prog23
onyxia@vscode-python-xxxx:~/work/ensae-prog23$ python delivery_network/main.py
<exécution du code python>
```

Notez que la première commande `cd .....` n'est pas nécessaire à chaque fois puisqu'à la fin de l'exécution du code python on reste dans le dossier `~/work/ensae-prog23/delivery_network`.

Le dossier où on exécute le code python est important : certains scripts python peuvent être exécutés dans leur dossier directement (c'est le cas de `main.py`) tandis que d'autres sont prévus pour être exécuté uniquement dans le dossier parent, c'est le cas des fichiers de tests qu'on exécute depuis le dossier `~/work/ensae-prog23` :

```
onyxia@vscode-python-xxxx:~/work/ensae-prog23$ python tests/test_s1q1_graph_loading.py
```

2.3 Classes

En Python, tout est un objet. Il existe plein de types d'objets prédéfinis, tels que les entiers, les listes, les fonctions, etc. Les classes servent à définir de nouveaux types d'objets.

Définition et instanciation de la classe Dans notre cas, on veut définir le type d'objet correspondant à un graphe, dans la classe `Graph` du fichier `graph.py` (le plus souvent, chaque classe est dans son propre fichier). On peut alors instancier la classe, c'est-à-dire créer un objet de la classe, avec l'instruction suivante :

```
g = Graph([])
```

Dans ce cas, la variable `g` contient maintenant un objet de type graphe. La liste vide `[]` est l'argument du constructeur de la classe `Graph` (c'est la liste des nœuds initiaux du graphe).

Attributs et méthodes Cet objet a des attributs auxquels on accède par la notation `.`, par exemple `g.nb_nodes` pour l'attribut nombre de sommets. Enfin, on peut définir des méthodes, c'est-à-dire des fonctions qui agissent sur un élément de la classe, auxquelles on accède aussi par la notation `.`. La définition des attributs et des méthodes de la classe `Graph` est faite sous la ligne `class Graph`. Par exemple, une méthode `add_edge` est définie dans la classe de façon similaire à une fonction :

```
class Graph:
    ...
    def add_edge(self, node1, node2, power_min, dist=1):
        code de la methode
```

Le premier argument de la méthode est obligatoirement `self` et représente l'objet sur lequel la méthode est appelée.

La méthode peut-être exécutée sur l'objet `g` avec :

```
g.add_edge("Paris", "Palaiseau", 4, 20)
```

où `"Paris"` correspond au premier argument, etc.

Dans notre cas, cet appel doit ajouter l'arête `"Paris"`—`"Palaiseau"` de puissance minimale 4 et de distance 20 au graphe `g`.

Il est important de savoir écrire des classes, en particulier des méthodes dans les classes. Cela a été vu au 1er semestre et doit être acquis. Toutefois, en cas d'oubli, voici un lien vers courte introduction aux classes : https://courspython.com/classes-et-objets.html.

2.4 Indications pour la question 1 de la première séance

Pour finir la question 1 de la première séance, vous devez compléter la définition de la méthode `add_edge` et la fonction `graph_from_file`.

Structure de donnée pour un graphe Pour ce projet, on vous propose de représenter un graphe par ses listes d'adjacence, c'est-à-dire que pour chaque sommet x du graphe, on a une liste de ses voisins avec la puissance minimale et la distance de l'arête qui relie x à chacun de ses voisins. Cette liste est contenue dans un dictionnaire. Ainsi, pour le graphe suivant, avec des puissances minimales de 18 et des distances de 21 pour chacune des arêtes :

$$1 \text{ — } 2 \text{ — } 3$$

On doit obtenir, pour le sommet 2, une liste d'adjacence de la forme `[(1, 18, 21), (3, 18, 21)]`. Les listes d'adjacence sont stockées dans le dictionnaire `graph` qui est un attribut de la classe `Graph`.

Tester la fonction `add_edge` Dans votre fichier `main.py`, vous pouvez tester votre code de `add_edge` avec les instructions suivantes :

```
from graph import Graph # on importe la classe graphe du fichier graph.py

g = Graph([]) # creation d'un objet de type Graph
g.add_edge("Paris", "Palaiseau", 4, 20)

print(g) # affichage du graphe
```

Ce code doit être exécuté comme dans la section 2.2.

La fonction `graph_from_file` Cette fonction est à la fin du fichier `graph.py`, vous devez la compléter pour ouvrir un fichier de données. Cette fonction doit créer un objet de type `Graph`, puis utiliser la méthode `add_edge` pour ajouter chacune des arêtes du graphe et enfin renvoyer l'objet de type `graph`.

Vous pouvez ensuite tester cette fonction en écrivant, dans `main.py`, un code qui importe la fonction `graph_from_file`, ouvre un fichier avec `g = graph_from_file("input/network.00.in")` par exemple, puis l'affiche.

De façon plus systématique, vous pouvez la tester en exécutant le test `test_s1q1_graph_loading.py` en tapant, depuis le répertoire racine `ensae-prog23` la commande :

```
python tests/test_s1q1_graph_loading.py
```

Ce test fait plus ou moins la même chose (exécute la fonction et vérifie que ce qui est retourné est ce qui est attendu) mais en utilisant le framework `unittest` qui est le standard pour les tests. Si vous n'êtes pas familier avec le framework `unittest`, cherchez de la documentation sur Internet pour vous familiariser, et utiliser l'example fourni par `tests/test_s1q1_graph_loading.py` pour mieux comprendre.

Séance 2 : 24/2

Les solutions trouvées dans la séance 1 sont intéressantes car pour un trajet donné elle donne la réponse correcte. Toutefois, si le nombre de trajets T est grand (par example de l'ordre de quelques centaines de milliers comme dans les fichiers `routes.x.in`), elles deviennent trop coûteuses (voir la question 10). On aimerait pouvoir avoir une solution plus efficace. Dans cette séance et la suivante, on s'intéresse trouver des algorithmes permettant de réduire la complexité *par trajet*, à l'aide d'une étape de pré-processing. Dans cette partie on ne s'intéresse pas à la distance d sur les arêtes mais uniquement à leur poids $\bar{p}$.

Question 10. Estimer le temps nécessaire pour calculer (à l'aide du code développé dans la séance 1) la puissance minimale (et le chemin associé) sur l'ensemble des trajets pour chacun des fichiers `routes.x.in` donnés.

Indice : Il ne faut pas chercher à calculer la puissance minimale sur l'ensemble des trajets ; ce serait beaucoup trop long (pensez également aux aspects écologiques !). On pourra se contenter de faire tourner pour quelques trajets pour estimer le temps moyen par trajet et multiplier par le nombre de trajets. Pour le calcul du temps d'execution, une option simple est l'utilisation de `time.perf_counter()` disponible dans le package `time` (ne pas oublier de l'importer !).

On voit bien que ce temps n'est pas raisonnable. On va maintenant chercher à l'améliorer significativement. Une clef pour améliorer est l'observation suivante : soit $\mathcal{A}$ un arbre couvrant de poids minimal du graphe $\mathcal{G}$. Alors, la puissance minimale pour couvrir un trajet t dans le graphe $\mathcal{G}$ est égale à la puissance minimale pour couvrir ce trajet t dans le graphe $\mathcal{A}$. On rappelle qu'un arbre est un graphe acyclique connecté (qui a la forme classique sous forme d'ancêtres et descendants). Un arbre couvrant de poids minimal (minimum spanning tree en anglais) est un arbre construit avec des arêtes du graphe initiale (en n'en retenant que certaines) qui couvre tous les sommets du graphe et dont la somme des poids des arêtes (les $\bar{p}$ pour nous) est minimale parmi tous les arbres couvrants. Il a exactement $V - 1$ arêtes (comme tous les arbres ayant V sommets). Pour ceux qui ne maitrisent pas bien ces notions, on conseille le livre de Dasgupta et al., très court et clair et disponible ici.

Question 11 (bonus)**.** Prouver l'affirmation ci-dessus.

Avec cette observation, nous pouvons maintenant suivre une procédure en 2 étapes : (i) on calcule d'abord un arbre couvrant de poids minimal et (ii) on cherche ensuite un chemin pour couvrir chaque trajet dans ce graphe, ce qui est beaucoup plus rapide que de chercher dans le graphe initial.

Question 12. Ecrire une fonction `kruskal` qui prend en entrée un graphe au format de la classe `Graph` (e.g., `g`) et qui retourne un autre élément de cette classe (e.g., `g_mst`) correspondant à un arbre couvrant de poids minimal de `g`. Analyser la complexité de l'algorithme implémenté.

Indice : On suggère d'implémenter l'algorithme de Kruskal. Celui-ci est très simple : on prend chaque arête dans l'ordre des poids croissant seulement si l'ajouter aux arêtes déjà prises ne crée pas de cycle (sinon on la jète et on passe à la suivante). La difficulté est de, quand on considère une arête, déterminer rapidement si l'ajouter aux arêtes déjà prises crée un cycle. Ceci peut être fait efficacement avec une structure de donnée particulière appelé Union-Find. Toutes les informations sont disponibles dans les pages 127-134 du livre de Dasgupta et al, mais vous êtes libre de consulter tout autre source d'information sur l'algorithme de Kruskal et la structure Union-Find.

Question 13. Implémenter des tests de la fonction ci-dessus sur les petits graphes fournis.

Note : Cette question est écrite ici pour rappel mais rappelons qu'il faut toujours implémenter des tests unitaires, même quand ça n'est pas demandé.

Question 14. Ecrire une nouvelle fonction basée sur l'arbre couvrant de poids minimal qui calcule, pour un trajet t donné, la puissance minimale d'un camion pouvant couvrir ce trajet (et le chemin associé).

Indice : On ne considère plus ici que l'arbre couvrant, qui contient potentiellement beaucoup moins d'arêtes. Par ailleurs, puisqu'on sait que c'est un arbre, il est possible d'optimiser la recherche d'un chemin (unique maintenant) entre deux sommets.

Question 15. Analyser la complexité de la solution précédente théoriquement, et reprendre en pratiques les estimations de temps de la question 10 pour cette nouvelle fonction. Comparer.

Note : Ici, on cherchera à stocker les résultats des puissances minimum pour couvrir chaque trajet sous la forme suivante : pour chaque fichier `routes.x.in`, écrire un fichier `routes.x.out` qui contient T lignes avec sur chaque ligne un seul nombre correspondant à la puissance minimale pour couvrir le trajet.

Séance 3 : 10/3

Note : le contenu de cette séance est **bonus à condition** d'avoir obtenu à la fin de la séance précédente un code permettant de calculer les résultats pour tous les trajets (jusqu'à 500k dans certains fichiers `routes.x.in`) en un temps raisonnable (de l'ordre de la minute). On ne s'intéresse dans cette séance qu'à la puissance minimal d'un camion pouvant couvrir chaque trajet (i.e., on ne cherche pas à calculer le chemin associé).

La méthode trouvée dans la séance 2 devrait avoir permis une amélioration très significative par rapport à la séance 1, mais reste en théorie d'ordre $O(V)$ par trajet dans le pire cas. On essaie ici de réduire cette complexité à $O(\log V)$. Pour cela, on observe d'abord qu'un chemin v-v' dans un arbre est unique et peut être décomposé en deux parties par rapport au *plus petit ancêtre commun* de v et v' (lowest common ancestor ou LCA en anglais), défini comme suit : Le *plus petit ancêtre commun* de deux sommets v et v' dans un arbre $\mathcal{A}$ est le plus petit (i.e., plus profond) sommet qui a v et v' comme descendants. [2] Pour réduire le temps par trajet à $O(\log V)$, on doit pouvoir trouver le plus petit ancêtre commun de deux sommets et calculer le poids maximal sur le chemin entre ces deux sommets dans l'arbre, en temps $O(\log V)$.

Question 16. Ecrire un programme qui calcule, pour un trajet t donné, la puissance minimale d'un camion pouvant couvrir ce trajet, tel que le temps de pré-processing soit $O(V \log V)$ et ensuite le temps de réponse pour chaque trajet soit $O(\log V)$. Justifier cette complexité.

Indice : On pourra utiliser un pré-processing qui calcule, pour chaque sommet v l'ancêtre de v qui est à distance 2^i de v pour tout $i = 1, 2, \cdots$ (et garder le poids maximal associé). Notez que ce pré-processing ajoute une complexité en espace en $O(V \log V)$ (information à garder en mémoire), et qu'il faut qu'il soit fait en temps $O(V \log V)$.

Question 17. Tester l'algorithme ci-dessus sur les examples fournis et calculer le temps d'exécution. Comparer à la version de la séance 2.

2. On dit par convention que chaque sommet est descendant de lui-même. Ainsi, si v' a une connection directe depuis v alors v est le plus petit ancêtre commun.

Séances 4-6 : 16/3, 30/3, 13/4

Pour les séances 4 et 5 (16/3 et 30/3), vous travaillerez sur le problème ci-dessous. Le problème est volontairement plus ouvert. Il est attendu que vous analysiez le problème, proposiez une solution et décriviez sa complexité théorique. Vous devez ensuite implémenter et tester votre algorithme, en le comparant éventuellement avec d'autres solutions possibles telles que des solutions plus naïves (à la fois théoriquement et en pratique). Dans cette partie, on peut évidement réutiliser tout ce qui a été fait précédemment.

Le problème est le suivant : on dispose d'un catalogue de camions, où chaque camion a une puissance p et un coût c. On suppose que chaque modèle de camion est en stock illimité. Pour chaque camion acheté, on peut l'affecter à un trajet au maximum, et on obtient le profit du trajet (à condition bien-sûr que la puissance du camion soit suffisante pour couvrir le trajet). Par contre, on ne peut pas gagner plus de profit en affectant plusieurs camions à un trajet donné.

La compagnie de transport possède un budget fixe $B = 25 \cdot 10^9$ pour acheter des camions et souhaite savoir quels camions acheter pour maximiser le profit obtenu.

Pour les tests, on donne plusieurs catalogues de camions dans les fichiers `trucks.x.in`, qui ont la structure suivante :

— la première ligne est composée d'un entier correspondant au nombre de modèles de camions disponibles dans le catalogue ;
— les lignes suivantes représentent chacune un camion et sont composées de 2 entiers : la puissance et le coût du camion.

Chacun des catalogues peut être utilisé pour tous les fichiers des précédentes séances `network.x.in` (et les `routes.x.in` associés).

Question 18. Écrire un programme qui retourne une collection de camions à acheter ainsi que leurs affectations sur des trajets, pour maximiser la somme des profits obtenus sur les trajets couverts. Analyser la solution et tester le programme sur les différents fichiers `network.x.in` / `routes.x.in` et avec, pour chacun, les différents catalogues `trucks.x.in`.

Indice : Pour trouver l'optimum exact, on pourra réfléchir à des approches de type force brute, ou à des analogies avec des problèmes connues, par exemple le problème du sac à dos (knapsack). Toutefois, ces méthodes ne marcheront pas nécessairement en un temps raisonnable pour tous les fichiers d'entrée. On pourra alors réfléchir à des méthodes rapides (e.g., de type greedy, ou des heuristiques de type local search, ou d'autres idées) mais qui ne garantissent pas de trouver le maximum global.

Question 19 (bonus)**.** Implémenter une visualisation de l'allocation sur des petits exemples (par exemple `network.0.in` et `network.1.in`).

Indice : Vous pouvez utiliser comme base ce que vous avez fait à la question 7.

Question 20 (bonus)**.** Reprendre la question 18 dans un cas plus réaliste suivant : on suppose maintenant que

(i) Chaque arête a une probabilité ϵ de se "casser" (i.e., la route est bloquée), indépendamment des autres arêtes. On acquiert le profit correspondant à un trajet seulement si aucune arête sur le chemin entre les deux villes n'est cassée.
(ii) Le passage d'un camion sur un trajet t donne un coût (de carburant) proportionnel à la distance totale du chemin couvrant ce trajet (i.e., la somme des distances sur les arêtes de ce chemin).

On cherche à maximiser le profit en espérance du (i) moins le coût de carburant du (ii). On pourra prendre un coût de carburant de 0.01 par unité de distance et $\epsilon = 0.001$ pour les tests.

Indice : On pourra commencer par recalculer les profits en espérance pour chaque trajet, tout en gardant les chemins trouvés dans les séances précédentes à partir de l'arbre couvrant de poids minimal. Ceci est déjà une première approche, mais ne garanti pas une solution optimale même si on a l'optimum

exact dans la question 18. Ensuite, on pourra se poser la question de comment optimiser le profit en espérance (moins le coût de carburant) en s'autorisant à revenir sur le choix du chemin—mais cette deuxième partie est beaucoup plus difficile.

Recalculer les profits est faisable et pas trop dur. Optimiser est beaucoup plus dur car il faudrait, pour chaque trajet, avoir une liste des chemins avec leurs distances associées.

On rappelle que la deadline pour le rendu final (incluant le rapport au format PDF) est le jeudi 06/04 à 23h59. La séance 6, qui est après la deadline pour le rendu final, sera consacré à l'évaluation. Les enseignants auront lu les projets et passeront dans les groupes pour poser des questions. Cela sera aussi l'occasion de donner un retour aux étudiants sur leurs projets.