# Projet

March 18, 2025

## 1 Parcours de graphe

L'objectif de ce projet est d'étudier des algorithmes de calcul de plus court chemin.

Ce projet est :

- À faire seul ou en binôme
- À rendre sous forme de notebook Jupyter sur Moodle au plus tard le vendredi 12 avril à 23h59.

Ce sujet a été testé sur la machine virtuelle "ULR Ubuntu 20.04". Sur cette machine, pour la première exécution **uniquement**, avant de lancer Jupyter, dans un terminal déplacez-vous dans le dossier où se trouve ce fichier Jupyter, et faites :

- `python -m venv env`
- `source env/bin/activate`
- `pip install ipykernel`
- `python -m ipykernel install --name=env --user`
- `jupyter-notebook`
- Ouvrez ce fichier jupyter
- enfin, dans le menu Noyau -> Changer de noyau -> env

Pour les fois suivantes, vous aurez juste à ouvrir un terminal, vous déplacer dans le dossier où se trouve votre fichier jupyter, puis :

- `source env/bin/activate`
- `jupyter-notebook`

Ce sujet devrait aussi marcher sur les installations Linux récentes, et sur Google Collab.

[1]:
```
# Installation des dépendances

!pip install partition-networkx
!pip install matplotlib

import networkx as nx
import matplotlib.pyplot as plt
```

```
error: externally-managed-environment

× This environment is externally managed
 > To install Python packages system-wide, try apt install
```

```
    python3-xyz, where xyz is the package you are trying to
    install.

    If you wish to install a non-Debian-packaged Python package,
    create a virtual environment using python3 -m venv path/to/venv.
    Then use path/to/venv/bin/python and path/to/venv/bin/pip. Make
    sure you have python3-full installed.

    If you wish to install a non-Debian packaged Python application,
    it may be easiest to use pipx install xyz, which will manage a
    virtual environment for you. Make sure you have pipx installed.

    See /usr/share/doc/python3.11/README.venv for more information.

note: If you believe this is a mistake, please contact your Python
installation or OS distribution provider. You can override this, at the risk of
breaking your Python installation or OS, by passing --break-system-packages.
hint: See PEP 668 for the detailed specification.
error: externally-managed-environment

× This environment is externally managed
 > To install Python packages system-wide, try apt install
    python3-xyz, where xyz is the package you are trying to
    install.

    If you wish to install a non-Debian-packaged Python package,
    create a virtual environment using python3 -m venv path/to/venv.
    Then use path/to/venv/bin/python and path/to/venv/bin/pip. Make
    sure you have python3-full installed.

    If you wish to install a non-Debian packaged Python application,
    it may be easiest to use pipx install xyz, which will manage a
    virtual environment for you. Make sure you have pipx installed.

    See /usr/share/doc/python3.11/README.venv for more information.

note: If you believe this is a mistake, please contact your Python
installation or OS distribution provider. You can override this, at the risk of
breaking your Python installation or OS, by passing --break-system-packages.
hint: See PEP 668 for the detailed specification.
```

### 1.1 Échauffement

Le code ci dessous permet de générer un graphe $G$. Vous utiliserez ce graphe pour tout le reste du projet.

[3]:
```
# Génération d'un graphe aléatoire contenant 200 sommets

G = nx.gnp_random_graph(200, 0.6)

sommet1 = 10
sommet2 = 20
```

### 1.1.1 Implémentation de l'algorithme de Dijkstra

L'algorithme de Dijkstra est un algorithme de calcul de plus court chemin dans un graphe, propose par Edsger Dijkstra en 1959.

Je vous propose de commencer par implémenter cet algorithme. Vous avez normalement vu cet algorithme en CM et en TD. Si vous en avez besoin, vous pouvez aussi consulter la page Wikipédia, ou même la publication originale. Votre implémentation retournera un plus court chemin de la forme `[source, sommet_1, sommet_2, ..., target]`.

[63]:
```
def dijkstra(G, source, target):
    L = []
    # TODO
    return L
```

Vérifiez vos résultats en les comparant au plus court chemin fourni par l'implémentation de l'algorithme de Dijkstra par la librairie Networkx à l'aide du code ci-dessous.

Ne touchez pas au code ci-dessous, je m'en servirai pour noter vos travaux.

[7]:
```
path1 = nx.dijkstra_path(G, sommet1, sommet2)
print(path1)
path2 = dijkstra(G, sommet1, sommet2)
print(path2)
print(path1 == path2)
```

```
[10, 6, 64, 20]
[10, 6, 64, 20]
True
```

L'inconvévient de l'algorithme de Dijkstra est qu'il explore une grande partie du graphe avant de trouver le plus court chemin. C'est à dire qu'en partant d'un sommet source, il va passer en revue beaucoup de sommets avant de trouver le sommet destination. D'autres algorithmes, comme l'algorithme A* proposent des optimisations permettant d'explorer moins de sommets, et donc de trouver un plus court chemin plus rapidement.

### 1.1.2 Étude du temps d'exécution

Affichez maintenant le temps d'exécution de votre implémentation de Dijkstra en fonction de la taille d'un graphe $H$ (en nombre de sommets).

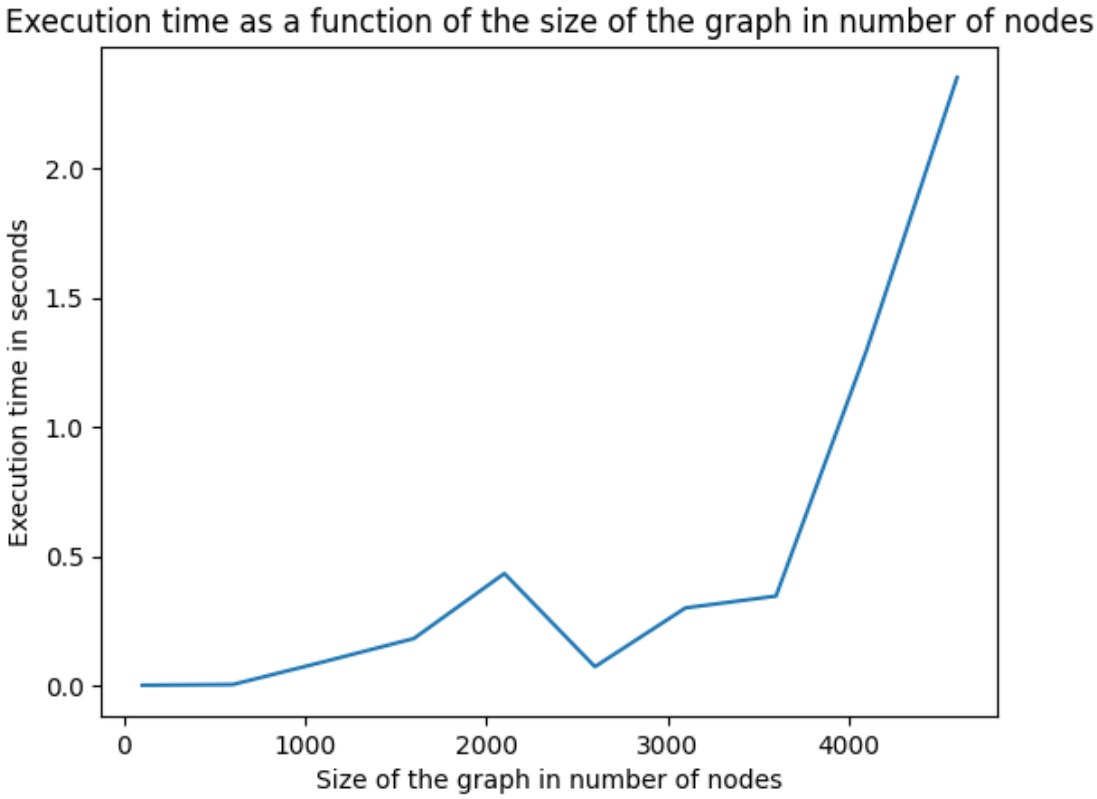
Vous pouvez générer de nouveaux graphes pour cette question uniquement. Dans le reste de votre projet, vous devez utiliser le graphe $G$ généré au début de ce document.

Vous pouvez utiliser la librairie time et/ou datetime pour mesurer des temps d'exécution.

Vous pouvez utiliser la librairie matplotlib pour afficher vos résultats. J'ai utilisé plt.plot() pour la figure ci-dessous.

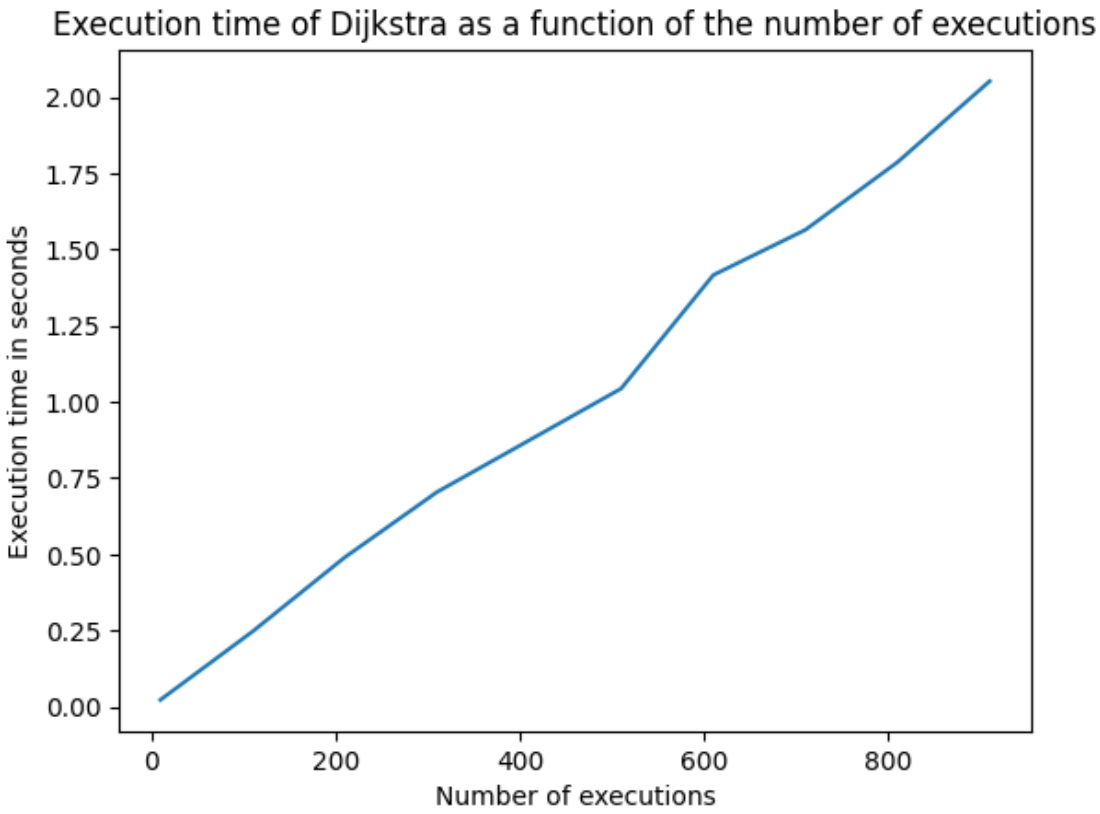
J'ai volontairement laissé la figure que j'obtiens avec mon code pour vous donner une idée du résultat attendu.

[20]:




Affichez maintenant le temps d'exécution de votre implémentation de Dijkstra en fonction du nombre de plus court chemins demandés (donc en fonction du nombre d'exécutions de Dijkstra), pour le graphe $G$ (généré lors de l'échauffement).

[9]:




Sans surprise, le temps d'exécution augmente de manière linéaire par rapport au nombre d'exécution de Dijkstra demandé : deux exécutions de Dijkstra prennent environ deux fois plus de temps qu'une seule exécution. Ceci n'est pas vrai pour tous les algorithmes. Certaines approches réalisent une étape de précalcul, qui permet d'économiser du temps lors des futurs exécutions.

### 1.2 Algorithme d'Akiba

Nous allons maintenant étudier l'algorithme d'Akiba, qui est un algorithme de calcul de distance (longueur d'un plus court chemin) dans des graphes, proposé en 2013 par Akiba *et al.* Cet algorithme réalise d'abord une étape de précalcul, pendant laquelle il ajoute des étiquettes (*labels*) sur les sommets du graphe. Ces étiquettes permettent ensuite de répondre rapidement à des requêtes demandant d'identifier la distance entre deux sommets.

Plus précisément, cet algorithme :

- Calcule pour chaque sommet $u$ de $G$, des étiquettes (pas toutes) de la forme $(v_i,\ (u, v_i))$, où $(u, v_i)$ représente la distance entre u et $v_i$ dans $G$.
- L'ensemble d'étiquettes $L$ est calculé en faisant un parcours en largeur partiel, depuis chaque sommet $u$ de $G$.
- Une fois $L$ calculé, la distance entre deux sommets $u$ et $v$ est calculée en faisant `requete(u, v, L)`. Cette fonction consiste simplement à trouver un sommet $z$ tel que :
  - $(z,\ (u, z))$ et $(z,\ (v, z))$ existent, c'est à dire que $z$ est dans les étiquettes de $u$ et de $v$
  - $z$ minimise $(u, z) + (v, z)$, c'est à dire la distance en $u$ et $z$ puis la distance entre $z$ et $v$.

L'algorithme d'Akiba ne calcule pas toutes les étiquettes de chacun des sommets, ce serait très couteux en temps et en mémoire. L'algorithme effectue des parcours en largeur partiels, comme suit :

- Initialement, l'ensemble d'étiquettes $L_0$ est vide. L'ensemble d'étiquettes $L_k$ est généré lors du $k$ ième parcours en largeur. Lors du $k+1$ ième parcours en largeur, effectué depuis un sommet $u$, l'algorithme ne parcours pas les voisins d'un sommet $v$ (donc dans l'arbre généré par ce parcours, $v$ sera une feuille) quand `requete(u, v, L_k)` est inférieur ou égal à la distance mesurée lors du $k+1$ ième parcours. Autrement dit, on ne parcours pas au dela du sommet $v$, quand on sait déjà qu'il existe un chemin plus court reliant $u$ à $v$.

Pour plus de détails, je vous invite à consulter la publication originale (vous pouvez bien sûr aussi me poser des questions).

### 1.2.1 Parcours en largeur

Commencez par implémenter un parcours en largeur complet d'un graphe $G$, sans l'optimisation mentionée ci-dessus. `BFS(G, u)` doit retourner un dictionnaire $L$ tel que `L[v]` est égal à la distance entre le sommet $u$ et le sommet $v$ dans le graphe $G$, pour tous les sommets $v$ du graphe $G$.

[11]:
```
def BFS(G, u):
    L = {}
    # TODO
    return L
```

### 1.2.2 Approche naïve

Effectuez un parcours en largeur depuis chaque sommet $u$ de $G$, puis remplissez l'ensemble d'étiquettes $L$. `L[u][v]` est égal à la distance entre $u$ et $v$ dans $G$. Retournez $L$.

[12]:
```
def naive(G):
    L = {}
    # TODO
    return L
```

**Question :** Quelle est la complexité de `naive(G)` ? Donnez simplement l'ordre de grandeur (par exemple $O(n)$), ainsi qu'une justification en quelques lignes, pas besoin de fournir de coefficients ni de constantes.

**Réponse :**

### 1.2.3 Requête

Implémentez une méthode `requete(u, v, L)` permettant de récupérer la distance entre $u$ et $v$ dans $G$ depuis $L$. Pour ce faire, à partir de `L[u]` et `L[v]`, vous devez trouver le sommet $z$ qui minimise la distance `L[u][z] + L[v][z]`. Retournez cette distance.

[13]:
```
def requete(u, v, L):
    distance = 0
    # TODO
    return distance
```

### 1.2.4 Akiba

Vous allez maintenant implémenter l'algorithme d'Akiba. Cet algorithme est similaire à l'approche naïve que vous avez déjà implémentée. L'idée est d'effectuer des parcours en largeur partiels, de manière à éviter de renseigner deux fois la même information dans l'ensemble d'étiquettes $L$, comme expliqué ci-dessus.

Commencez par implémenter la méthode `pruned_BFS(G, u, L)`, qui va effectuer un parcours en largeur partiel depuis le sommet $u$ dans le graphe $G$. Au moment où le parcours considère le sommet $v$, vous ne devez pas explorer les voisins de $v$ si `requete(u, v, L)` est inférieur ou égal à la distance entre $u$ et $v$ mesurée lors de ce parcours. Sinon, vous devez rajouter l'étiquette `L[u][v]` (donc la distance entre $u$ et $v$) à $L$, comme expliqué dans la publication originale.

Retournez l'ensemble d'étiquette $L_{new}$, qui est l'ensemble d'étiquettes $L$, auquel vous aurez rajouté les étiquettes du parcours en largeur partiel depuis $u$.

[14]:
```
def pruned_BFS(G, vk, L):
    L_new = L
    # TODO
    return L_new
```

Vous pouvez maintenant effectuer un `pruned_BFS(G, u, L)` depuis tous les sommets $u$ de $G$, de manière à remplir l'ensemble d'étiquettes $L$.

[15]:
```
def akiba(G):
    L = {}
    return L
```

Votre implémentation de l'algorithme d'Akiba devrait maintenant être fonctionnelle. Vous pouvez vérifier son bon fonctionnement en comparant son résultat avec le résultat de la fonction shortest_path_length de la librairie Networkx. Je vous conseille de vérifier vos résultats sur plusieurs paires de sommets car j'utiliserai une paire de sommets aléatoire, ainsi qu'un autre graphe quand je noterai vos travaux.

Ne touchez pas au code ci-dessous, je m'en servirai pour noter vos travaux.

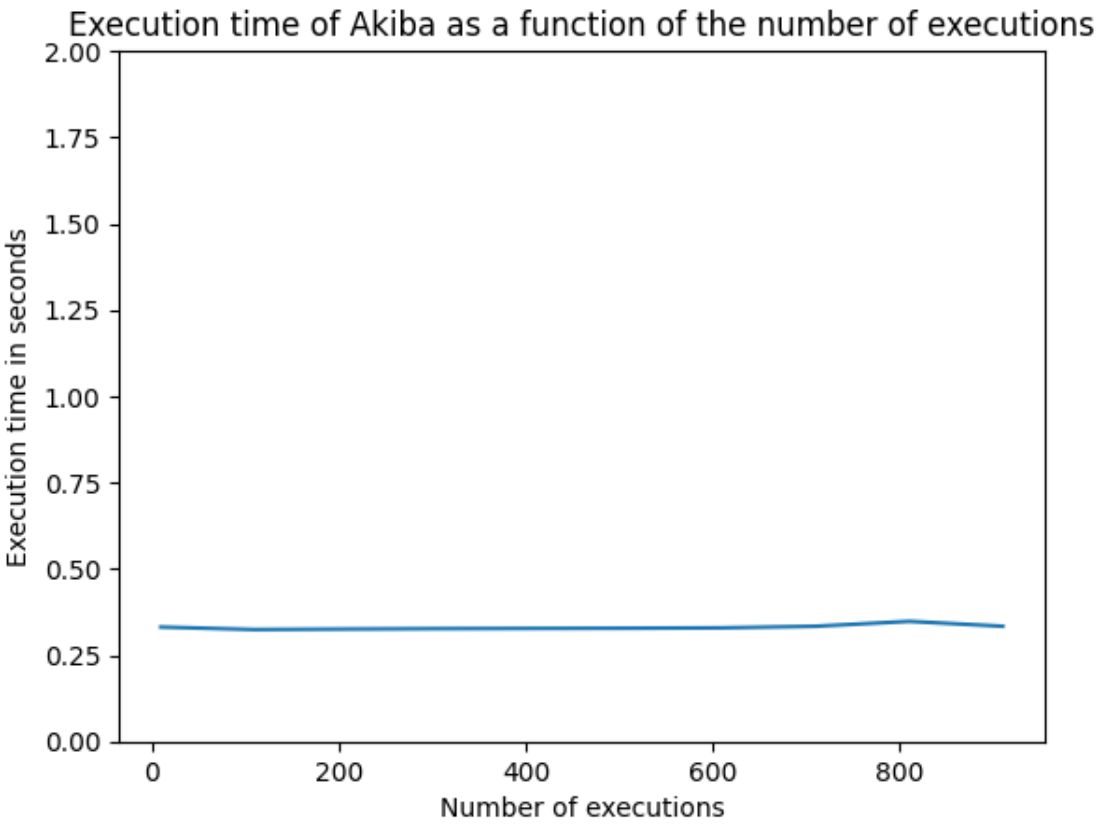
[88]:
```
distance1 = nx.shortest_path_length(G, sommet1, sommet2)
print(distance1)
L = akiba(G)
distance2 = requete(sommet1, sommet2, L)
print(distance2)
print(distance1 == distance2)
```

```
2
2
True
```

Pour finir, affichez le temps d'éxecution de votre implémentation d'Akiba en fonction du nombre de requêtes (le nombre d'exécutions de `requete(u, v, L)`).

[17]:
```

```




L'algorithme d'Akiba prend plus de temps que Dijkstra lors de la première requête car il doit construire l'ensemble d'étiquettes. Par contre, les requêtes suivantes prennent très peu de temps. Cet algorithme est donc rentable si le nombre de requêtes est élevé.