Question 4

Introduction

L'idée de cet exercice est de simuler l'envoie de plusieurs messages à travers un réseau lors d'une catastrophe naturelle. L'objectif est donc de transmettre plusieurs messages entre deux points d'un réseau. Le réseau sera représenté par un graphe de plusieurs noeuds qui représentent respectivement une région et les villes au sein de cette région.

Le noeud d'entré symbolise la ville dans laquelle la catastrophe naturelle a lieu. Le noeud de sortie symbolise la ville dans laquelle se trouve le bureau d'émergence de la région. Il faut donc trouver le chemin le plus court entre les deux villes afin que le bureau d'émergence soit tenu informé de ce qu'il se passe dans la zone dangereuse.

Implémentation

Cette simulation a été réalisée en Python dans un environnement virtuel. De cette manière, les librairies nécessaires pour lancer le programme ne sont pas installées globalement sur le système mais seront uniquement accessible pour ce projet spécifiquement. La démarche pour installer l'environnement virtuel et exécuter le programme est disponible à la section Installation et exécution.

Création d'un noeud

Les noeuds d'un graphe représentent les points de passage par lesquels un message peut transiter afin que l'information soit transmise entre le noeud de départ et d'arrivé.

Pour que l'on puisse créer le graphe mais également pour pouvoir transiter d'un noeud à un autre, nous allons devoir stocker certains informations :

- Les coordonnées afin de calculer la distances entre chaque noeud,
- L'index, afin d'identifier un noeud dans la liste,
- Le score du noeud courant pour, une fois l'ensemble des messages envoyés, pouvoir déterminer le noeud gagnant,
- Les voisins du noeud sous la forme d'un dictionnaire avec comme clé, l'index du noeud voisin et en valeur, la distance.

Création du réseau

La création du réseau a été faite de manière à ce que celui-ci soit aléatoire à chaque exécution du programme. Afin de créer un réseau aléatoire, nous avons décidé de nous baser sur une grille de n×n cases. Ainsi, nous pouvons facilement placer les noeuds du graphe de manière aléatoire sur la grille tout en ayant accès à la distance entre chacun d'entres-eux grâce à la formule de la distance euclidienne.

Ensuite, à chaque création d'un nouveau noeud dans le graphe, nous allons le lier à un certain nombre de noeuds déjà présentent dans le graphe. Ce nombre sera choisi aléatoirement entre 1 et 10.

Au sein de notre réseau, nous devons également sauvegarder certains paramètres afin de pouvoir facilement travailler avec :

- Le nombre de noeuds à un instant précis,
- Le nombre de noeuds lors de la création du graphe,
- La liste des noeuds du graphe,
- Le seuil de distance à partir duquel un message ne peut plus transiter d'un noeud à un autre,
- Le noeud de départ,
- Le noeud d'arrivé,
- La liste des index des noeuds supprimés,
- La liste des emplacements des noeuds supprimés dans la liste des noeuds du graphe.

Logique principale de la simulation

Comme nous avons pu l'évoquer, le but est de transmettre *n* messages entre deux noeuds d'un graphe. Le chemin doit être le plus court possible et la distance entre chaque noeud du chemin doit être inférieure au seuil que nous allons définir.

```
network = Network(nb_nodes=NB_NODES, distance_threshold=DISTANCE_THRESHOLD)
```

Dans un premier temps, nous commençons pas créer un graphe aléatoire en utilisant la méthode que nous avons décrite précédemment.

```
shortest_path = None
```

Nous déclarons et initialisons ensuite notre variable `shortest_path` afin de pouvoir conserver le plus cours chemin d'un envoie d'un message à un autre.

Nos variables principales étant définies, nous allons maintenant pouvoir passer au coeur du programme : la logique de sélection du chemin et l'envoi des messages. Toute cette logique est implémentée dans une boucle for qui boucle *k* fois, *k* étant le nombre de message que l'on souhaite envoyer.

Pour rappel, l'idée de cet exercice est de simuler l'envoie de plusieurs message entre deux points pendant une catastrophe naturelle.

```
if random.randint(1, 2) == 1:
    network.remove_random_node()
```

Nous avons donc décidé de, de manière aléatoire, supprimer un noeud du graphe afin de simuler l'inaccessibilité d'une ville de la région pendant l'envoie des messages. Ainsi, à chaque envoie d'un nouveau

message, nous avons une probabilité de 0.5 qu'un noeud soit supprimé du graphe. Le noeud sélectionné et supprimé sera, bien évidemment, différent de ceux de départ et d'arrivé.

Ensuite, dans le cas ou il n'existe aucun chemin optimal entre les deux points, nous essayons d'en déterminer un. Pour ce faire, nous avons utilisé l'algorithme de **Dijkstra**. Cet algorithme est idéal dans ce cas de figure car il permet, précisement, de déterminer le chemin le plus court dans un graphe connexe.

L'algorithme de Dijkstra est relativement simple à comprendre. L'idée est de, successivement, déterminer le chemin le plus court entre le noeud actuel et le point de départ en sauvegardant le prédécesseur par lequel le chemin est passé pour atteindre le noeud actuel. Une fois tous les noeuds parcourus, il ne reste plus qu'à remonter le chemin dans le sens inverse en partant du point d'arrivé puis en remontant les prédécesseurs jusqu'à atteindre le point de départ.

Après avoir déterminé notre chemin, deux cas de figures peuvent se présenter :

- Après avoir supprimé un certain nombre de noeuds du graphe, il est possible que celui-ci devienne non-connexe et que les noeuds de départ et d'arrivé soient dans deux sous-graphe différent. Dans ce cas, aucun chemin n'existe.
- L'algorithme de Dijkstra trouve un chemin optimal entre les deux points mais la distance entre deux noeuds est supérieur au seuil de distance. Dans ce cas, le message ne peut pas transiter entre les deux noeuds impliqué et donc, le chemin n'est pas valide.

Ainsi, si aucun chemin n'existe ou s'il n'est pas valide, alors le message est perdu. Il faudra alors attendre le prochain envoie d'un message pour réessayer de trouver un chemin optimal.

Optimisation

Pour des raisons de performances, nous avons dû optimiser notre algorithme. En effet, plutôt que de re-calculer le chemin optimal à chaque envoie d'un message, nous avons opté pour une approche légerement différente.

En effet, nous re-calculons un nouveau chemin avec l'algorithme de Dijkstra dès que notre chemin n'est plus valide. Autrement-dit, tant que notre chemin existe et est valide, nous le concervons pour envoyer les messages du point de départ au point d'arrivé. Malheureusement, dès lors ou un chemin n'est pas valide, nous perdons un message. En d'autres termes, si nous avons déterminé un chemin valide lors de l'envoie du message 1, puis que, de manière aléatoire, lors de l'envoie du message 23, un noeud de ce chemin n'est plus accessible, alors ce même message 23 sera perdu. Toutefois, la perte de ce message donnera l'information au réseau afin de re-calculer un nouveau chemin lors de l'envoie du prochain message.

Ainsi, tant que les noeuds du chemin déterminé lors de l'envoie du premier message existent, alors nous n'aurons qu'à déterminer un seul chemin optimal ce augmente les performances générales de la simulation.

Description des classes

Node.py

La classe `Node` sert à symboliser les différentes intersections du graphe par lesquelles un message peut passer afin d'atteindre le points d'arrivé.

Plusieurs attributs seront créés grâces aux paramètres qui doivent être renseignés pour créer cet objet :

- L'index du noeud, qui sert à identifier le noeud,
- Les coordonnées du noeud dans la grille lors de la création du graphe. Cet attribut correspond à une instance de la classe **Coordinates **qui sert à plus facilement manipuler l'abscisse et les ordonnées du noeud,
- Le score,
- Les noeuds voisins du noeud actuel.

Également, différentes méthodes sont disponibles pour facilement intéragir et modifier cette classe :

- Une méthode permettant d'incrémenter le score du noeud courant,
- Une méthode pour supprimer un certain voisin lorsqu'un noeud est supprimé du graphe.

Coordinates.py

La classe `Coordinates` est très simple car elle permet de facilement récupérer les coordonnées d'un noeud dans la grille de départ.

Seulement 2 paramètres sont nécessaires afin de créer un objet `Coordinates` :

- La valeur en abscisse,
- La valeur en ordonnée.

Ainsi, pour récupérer les coordonnées d'un noeud, nous n'aurons qu'à utiliser les commandes suivantes : `node.coords.x` et `node.coords.y`.

Network.py

La classe `Network` est sans doute la plus complexe. C'est elle qui contient la logique permettant de générer un graphe aléatoire, d'afficher l'état de ce graphe à un instant ou encore de déterminer le chemin le plus court entre deux points.

Comme nous avons pu le voir dans les parties précédentes, cette classes nécessite un certain nombre de d'attributs :

- Le nombre de noeuds à un instant donné,
- Le nombre de noeuds lors de la création du graphe,
- La liste des noeuds du graphe
- Le seuil de distance au delà duquel un message ne peut plus transiter entre deux noeuds,
- L'index du noeud de départ,
- L'index du noeud d'arrivé,
- La liste des indexes des noeuds supprimés,
- La liste des emplacements dans la liste des noeuds des noeuds supprimés.

Pour ce qui est des méthodes, nous nous focaliserons principalement sur les plus importantes :

- Une première méthode `_generate_random_graph` qui permet de générer un graphe aléatoire en fonction du nombre de noeuds souhaité. Pour chaque noeuds que nous créons, nous allons aléatoirement le placer sur une grille puis le lier, également de manière aléatoire, à d'autres noeuds déjà présents dans le graphe.
- Une autre méthode (`remove_random_node`) permet de supprimer de manière aléatoire un noeud du graphe afin de simuler l'inaccessibilité d'un noeud. Le noeud supprimé sera, bien évidemment, différent de ceux de départ et d'arrivé.
- Nous retrouvons ensuite la méthode la plus importante : `custom_dijkstra`. Cette méthode permet de déterminer le chemin le plus court entre les deux noeuds de départ et d'arrivé. Nous appliquons donc l'algorithme de Dijkstra puis nous retournons le chemin ainsi que la distance des points de départ et d'arrivé en passant par le chemin sélectionné.
- `is_path_reachable` est la méthode qui permet de déterminer si un chemin est valide ou non.
- Enfin, la méthode `send_message` permet d'incrémenter le score des noeuds contenus dans le chemin que l'on aura déterminé à l'aide de l'algorithme de Dijkstra.

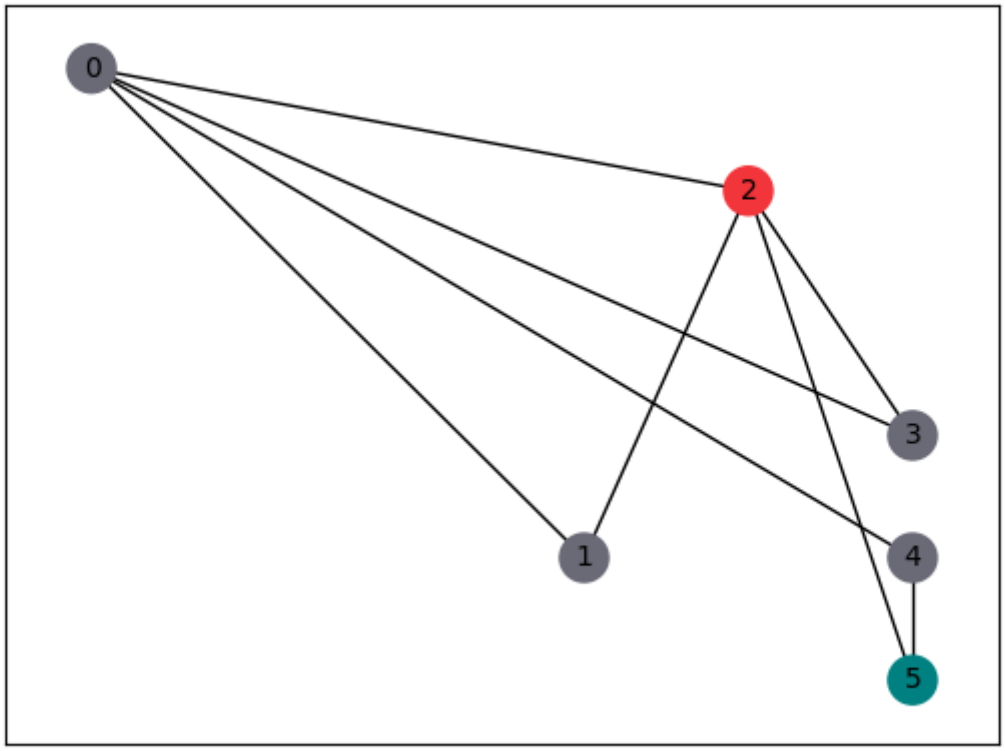
Nous avons également créé plusieurs méthodes utilitaires :

- `pretty_print` qui sert à afficher l'état courant du graphe avec les noeuds présents, les voisins, les noeuds de départ et d'arrivé. Pour un graphe de 3 noeuds, la méthode pourrait retourner quelque comme cela :

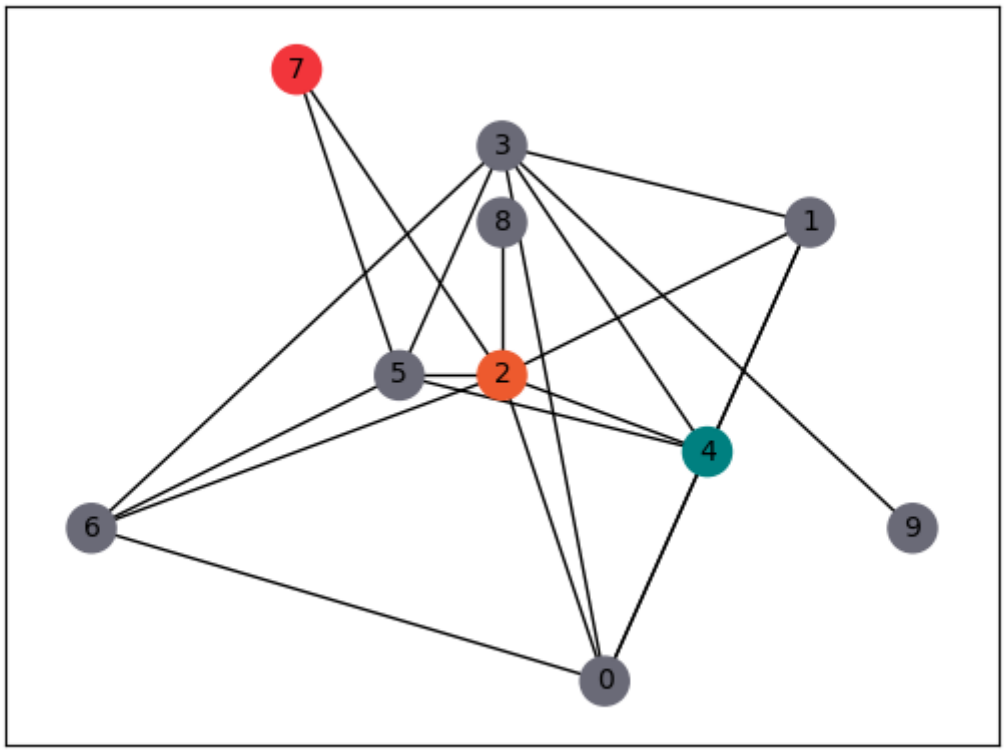

```
{
   "nodes": {
      "0": {
         "neighbors": {
            "1": 1.0,
            "2": 2.0
         },
         "coords": {
            "x": 2,
            "y": 0
         },
         "score": 0
      },
      "1": {
         "neighbors": {
            "0": 1.0
         },
         "coords": {
            "x": 1,
            "y": 0
         },
         "score": 0
      },
      "2": {
         "neighbors": {
```

```
            "0": 2.0
         },
         "coords": {
            "x": 0,
            "y": 0
         },
         "score": 0
      }
   },
   "start": 2,
   "end": 0
}
```


- Enfin, la méthode `plot_print` permet d'afficher une représentation graphique du graphe grâce aux librairies `networkx` et `matplotlib`. Le but étant d'avoir une vision précise du graphe que l'on va devoir manipuler par la suite. Par exemple, pour un graphe de 6 noeuds, la méthode pourrait afficher le graphe suivant.



Dans le graphe ci-dessus, le noeud bleu représente le noeud de départ tandis que le noeud rouge représente celui d'arrivé. Nous avons également fait en sorte de pouvoir colorer différement les noeuds contenus dans le chemin. En voici un exemple pour un graphe de 10 noeuds.



Également, la position des noeuds du graphe dans la fenêtre correspond bien aux coordonnées que nous avons renseignés plus tôt lors de la création des noeuds du graphe.

Installation et exécution

Prérequis

- Python (3.9 idéalement)
- Pip

Installation

⚠ Ce projet, pour fonctionner nécessite l'installation de plusieurs librairies : `numpy`, `networkx` et `matplotlib`. Deux options sont possibles : les installer globalemet sur l'ordinateur ou bien les installer dans un environnement virtuel. L'intérêt de l'environnement virtuel est que les librairies ne sont pas installées globalement sur l'ordinateur.

Si vous souhaitez installer les librairies globalement sur l'ordinateur, il suffit de taper les commandes suivantes :

```
pip install numpy
pip install networkx
pip install matplotlib
```

Il est aussi possible de les installer via le fichier `requirements.txt` avec la commande : `pip install -r requirements.txt`. Cette dernière commande installera automatiquement les librairies nécessaires

Dans le cas ou vous voulez installer les librairies dans un environnement virtuel, il va tout d'abord falloir installer la librairie permettant de créer des environnements virtuels : `virtualenv` avec la commande suivante : `pip install virtualenv`. Ensuite, il ne reste plus qu'à suivre les étapes ci-dessous.

- Création de l'environment virtuel

```
python -m venv venv
```

- Activer l'environment virtuel

```
venv/Scripts/activate      # Windows
. ./venv/Scripts/activate  # Linux
```

- Installation des dépendances

```
pip install -r requirements.txt
```

ou bien

```
pip install numpy
pip install networkx
pip install matplotlib
```

Et voila !

Il ne reste plus qu'à lancer le fichier `main.py` ou `main_single_message.py`.

References

- Networkx official documentation
- Cours 8INF840-01 - Structure de données (Cours sur les graphes de Djamal Rebaine)