TD 2 : graphes et algorithme de Dijkstra

Les sujets de TD ne sont ni relevés, ni notés. Ils sont volontairement conçus pour ne pas pouvoir être traités en une seule séance ; n'hésitez cependant pas, si vous le souhaitez, à tenter de les finir chez vous.

Dans ce TP, nous allons découvrir et utiliser des graphes, ainsi qu'un algorithme très commun : l'algorithme de Dijkstra.

Rappel : respectez les conventions de style (PEP 8 et *Zen of Python*), écrivez des tests, documentez votre code.

Qu'est-ce qu'un graphe ?

En informatique, l'on entend par *graphe* un ensemble d'éléments reliés entre eux. Il existe un grand nombre de définitions alternatives ; pour la suite nous considérerons des graphes *finis*, *simples*, *non-orientés* et *pondérés* :

non-orientés les traits reliant deux éléments ne portent pas de flèche. C'est-à-dire que relier un élément x à un élément distinct y signifie la même chose que relier deux éléments y à x. Les traits sont appelés *arêtes*.

simples pour deux éléments x et y, il existe au plus une arête de x vers y. Par ailleurs, il n'existe pas d'arc de x vers x lui-même.

pondérés chaque arête se voit attaché un nombre réel positif, que l'on appelle son *poids* ou parfois son *coût*

finis il n'y a qu'un nombre fini d'éléments.

Formellement, un graphe Γ est donc un couple (V, E) où V est un ensemble fini de *sommets* ou *nœud* (*vertices* en anglais), et E d'*arêtes* (*edges*), E étant composé de couples de la forme $(\{x, y\}, \alpha)$ avec x et y des éléments de V et $\alpha \in \mathbf{R}_+$.

Et moins formellement, un graphe est un dessin qui peut ressembler au suivant :

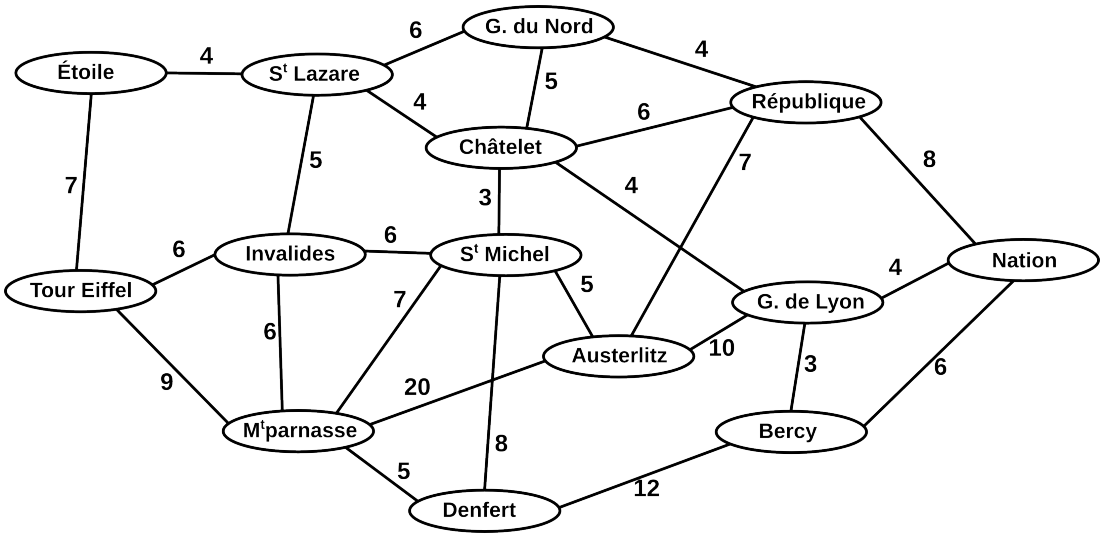

FIGURE 1 – Quelques lieux notables de Paris et le temps, en minutes, nécessaire pour les relier par les transports publics — grèves et colis suspects exclus.

 Remarques et corrections bienvenues à l'adresse <valentin@melot.tf>.

1. Démontrer que dans un tel graphe, il n'y a qu'un nombre fini d'arêtes. Majorer le nombre d'arcs en fonction du nombre de nœuds.

Représenter un graphe en mémoire

Avant de pouvoir travailler avec des graphes, il est nécessaire de comprendre comment les représenter en mémoire, d'identifier les structures de données à utiliser.

L'ensemble V étant fini, il peut être mis en bijection avec un ensemble de la forme $[\![1, n]\!]$. En d'autres termes, on choisit une numérotation pour les sommets, ce qui permet d'accéder aux éléments de V par une liste ou un dictionnaire. Dans l'exemple de la figure 1, l'on aurait par exemple :

```
V = ["Étoile", "Saint-Lazare", "Gare␣du␣nord", "Châtelet", "République",
     "Tour␣Eiffel", "Invalides", "Saint-Michel", "Austerlitz", "Gare␣de␣Lyon",
     "Nation", "Montparnasse", "Denfert-Rochereau", "Bercy"]
```

Pour représenter les arêtes, trois choix existent :

— Sous la forme d'une liste de triplets `(x, y, a)` avec `x` et `y` les extrémités de l'arête et `a` son étiquette. Pour la figure 1, le début de la liste des arêtes serait :

```
E = [(0, 1, 4), (1, 2, 6), (1, 3, 4), (2, 3, 5), (2, 4, 4), ...]
```

Lire : le poids de l'arête reliant les éléments 0 à 1 est 4 (il faut quatre minutes pour aller de l'Étoile à Saint-Lazare par le RER A), le poids de l'arête reliant 1 à 2 est 6 (il faut six minutes pour aller de Saint-Lazare à la gare du nord par le RER E), etc. Cette représentation est la *liste des arêtes*.

— L'on peut également représenter les arêtes de notre graphe sous la forme d'une matrice contenant les poids des différentes arêtes, par exemple :

$$\begin{pmatrix}
\cdot & 4 & \cdot & \cdot & \cdot & 7 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
4 & \cdot & 6 & 4 & \cdot & \cdot & 5 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & 6 & \cdot & 5 & 4 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & 4 & 5 & \cdot & 6 & \cdot & \cdot & 3 & \cdot & 4 & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & 4 & 6 & \cdot & \cdot & \cdot & \cdot & 7 & \cdot & 8 & \cdot & \cdot & \cdot \\
7 & \cdot & \cdot & \cdot & \cdot & \cdot & 6 & \cdot & \cdot & \cdot & \cdot & 9 & \cdot & \cdot \\
\cdot & 5 & \cdot & \cdot & \cdot & 6 & \cdot & 6 & \cdot & \cdot & \cdot & 6 & \cdot & \cdot \\
\cdot & \cdot & \cdot & 3 & \cdot & \cdot & 6 & \cdot & 5 & \cdot & \cdot & 7 & 8 & \cdot \\
\cdot & \cdot & \cdot & \cdot & 7 & \cdot & \cdot & 5 & \cdot & 10 & \cdot & 20 & \cdot & \cdot \\
\cdot & \cdot & \cdot & 4 & \cdot & \cdot & \cdot & \cdot & 10 & \cdot & 4 & \cdot & \cdot & 3 \\
\cdot & \cdot & \cdot & \cdot & 8 & \cdot & \cdot & \cdot & \cdot & 4 & \cdot & \cdot & \cdot & 6 \\
\cdot & \cdot & \cdot & \cdot & \cdot & 9 & 6 & 7 & 20 & \cdot & \cdot & \cdot & 5 & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 8 & \cdot & \cdot & \cdot & 5 & \cdot & 12 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 3 & 6 & \cdot & 12 & \cdot
\end{pmatrix}$$

À noter qu'il a fallu utiliser une valeur spéciale pour représenter l'absence d'arête. En Python, l'on peut représenter cette matrice par une simple liste bidimensionnelle en utilisant `None` pour marquer l'absence d'arête :

```
E = [[None, 4, None, None, None, 6, None, None,
      None, None, None, None, None, None],
     [4, None, 6, 4, None, None, 5, None, None,
      None, None, None, None, None],
     ...
    ]
```

À noter que, puisque le graphe est non-orienté, la matrice est symétrique. De plus, la diagonale ne contient que des `None` en raison de la simplicité.

L'on dit que E est représenté par sa *matrice d'adjacence*.

— Enfin, l'on peut définir une liste de liste de couples. La x^{e} liste de couples contient des triplets (y, a) où y représente un nœud relié à x et a le poids de l'arête. Toujours dans notre exemple, l'on obtiendrait :

```
E = [[(1, 4), (5, 7)],
     [(0, 4), (2, 6), (3, 4), (6, 5)],
     [(1, 6), (3, 5), (4, 3)],
     ...
    ]
```

Ici, E est représenté sous la forme d'une *liste d'adjacence*.

2. Quels sont les avantages et inconvénients de ces trois modes de représentation des arêtes ? L'on s'intéressera en particulier à la quantité de mémoire utilisée (qui peut dépendre des caractéristiques du graphe), et à la vitesse à laquelle peuvent être faites certaines opérations telles que la récupération de l'ensemble des nœuds connectés à un nœud donné.

   En réalité, l'on n'utilise quasiment jamais la liste des arêtes.

3. Écrire une fonction `mat2list` transformant une matrice d'adjacence en liste d'adjacence.

4. Écrire une fonction `list2mat` transformant une liste d'adjacence en matrice d'adjacence. L'on soulèvera une exception si la liste d'adjacence ne correspond pas à un graphe simple.

Calcul d'un chemin de poids minimal : l'algorithme de Dijkstra

Nota : Edsger Dijkstra était hollandais, son nom de famille se prononce donc ['dɛɪkstra].

Sur la figure 1, puisque les arêtes représentent des liaisons directes entre deux lieux (par le métro, le RER ou en autobus), une famille d'arêtes peut définir un chemin : un chemin reliant v_d à v_a est une famille d'arêtes $(e_1, \dots, e_n)$ telles que e_1 ait pour extrémité v_d, e_n ait pour extrémité v_a, et telle que pour tout $i \in [\![1, n-1]\!]$, e_i et e_{i+1} aient un sommet en commun.

Pour un tel chemin $(e_0, \dots, e_n)$, le *poids* (ou le *coût*) du chemin est la somme des coûts des arêtes par lesquelles il passe. Attention, même si les poids des arêtes représentent des distances ou des durées, on ne parle pas de *longueur* du chemin : celle-ci désigne simplement le nombre d'arêtes constituant le chemin (ici, n).

Un problème naturel revient donc à chercher, pour deux sommets fixés, un chemin de poids minimal. Dans notre exemple, un tel chemin désigne l'itinéraire le plus rapide pour relier deux stations. Selon ce que représente le graphe, il peut également s'agir d'un itinéraire le plus court, ou le moins coûteux, etc. Il s'agit en fait du problème que résolvent les programmes de cartographie lorsqu'on leur demande un itinéraire !

L'idée de base de l'algorithme est d'essayer de *parcourir*, les sommets du graphe à partir d'un sommet de départ, en explorant d'abord le plus proche, puis le deuxième plus proche, puis le troisième plus proche, etc. jusqu'à tomber sur le sommet d'arrivée souhaité. Au fur et à mesure que les sommets sont visités, des chemins de poids minimaux sont construits entre le point de départ et chaque sommet.

Tentons de comprendre le fonctionnement de l'algorithme sur le graphe de la figure 1, avec comme point de départ l'Étoile et comme arrivée Montparnasse.

0. L'on part de l'Étoile. Ses deux voisins immédiats sont Saint-Lazare, à quatre minutes, et la tour Eiffel, à sept minutes. Les sommets les plus proches de l'Étoile sont pour l'instant, dans l'ordre : Étoile (0, visité), Saint-Lazare (4) et tour Eiffel (7).

1. Depuis Saint-Lazare, on atteint la gare du Nord en six minutes, Châtelet en quatre, les Invalides en cinq et l'Étoile en quatre. Cela nous donne donc des chemins depuis l'Étoile vers la gare du Nord en dix minutes, Châtelet en huit minutes, les Invalides en neuf minutes et l'Étoile en huit. Ce dernier chemin vers l'Étoile est inintéressant : il est plus long que le chemin déjà calculé, qui consistait à ne pas bouger ! À ce moment, les sommets les plus proches de l'Étoile sont : Étoile (0, visité), Saint-Lazare (4, visité), tour Eiffel (6), Châtelet (8), Invalides (9) et gare du Nord (10).
2. On regarde maintenant ce qui se passe depuis la tour Eiffel. L'on peut atteindre l'Étoile, les Invalides et Montparnasse. Depuis l'Étoile, cela donne des chemins en quatorze minutes pour revenir à l'Étoile (donc inintéressant), treize minutes pour les Invalides (donc inintéressant, puisque l'on avait déjà un chemin en neuf minutes !), et seize minutes pour Montparnasse. La liste des chemins les plus proches de l'Étoile est donc : Étoile (0, visité), Saint-Lazare (4, visité), tour Eiffel (7, visité), Châtelet (8), Invalides (9), gare du Nord (10), Montparnasse (16). Mais avons-nous pour autant trouvé le chemin le plus rapide vers Montparnasse ?
3. Le sommet suivant à explorer est Châtelet ; les sommets les plus proches de l'Étoile sont donc : Étoile (0, visité), Saint-Lazare (4, visité), tour Eiffel (7, visité), Châtelet (8, visité), Invalides (9), gare du Nord (10), Saint-Michel (10), gare de Lyon (12), Montparnasse (16), République (16).
4. Puis les Invalides : depuis celles-ci, l'on trouve des chemins vers la tour Eiffel (à quinze minutes, donc inintéressant), Saint-Lazare (quatorze minutes, donc inintéressant), Saint-Michel (quinze minutes, donc inintéressant), et Montparnasse… en quinze minutes. Ce dernier temps est plus intéressant que celui du chemin précédemment calculé *via* la tour Eiffel, qui prenait seize minutes. Les temps mis à jour sont donc : Étoile (0, visité), Saint-Lazare (4, visité), tour Eiffel (7, visité), Châtelet (8, visité), Invalides (9, visité), gare du Nord (10), Saint-Michel (10), gare de Lyon (12), Montparnasse (15), République (16).
5. Et l'on continue jusqu'à avoir parcouru tout le graphe. Finalement, Montparnasse ne peut pas être atteint depuis l'Étoile en moins de quinze minutes, *via* Saint-Lazare [1].

5. L'on se donne deux listes de même longueur n, l'une appelée `visited` contenant des booléens (`True` ou `False`), et l'autre `costs` contenant des nombres positifs et la valeur spéciale `None`. Écrire une fonction `find_min` renvoyant (s'il en existe) un indice `i` tel que `costs[i]` soit minimal parmi l'ensemble des `j` vérifiant `visited[j] == True`.
6. En s'inspirant des explications précédentes, écrire une fonction `min_cost`, prenant comme paramètre une liste d'adjacence, un sommet de départ et un sommet d'arrivée, et renvoie le poids minimal d'un chemin entre les deux. L'on pourra travailler avec deux listes : l'une contient des booléens indiquant si un sommet a déjà été visité ou non, l'autre contient des entiers indiquant les plus courtes durées calculées depuis le point de départ. Ne pas hésiter à afficher à l'écran un maximum d'informations (par exemple la valeur des deux listes à chaque tour de boucle ou le sommet choisi).
7. Modifier la fonction `min_cost` en une fonction `dijkstra` prenant les mêmes paramètres, et renvoyant un chemin de poids minimal entre le sommet de départ et le sommet d'arrivée. Un tel chemin sera représenté sous la forme d'une liste ordonnée de sommets.

Preuve de l'algorithme

Il est vital de se poser une question lorsque l'on implémente un algorithme manipulant des structures de données un peu complexes en informatique : pourquoi cela fonctionne-t-il ? Dans le cas présent, cela est d'autant plus important que le jury du concours Agro-Véto constate dans son rapport de 2016 [2] qu'à peu près aucun candidat n'est capable de donner de réponse.

1. En fait, il serait assez idiot de passer par Saint-Lazare pour faire ce trajet, la correspondance entre le RER A et la ligne 13 est horriblement longue, alors que le trajet *via* la tour Eiffel se fait sans quitter la ligne 6. La raison est simple : nous avons négligé les correspondances.
2. Accessible à l'adresse `https://www.concours-agro-veto.net/IMG/pdf_2016_maths_info.pdf`. Voir section 4.4, page 6.

Dans tous les algorithmes faisant intervenir une boucle ou presque, l'argument repose sur un *invariant de boucle*, c'est-à-dire le constat qu'une certaine propriété concernant les données en mémoire est vérifiée à la fin de chaque tour de boucle.

8. Dans notre liste `min_costs`, les coûts minimaux associés à chaque sommet évoluent à chaque tour de boucle. Mais que peut-on dire du coût minimal d'un sommet marqué comme exploré ? Partant de là, énoncer un *invariant de boucle*, dont découle directement le fait que l'algorithme produit le résultat attendu.
9. Comment peut-on prouver que cet invariant de boucle est vrai ?

Complexité de l'algorithme

10. Montrer que la quantité de mémoire consommée par l'algorithme est dominée (très grossièrement !) par n^2, où n est le nombre de sommets du graphe.

    L'on s'intéresse maintenant à la complexité temporelle de l'algorithme.
11. Est-il vraiment nécessaire de traiter tous les sommets ? Modifier la fonction pour éviter des tours de boucle inutiles.
12. Quelle est la complexité temporelle de la fonction `find_min` dans le pire des cas ? En exploitant la question 1, en déduire que la complexité temporelle de cette implémentation de l'algorithme de Dijkstra est en $O(n^2)$ dans le pire des cas.

    En fait, il est possible d'améliorer grandement notre procédure de détermination du sommet suivant à explorer, en gérant mieux la liste `min_costs`. Cela permet de réduire la complexité dans le pire des cas à $O(a + n \log n)$, où a représente le nombre d'arêtes du graphe.

D'autres façon de parcourir un graphe

Les algorithmes dans lesquelles il est nécessaire de visiter les sommets d'un graphe sont nombreux. À chaque fois, l'ordre de parcours des sommets est déterminant. Deux façons d'ordonner les sommets en partant d'un sommet de départ sont très fréquentes :

Parcours en profondeur Il consiste à s'éloigner le plus possible du sommet de départ puis, une fois que ce n'est plus possible. Partant du sommet initial (sommet 0), l'on visite un premier voisin (sommet 1), puis un voisin de 1 (sommet 2), etc. en ne repassant jamais sur un sommet déjà visité. Fatalement, vient un moment où l'on arrive à un sommet n dont tous les voisins ont déjà été visités. L'on *remonte* alors : si le sommet $n-1$ a un voisin non encore exploré, l'on repart sur celui-ci, sinon, l'on regarde $n-2$, etc.

Parcours en largeur Il consiste au contraire à regarder d'abord tous les voisins du sommet de départ (distance 1), puis les voisins des voisins (distance 2), puis les voisins des voisins des voisins (distance 3), etc.

13. Implémenter deux fonctions `parcours_largeur` et `parcours_profondeur`. L'on devra dans tous les cas maintenir un tableau de booléens `visited` pour ne pas repasser deux fois au même endroit. Pour faciliter l'implémentation de ces fonctions, l'on pourra (sans obligation) utiliser l'une des astuces suivantes (toutes hors-programme) :
    - Maintenir une *file de priorité*, à l'aide des fonctions `L.append(x)` (qui rajoute un élément `x` à la fin d'une liste) et `L.pop(0)` (qui renvoie le premier élément de `L` et le supprime de `L`). Une telle file fonctionne sur le principe du « premier entré, premier sorti ».
    - Maintenir une *pile d'assiettes*, à l'aide des fonctions `L.append(x)` et `L.pop()` (qui renvoie le dernier élément de `L` et le supprime de `L`). Une telle pile fonctionne sur le principe du « premier entré, dernier sorti ».

 Remarques et corrections bienvenues à l'adresse <valentin@melot.tf>.

— Faire appel à une fonction récursive, c'est-à-dire une fonction qui, au cours de son exécution, se rappelle elle-même avec des paramètres différents.

14. Imaginons que le graphe ne soit pas connexe, c'est-à-dire que l'on puisse trouver deux points qui ne soient reliés par aucun chemin. C'est le cas, par exemple, si l'on regarde le graphe des stations du métro de Paris *et* de Toulouse : aucun chemin ne permet de relier Châtelet – Les Halles à une station quelconque du métro de Toulouse. Que se passe-t-il au cours de l'exécution des fonctions `parcours_largeur` et `parcours_profondeur` ? Comment nous assurer que nous explorerons bien tous les nœuds malgré tout ?
15. Écrire une fonction indiquant si un graphe est connexe.

    Un graphe est dit *acyclique* s'il ne contient pas de boucle ou plus formellement, s'il n'existe pas de chemin de la forme $(e_1, \dots, e_n)$ tel que en notant x_{i-1} et x_i les extrémités de l'arête e_i pour tout $i \in [\![1, n]\!]$, on ait $x_0 = x_n$ et $x_0, x_1, \dots, x_n$ distincts deux à deux.
16. Écrire une fonction déterminant si un graphe est acyclique.