From Maths et Python

TP: Algorithmes De Plus Court Chemin

Sur cette page... (hide)

1. Introduction

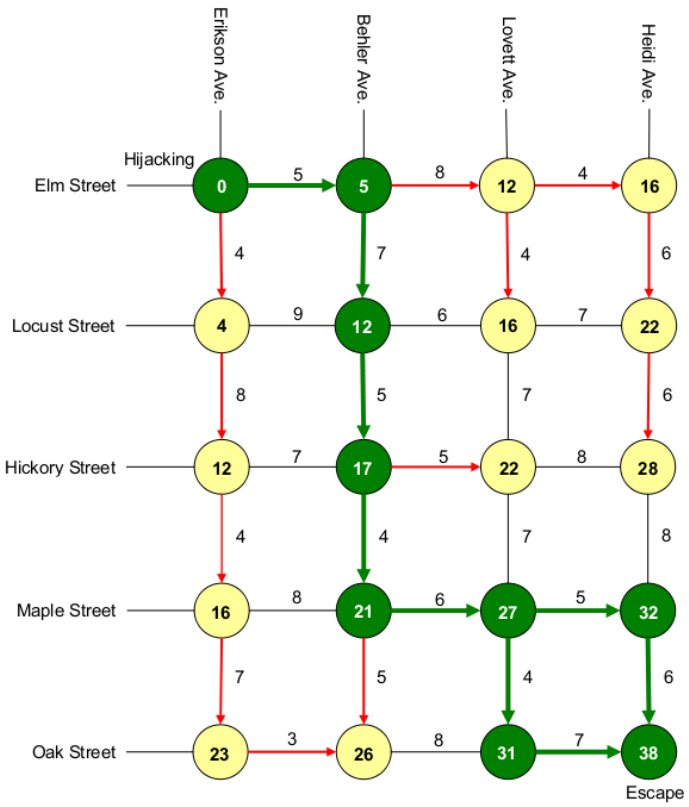
Dans l'épisode 23 de la saison 3 de **Numb3rs**, intitulé *Money for Nothing*, un camion contenant de l'argent et des médicaments a été attaqué.

Comme les voleurs souhaitent quitter la ville (Los Angeles) le plus rapidement possible, et qu'ils ont prémédité leur coup, le héros (Charlie) essaye de déterminer le plus rapide des chemins possibles, reliant le lieu de l'embuscade à la porte de sortie de la ville la plus proche.

L'embuscade a eu lieu dans le noeud (croisement) supérieur gauche de l'image ci-dessous, et la sortie est le noeur inférieur droit ;

- une flèche représente une rue à sens unique,
- un arc représente une rue empruntable dans les deux sens.

Les valeurs sur les arcs/flèches représentent les temps nécessaires pour rejoindre chaque croisement.



Tester toutes les solutions devient vite inextricable. L'idée est alors d'utiliser un algorithme de recherche du plus court chemin : Dijkstra.

Il y a deux solutions possibles, elles sont indiquées sur le schéma en vert.

2. Le module networkx

2.1 Présentation du module

Networks est un module python permettant de manipuler les graphes, qu'ils soient orientés ou non. Les algorithmes classiques sur les graphes sont implantés.

2.2 Installation

On récupère ce module ici : **https://networkx.lanl.gov/download/networkx/networkx-0.37.tar.gz**. On le décompresse :

```
tar zxvf networkx-0.37.tar.gz
```

et on copie le répertoire **networkx-0.37/networkx** dans son répertoire de travail python :

```
cp -R networkx-0.37/networkx ~/python
```

Reste à aller dans le répertoire python, et lancer python :

```
cd ~/python
python
```

2.3 Vers une résolution du problème

On importe le module **networks**, et on crée un graphe orienté :

```
>>> from networkx import *
>>> G=XDiGraph()
```

On commence par saisir les 20 noeuds dans une liste. Par exemple, le noeud correspondant au croisement des rues *Maple Street* et *Behler Ave.*, numéroté 21 sur le dessin, correspond à l'élément (1,3) dans notre liste (le point (0,0) est le noeud supérieur gauche, lieu de l'attaque).

```
>>> L=[]
>>> for x in range(4):
...     for y in range(5):
...         L.append((x,y))
...
```

On ajoute cette liste de noeuds à notre digraphe :

```
>>> G.add_nodes_from(L)
```

Il faut maintenant mettre les poids sur les flèches, c'est-à-dire mettre le temps nécessaire à rejoindre deux croisements adjacents. C'est un peu long, mais bon...

```
>>> G.add_edge((0,0),(1,0),5)
>>> G.add_edge((1,0),(2,0),8)
>>> G.add_edge((2,0),(3,0),4)
>>> G.add_edge((0,0),(0,1),4)
>>> G.add_edge((1,0),(1,1),7)
>>> G.add_edge((2,0),(2,1),4)
>>> G.add_edge((3,0),(3,1),6)
>>> G.add_edge((0,1),(1,1),9)
>>> G.add_edge((1,1),(2,1),6)
>>> G.add_edge((2,1),(3,1),7)
>>> G.add_edge((0,1),(0,2),8)
>>> G.add_edge((1,1),(1,2),5)
>>> G.add_edge((2,1),(2,2),7)
>>> G.add_edge((3,1),(3,2),6)
>>> G.add_edge((0,2),(1,2),7)
>>> G.add_edge((1,2),(2,2),5)
>>> G.add_edge((2,2),(3,2),8)
>>> G.add_edge((0,2),(0,3),4)
>>> G.add_edge((1,2),(1,3),4)
>>> G.add_edge((2,2),(2,3),7)
>>> G.add_edge((3,2),(3,3),8)
>>> G.add_edge((0,3),(1,3),8)
>>> G.add_edge((1,3),(2,3),6)
>>> G.add_edge((2,3),(3,3),5)
>>> G.add_edge((0,3),(0,4),7)
>>> G.add_edge((1,3),(1,4),5)
>>> G.add_edge((2,3),(2,4),4)
>>> G.add_edge((3,3),(3,4),6)
>>> G.add_edge((0,4),(1,4),3)
>>> G.add_edge((1,4),(2,4),8)
>>> G.add_edge((2,4),(3,4),7)
```

Astuce : Ne recopiez pas ce qui précède, ligne par ligne. Faites un copier-coller dans un éditeur, et remplacez tous les **>>>** par rien du tout.

2.4 Travaux pratiques

On a, dans ce qui précède, supposé que toutes les rues étaient à sens unique. Rajoutez des arcs orientés au graphe **G** pour coller exactement au problème de Charlie.

Il reste maintenant à réfléchir à un algorithme trouvant le plus court chemin. Une étude systématique de tous les chemins possibles n'étant évidemment pas envisageable, on va s'orienter vers des algorithmes évolués de recherche de plus court chemin.

3. L'algorithme de Floyd

3.1 Présentation

L'algorithme de Floyd est un premier algorithme, naïf, permettant de répondre au problème de la recherche de plus court chemin. Il est simple, mais coûteux.

3.2 Remarques préliminaires

Il faut d'abord qu'un plus court chemin existe, donc supposer les graphes :
- connexes, (il existe bien un chemin entre les sommets considérés)
- à poids positifs (sans cela, l'existence d'un chemin minimal n'est pas assuré).

3.3 Description de l'algorithme

Fondements

Il y a, à la base de l'algorithme, les remarques suivantes :

- si $(a_0, ..., a_i, ..., a_p)$ est un plus court chemin de a_0 à a_p, alors :
  - $(a_0, ..., a_i)$ est un plus court chemin de a_0 à a_i,
  - $(a_i, ..., a_p)$ est un plus court chemin de a_i à a_p.

- Les poids des arêtes étant positives, tout chemin contenant un cycle est nécessairement plus long que le même chemin sans le cycle. On peut, de ce fait, se limiter à la recherche de plus courts chemins passant par des sommets deux à deux distincts.

Matrice M des distances

La suite de matrices $M^{(k)}$ est initialisée par la matrice $M^{(0)}$, de taille $n \times n$ (où n est le nombre de sommets du graphe), telle que
- $M^{(0)}_{i,j} = k$ s'il existe un arc de poids k entre i et j,
- $M^{(0)}_{i,j} = +\infty$ sinon.

Il suffit ensuite de calculer la suite de matrices $M^{(k)}$, $k \geq sqlant1$ définies par :

$$M^{(k)}_{i,j} = min\left(M^{(k-1)}_{i,j}, M^{(k-1)}_{i,k} + M^{(k-1)}_{k,j}\right)$$

C'est-à-dire...

- $M^{(1)}$ vaut :

$$\begin{pmatrix} \min\left(M^{(0)}_{1,1}, M^{(0)}_{1,1} + M^{(0)}_{1,1}\right) & \min\left(M^{(0)}_{1,2}, M^{(0)}_{1,1} + M^{(0)}_{1,2}\right) & \dots & \min\left(M^{(0)}_{1,n}, M^{(0)}_{1,1} + M^{(0)}_{1,n}\right) \\ \min\left(M^{(0)}_{2,1}, M^{(0)}_{2,1} + M^{(0)}_{1,1}\right) & \min\left(M^{(0)}_{2,2}, M^{(0)}_{2,1} + M^{(0)}_{1,2}\right) & \dots & \min\left(M^{(0)}_{2,n}, M^{(0)}_{2,1} + M^{(0)}_{1,n}\right) \\ \min\left(M^{(0)}_{3,1}, M^{(0)}_{3,1} + M^{(0)}_{1,1}\right) & \min\left(M^{(0)}_{3,2}, M^{(0)}_{3,1} + M^{(0)}_{1,2}\right) & \dots & \min\left(M^{(0)}_{3,n}, M^{(0)}_{3,1} + M^{(0)}_{1,n}\right) \\ \vdots & \vdots & \dots & \vdots \\ \min\left(M^{(0)}_{n,1}, M^{(0)}_{n,1} + M^{(0)}_{1,1}\right) & \min\left(M^{(0)}_{n,2}, M^{(0)}_{n,1} + M^{(0)}_{1,2}\right) & \dots & \min\left(M^{(0)}_{n,n}, M^{(0)}_{n,1} + M^{(0)}_{1,n}\right) \end{pmatrix}$$

- $M^{(2)}$ vaut :

$$\begin{pmatrix} \min\left(M^{(1)}_{1,1}, M^{(1)}_{1,2} + M^{(1)}_{2,1}\right) & \min\left(M^{(1)}_{1,2}, M^{(1)}_{1,2} + M^{(1)}_{2,2}\right) & \dots & \min\left(M^{(1)}_{1,n}, M^{(1)}_{1,2} + M^{(1)}_{2,n}\right) \\ \min\left(M^{(1)}_{2,1}, M^{(1)}_{2,2} + M^{(1)}_{2,1}\right) & \min\left(M^{(1)}_{2,2}, M^{(1)}_{2,2} + M^{(1)}_{2,2}\right) & \dots & \min\left(M^{(1)}_{2,n}, M^{(1)}_{2,2} + M^{(1)}_{2,n}\right) \\ \min\left(M^{(1)}_{3,1}, M^{(1)}_{3,2} + M^{(1)}_{2,1}\right) & \min\left(M^{(1)}_{3,2}, M^{(1)}_{3,2} + M^{(1)}_{2,2}\right) & \dots & \min\left(M^{(1)}_{3,n}, M^{(1)}_{3,2} + M^{(1)}_{2,n}\right) \\ \vdots & \vdots & \dots & \vdots \\ \min\left(M^{(1)}_{n,1}, M^{(1)}_{n,2} + M^{(1)}_{2,1}\right) & \min\left(M^{(1)}_{n,2}, M^{(1)}_{n,2} + M^{(1)}_{2,2}\right) & \dots & \min\left(M^{(1)}_{n,n}, M^{(1)}_{n,2} + M^{(1)}_{2,n}\right) \end{pmatrix}$$

- Et ainsi de suite...

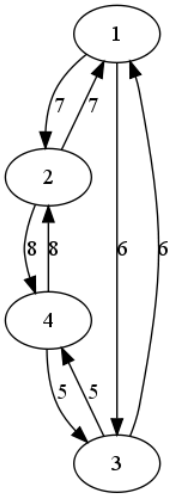
En effet,pour $k > 0$, $M_{i,j}^{(k)}$ est la longueur du plus court chemin entre i et j, en ne considérant que les sommets intermédiaires 0, 1, ..., k.

Ainsi, après n calculs, on obtient la matrice des longueurs des plus courts chemins.

Matrice P des poids

3.4 Exemple

On souhaite appliquer l'algorithme de Floyd-Warshall au graphe :



Initialisation

- On initialise notre suite de matrices par les distances contenues dans le graphe, en mettant ∞ s'il n'y a pas d'arêtes entre les sommets concernés :

$$M^{(0)} = \begin{pmatrix} 0 & 7 & 6 & \infty \\ 7 & 0 & \infty & 8 \\ 6 & \infty & 0 & 5 \\ \infty & 8 & 5 & 0 \end{pmatrix}$$

  Par exemple $M_{2,1}^{(0)} = 7$ puisqu'il existe une arête, de poids 7, allant de 2 à 1.

- La première matrice des prédécesseurs contient :
  - *None* s'il n'y a pas d'arêtes menant de i à j,
  - i si $i = j$,
  - i s'il existe une arête de i à $j \neq i$.

  Ce qui donne

$$P^{(0)} = \begin{pmatrix} 1 & 1 & 1 & None \\ 2 & 2 & None & 2 \\ 3 & None & 3 & 3 \\ None & 4 & 4 & 4 \end{pmatrix}$$

Deuxième étape

- Pour déterminer l'élément (i, j) de $M^{(1)}$, on prend la plus petite valeur entre $M_{i,j}^{(0)}$ et $M_{i,1}^{(0)}$ et $M_{1,j}^{(0)}$...

$$M^{(1)} = \begin{pmatrix} 0 & 7 & 6 & \infty \\ 7 & 0 & 13 & 8 \\ 6 & 13 & 0 & 5 \\ \infty & 8 & 5 & 0 \end{pmatrix}$$

  Par exemple, $M_{2,3}^{(1)} = Min\left(M_{2,3}^{(0)}; M_{2,1}^{(0)} + M_{1,3}^{(0)}\right) = Min(\infty; 7 + 6) = 13$

- À la base, $P^{(1)} = P^{(0)}$, et à chaque fois qu'un élément (i, j) a été modifié dans $M^{(1)}$, on met 1 en position (i, j) dans $P^{(1)}$.

En effet, modifier $M_{i,j}^{(1)}$ signifie que faire $i \to 1 \to j$ est plus court que $i \to j$.

Cela donne, au final :

$$P^{(1)} = \begin{pmatrix} 1 & 1 & 1 & None \\ 2 & 2 & 1 & 2 \\ 3 & 1 & 3 & 3 \\ None & 4 & 4 & 4 \end{pmatrix}$$

Troisième étape

- Pour déterminer l'élément (i, j) de $M^{(2)}$, on prend la plus petite valeur entre $M_{i,j}^{(1)}$ et $M_{i,2}^{(1)}$ et $M_{2,j}^{(1)}$.
  Cela revient en effet à se poser la question de savoir s'il est plus court d'aller de i à j directement, ou en s'autorisant d'emprunter &ventuellement des sommets parmi 1 et 2.

$$M^{(2)} = \begin{pmatrix} 0 & 7 & 6 & 15 \\ 7 & 0 & 13 & 8 \\ 6 & 13 & 0 & 5 \\ 15 & 8 & 5 & 0 \end{pmatrix}$$

  Par exemple, $M_{2,3}^{(2)} = Min\left(M_{2,3}^{(1)}; M_{2,2}^{(1)} + M_{2,3}^{(1)}\right) = Min(13; 0 + 13) = 13$

- On part de $P^{(2)} = P^{(1)}$, et à chaque fois qu'un élément (i, j) a été modifié dans $M^{(2)}$, on met 2 en position (i, j) dans $P^{(2)}$.
  Cela donne :

$$P^{(2)} = \begin{pmatrix} 1 & 1 & 1 & 2 \\ 2 & 2 & 1 & 2 \\ 3 & 1 & 3 & 3 \\ 2 & 4 & 4 & 4 \end{pmatrix}$$

Quatrième étape

- On poursuit l'algorithme : $M_{i,j}^{(3)} = Min(M_{i,3}^{(2)}; M_{3,j}^{(2)}$:

$$M^{(3)} = \begin{pmatrix} 0 & 7 & 6 & 11 \\ 7 & 0 & 13 & 8 \\ 6 & 13 & 0 & 5 \\ 11 & 8 & 5 & 0 \end{pmatrix}$$

- De même,

$$P^{(3)} = \begin{pmatrix} 1 & 1 & 1 & 3 \\ 2 & 2 & 1 & 2 \\ 3 & 1 & 3 & 3 \\ 3 & 4 & 4 & 4 \end{pmatrix}$$

Dernième étape

Il nous reste un dernier sommet à envisager : le 4. On ne trouve aucun changement...

$$M^{(4)} = \begin{pmatrix} 0 & 7 & 6 & 11 \\ 7 & 0 & 13 & 8 \\ 6 & 13 & 0 & 5 \\ 11 & 8 & 5 & 0 \end{pmatrix} [[\ ChristopheGuyeux]] P^{(4)} = \begin{pmatrix} 1 & 1 & 1 & 3 \\ 2 & 2 & 1 & 2 \\ 3 & 1 & 3 & 3 \\ 3 & 4 & 4 & 4 \end{pmatrix}$$

Exploitation des résultats

3.5 Evaluation de l'algorithme

1. Chaque matrice se calculant en $O(n^2)$, l'algorithme final est en $O(n^3)$ opérations (et $O(n^2)$ en espace). On verra que Dijkstra sera meilleur.

2. Cet algorithme fournit tous les poids des chemins les plus courts.

3. Cependant, pour déterminer ces chemins, il faut une autre suite de matrices, contenant en position (i, j) le sommet k par

lequel il faut passer dans un chemin le plus court de i à j : par appel récursif, on pourra reconstruire le chemin proprement dit (ce qui engendre un coût supplémentaire).

4. Enfin, cet algorithme est plus approprié pour les graphes denses.

3.6 Résolution du problème de Charlie

Le module **networkx** possède une méthode permettant d'appliquer l'algorithme de Floyd à notre graphe. Cette méthode s'appelle **path.floyd_warshall** :

```
>>> path.floyd_warshall(G)
({(0, 1): {(0, 1): 0, (1, 2): 14, (3, 2): 27, (0, 0): inf, (3, 3): 29, (3, 0): inf, (2, 4): 28, (3, 1): 22, (1, 4): 22, (0, 2): 8, (2, 0): inf, (1, 3): 18, (2,
```

Le retour semble assez compliqué. En fait, on retrouve les deux éléments renvoyés par l'algorithme de Floyd : la distance, et le chemin (un couple de dictionnaires)...

```
>>> (distance, chemin)=path.floyd_warshall(G)
>>> distance[(0,1)]
{(0, 1): 0, (1, 2): 14, (3, 2): 27, (0, 0): inf, (3, 3): 29, (3, 0): inf, (2, 4): 28, (3, 1): 22, (1, 4): 22, (0, 2): 8, (2, 0): inf, (1, 3): 18,
(2, 3): 24, (2, 1): 15, (2, 2): 19, (1, 0): inf, (0, 4): 19, (0, 3): 12, (3, 4): 35, (1, 1): 9}
```

On trouve les distances du sommet (0,1) à tous les autres sommets :
- à lui-même (0,1), la distance est nulle,
- au sommet (1,2), la distance vaut 14,
- *etc.*

On peut aussi préciser le sommet d'arrivée qui nous intéresse :

```
>>> distance[(0,1)][(1,1)]
9
```

On peut donc trouver le temps (distance) du plus court chemin entre notre lieu d'attaque, et la sortie de la ville :

```
>>> distance[(0,0)][(3,4)]
38
```

Il reste à obtenir ce chemin. On se souvient que, dans l'algorithme ci-dessus, la matrice des chemins doit être remontée récursivement. Ainsi,

```
>>> chemin[(0,0)][(3,4)]
(3, 3)
```

signifie que, quand on emprunte le plus court chemin entre le croisement (0,0) et le croisement (3,4), le dernier croisement visité est (3,3).

On peut recommencer pour trouver le plus court chemin entre (0,0) et (3,3) - on connaît la fin du parcours :

```
>>> chemin[(0,0)][(3,3)]
(2, 3)
```

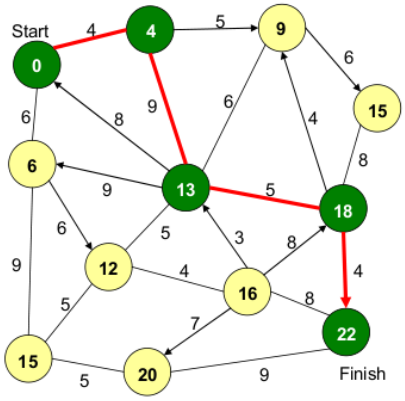
On continue ainsi, récursivement :

```
>>> chemin[(0,0)][(2,3)]
(1, 3)
>>> chemin[(0,0)][(1,3)]
(1, 2)
>>> chemin[(0,0)][(1,2)]
(1, 1)
>>> chemin[(0,0)][(1,1)]
(1, 0)
>>> chemin[(0,0)][(1,0)]
(0, 0)
```

On retrouve bien le chemin indiqué en vert. Connaissant le moment de l'attaque, on sait donc à quel carrefour se placer pour avoir le plus de chances de retrouver les agresseurs.

3.7 Travaux pratiques

1. Réalisez l'algorithme de Floyd. On utilisera **networkx** pour le graphe, et **numpy** pour les matrices.
2. L'appliquer à l'exemple précédent.
3. Faire de même avec :



4. Algorithme de Dijkstra

Présentation

Publié en 1959, l'algorithme de Edgser Wybe Dijkstra (1930-2002) est une alternative à Floyd, plus complexe mais plus rapide.

Ici, on se fixe un sommet source, et l'algorithme retourne tous les chemins les plus courts de ce sommet à chacun des autres sommets (légèrement différent du retour de Floyd)

Description de l'algorithme

4.1 Le principe

L'algorithme de Dijkstra est un algorithme de type glouton : à chaque nouvelle étape, on traite un nouveau sommet. Reste à définir le choix du sommet à traiter, et le traitement à lui infliger…

Tout au long du calcul, on met à jour deux ensembles :

C
: Ensemble des sommets restant à visiter à visiter (au départ $C = S \backslash \{source\}$).

D
: Ensemble des sommets pour lesquels on connaît déjà leur plus petite distance à la source (au départ, $D = \{source\}$).

L'algorithme se termine quand *C* est vide.

Pour chaque sommet *s* dans *D*, on conservera :
- dans un tableau *distances*, le poids du plus court chemin jusqu'à la source,
- dans un tableau *parcours*, le sommet *p* qui le précède dans un plus court chemin de la source à *s*.

Ainsi, pour retrouver un chemin le plus court, il suffira de remonter de prédécesseur en prédécesseur jusqu'à la source, ce qui pourra se faire grâce à un unique appel récursif (beaucoup moins coûteux que dans le cas de Floyd).

4.2 Initialisation

Au début de l'algorithme, le chemin le plus court connu entre la source et chacun des sommets est le chemin direct, avec une arête de poids infini s'il n'y a pas de liaison entre les deux sommets.

On initialise donc le tableau distances par les poids des arêtes reliant la source à chacun des sommets, et le tableau parcours par *source* pour tous les sommets.

4.3 $i^{ème}$ étape

On suppose avoir déjà traité *i* sommets, *parcours* et *distances* contiennent respectivement les poids et le prédécesseur des plus courts chemins pour chacun des sommets déjà traités.

Soit *s* le sommet de *C* réalisant le minimum de *distances[s]*.

- On supprime *s* de *C* et on l'ajoute à *D*.
- Reste à mettre à jour les tableaux *distances* et *parcours* pour les sommets *t* reliés directement à *s* par une arête, comme suit… Si

$$distances[s] + F(s,t) < distances[t],$$

alors on remplace :

- *distances[t]* par $distances[s] + F(s,t)$,

- *parcours[t]* par *s*.

Et c'est tout !

4.4 (n-2)$^{\text{ème}}$ étape

Au départ, il y a *(n-1)* sommets à visiter, mais la dernière étape est inutile (elle n'apporte rien). Dès la *(n-2)*$^{\text{ème}}$ étape, *distances* et *parcours* contiennent toute l'information nécessaire pour trouver des plus courts chemins.

Evaluation

Si n est le nombre de sommets du graphe, et a son nombre d'arêtes, alors l'algorithme de Dijkstra renvoie le résultat au bout de $O(a^2 lnn)$ opérations.

Pour un même résultat, Floyd nécessite $O(n^3)$ opérations.

Dijkstra et networkx

Montrons comment calculer le plus court chemin d'un graphe pondéré, et sa longueur (son temps), en utilisant la méthode de Dijkstra de **networkx** :

```
>>> path.dijkstra_path(G,(0,0),(3,4))
[(0, 0), (1, 0), (1, 1), (1, 2), (1, 3), (2, 3), (2, 4), (3, 4)]
>>> path.dijkstra_path_length(G,(0,0),(3,4))
38
```

Si la programmation de cet algorithme est un peu plus compliquée, son utilisation est plus simple.

Utilisation

Le protocole *open shortest path first*, qui permet un routage internet très efficace des informations, utilise Dijkstra.

Le réseau Internet utilise pour le moment un autre type d'algorithme pour déterminer le chemin à suivre pour transmettre des données entre deux serveurs : *Routing Information Protocol* (RIP). Dijkstra commence cependant à s'implanter, et remplace progressivement ce protocole : il est le plus rapide.

Les sites d'itinéraires routiers l'utilisent de la même manière et permettent de nombreuses adaptations en ajustant le poids des arcs (trajet le plus économique, le plus rapide, etc.)

Travaux pratiques

1. Reprogrammez cet algorithmes, en utilisant la structure de graphe propre à **networkx**.
2. L'appliquer aux deux exemples précédents, en comparant le temps d'exécution avec l'algorithme de **networkx** (on pourra se renseigner sur le module **timeit**).