TEA_Graphs

October 30, 2018

1 TEA 1 - Compte rendu:

Binôme: Amina KACIMI, Minh-Hoang DANG

Travail réalisé avec **Google Colabolatory**

Travail demandé : - Des salles de TP sont réservées pour un travail en autonomie. Votre présence dans ces salles n'est pas obligatoire. Par ailleurs un travail personnel supplémentaire devra être fourni. - Ce TEA est à réaliser en binôme, et à rendre sous Moodle au plus tard le 26 octobre 2018. Pour chacune des trois parties le rendu attendu est indiqué avec à chaque fois une version minimale et des versions plus complètes. Vous devez rendre : 1 rapport (partie 1) et deux fichiers python (parties 2 et 3) dans une archive. Pensez à indiquer les noms des deux participants.

1.1 1. Analyse de l'algorithme de Dijkstra

```
Entrée : un graphe pondéré G=(S,A,w) et un sommet origine x
Sortie : le poids du plus court chemin entre u et v pour chaque sommet v de
G
Initialiser un tableau T destimation des plus courts chemins
Pour chaque sommet v de G
    T[v] = inf
    T[x] = 0
Initialiser la liste L des sommets à traiter avec lensemble S
Tant que la liste L nest pas vide
    Extraire dans L le sommet u tel que T[u] est minimal
    Pour chaque sommet v voisin de u
        si (T(v) > T(u) + w(u,v))
        T(v) = T(u) + w(u,v)
Retourner T
```

Il s'agira de mener une étude de cet algorithme en répondant aux questions suivantes : + **Jeu d'essai** : Création d'un jeu d'essai (plusieurs graphes) représentatif de tous les cas possibles (graphe non connexe, arêtes doubles, multiplicité des plus courts chemins), et identification des valeurs d'estimation T à chaque étape de l'algorithme + **Complexité** : Etude de la complexité de cet algorithme, en justifiant le choix d'implémentation de la liste L des sommets restant à traiter (liste chaînée, table de hachage, arbre binaire de recherche...) + **Extension de l'algorithme** : proposer une extension de l'algorithme permettant de calculer le plus court chemin entre deux sommets du graphe et donner sa complexité.

Rendu minimal : un rapport (1 page maximum) contenant les dessins des graphes du jeu d'essai et l'algorithme version étendue.

Rendu v2 : idem avec des justifications sur les complexités des différentes structures de données (1 page maximum pour les complexités).

1.2 2. Implémentation de l'algorithme

```
In [1]: # https://pypi.python.org/pypi/pydot
        #!apt-get -qq install -y graphviz && pip install -q pydot
        import pydot

        import matplotlib.pyplot as plt
        import networkx as nx
        from networkx.drawing import nx_pydot
        import matplotlib.image as mpimg
        import numpy as np
        import math
        from datetime import datetime as dt, timedelta

        from bokeh.io import show, output_notebook, output_file, reset_output
        from bokeh.models import Plot, Range1d, MultiLine, Circle, WheelZoomTool, PanTool,
        HoverTool, ResetTool, Arrow, VeeHead, CustomJSTransform, LabelSet
        from bokeh.models.graphs import from_networkx, NodesAndLinkedEdges, EdgesAndLinkedNodes
        from bokeh.palettes import Spectral4
        from bokeh.models.widgets import Dropdown
        from bokeh.plotting import curdoc
        from bokeh.layouts import widgetbox
        from bokeh.transform import transform

In [14]: def get_edge_data(G, nodes, attr):
             if G.__class__ == nx.classes.multigraph.MultiGraph: return
         G[nodes[0]][nodes[1]][0][attr]
             else: return G[nodes[0]][nodes[1]][attr]

         def networkx_to_pydot(G):

             # Convertir networkx.Graph en pydot.Dot
             graph = nx_pydot.to_pydot(G)

             # Ajouter les informations
             data = []
             for edge in G.edges():
                 data = np.hstack((data, get_edge_data(G, edge, 'weight')))

             for i, edge in enumerate(graph.get_edges()):
                 edge.set_label( "%d" % data[i])

             return graph


         def write_dot(G, gName, writeFile=False):

             if writeFile:
                 fileName = "./pdfsrc/"+gName+".dot"
                 #fileName = "pdfsrc\\"+gName+".dot"
                 nx_pydot.write_dot(G, fileName)

             graph = networkx_to_pydot(G)
             graph.write_png("./pdfsrc/"+gName+'.png')
             #graph.write_png("pdfsrc\\"+gName+'.png')

             # img=mpimg.imread(gName+'.png')
             # imgplot = plt.imshow(img)
             # plt.axis('off')
             # plt.show()
```

```
def drawGraph(G, title, path=None):
    plot = Plot(plot_width=840, plot_height=480, x_range=Range1d(-2, 2),
y_range=Range1d(-2, 2))
    plot.title.text = title

    #node_hover_tool = HoverTool(tooltips=[("Sommet", "@index")], line_policy="next")
    edge_hover_tool = HoverTool(tooltips=[("Weight", "@weight")], line_policy="interp")

    graph_renderer = from_networkx(G, nx.spring_layout)
    graph_renderer.node_renderer.glyph = Circle(size=15, fill_color=Spectral4[0])
    graph_renderer.edge_renderer.glyph = MultiLine(line_color='black',
line_join='miter')

    graph_renderer.inspection_policy = EdgesAndLinkedNodes()
    #graph_renderer.selection_policy = NodesAndLinkedEdges()

    # add the labels to the node renderer data source
    source = graph_renderer.node_renderer.data_source
    source.data['names'] = [str(x) for x in source.data['index']]

    # create a transform that can extract the actual x,y positions
    code = """
        var result = new Float64Array(xs.length)
        for (var i = 0; i < xs.length; i++) {
            result[i] = provider.graph_layout[xs[i]][%s]
        }
        return result
    """
    xcoord = CustomJSTransform(v_func=code % "0",
args=dict(provider=graph_renderer.layout_provider))
    ycoord = CustomJSTransform(v_func=code % "1",
args=dict(provider=graph_renderer.layout_provider))

    # Use the transforms to supply coords to a LabelSet
    labels = LabelSet(x=transform('index', xcoord),
                      y=transform('index', ycoord),
                      text='names', text_font_size="12px",
                      x_offset=5, y_offset=5,
                      source=source, render_mode='canvas')

    plot.add_layout(labels)

    plot.renderers.append(graph_renderer)
    plot.add_tools(edge_hover_tool, ResetTool(), PanTool(), WheelZoomTool())

    if path:
        pos = graph_renderer.layout_provider.graph_layout

        path_edges = [(path[i-1], path[i]) for i in range(1, len(path))]

        for edge in path_edges:
            arrows = Arrow(end = VeeHead(fill_color='black', line_color='black',
size=10),
                           x_start = pos[edge[0]][0],
                           y_start = pos[edge[0]][1],
                           x_end = pos[edge[1]][0],
                           y_end = pos[edge[1]][0],
                           line_color = 'red')
            plot.add_layout(arrows)

    #reset_output()
    output_notebook()
    show(plot)

# Create elements
g = nx.Graph()

A = g.add_node("A")
```

```
B = g.add_node("B")
C = g.add_node("C")
D = g.add_node("D")
E = g.add_node("E")
F = g.add_node("F")
G = g.add_node("G")
```

```
In [3]: def minimal_node(T, nodes):
            """
            Synopsis: Trouver le sommet de degree minimal
            """
            minN = None
            for node in nodes:
                if node in T:
                    if minN is None:
                        minN = node
                    elif T[node] < T[minN]:
                        minN = node

            return minN

        def dijkstra(G, init_node):
            """
            Synopsis:
            + G: un graphe pondéré
            + init_node: un sommet origine de G
            """

            T = { init_node: 0 }
            current_node = init_node
            path = {}

            L = set(G.nodes())


            while L:
                u  = minimal_node(T, L)
                if u is None: break

                current_weight = T[u]

                for v in G.neighbors(u):
                    weight = current_weight + get_edge_data(G, (u, v), 'weight')
                    if v not in T or weight < T[v]:
                        T[v] = weight
                        path[v] = u

                L.remove(u)

            return T, path

        def dijkstra_shortest_path(G, init, goal):
            distances, paths = dijkstra(G, init)
            route = [goal]

            while goal != init:
                if goal not in paths: break
                route.append(paths[goal])
                goal = paths[goal]

            route.reverse()
            return route
```

1.2.1 Graphe non connexte:

```
In [15]: gnc = nx.Graph(g)
         gnc.add_edge("A", "B", weight=9.0)
```

```
        gnc.add_edge("B", "C", weight=5.0)

        gnc.add_edge("D", "E", weight=7.0)
        gnc.add_edge("E", "F", weight=11.0)
        gnc.add_edge("F", "G", weight=1.0)

        print(dijkstra(gnc, 'A'))
        dsp = dijkstra_shortest_path(gnc, 'A', 'G')
        print(dsp)
        drawGraph(gnc, "graphe_non_connexe", dsp)
```

```
({'A': 0, 'B': 9.0, 'C': 14.0}, {'B': 'A', 'C': 'B'})
['G']
```

1.3 Arretes doubles

```
In [5]: gad = nx.MultiGraph(g)
        gad.add_edge("A", "B", weight=18.0)
        gad.add_edge("B", "C", weight=14.0)
        gad.add_edge("B", "E", weight=14.0)
        gad.add_edge("C", "D", weight=8.0)
        gad.add_edge("D", "E", weight=12.0)
        gad.add_edge("D", "G", weight=17.0)
        gad.add_edge("E", "F", weight=16.0)
        gad.add_edge("E", "F", weight=4.0)
        gad.add_edge("G", "F", weight=17.0)

        drawGraph(gad, "arretes_doubles")

        print(dijkstra(gad, 'A'))
        print(dijkstra_shortest_path(gad, 'A', 'G'))
        print(nx.shortest_path(gad, source='A', target='G'))
```

```
({'A': 0, 'B': 18.0, 'C': 32.0, 'E': 32.0, 'D': 40.0, 'F': 48.0, 'G': 57.0}, {'B':
'A', 'C': 'B', 'E': 'B', 'D': 'C', 'F': 'E', 'G': 'D'})
['A', 'B', 'C', 'D', 'G']
['A', 'B', 'C', 'D', 'G']
```

1.4 Multiplicité des plus courts chemins

```
In [6]: mspg = nx.DiGraph(g)
        mspg.add_edge("A", "B", weight=5.0)
        mspg.add_edge("A", "C", weight=10.0)
        mspg.add_edge("B", "D", weight=6.0)
        mspg.add_edge("B", "E", weight=3.0)
        mspg.add_edge("D", "F", weight=6.0)
        mspg.add_edge("E", "D", weight=2.0)
        mspg.add_edge("E", "G", weight=2.0)
        mspg.add_edge("E", "C", weight=2.0)
        mspg.add_edge("G", "F", weight=2.0)

        print(dijkstra(mspg, 'A'))

        dsp = dijkstra_shortest_path(mspg, 'A', 'F')
        print(dsp)

        drawGraph(mspg, "multiple_shortest_paths")
```

```
({'A': 0, 'B': 5.0, 'C': 10.0, 'D': 10.0, 'E': 8.0, 'G': 10.0, 'F': 12.0}, {'B': 'A',
'C': 'A', 'D': 'E', 'E': 'B', 'G': 'E', 'F': 'G'})
['A', 'B', 'E', 'G', 'F']
```

1.4.1 Estimation de la complexité

```
def minimal_node(T, nodes):
    minN = None                             # n fois
    for node in nodes:
        if node in T:
            if minN is None:
                minN = node
            elif T[node] < T[minN]:
                minN = node

    return minN
```

$$T_{minimalNode}(n) = \mathcal{O}(n)$$

```
def dijkstra(G, init_node):
    T = { init_node: 0 }
    current_node = init_node
    path = {}

    L = set(G.nodes())


    while L:                                # n fois maximum
        u  = minimal_node(T, L)
        if u is None: break

        current_weight = T[u]

        for v in G.neighbors(u):            # k fois
            weight = current_weight + get_edge_data(G, (u, v), 'weight')
            if v not in T or weight < T[v]:
                T[v] = weight
                path[v] = u

        L.remove(u)

    return T, path
```

$$T_{dijkstra}(n) = \sum_{k=0}^{n}(k) + T_{minimalNode}(n)$$

$$T_{dijkstra}(n) = max(\frac{n \cdot (n+1)}{2}, n)$$

$$T_{dijkstra}(n) = \mathcal{O}(n^2)$$

```python
def dijkstra_shortest_path(G, init, goal):
    distances, paths = dijkstra(G, init)
    route = [goal]

    while goal != init:                 # n fois
        if goal not in paths: break
        route.append(paths[goal])
        goal = paths[goal]

    route.reverse()
    return route
```

$$T_{dsp}(n) = T_{dijkstra}(n) + n$$

$$T_{dsp}(n) = max(\frac{n \cdot (n+1)}{2}, n)$$

$$T_{dsp}(n) = \mathcal{O}(n^2)$$

1.5 3. Plus courts chemins dans le métro parisien

Il s'agira de proposer une interface graphique permettant de calculer des plus courts chemins dans le métro parisien. L'objectif est que l'utilisateur puisse choisir une station de départ et une station d'arrivée et que le logiciel lui indique un plus court trajet entre ces deux stations. Idéalement le graphe complet devrait être visualisé.

Vous devez donc lire le fichier **metro.graph** (disponible sous moodle, format indiqué ci-dessous), extraire les informations pour construire le graphe et le visualiser puis bâtir votre interface pour pouvoir récupérer la requête de l'utilisateur, calculer le plus court chemin et enfin l'afficher.

Rendu minimal : un unique fichier python permettant de (i) lire le fichier, (ii) construire le graphe valué et orienté correspondant aux liaisons entre les stations de métro et (iii) des exemples d'appels à la fonction dijkstra(x,y) retournant le plus court chemin entre x et y.

Rendu v2 (nécessite le rendu v2 de la partie 2) : au lieu d'afficher le plus court chemin au format texte, celui-ci est visualisé sur le graphe.

Rendu v3 : le graphe est affiché avec les coordonnées réelles des stations et pas avec un dessin par défaut du graphe. Un descriptif textuel du trajet est donné à l'utilisateur (ex : indication des changements, temps de trajet global, nombre de correspondances, etc.)

Rendu v4 : ce que vous voulez en plus pour rendre l'utilisation plus agréable, par exemple pour aider à choisir les stations de départ et d'arrivée, pour pouvoir choisir le trajet avec le moins de correspondances, etc.

Il n'est pas demandé de rapport pour cette partie. Si vous avez des remarques à faire, faites-les en commentaires au début du fichier python.

Fichier metro.graph : + Chaque sommet correspond à une station pour une ligne donnée (par exemple, République (ligne 3) et République (ligne 5) sont deux sommets différents). A chaque sommet est associé le nom de la station (chaîne de caractères) et la position de la station sur une carte (échelle :1~25.7m). + Un arc orienté et valué connecte deux sommets si le métro relie directement les stations correspondantes. Le graphe n'est pas symétrique à cause de quelques "sens uniques", par exemple du côté de la porte d'Auteuil. Les arcs sont valués par le temps estimé du

trajet en secondes (en prenant pour base une vitesse moyenne de 10m/s, soit 36km/h). + Deux arcs symétriques connectent deux sommets s'il est possible de passer à pied sans changer de billet entre les stations correspondantes. Ces arcs sont alors valués par une estimation du temps moyen de trajet et d'attente (120s).

```
In [7]: import re

        file = open('metro.graph', 'r')
        lines = file.readlines()

        # Récupérer les données
        noms_sommets = np.array([ re.split(' ', ns.strip(), 1) for ns in lines[2: 378] ])
        noms_coordonnees = np.array([ re.split(' ', ns.strip()) for ns in lines[379: 755] ],
        dtype=float)
        arcs_values = np.array([ re.split(' ', ns.strip()) for ns in lines[756: 1689] ],
        dtype=float)

        # Préparer les parametres pour le plot
        minX = np.min(noms_coordonnees[:, 1])
        maxX = np.max(noms_coordonnees[:, 1])

        minY = np.min(noms_coordonnees[:, 2])
        maxY = np.max(noms_coordonnees[:, 2])

        sommet_values = np.concatenate((noms_sommets, np.delete(noms_coordonnees, 0, 1)),
        axis=1)
        del noms_sommets
        del noms_coordonnees
        #print(sommet_values)

In [8]: def harmoniser(s):
            """
            Synopsis: les identifiants des stations sont inconsistents d'un array à un autre
                    et on souhaite les harmoniser. Ex: '0123' => '123'
            """
            return str(int(s))

        metro = nx.DiGraph()

        for v in sommet_values:
            metro.add_node(harmoniser(v[0]), nom=v[1], X=float(v[2]), Y=float(v[3]))

        for e in arcs_values:
            metro.add_edge(harmoniser(e[0]), harmoniser(e[1]), weight=e[2])

        del sommet_values
        del arcs_values

        #write_dot(metro, "metro_parisien")

In [9]: def metro_layout(G):
            return { node[0]: np.array([float(node[1]['X']), float(node[1]['Y'])]) for node in
        G.nodes(data=True) }

        def metro_graph_draw(G, clr_nodes, interactive=False):

            # Colorier en rouge les sommets:
            red_nodes = clr_nodes

            # Colorier en rouge les arrets
            red_edges = [(clr_nodes[node-1], clr_nodes[node]) for node in range(1,
        len(clr_nodes))]


            # Static plot Matplotlib
            if interactive is False:
```

```
        node_positions = metro_layout(G)

        black_nodes = [node for node in G.nodes() if node not in red_nodes]
        black_edges = [edge for edge in G.edges() if edge not in red_edges]

        node_colours = ['black' if not node in red_nodes else 'red' for node in
G.nodes()]
        edge_colours = ['black' if not edge in red_edges else 'red' for edge in
G.edges()]

        plt.figure(figsize=(15, 10))
        nx.draw_networkx_nodes(G, pos=node_positions, nodelist=black_nodes,
node_color='b', node_size = 250, alpha=0.25)
        nx.draw_networkx_nodes(G, pos=node_positions, nodelist=red_nodes,
node_color='r', node_size = 500, alpha=1)

        nx.draw_networkx_edges(G, pos=node_positions, edgelist=black_edges,
edge_color='b', arrows=False, alpha=0.25)
        nx.draw_networkx_edges(G, pos=node_positions, edgelist=red_edges,
edge_color='r', arrows=True, alpha=1)

        node_labels = { node[0]: node[1]['nom'] if node[0] in red_nodes else "" for node
in G.nodes(data=True)}
        nx.draw_networkx_labels(G, pos=node_positions, labels=node_labels)


        #nx.draw(metro, pos=node_positions, edge_color=edge_colours, arrows=True,
node_size=50, node_color=node_colours)
        plt.title('Trajet de \"'+ G.node[depart]['nom']+'\" à \"'+
G.node[arrivee]['nom']+'\": ', size=15)
        plt.xlim(minX, maxX)
        plt.ylim(minY, maxY)
        plt.axis('off')
        plt.show()

    # Plot interactive Bokeh
    else:

        node_colours = { node[0]: 'red' if node[0] in red_nodes else 'black' for node in
G.nodes(data=True) }
        node_alpha = { node[0]: 1 if node[0] in red_nodes else 0.25 for node in
G.nodes(data=True) }

        edge_colours = { (edge[0], edge[1]): 'red' if edge in red_edges else 'black' for
edge in G.edges() }
        edge_thickness = { (edge[0], edge[1]): 3 if edge in red_edges else 0.5 for edge
in G.edges() }

        nx.set_node_attributes(G, node_colours, "node_color")
        nx.set_node_attributes(G, node_alpha, "node_alpha")

        nx.set_edge_attributes(G, edge_colours, "edge_color")
        nx.set_edge_attributes(G, edge_thickness, "edge_width")


        plot = Plot(plot_width=840, plot_height=480, x_range=Range1d(minX, maxX),
y_range=Range1d(minY, maxY))
        plot.title.text = 'Trajet de \"' + G.node[depart]['nom']+'\" à \"'+
G.node[arrivee]['nom']+'\": '

        node_hover_tool = HoverTool(tooltips=[("Station", "@nom"),("ID", "@index"),
("CoordX", "@X"), ('CoordY', "@Y")])
        plot.add_tools(node_hover_tool, ResetTool(), PanTool(), WheelZoomTool())

        graph_renderer = from_networkx(G, metro_layout)
        graph_renderer.node_renderer.glyph = Circle(size=15, fill_color="node_color",
fill_alpha='node_alpha')
        graph_renderer.edge_renderer.glyph = MultiLine(line_color="edge_color",
```

```
        line_width='edge_width', line_join='miter')

                plot.renderers.append(graph_renderer)

                # Dessiner les fleches uniquement sur les arrets rouges
                for edge in red_edges:
                    arrows = Arrow(end = VeeHead(fill_color='red', line_color='red', size=10),
                                   x_start = G.node[edge[0]]['X'],
                                   y_start = G.node[edge[0]]['Y'],
                                   x_end = G.node[edge[1]]['X'],
                                   y_end = G.node[edge[1]]['Y'],
                                   line_width = get_edge_data(G, edge,'edge_width'),
                                   line_color = 'red')
                    plot.add_layout(arrows)

                reset_output()
                output_notebook()
                show(plot)

            # Afficher les informations textuelles
            duree = np.sum(
                [G[clr_nodes[i-1]][clr_nodes[i]]['weight'] for i in range(1, len(clr_nodes))]
            )

            now = dt.now()
            print("Il est maintenant: "+ dt.strftime(now, '%Hh%M'))
            eta = now + timedelta(seconds=duree)
            print("Duree: ~", int(duree/60), 'mins')
            print("Estimation du temps d'arrivee: "+dt.strftime(eta, '%Hh%M'))

            itineraires = [ G.node[node]['nom'] for node in clr_nodes ]
            print("Il déservira ", len(itineraires) ," itinéraire(s)", itineraires)
```

```
In [10]: #!pip install ipywidgets
         !jupyter nbextension enable --py widgetsnbextension
         # Il fait rafraichir le navigateur
```

```
Enabling notebook extension jupyter-js-widgets/extension...
      - Validating: OK
```

```
In [11]: from ipywidgets import interact, interactive, IntSlider, Layout, interact_manual
         from IPython.display import display

         def metro_dijkstra(G, init_node, mode):
             T = { init_node: 0 }
             current_node = init_node
             path = {}
             L = set(G.nodes())
             while L:
                 u  = minimal_node(T, L)
                 if u is None: break
                 current_weight = T[u]
                 for v in G.neighbors(u):
                     # On va calculer par le degree de chaque voisins
                     pts = G.degree(v) if mode != "weight" else G[u][v]['weight']
                     weight = current_weight + pts
                     if v not in T or weight < T[v]:
                         T[v] = weight
                         path[v] = u
                 L.remove(u)
             return T, path

         def metro_dijkstra_shortest_path(G, init, goal, mode="weight"):
             distances, paths = metro_dijkstra(G, init, mode)
             route = [goal]
             while goal != init:
```

```
            if goal not in paths: break
            route.append(paths[goal])
            goal = paths[goal]
        route.reverse()
        return route


menu = {node[1]['nom']: node[0] for node in metro.nodes(data=True)}
depart = '0'
arrivee = '1'
Interactive = False
correspondance = 'weight'

def metro_paris_interactive(depart, arrivee, Interactive, correspondance):
    d = str(harmoniser(depart))
    a = str(harmoniser(arrivee))

    parcours = metro_dijkstra_shortest_path(metro, d, a, correspondance)
    metro_graph_draw(metro, parcours, Interactive)


p = interactive(metro_paris_interactive,
                depart=menu,
                arrivee=menu,
                Interactive = {'Oui': True, 'Non': False},
                correspondance = {'Moins de temps': "weight", 'Moins de
correspondences': "lstop"})
display(p)
```

```
interactive(children=(Dropdown(description='depart', options={'Abbesses': '0', 'Alexandre Dumas
```

In [ ]: