TEA_Graphs

November 1, 2018

Binôme: Amina KACIMI, Minh-Hoang DANG
Travail réalisé avec **Google Colabolatory**

```
In [1]: from IPython.display import HTML, display
        from wand.image import Image as WImage

        def showImage(fileName, scale=1):
            img = WImage(filename=fileName)
            width, height = img.size
            img.resize(int(width*scale), int(height*scale))
            return img
```

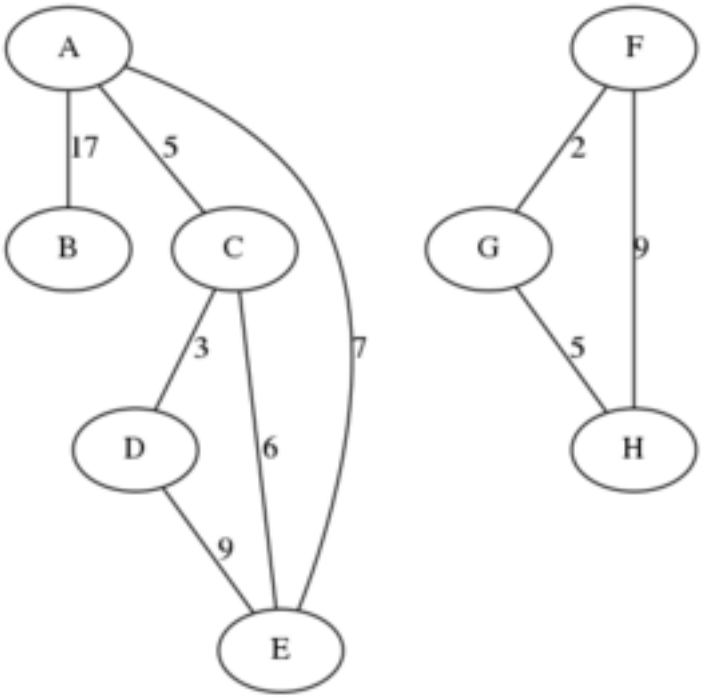
1 Analyse de l'algorithme de Dijkstra

1.1 Jeu d'essai:

1.1.1 Graphe non connexe:

```
In [2]: showImage('pdfsrc/graphe_non_connexe.svg', 0.65)
```

Out[2]:



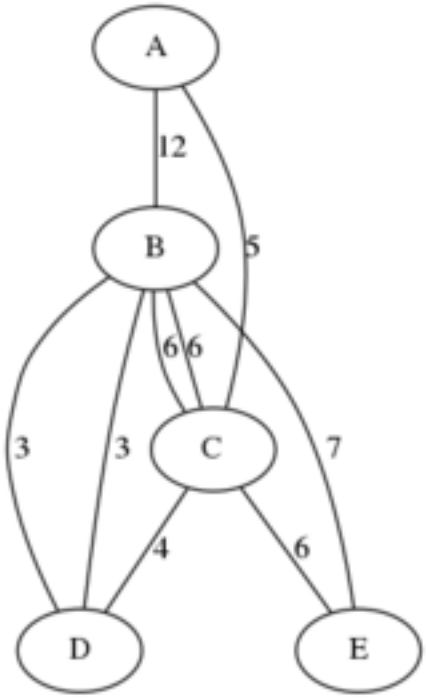
En partant de A

etape	L	T	u	v	T[u]	weight	path
0	{'G', 'H', 'F'}	{'A': 0, 'B': 17, 'C': 5, 'E': 7, 'D': 8}	A	B	0	17	{'B': 'A', 'C': 'A', 'E': 'A', 'D': 'C'}
1	{'G', 'H', 'F'}	{'A': 0, 'B': 17, 'C': 5, 'E': 7, 'D': 8}	A	C	0	5	{'B': 'A', 'C': 'A', 'E': 'A', 'D': 'C'}
2	{'G', 'H', 'F'}	{'A': 0, 'B': 17, 'C': 5, 'E': 7, 'D': 8}	A	E	0	7	{'B': 'A', 'C': 'A', 'E': 'A', 'D': 'C'}
3	{'G', 'H', 'F'}	{'A': 0, 'B': 17, 'C': 5, 'E': 7, 'D': 8}	C	A	5	10	{'B': 'A', 'C': 'A', 'E': 'A', 'D': 'C'}
4	{'G', 'H', 'F'}	{'A': 0, 'B': 17, 'C': 5, 'E': 7, 'D': 8}	C	D	5	8	{'B': 'A', 'C': 'A', 'E': 'A', 'D': 'C'}
5	{'G', 'H', 'F'}	{'A': 0, 'B': 17, 'C': 5, 'E': 7, 'D': 8}	C	E	5	11	{'B': 'A', 'C': 'A', 'E': 'A', 'D': 'C'}
6	{'G', 'H', 'F'}	{'A': 0, 'B': 17, 'C': 5, 'E': 7, 'D': 8}	E	C	7	13	{'B': 'A', 'C': 'A', 'E': 'A', 'D': 'C'}
7	{'G', 'H', 'F'}	{'A': 0, 'B': 17, 'C': 5, 'E': 7, 'D': 8}	E	D	7	16	{'B': 'A', 'C': 'A', 'E': 'A', 'D': 'C'}
8	{'G', 'H', 'F'}	{'A': 0, 'B': 17, 'C': 5, 'E': 7, 'D': 8}	E	A	7	14	{'B': 'A', 'C': 'A', 'E': 'A', 'D': 'C'}
9	{'G', 'H', 'F'}	{'A': 0, 'B': 17, 'C': 5, 'E': 7, 'D': 8}	D	C	8	11	{'B': 'A', 'C': 'A', 'E': 'A', 'D': 'C'}
10	{'G', 'H', 'F'}	{'A': 0, 'B': 17, 'C': 5, 'E': 7, 'D': 8}	D	E	8	17	{'B': 'A', 'C': 'A', 'E': 'A', 'D': 'C'}
11	{'G', 'H', 'F'}	{'A': 0, 'B': 17, 'C': 5, 'E': 7, 'D': 8}	B	A	17	34	{'B': 'A', 'C': 'A', 'E': 'A', 'D': 'C'}

1.1.2 Arêtes doubles:

```
In [3]: showImage('pdfsrc/arretes_doubles.svg', 0.65)
```

Out[3]:



En partant de A

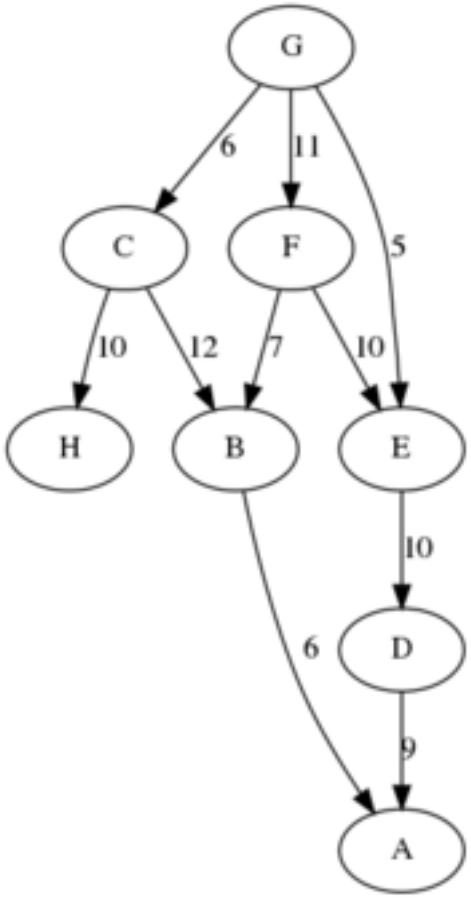
etape	L	T	u	v	T[u]	weight	path
0	{'D', 'B', 'E', 'A', 'C'}	{'A': 0, 'B': 11, 'C': 5, 'D': 9, 'E': 11}	A	B	0	12	{'B': 'C', 'C': 'A', 'D': 'C', 'E': 'C'}
1	{'D', 'B', 'E', 'A', 'C'}	{'A': 0, 'B': 11, 'C': 5, 'D': 9, 'E': 11}	A	C	0	5	{'B': 'C', 'C': 'A', 'D': 'C', 'E': 'C'}
2	{'D', 'B', 'E', 'C'}	{'A': 0, 'B': 11, 'C': 5, 'D': 9, 'E': 11}	C	A	5	10	{'B': 'C', 'C': 'A', 'D': 'C', 'E': 'C'}
3	{'D', 'B', 'E', 'C'}	{'A': 0, 'B': 11, 'C': 5, 'D': 9, 'E': 11}	C	B	5	11	{'B': 'C', 'C': 'A', 'D': 'C', 'E': 'C'}
4	{'D', 'B', 'E', 'C'}	{'A': 0, 'B': 11, 'C': 5, 'D': 9, 'E': 11}	C	D	5	9	{'B': 'C', 'C': 'A', 'D': 'C', 'E': 'C'}
5	{'D', 'B', 'E', 'C'}	{'A': 0, 'B': 11, 'C': 5, 'D': 9, 'E': 11}	C	E	5	11	{'B': 'C', 'C': 'A', 'D': 'C', 'E': 'C'}
6	{'D', 'B', 'E'}	{'A': 0, 'B': 11, 'C': 5, 'D': 9, 'E': 11}	D	B	9	12	{'B': 'C', 'C': 'A', 'D': 'C', 'E': 'C'}
7	{'D', 'B', 'E'}	{'A': 0, 'B': 11, 'C': 5, 'D': 9, 'E': 11}	D	C	9	13	{'B': 'C', 'C': 'A', 'D': 'C', 'E': 'C'}
8	{'B', 'E'}	{'A': 0, 'B': 11, 'C': 5, 'D': 9, 'E': 11}	B	A	11	23	{'B': 'C', 'C': 'A', 'D': 'C', 'E': 'C'}
9	{'B', 'E'}	{'A': 0, 'B': 11, 'C': 5, 'D': 9, 'E': 11}	B	C	11	17	{'B': 'C', 'C': 'A', 'D': 'C', 'E': 'C'}
10	{'B', 'E'}	{'A': 0, 'B': 11, 'C': 5, 'D': 9, 'E': 11}	B	D	11	14	{'B': 'C', 'C': 'A', 'D': 'C', 'E': 'C'}

etape	L	T	u	v	T[u]	weight	path
11	{'B', 'E'}	{'A': 0, 'B': 11, 'C': 5, 'D': 9, 'E': 11}	B	E	11	18	{'B': 'C', 'C': 'A', 'D': 'C', 'E': 'C'}
12	{'E'}	{'A': 0, 'B': 11, 'C': 5, 'D': 9, 'E': 11}	E	C	11	17	{'B': 'C', 'C': 'A', 'D': 'C', 'E': 'C'}
13	{'E'}	{'A': 0, 'B': 11, 'C': 5, 'D': 9, 'E': 11}	E	B	11	18	{'B': 'C', 'C': 'A', 'D': 'C', 'E': 'C'}

1.1.3 Multiplicité des plus courts chemins

```
In [4]: showImage('pdfsrc/multiple_shortest_paths.svg', 0.65)
```

Out[4]:



En partant de 'C':

etape	L	T	u	v	T[u]	weight	path
0	{'D', 'B', 'E', 'A', 'H', 'C', 'F', 'G'}	{'C': 0, 'B': 12, 'H': 10, 'A': 18}	C	B	0	12	{'B': 'C', 'H': 'C', 'A': 'B'}

etape	L	T	u	v	T[u]	weight	path
1	{'D', 'B', 'E', 'A', 'H', 'C', 'F', 'G'}	{'C': 0, 'B': 12, 'H': 10, 'A': 18}	C	H	0	10	{'B': 'C', 'H': 'C', 'A': 'B'}
2	{'D', 'B', 'E', 'A', 'F', 'G'}	{'C': 0, 'B': 12, 'H': 10, 'A': 18}	B	A	12	18	{'B': 'C', 'H': 'C', 'A': 'B'}

1.2 Complexité

```
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                          // n
    Extraire dans L le sommet u tel que T[u] est minimal  // m, NodeMinimal(L)
    Pour chaque sommet v voisin de u                       // k
        si (T(v) > T(u) + w(u,v))                          // Relax(u, v, w)
        T(v) = T(u) + w(u,v)
Retourner T
```

Un graph **NetworkX** est representé de manière suivante: + Un sommet est un array composant d'un index et un dictionaire des attributs:

$$node : [index, \{attr1 : val1, attr2 : val2, ...\}]$$

+ Un arret est un array composant d'un couple de sommets (index) et un dictionnaire des attribut:

$$edge : [(node_start, node_end), \{attr1 : val1, attr2 : val2, ...\}]$$

Avec un *array*, on peut faire des manipulations facilement avec numpy. Le *dict* permet de effectuer les recherche de valeur par une clé. En outre, on peut affecter n'importe quel type de donné dans la clé d'un *dict*.

Pour les jeux d'essai, on utilise les lettres pour les sommets (A-G). De ce fait, le tableau T et le chemin seront un *dict*

$$T : \{sommet1 : cout1, sommet2 : cout2, ...\}$$

$$DSP : \{sommet1, sommet2, ...\}$$

Note: DSP peut etre convertit en un *array* avec `python list()`

Cet algorithme repose sur deux composants, `NodeMinimal(L)` de coût $T(m) = \sum_{m=0}^{i}(1) = \mathcal{O}(|S|)$, et k fois `Relax(u, v, w)` de coût $T(v) = \sum_{j=0}^{v}(1) = \mathcal{O}(|A|)$. Le temps d'exécution (cas général) sera $\mathcal{O}(|S| \cdot T_{NodeMinimal(L)} + |A| \cdot T_{Relax(u,v,w)})$

Dans notre cas, avec les arrays, on aura:

$$T(n) = \sum_{i=0}^{n}(\sum_{m=0}^{i}(1) + \sum_{j=0}^{v}(1))$$

$$T(n) = \sum_{i=0}^{n}(i+v)$$

$$T(n) = vn + \sum_{i=0}^{n}(i)$$

$$T(n) = vn + \frac{n \cdot (n+1)}{2}$$

$$T(n) = \mathcal{O}(n^2 + vn)$$

Autrement dit, le temps total d'exécution est en

$$\mathcal{O}(|S| \cdot |S| \cdot 1 + |A| \cdot 1)$$

$$\mathcal{O}(|S|^2 + |A|)$$

1.3 Extension

1.3.1 Pseudo-algorithme

```
Entrée : un graphe pondéré G=(S,A,w), un sommet origine x, un sommet d'arrivee d
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

    Si u = d alors fin tant que

    Pour chaque sommet v voisin de u
        si (T(v) > T(u) + w(u,v))
        T(v) = T(u) + w(u,v)

DSP : le chemin le plus court
Inserer u dans DSP
Si T[u] is defined or u = x:
      Tant que u est distinct de x:  // l fois
          Inserer u dans DSP
          u := T[u]
Retourner DSP
```

1.3.2 Complexité

$$T(n) = \sum_{i=0}^{n}(\sum_{m=0}^{i}(1) + \sum_{j=0}^{v}(1)) + l$$

$$T(n) = \sum_{i=0}^{n}(i+v) + l$$

$$T(n) = vn + \sum_{i=0}^{n}(i) + l$$

$$T(n) = vn + \frac{n \cdot (n+1)}{2} + l$$

$$T(n) = \mathcal{O}(n^2 + vn)$$

De même, le temps total d'exécution est en $\mathcal{O}(|S|^2 + |A| + l) = \mathcal{O}(|S|^2 + |A|)$

1.3.3 Complexité sous autres structure de données

Soit la table de comparaison asymtotique suivante (Wikipédia):

Data Structure	Insert	Search	Find minimum	Space usage
Unsorted array	O(1)	O(n)	O(n)	O(n)
Sorted array	O(n)	O(log n)	O(1)	O(n)
Unsorted linked list	O(1)	O(n)	O(n)	O(n)
Sorted linked list	O(n)	O(n)	O(1)	O(n)
Self-balancing binary search tree	O(log n)	O(log n)	O(log n)	O(n)
Heap	O(log n)	O(n)	O(1): min-heap, O(n): max-heap	O(n)
Hash table	O(1)	O(1)	O(n)	O(n)
Trie (k = average length of key)	O(k)	O(k)	O(k)	O(k n)

Arbre binaire de recherche (Heap): Dans un arbre binaire de recherche, les fonction d'extraction (dans `MinimalNode(L)`) et d'insertion (dans `Relax(u, v, w)` sont des fonctions recursives qui s'exécute en temps $\mathcal{O}(log|S|)$

Le temps d'éxécution sera donc:

$$\mathcal{O}(|S| \cdot log|S| + |A| \cdot log|S|)$$

$$\mathcal{O}((|S| + |A|) \cdot log|S|)$$

On aura $\mathcal{O}(|S|log|S|)$ si $|S| > |A|$ et $\mathcal{O}(|A|log|S|)$ sinon.

Table hachage: `MinimalNode(L)` coût $\mathcal{O}(|S|)$ et `Relax(u, v, w)` coût $\mathcal{O}(1)$

Le temps d'éxécution sera donc:

$$\mathcal{O}(|S| \cdot |S| + |A| \cdot 1)$$

$$\mathcal{O}(|S|^2 + |A|)$$

Liste chaînée (triée): `MinimalNode(L)` coût $\mathcal{O}(1)$ et `Relax(u, v, w)` coût $\mathcal{O}(|S|)$

Le temps d'éxécution sera donc:

$$\mathcal{O}(|S| \cdot 1 + |A| \cdot |S|)$$

$$\mathcal{O}(|S| + |A| \cdot |S|)$$

Liste chaînée (non-triée): `MinimalNode(L)` coût $\mathcal{O}(|S|)$ et `Relax(u, v, w)` coût $\mathcal{O}(1)$
Le temps d'éxécution sera donc:

$$\mathcal{O}(|S| \cdot |S| + |A| \cdot 1)$$

$$\mathcal{O}(|S|^2 + |A|)$$

2 Implémentation de l'algorithme

```
In [5]: # List des programmes à installer
        """!apt-get -qq install -y graphviz
        !pip install -q pydot
        !pip install -q ipywidgets
        !pip install -q matplotlib
        !pip install -q networkx
        !pip install -q numpy
        !pip install -q bokeh
        !pip install -q ipywidgets
        !pip install -q Wand
        """

        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, export_png
        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

        from ipywidgets import interact, interactive, IntSlider, Layout, interact_manual

        !jupyter nbextension enable --py widgetsnbextension
        # Il faut rafraichir le navigateur
```

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

```
In [6]: 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 drawGraph(G, title, path=None, layout=nx.circular_layout):

            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, 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='red', line_color='red', 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]][1],
                           line_color = 'red')
            plot.add_layout(arrows)

    reset_output()
    output_notebook()
    show(plot)

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_svg("./pdfsrc/"+gName+'.svg')
            #graph.write_png("pdfsrc\\"+gName+'.png')

        # Create elements
        g = nx.Graph()

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

In [7]:
```
        # Session interactive
        def make_graph_menu(G):
            return {node[0]: node[0] for node in G.nodes(data=True)}

        graph_menu = make_graph_menu(g)
        graph_layout_menu = {"Circular": nx.circular_layout, "Spectral": nx.spectral_layout,
        "Spring": nx.spring_layout}
        graph_depart = 'A'
        graph_arrivee = 'G'
        graph_target = g
        graph_name = 'Example'
        graph_layout = nx.spectral_layout

        def graph_interactive(graph_target, graph_depart, graph_arrivee, graph_name,
        graph_layout):
            parcours = dijkstra_shortest_path(graph_target, graph_depart, graph_arrivee)
            drawGraph(graph_target, graph_name, parcours, graph_layout)
```

In [8]:
```
            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())
```

```
            etape = 0

            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
                    etape+=1

                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
```

2.1 Graphe non connexte:

```
In [9]: gnc = nx.Graph(g)
        gnc.add_edge('A','B',weight=17)
        gnc.add_edge('A','C',weight=5)
        gnc.add_edge('C','D',weight=3)
        gnc.add_edge('C','E',weight=6)
        gnc.add_edge('E','D',weight=9)
        gnc.add_edge('E','A',weight=7)
        gnc.add_edge('F','G',weight=2)
        gnc.add_edge('H','G',weight=5)
        gnc.add_edge('F','H',weight=9)

        print(dijkstra(gnc, 'A'))
        write_dot(gnc, 'graphe_non_connexe')

        p = interactive(graph_interactive,
                        graph_target=[gnc],
                        graph_depart=graph_menu,
                        graph_arrivee=graph_menu,
                        graph_name="graphe_non_connexe",
                        graph_layout=graph_layout_menu
                       )
        display(p)
```

```
({'A': 0, 'B': 17, 'C': 5, 'E': 7, 'D': 8}, {'B': 'A', 'C': 'A', 'E': 'A', 'D': 'C'})
```

```
interactive(children=(Dropdown(description='graph_target', options=(<networkx.classes.graph.Gra
```

```
In [10]: showImage('pdfsrc/screenshots/graphe_non_connexe.png')
```

Out[10]:

graph_target

graph_depart A

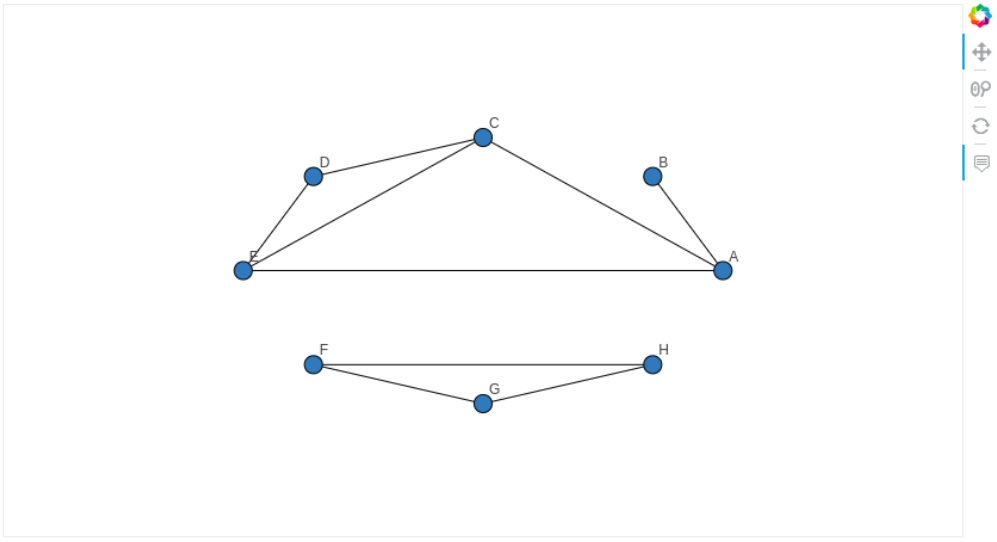
graph_arriv… A

graph_name graphe_non_connexe

graph_layout Circular

BokehJS 1.0.0 successfully loaded.

graphe_non_connexe



2.2 Arêtes doubles

```
In [11]: gad = nx.MultiGraph()

         gad.add_node('A',pos=(1,3))
         gad.add_node('B',pos=(1,2))
         gad.add_node('C',pos=(1,1))
         gad.add_node('D',pos=(2,2))
         gad.add_node('E',pos=(2,1))

         gad.add_edge('B','A',weight=12)
         gad.add_edge('C','A',weight=5)
         gad.add_edge('C','B',weight=6)
         gad.add_edge('D','B',weight=3)
         gad.add_edge('B','C',weight=8)
         gad.add_edge('D','C',weight=4)
         gad.add_edge('E','C',weight=6)
         gad.add_edge('B','D',weight=5)
         gad.add_edge('E','B',weight=7)

         print(dijkstra(gad, 'A'))
         write_dot(gad, 'arretes_doubles')

         p = interactive(graph_interactive,
                         graph_target=[gad],
                         graph_depart=make_graph_menu(gad),
                         graph_arrivee=make_graph_menu(gad),
                         graph_name="arretes_doubles",
                         graph_layout=graph_layout_menu
```

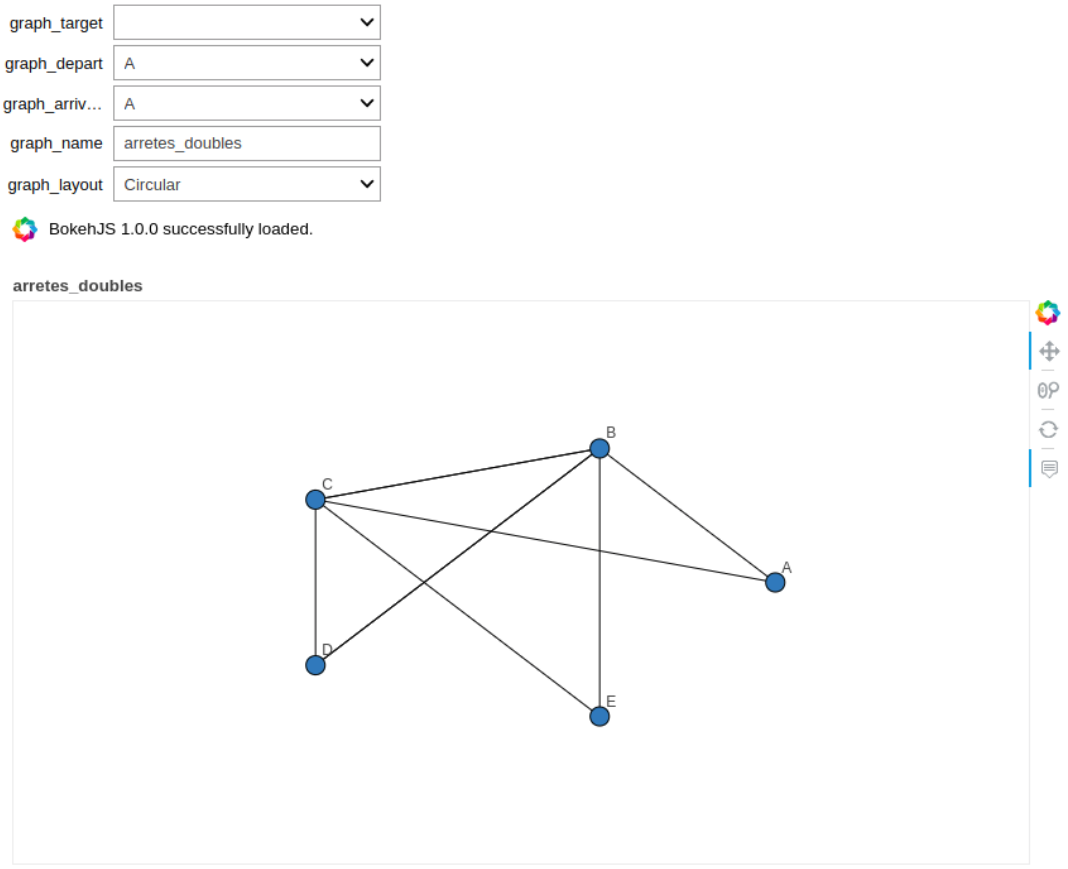
```
                    )
        display(p)
```

```
({'A': 0, 'B': 11, 'C': 5, 'D': 9, 'E': 11}, {'B': 'C', 'C': 'A', 'D': 'C', 'E': 'C'})
```

```
interactive(children=(Dropdown(description='graph_target', options=(<networkx.classes.multigra
```

In [12]: `showImage('pdfsrc/screenshots/arretes_doubles.png')`

Out[12]:



2.3 Multiplicité des plus courts chemins

In [13]:
```
mspg = nx.DiGraph()

        mspg.add_node('A',pos=(1,2))
        mspg.add_node('B',pos=(1,3))
        mspg.add_node('C',pos=(1,1))
        mspg.add_node('D',pos=(2,1))
        mspg.add_node('E',pos=(2,2))
        mspg.add_node('F',pos=(2,3))
        mspg.add_node('G',pos=(3,1))
```

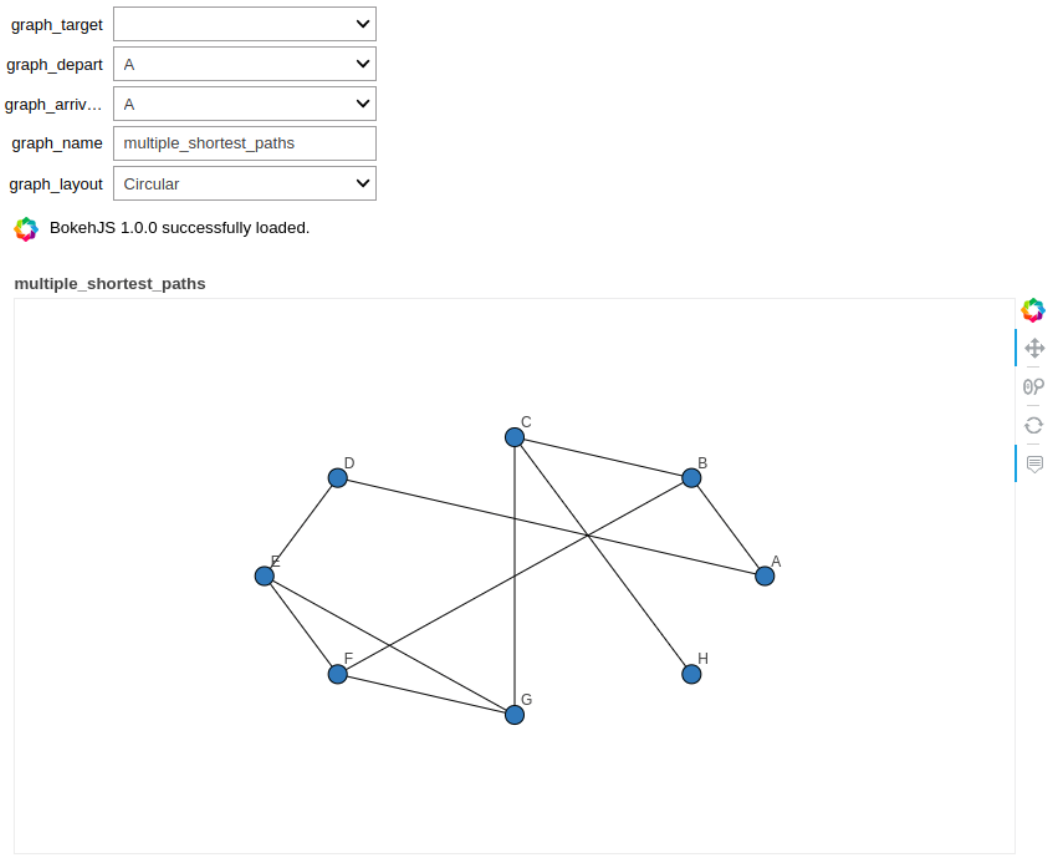
```
mspg.add_node('H',pos=(3,2))

mspg.add_edge('B','A',weight=6)
mspg.add_edge('D','A',weight=9)
mspg.add_edge('C','B',weight=12)
mspg.add_edge('F','B',weight=7)
mspg.add_edge('G','C',weight=6)
mspg.add_edge('E','D',weight=10)
mspg.add_edge('G','E',weight=5)
mspg.add_edge('F','E',weight=10)
mspg.add_edge('G','F',weight=11)
mspg.add_edge('C','H',weight=10)

print(dijkstra(mspg, 'C'))
write_dot(mspg, 'multiple_shortest_paths')

p = interactive(graph_interactive,
                graph_target=[mspg],
                graph_depart=make_graph_menu(mspg),
                graph_arrivee=make_graph_menu(mspg),
                graph_name="multiple_shortest_paths",
                graph_layout=graph_layout_menu
               )
display(p)
```

```
({'C': 0, 'B': 12, 'H': 10, 'A': 18}, {'B': 'C', 'H': 'C', 'A': 'B'})
```

```
interactive(children=(Dropdown(description='graph_target', options=(<networkx.classes.digraph.
```

In [14]:
```
showImage('pdfsrc/screenshots/multiple_shortest_paths.png')
```

Out[14]:



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 [15]: 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)

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


         # 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])

         del noms_sommets
         del noms_coordonnees
         #print(sommet_values)

In [16]: 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 [17]: 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()
            #export_png(plot, filename="pdfsrc/metro_plot.png")
            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 [18]:
```
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 = '175'
        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)
```

In [19]:
```
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 [20]:
```
showImage('pdfsrc/metro_plot.png')
```

Out[20]:

depart Abbesses

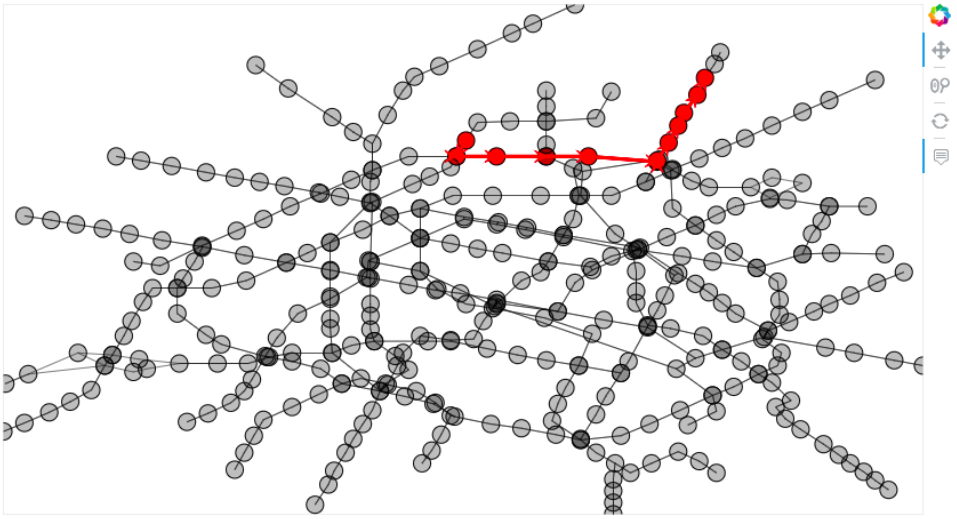
arrivee Aubervilliers-Pantin, Quatre Che

Interactive Oui

correspond… Moins de temps

BokehJS 1.0.0 successfully loaded.

Trajet de "Abbesses" à "Madeleine":



```
Il est maintenant: 00h01
Duree: ~ 12 mins
Estimation du temps d'arrivee: 00h13
Il déservira  13  itinéraire(s) ['Abbesses', 'Pigalle', 'Pigalle', 'Anvers', 'Barbès Rochechouart', 'La Chapell
e', 'Stalingrad', 'Stalingrad', 'Riquet', 'Crimée', 'Corentin-Cariou', 'Porte de la Villette', 'Aubervilliers-Pan
tin, Quatre Chemins']
```