RO202

TD Numéro 1

Valentin Quoniam-Barre

Février 2022

EXERCICE 1 : ARBRES

Q1.

Appliquons l'algorithme de Kruskal à notre arbre :

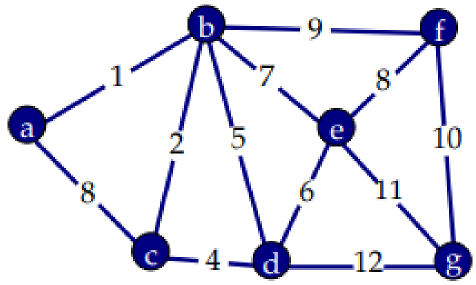

FIGURE 1 – Graphe de l'exercice 1

La liste des arêtes par poids croissant est : ["ab - 1", "bc - 2", "cd - 4","db - 5","de - 6", "be - 7", "ef - 8", "ac - 8", "bf -9", "fg - 10", "eg - 11", "gd - 11"]

Pour les 3 permières arêtes, on a pas de problèmes de cycle. En revanche, l'arête "db" générerait un cycle si on l'ajoutait à l'arbre. On ajoute ensuite "de" et on passe "be" car on aurait un cycle. De même, on ajoute "ef" et on enlève "bf" car on aurait un cycle. Il n'y a plus qu'à rajouter "fg" et l'arbre est complet.
L'arbre a donc un poids de 31 et est : [a-b, b-c, c-d, d-e, e-f, f-g]

Q2.

Ce type de réseau est très problématique pour une banque en raison de sa rigidité. Si une arête est défaillante, il n'y aura peu de moyens de la remplacer et un noeud pourrait devenir inatteignable. De plus, partager une donnée à tous les noeuds (ce qui est souvent fait dans la réalité), est très coûteux ici. Les banques utilisent plutôt un **réseau en étoile**.

EXERCICE 2 : CHEMINS

Appliquons l'algorithme de Dijkstra à notre graphe :

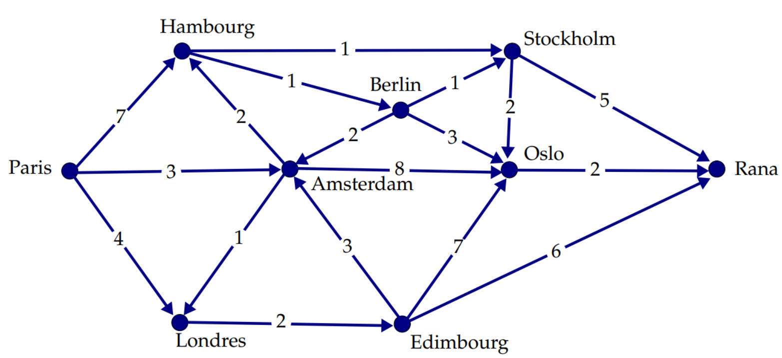

FIGURE 2 – Graphe de l'exercice 2

Valeurs de pi

j	pivot	Amsterdam	Londres	Hambourg	Edinbourg	Berlin	Stockholm	Oslo	Rana
1	Paris	**3**	4	7	|	|	|	|	|
2	Amsterdam	|	**4**	5	|	|	|	11	|
3	Londres	|	|	**5**	6	|	|	11	|
4	Hambourg	|	|	|	**6**	6	6	11	|
5	Edimbourg	|	|	|	|	**6**	6	11	12
6	Berlin	|	|	|	|	|	**6**	9	12
7	Stockholm	|	|	|	|	|	|	**8**	11
8	Oslo	|	|	|	|	|	|	|	**10**
9	Rana	3	4	5	6	6	6	8	10

Valeurs de pred

j	pivot	Amsterdam	Londres	Hambourg	Edinbourg	Berlin	Stockholm	Oslo	Rana
1	Paris	**Paris**	Paris	Paris	|	|	|	|	|
2	Amsterdam	|	**Paris**	Amsterdam	|	|	|	Amsterdam	|
3	Londres	|	|	**Amsterdam**	Londres	|	|	Amsterdam	|
4	Hambourg	|	|	|	**Londres**	Hambourg	Hambourg	Amsterdam	|
5	Edimbourg	|	|	|	|	**Hambourg**	Hambourg	Amsterdam	Edimbourg
6	Berlin	|	|	|	|	|	**Hambourg**	Berlin	Edimbourg
7	Stockholm	|	|	|	|	|	|	**Stockholm**	Stockholm
8	Oslo	|	|	|	|	|	|	|	**Oslo**
9	Rana	Paris	Paris	Amsterdam	Londres	Hambourg	Hambourg	Stockholm	Oslo

Le chemin le plus court pour parcourir tout l'arbre a un poids de 17. Il suffit de lire, pour chaque colonne, la dernière et première ligne de "Valeurs de pred" pour avoir toutes les arêtes du chemin. Pour aller à Rana le plus vite possible, il faut donc faire un trajet de 10 heures :

Paris -> Amsterdam -> Hambourg -> Stockholm -> Oslo -> Rana.

EXERCICE 3 : IMPLEMENTATION DE KRUSKAL

Voici l'algorithme de Kruskal implémenté assez simplement en python. Y sont présentes les fonctionnalités "computeMin" et "kruskalCC" comme demandé dans la suite de l'exercice.

```python
import numpy as np
import graph
import sys

def main():

    # Créer un graphe contenant les sommets a, b, c, d, e, f, g
    g = graph.Graph(np.array(["a", "b", "c", "d", "e", "f", "g"]))

    # Ajouter les arêtes
    g.addEdge("a", "b",  1.0)
    g.addEdge("a", "c",  3.0)
    g.addEdge("b", "c",  2.0)
    g.addEdge("b", "d",  5.0)
    g.addEdge("b", "e",  7.0)
    g.addEdge("b", "f",  9.0)
    g.addEdge("c", "d",  4.0)
    g.addEdge("d", "e",  6.0)
    g.addEdge("d", "g", 12.0)
```

```
    g.addEdge("e", "f",  8.0)
    g.addEdge("e", "g", 11.0)
    g.addEdge("f", "g", 10.0)

    """
    Le résulat pour computeMin = True:
                Le poids total de l'abre est : 31.0
                Voici l'arbre :

                'a' - 'b' (1.0)
                'b' - 'c' (2.0)
                'c' - 'd' (4.0)
                'd' - 'e' (6.0)
                'e' - 'f' (8.0)
                'f' - 'g' (10.0)
    """

    """
    Le résultat pour computeMin = False:
                Le poids total de l'abre est : 49.0
                Voici l'arbre :

                'a' - 'c' (3.0)
                'b' - 'f' (9.0)
                'c' - 'd' (4.0)
                'd' - 'g' (12.0)
                'e' - 'g' (11.0)
                'f' - 'g' (10.0)
    """

    ####### EXERCICE 3 - QUESTION 4 ET 5#####

    #1er arbre de test :
    fig1a = graph.Graph(np.array(["a", "b", "c", "d", "e", "f", "g","h"]))
    fig1a.addEdge("a", "b",  9.0)
    fig1a.addEdge("a", "f",  6.0)
    fig1a.addEdge("a", "h",  9.0)
    fig1a.addEdge("f", "e",  1.0)
    fig1a.addEdge("b", "e",  5.0)
    fig1a.addEdge("e", "g",  3.0)
    fig1a.addEdge("g", "c",  5.0)
    fig1a.addEdge("g", "d",  9.0)
    fig1a.addEdge("g", "h",  5.0)
    fig1a.addEdge("d", "h",  7.0)
    fig1a.addEdge("c", "d",  2.0)
    fig1a.addEdge("b", "d",  8.0)
    fig1a.addEdge("b", "c",  5.0)

    """
    Le résultat pour ComputeMin = True:
                    Le poids total de l'abre est : 27.0
                    Voici l'arbre :

                    'a' - 'f' (6.0)
                    'b' - 'e' (5.0)
                    'c' - 'd' (2.0)
                    'c' - 'g' (5.0)
                    'e' - 'f' (1.0)
                    'e' - 'g' (3.0)
                    'g' - 'h' (5.0)
    """

    #2ème arbre de test
    fig1b = graph.Graph(np.array(["A", "B", "C", "D", "E", "F"]))
    fig1b.addEdge("A", "B",  4.0)
    fig1b.addEdge("A", "C",  3.0)
    fig1b.addEdge("B", "C",  5.0)
    fig1b.addEdge("B", "F",  2.0)
    fig1b.addEdge("C", "F",  5.0)
    fig1b.addEdge("C", "D",  2.0)
    fig1b.addEdge("F", "D",  3.0)
```

```
    fig1b.addEdge("D", "E",  4.0)
    fig1b.addEdge("F", "E",  3.0)

    """
    Le résultat pour computeMin = True :
                Le poids total de l'abre est : 13.0
                Voici l'arbre :

                'A' - 'C' (3.0)
                'B' - 'F' (2.0)
                'C' - 'D' (2.0)
                'D' - 'F' (3.0)
                'E' - 'F' (3.0)
    """
    #####################################
    # Obtenir un arbre couvrant de poids minimal du graphe
    tree = kruskalCC(g)

    # S'il existe un tel arbre (i.e., si le graphe est connexe)
    if tree != None:

        # L'afficher
        print(tree)

    else:
        print("Pas d'arbre couvrant")

# Applique l'algorithme de Kruskal pour trouver un arbre couvrant de poids minimal d'un graphe
# Retourne: Un arbre couvrant de poids minimal du graphe ou None s'il n'en existe pas
def kruskal(g, computeMin):

    if type(computeMin)!=bool:
        print("Error : 2nd argument computeMin has to be a boolean")
        return 0
    ############### INITIALISATION ###############

    # Créer un nouveau graphe contenant les mêmes sommets que g
    tree = graph.Graph(g.nodes)

    # Récupérer toutes les arêtes de g
    edges = g.getEdges()

    # Trier les arêtes par poids croissant si computeMin est vrai, sinon on a juste à trier dans l'autre sens !
    if computeMin == False:
        edges.sort(reverse = True) #Tri dans l'ordre décroissant
    else:
        edges.sort()

    # Compter le poids de l'arbre
    addedWeights = 0

    ############### ALGORITHME ####################

    # On utilise l'implémentation Union-Find pour détecter les cycles dans le graphe
    for edge in edges:
       if tree.createACycle(edge) == False:  #On ajoute le chemin à l'arbre si il ne forme pas encore de cycle avec ce dernier
           tree.addCopyOfEdge(edge)
           addedWeights+=edge.weight          #On ajoute également son poids

    print(f"\nLe poids total de l'abre est : {addedWeights}")
    print("\nVoici l'arbre :\n")

    if addedWeights ==0:                      #Si il n'y a aucun arbre possible, on retourne None
        return None

    return tree


#################### EXERCICE 3 - Q6 ################
```

```python

def kruskalCC(g):

    # Créer un nouveau graphe contenant les mêmes sommets que g
    tree = graph.Graph(g.nodes)

    # Récupérer toutes les arêtes de g
    edges = g.getEdges()
    edges.sort()
    # Compter le poids de l'arbre
    addedWeights = 0

    # Liste des composantes connexes
    component = [k for k in range(0,g.n)]

    ############### ALGORITHME ####################

    # On utilise l'implémentation Union-Find pour détecter les cycles dans le graphe

    for edge in edges:
        if component[edge.id1] != component[edge.id2]:       #Si les 2 noeuds sont des des composantes connexes différentes, c'est good
            tree.addCopyOfEdge(edge)
            addedWeights += edge.weight
            component[edge.id2] = component[edge.id1]        #On met le dernier noeud visité dans la composante connexe de la racine et ainsi de suite


    print(f"\nLe poids total de l'abre est : {addedWeights}")
    print("\nVoici l'arbre :\n")

    if addedWeights ==0:                          #Si il n'y a aucun arbre possible, on retourne None
        return None
    return tree

if __name__ == '__main__':
    main()
```

EXERCICE 4 : IMPLEMENTATION DE DIJKSTRA

Voici l'algorithme de Dijkstra implémenté assez simplement en python.

```python
import graph
import sys
import numpy as np

def main():

    ################ GRAPHE DE L'EXERCICE 2 ##############
    cities = []
    cities.append("Paris")
    cities.append("Hambourg")
    cities.append("Londres")
    cities.append("Amsterdam")
    cities.append("Edimbourg")
    cities.append("Berlin")
    cities.append("Stockholm")
    cities.append("Rana")
    cities.append("Oslo")

    g = graph.Graph(cities)

    g.addArc("Paris", "Hambourg", 7)
    g.addArc("Paris",  "Londres", 4)
    g.addArc("Paris",  "Amsterdam", 3)
    g.addArc("Hambourg",  "Stockholm", 1)
    g.addArc("Hambourg",  "Berlin", 1)
    g.addArc("Londres",  "Edimbourg", 2)
    g.addArc("Amsterdam",  "Hambourg", 2)
```

```
    g.addArc("Amsterdam",  "Oslo", 8)
    g.addArc("Stockholm",  "Oslo", 2)
    g.addArc("Stockholm",  "Rana", 5)
    g.addArc("Berlin",  "Amsterdam", 2)
    g.addArc("Berlin",  "Stockholm", 1)
    g.addArc("Berlin",  "Oslo", 3)
    g.addArc("Edimbourg",  "Oslo", 7)
    g.addArc("Edimbourg",  "Amsterdam", 3)
    g.addArc("Edimbourg",  "Rana", 6)
    g.addArc("Oslo",  "Rana", 2)

    ################# GRAPHES DE L'EXERCICE 4 ###############

    # Premier graphe
    fig2a = graph.Graph(np.array(["r","a", "b", "c", "d", "e", "f", "g"]))
    fig2a.addArc("r","a",5)
    fig2a.addArc("r","b",4)
    fig2a.addArc("b","a",5)
    fig2a.addArc("b","c",3)
    fig2a.addArc("b","g",9)
    fig2a.addArc("a","c",3)
    fig2a.addArc("c","d",2)
    fig2a.addArc("c","f",6)
    fig2a.addArc("c","g",8)
    fig2a.addArc("d","e",2)
    fig2a.addArc("d","a",8)
    fig2a.addArc("e","c",4)
    fig2a.addArc("g","f",5)

    # Deuxième graphe
    fig2b = graph.Graph(np.array(["r","A", "B", "C", "D", "E", "F", "G"]))
    fig2b.addArc("r","A",2)
    fig2b.addArc("r","G",3)
    fig2b.addArc("F","G",3)
    fig2b.addArc("F","D",4)
    fig2b.addArc("A","F",1)
    fig2b.addArc("A","B",3)
    fig2b.addArc("B","C",2)
    fig2b.addArc("D","C",2)
    fig2b.addArc("E","D",3)
    fig2b.addArc("E","F",2)
    fig2b.addArc("G","E",2)

    #########################################################
    # Applique l'algorithme de Dijkstra pour obtenir une arborescence
    tree1 = dijkstra(g, "Paris")
    print(tree1)

    tree2 =  dijkstra(fig2a, "r")
    print(tree2)

    tree3 =  dijkstra(fig2b, "r")
    print(tree3)

########### RESULTATS OBTENUS ############
    """
1er arbre :
    Poids du chemin : 17.0

'Paris' - 'Londres' (4.0)
'Paris' - 'Amsterdam' (3.0)
'Hambourg' - 'Berlin' (1.0)
'Hambourg' - 'Stockholm' (1.0)
'Londres' - 'Edimbourg' (2.0)
'Amsterdam' - 'Hambourg' (2.0)
'Stockholm' - 'Oslo' (2.0)
'Oslo' - 'Rana' (2.0)

2ème arbre :
    Poids du chemin : 31.0

```

```
'r' - 'a' (5.0)
'r' - 'b' (4.0)
'b' - 'c' (3.0)
'b' - 'g' (9.0)
'c' - 'd' (2.0)
'c' - 'f' (6.0)
'd' - 'e' (2.0)

3ème arbre :
    Poids du chemin : 17.0

'r' - 'A' (2.0)
'r' - 'G' (3.0)
'A' - 'B' (3.0)
'A' - 'F' (1.0)
'B' - 'C' (2.0)
'F' - 'D' (4.0)
'G' - 'E' (2.0)
    """

def dijkstra(g, origin):

    ########### INITIALISATION ##############

    # Créer un nouveau graphe contenant les mêmes sommets que g
    tree = graph.Graph(g.nodes)

    # Get the index of the origin
    r = g.indexOf(origin)

    #Build the list of indexes, as we'll work with them
    v = [k for k in range(0,g.n)]

    # Next node considered
    pivot = r

    # Liste qui contiendra les sommets ayant été considérés comme pivot
    v2 = []
    v2.append(r)

    # Initialisation de la liste des prédecesseurs (cest fait un peu brutalement)
    pred = [0] * g.n

    # Initialisation du tableau des distances des sommets à r
    # Les distances entre r et les autres sommets sont initialement infinies
    pi = [sys.float_info.max] * g.n
    pi[r] = 0

    ############### ALGORITHME ###############

    for i in range(1,g.n):                                              # Sélection des n-1 pivots du chemin
        for node in (list(set(v) - set(v2))):                           # Sommets pas encore pivots (Attention, on travaille en index !)
            # On renomme les variables pour plus de lisibilité
            weight = g.adjacency[pivot,node]

            #Actualisation de pi et de pred
            if weight != 0.0 and pi[pivot] + weight < pi[node]:        # Sommets reliés au pivot avec nouveau chemin détecté
                pi[node] = pi[pivot] + weight                           # On acutalise le nouveau meilleur chemin
                pred[node] = pivot                                      # On actualise aussi le parent

        # Choix du meilleur pivot en codant notre propre fonction argmin, avec les indices uniquement dans v-v2
        pi_min = sys.float_info.max
        for index in list(set(v) - set(v2)):
            if pi[index] <pi_min:
                pi_min = pi[index]

```

```python
        # Traitement des villes ayant la même distance à l'origine : On les prend une par une
        occurences  = [k for (k, item) in enumerate(pi) if item == pi_min]  # On récupère les indices des villes à distances égales
        if len(occurences)!=1:                                              # Si il y a plusieurs occurences, on récupère les indices un par un
            k=0
            while(occurences[k] in v2):    #On fait la récupération "un par un" avec une boucle while
                k+=1
            pivot = occurences[k]
        else:
            pivot = occurences[0]                                           # Sinon, tout va bien et on prend le seul indice dans "occurences"

        v2.append(pivot)                                                    # Ajout du nouveau pivot

    #Construction de l'arborescence et affichage de l'abre (on affiche aussi le poids)
    total_weight = 0
    for i in range(0,g.n):
        weight = g.adjacency[pred[i],i]
        tree.addArcByIndex(pred[i], i, weight)       #On ajoute, pour chaque node, sa node parente trouvée par l'algorithme
        total_weight += weight

    if total_weight == 0:
        return None

    print(f"Poids du chemin : {total_weight} \n")
    return tree

if __name__ == '__main__':
    main()
```

EXERCICE 5 : ORDONNANCEMENT

Q1. et Q2.

Voici le graphe de la question 2. Nous avons pris en compte la durée de chaque tâche, et les arêtes non numérotées ont un poids de 0.

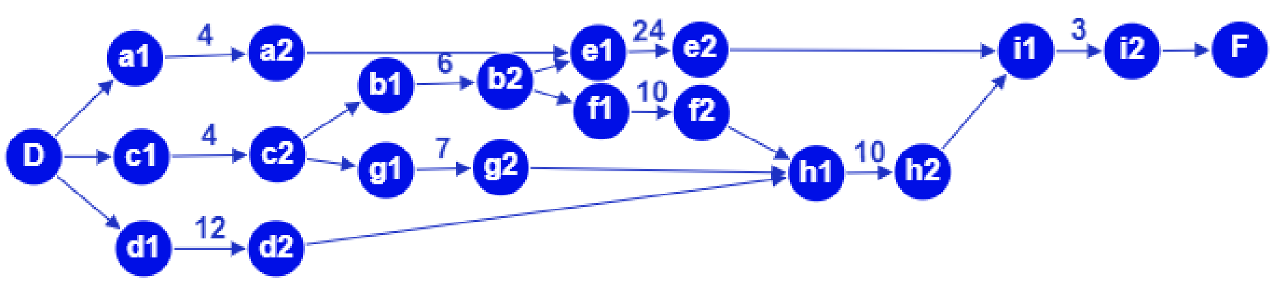

Figure 3 – Graphe de l'exercice 5

Q3.

Pour trouver la durée minimale du projet, il faut savoir quel est l'ordre optimal pour exécuter les tâches. Dans le cadre d'un arbre d'ordonnancement sans circuit, il faut pour cela utiliser **l'algorithme de Bellman**, après avoir appliqué l'algorithme de tri topologique sur le graphe.

Ici, passer d'une tâche à l'autre ne coûte rien. La durée totale du projet sera donc la somme des durées des tâches (Ici, 80 mois). Il suffit de suivre le tri topologique et on aura notre organisation optimale du projet.

Vu comment le graphe est agencé (pas de retour en arrière possible), on aura également pu utiliser l'algorithme de Dijkstra.

Q4.

Le chemin critique (celui de plus longue durée) est ici : D->c->b->e->i>-F.

EXERCICE 6

Dans ce cas, on utilisera le dernier algorithme du CM1 : Roy-Warshall-Floyd. On veut les distances maximales ici, donc on initialise toutes les distances entre noeuds non-reliés à -infini.
Voici le résultat obtenu :

Valeurs de m(ligne vers colonne), pred(ligne vers colonne)

	a	b	c	d	e
a	-	17,c	14,d	9,b	19,b
b	-	11,c	8,d	2,b	13,d
c	-	13,c	11,d	5,b	16,d
d	-	9,c	6,d	10,c	12,d
e	-	-	-	-	-

Pour minimiser, on mettrait les distances à +infini de base, et on inverserait le signe > lorsque l'on compare les différents chemins possibles entre eux. Cet algorithme peut calculer des chemins aussi bien minimaux que maximaux.