UCLouvain

LINFO1114
Projet 1
Rapport groupe 28

Distance du plus court chemin : Dijkstra, Bellman-Ford et Floyd-Warshall

---

Élèves :
Henri Pihet 66151900
Thibault Van Raemdonck
38542000

Enseignant :
Marco Saerens

20 décembre 2022

Table des matières

1 Introduction

Ce travail a été réalisé par Henri Pihet et Thibault Van Raemdonck dans le cadre du cours de Mathématique discrètes dispensé par le professeur Marco Saerens en l'année académique 2022-2023. Moodle, Github.

Il nous a été demandé d'implémenter trois algorithmes permettant de répondre à un problème bien connu dans le domaine de l'informatique : le problème du plus court chemin. Étant donné un graphe, orienté ou non, dont les arêtes sont pondérées, le but est de trouver la matrice de distance des plus courts chemins entre toutes les paires de noeuds de ce graphe. Les trois algorithmes implémentés sont ceux de Dijsktra, de Bellman-Ford et de Floyd-Warshall, décrits dans la section ci-dessous.

2 Rappels théoriques

Dans cette section nous allons décrire le fonctionnement des trois algorithmes implémentés, c'est-à-dire celui de Dijsktra, de Bellman-Ford et de Floyd-Warshall.

2.1 Algorithme de Dijsktra

Étant donné un noeud de départ a, nous voulons connaître la distance de ce noeud par rapport à un noeud d'arrivée z, et par extension à tous les autres noeuds du graphe. Nous allons donc associer à chaque noeud un label qui équivaut à la distance du noeud A par rapport à ce noeud. Lors de la première itération (k=0), $L_0(a)$ vaut donc 0 et tous les autres valent $+\infty$. L'ensemble des noeuds dont le label est déjà minimal se trouve dans S_k, donc à la première itération S_0 est l'ensemble vide. Soit v un noeud qui n'est pas encore dans S_k, alors $L_k(v)$ est la longueur du plus court chemin entre a et v en utilisant uniquement des noeuds déjà présents dans S_k. Lorsqu'un noeud u est ajouté à S_{k-1}, un chemin plus court entre a et v vaut soit $L_{k-1}(v)$ (c'est-à-dire que le noeud u ajouté n'améliore pas la distance entre a et v), ou bien $L_{k-1}(u)$ + w(u,v) (dans ce cas, on remplace la distance entre a et v par la distance entre a et u, à laquelle on rajoute la distance entre u et v). En d'autres termes, lorsqu'on rajoute un noeud u à S_k, on met à jour la valeur des noeuds qui n'y sont pas en passant par ce noeud u SI passer par ce noeud u diminue la distance entre a et les autres noeuds.

$$L_k(v) = min\{L_{k-1}(v), L_{k-1}(u) + w(u, v)\} \tag{1}$$

A la fin des itérations, c'est-à-dire lorsque tous les noeuds ont été placés dans S_k, le label associé à chaque noeud correspond à la distance la plus courte entre a et ce noeud, et donc $L^*(z)$ vaut bel et bien la plus courte distance entre a et z.

2.2 Algorithme de Bellman-Ford

Cet algorithme permet de calculer des plus courts chemins dans un circuit pondéré orienté. Contrairement à celui de Dijsktra, il autorise la présence d'arcs négatifs et donc potentiellement de circuits absorbants. Dans le cadre du cours, nous ne nous intéressons pas aux graphes contenant des valeurs négatives, c'est pourquoi nous renvoyons ici simplement une erreur lorsque l'algorithme détecte un arc pondéré négativement.
Étant donné un graphe G=(V,E), l'algorithme de Bellman-Ford calcule la distance la plus

courte entre un noeud source s $\in V$ à chaque sommet de G. A l'initialisation, on associe à chaque noeud u $\in V$ une valeur d[u,0] = 0 et un noeud prédécesseur pred[u] = null. La variable d[u,0] représente la distance entre le noeud s et ce noeud u à l'itération k, donc d[s,0] = 0. Nous allons ensuite lancer une boucle allant de k=1 jusque n-1, n étant le nombre de noeuds présents dans G. Pour chaque itération, nous regardons l'inégalité suivante pour chaque arête allant de u à v : si d[u,k-1] ajouté à la distance entre u et v (=w(u,v)) est inférieur à d[v,k-1], alors la valeur de d[v,k] est remplacée par d[u,k-1] + w(u,v), et pred[v] est remplacé par le noeud u. La formule associée est la suivante :

$$d[v,k] = min\{d[v,k-1], d[u,k-1] + w(u,v)\} \tag{2}$$

2.3 Algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall consiste à déterminer les distances des plus courts chemins entre toutes les paires de noeuds dans un graphe orienté et pondéré. Soit W^k la matrice des W^k_{ij}, le coût minimal du noeud i au noeud j en passant uniquement par des noeuds intermédiaires dans les {1,2,3,...,k} noeuds de G s'il en existe un, et $+\infty$ sinon. Choisissons maintenant un chemin p entre i et j de poids minimal dont les noeuds intermédiaires sont dans {1,2,3,...,k}. Alors, si la somme des poids des chemins entre i et k et k et j respectivement, dont les noeuds intermédiaires se trouvent dans {1,2,3,...,k-1} est inférieure au poids du chemin allant entre i et j passant par les mêmes noeuds intermédiaires, on remplace la valeur de W^k_{ij} de la manière suivante :

$$W^k_{ij} = min\{W^{k-1}_{ij}, W^{k-1}_{ik} + W^{k-1}_{kj}\} \tag{3}$$

En d'autres termes, cet algorithme consiste à choisir les noeuds un à un et de mettre à jour tous les plus courts chemins qui incluent le noeud choisi comme noeud intermédiaire du plus court chemin.

3 Calcul théorique et numérique de Dijkstra

La tableau 1 représente la solution du calcul du chemin le plus court entre le point a et tous les autres points. La dernier colonne vérifie que les résultats obtenus aux programmes de dijkstra, bellman ford et floyd warshall sont correcte. Le chemin le plus court entre a et j a été mise en valeur en rose dans le tableau. Le chemin le plus court est de 9 entre a et j.

TABLE 1 – Matrice des chemins les plus courts depuis le point a, Dijkstra

	L0	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10
a	0	0	0	0	0	0	0	0	0	0	0
b	inf	4	4	4	4	4	4	4	4	4	4
c	inf	4	4	4	4	4	4	4	4	4	4
d	inf	1	1	1	1	1	1	1	1	1	1
e	inf	inf	6	6	6	6	6	6	6	6	6
f	inf	inf	inf	7	4	4	4	4	4	4	4
g	inf	inf	8	8	8	8	8	8	8	8	8
h	inf	inf	inf	inf	inf	7	7	7	7	7	7
i	inf	inf	inf	inf	5	5	5	5	5	5	5
j	inf	inf	inf	inf	inf	inf	inf	10	9	9	9

4 Annexe

Fichiers source, historique des contributions et informations supplémentaires disponible sur Github.

main.py

```
import numpy as np
import csv
from algorithms.dijkstra import Dijkstra
from algorithms.bellman_ford import Bellman_Ford
from algorithms.floyd_warshall import Floyd_Warshall


def csv_to_matrix(filename):
    csv_file = open(filename, 'r')
    data = csv.reader(csv_file)

    # Create empty list to store the data
    data_matrix = []

    # Loop through the rows in the csv and append them to the
       list
    for row in data:
        data_matrix.append(row)

    # Convert the list to a numpy array
    data_matrix = np.array(data_matrix)

    # Convert each element of the array to a numpy float
    data_matrix = data_matrix.astype(float)

    # Return the resulting matrix
    return data_matrix
```

```
#funtion that help in the process of manually translating the .
   jpg source graph to the numpy matrix
def is_symmetric(matrix, N):
  for i in range(N):
    for j in range(N):
      if (matrix[i][j] != matrix[j][i]):
        #print("i j")
        #print(i)
        #print(j)
        #print(mat[i][j])
        #print(mat[j][j])
        return False
  return True

def check_matrix_equality(matrix_list):
    # initialize the flag
    flag_equal = True
    # check the equality of each matrix in the list
    for i in range(len(matrix_list)-1):
        if not np.array_equal(matrix_list[i], matrix_list[i+1]):
            flag_equal = False
            break
    # return the flag
    return flag_equal


def main(input_file):
    try:
        # Test converting csv to matrix
        matrix = csv_to_matrix(input_file)
        # Testing symmetry of the matrix
        if (is_symmetric(matrix, 10) != True):
            print("Error: Matrix is not symmetric")
            return -1
        else:
            print("Input Matrix:")
            print(matrix)

            print("Dijkstra output: ")
            dijkstra = Dijkstra(matrix)
            print(dijkstra)

            print("Belleman Ford output: ")
            bellman_ford = Bellman_Ford(matrix)
```

```
            print(bellman_ford)

            print("Floyd Warshall output: ")
            floyd_warshall = Floyd_Warshall(matrix)
            print(floyd_warshall)

            matrix_list=[dijkstra, bellman_ford, floyd_warshall]
            flag = check_matrix_equality(matrix_list)
            print("Output matrix are the same: ", flag)

            return 0

    except:
        # Return error message if converting to matrix failed
        # May be caused by a non square matrix, common error
        print("Error: converting csv to matrix")
        return -1


# Run
main("graph28.csv")
```

dijkstra.py

```
import numpy as np
import heapq


def Dijkstra(graph):
  # Initialize a matrix to store the distances of the shortest
     paths between all pairs of nodes
  distances = np.full(graph.shape, np.inf)

  # Loop over all nodes in the graph
  for start in range(graph.shape[0]):
    # Initialize distances and visited arrays
    distances[start] = np.full(graph.shape[0], np.inf)
    visited = np.zeros(graph.shape[0], bool)

    # Set the distance to the start node to be 0
    distances[start][start] = 0

    # Create a priority queue to store the nodes to be processed
    # This queue will be sorted by the distance to the node,
    # so that the node with the shortest distance will be
       processed first
    # It's my first time using it, not sure of what i'm doing.
       But it's said to be efficient
```

```
    queue = [(0, start)]

    # Loop until the priority queue is empty
    while queue:
      # Get the node with the shortest distance from the queue
      dist, node = heapq.heappop(queue)

      # Skip the node if it has already been visited
      if visited[node]:
        continue

      # Mark the node as visited
      visited[node] = True

      # Update the distances of the neighboring nodes
      for i, d in enumerate(graph[node]):
        if d > 0:  # Check if there is an edge between node and
            i
          new_dist = dist + d  # Calculate the distance to i
              through node
          if new_dist < distances[start][i]:  # Check if the new
               distance is shorter than the current distance
            distances[start][i] = new_dist  # Update the
                distance to i
            heapq.heappush(queue, (new_dist, i))  # Add i to the
                 queue to be processed

  # Return the distances matrix
  return distances
```

bellman_ford.py

```
import numpy as np

def Bellman_Ford(matrix):
    num_nodes = matrix.shape[0]

    # Initialize distance matrix with infinity values
    distances = np.full((num_nodes, num_nodes), np.inf)

    # Set distance between each node and itself to 0
    for i in range(num_nodes):
        distances[i, i] = 0

    # Iterate over the matrix and update distances
    for i in range(num_nodes):
        for j in range(num_nodes):
            if matrix[i, j] != np.inf:
```

```
                distances[i, j] = matrix[i, j]

    # Perform relaxation step num_nodes-1 times
    for k in range(num_nodes-1):
        for i in range(num_nodes):
            for j in range(num_nodes):
                distances[i, j] = min(distances[i, j], distances
                    [i, k] + distances[k, j])

    # Check if the graph has negative cycles by looking for
    # distances that can be further relaxed
    for i in range(num_nodes):
        for j in range(num_nodes):
            if distances[i, j] > distances[i, k] + distances[k,
                j]:
                raise ValueError('Graph contains negative cycle'
                    )

    return distances
```

floyd_warshall.py

```
import numpy as np

def Floyd_Warshall(graph):
    num_nodes = np.shape(graph)[0]
    # Initialize distance matrix
    dist = np.array(graph)
    # Add intermediate nodes to paths
    for k in range(num_nodes):
        for i in range(num_nodes):
            for j in range(num_nodes):
                dist[i, j] = min(dist[i, j], dist[i, k] + dist[k
                    , j])
    return dist
```

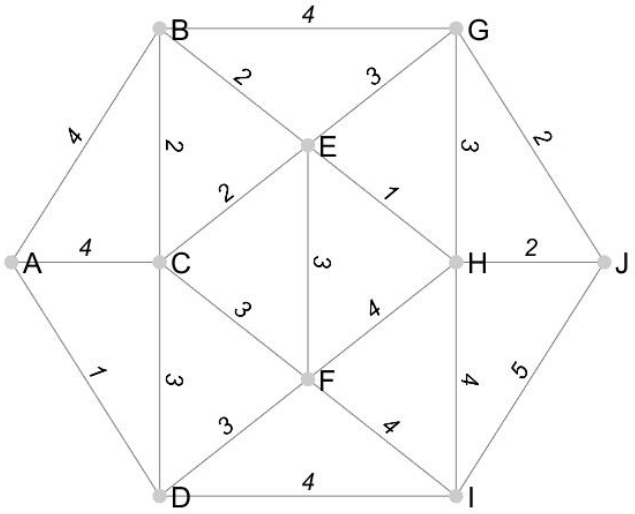

FIGURE 1 – Graphique des coûts

TABLE 2 – Matrice des coûts

0	4	4	1	inf	inf	inf	inf	inf	inf
4	0	2	inf	2	inf	4	inf	inf	inf
4	2	0	3	2	3	inf	inf	inf	inf
1	inf	3	0	inf	3	inf	inf	4	inf
inf	2	2	inf	0	3	3	1	inf	inf
inf	inf	3	3	3	0	inf	4	4	inf
inf	4	inf	inf	3	inf	0	3	inf	2
inf	inf	inf	inf	1	4	3	0	4	2
inf	inf	inf	4	inf	4	inf	4	0	5
inf	inf	inf	inf	inf	inf	2	2	5	0