SAE 4 : Notions Mathématiques

2023-03-06

Préambule

1. Les serveurs

Dans le cadre de notre projet informatique, nous allons devoir utiliser 13 machines, donc nous avons choisi de modéliser 13 serveurs sur lesquels vont se réaliser les tâches.

Nos serveurs se divisent en plusieurs opérations différentes: - 1. Un serveur web - 2. Un routeur - 3. Un serveur de gestion de base de données - 4. Deux postes de travail - 5. Deux serveurs DNS - 6. Un serveur DHCP - 7. Un serveur LDAP - 8. Un serveur d'authentification (Kerberos) - 9. Un serveur de logs - 10. Un serveur de surveillance du réseau - 11. Un serveur de fichiers

2. Les tâches

Nous allons modéliser 13 tâches qui vont tourner sur les différents serveurs.

[… Ajouter les tâches en question? …]

3. Définitions

Posons que nous modélisons notre problème sur une durée de 1000 millisecondes.

Nous attribuons à nos serveurs une valeur entière, représentant le temps total d'execution possible sur notre durée totale. Par exemple, une valeur de 60 signifie que sur 1000 millisecondes, le serveur peut effectuer 60 millisecondes de tâches.

Nous faisons de même pour les tâches, bien que cette fois la réprésentation est faite sur les temps d'execution nécessaire pour terminer la tâche.

4. Hypothèses

Afin de résoudre ce problème, nous allons utiliser un algorithme de méthode hongroise, qui permet de résoudre le problème en optimisant les tâches à faire tourner sur les serveurs.

Mathématiques

Le problème se base dans l'optimisation d'un flux de transport. Afin de résoudre ce problème, nous utilisons la méthode hongroise. Généralement, on peut écrire le problème du flux de transport sous la forme d'un graphe orienté sans cycles $G = (S, A, c)$, où S est l'ensemble des sommets, A l'ensemble des arcs, et c la fonction coût.

A l'aide des flots (arcs entre deux noeuds du graphe), symbolisés par f, qui représente le flux de transport, on peut écrire la fonction coût c sous la forme $c(f) = \sum_{(i,j)\in A} c_{ij} f_{ij}$. On cherche alors à minimiser cette fonction coût (dans notre cas).

La méthode hongroise reprend ces notions, mais doit utilser un graphe biparti. C'est à dire ou pour $G = (S, A)$, on peut créer 2 parties disjointes U et V tels que chaque arrête ait une extremité dans U et dans V.

Nous reprenons le problème des serveurs et tâches.

Supposons un graphe biparti ayant $2n$ sommets, qui affecte n tâches à n serveurs. On peut alors écrire le graphe sous la forme $G = (S, A, c)$, où :

$S = \{1, 2, ..., 2n\}$,

$A = \{(i, j) \in S \times S | i \in U, j \in V\}$,

$c = \{c_{ij} | (i, j) \in A\}$

Ce graphe peut prendre la forme d'une matrice carrée reprenant les coûts de chaque arc

$$M = \begin{pmatrix} c_{11} & c_{12} & ... & c_{1n} \\ c_{21} & c_{22} & ... & c_{2n} \\ ... & ... & ... & ... \\ c_{n1} & c_{n2} & ... & c_{nn} \end{pmatrix}.$$

Cette visualisation permet alors de traiter informatiquement le problème de la minimisation des coûts dans le réseau de transport.

Le programme python

Le programme Python : initialisation

Pour effectuer le programme, nous avons utilisé le langage Python, et nous avons utilisé la librairie Numpy pour simplifier la visualisation et aider sur certaines opérations.

Nous avons aussi importé la librairie random pour générer des nombres aléatoires, et sys pour pouvoir afficher des tableaux de grande taille.

```
import numpy as np
import sys
import random
np.set_printoptions(threshold=sys.maxsize) # Affichage de toutes les valeurs d'un array numpy
```

Le programme Python : création des matrices

Afin de modéliser le problème de manière simple, nous avons établi une série de fonctions permettant de générer des matrices selon les choix de l'utilisateur.

Nous permettons à l'utilisateur soit d'avoir une matrice générée automatiquement, en spécifiant la taille de la matrice et l'intervalle de valeurs, soit de créer une matrice en spécifiant les valeurs des serveurs et des tâches.

L'utilisateur peut aussi renseigner une matrice pré-configurée plus loin dans le programme (ceci n'est pas une fonction)

La fonction `generate_auto_matrix` permet de générer une matrice de taille variable avec des valeurs aléatoires comprises entre 1 et l'intervalle spécifié par l'utilisateur.

La fonction `make_matrice` permet de créer une matrice en utilisant un lien entre array de serveurs et array de tâches.

La méthode hongroise est particulièrement efficace sur des matrices carrées, donc nous avons créé une fonction `make_square_matrice` qui permet de créer une matrice carrée en ajoutant des valeurs à la fin de la matrice (ajout de serveurs 'virtuels' pour attribuer les tâches restantes, ou de tâches 'virtuelles' pour attribuer les serveurs restants)

```
def generate_auto_matrix(taillex, tailley, intervalle): # Génération automatique d'une matrice de taill
    matrice = np.random.randint(1, intervalle, size=(taillex, tailley)) # Nombre minimum = 1; Nombre ma
    return make_square_matrice(matrice)

def make_matrice(serveurs, taches): # Création d'une matrice en utilisant un lien entre array de serveu
    matrice = []
    for i in serveurs: # Pour chaque serveur
        ligne = []
        for j in taches: # Pour chaque tâche qui va être effectuée sur le serveur
            ligne.append(i * j) # On ajoute le produit de la valeur du serveur et de la valeur de la tâ
        matrice.append(ligne) # On ajoute la ligne à la matrice
    matrice = np.array(matrice) # On transforme la matrice en array numpy
    return make_square_matrice(matrice)

def make_square_matrice(matrice): # Création d'une matrice carrée en ajoutant des valeurs à la fin de l
    if (len(matrice[0]) > len(matrice)): # Si le nombre de tâches est supérieur au nombre de serveurs
        print("Attention, la matrice n'est pas carrée, il y a plus de tâches que de serveurs : on rajou
        for i in range(len(matrice[0]) - len(matrice)): # Pour chaque ligne à rajouter
            matrice = np.append(matrice, [[matrice.max() + 100] * len(matrice[0])], axis=0) # On rajout
    elif (len(matrice[0]) < len(matrice)): # Si le nombre de tâches est inférieur au nombre de serveurs
        print("Attention, la matrice n'est pas carrée, il y a plus de serveurs que de tâches : on rajou
        for i in range(len(matrice) - len(matrice[0])): # Pour chaque colonne à rajouter
            matrice = np.append(matrice, [[matrice.max() + 100] * len(matrice)], axis=1) # On rajoute u
    return matrice
```

Le programme Python : méthode hongroise partie 1

La première partie de la méthode hongroise consiste à soustraire à chaque ligne et à chaque colonne le minimum de la ligne/colonne.

A cette effet, nous avons 2 fonctions prennant en paramètre la matrice, et renvoyant la matrice modifiée (soit avec les lignes modifiées, soit avec les colonnes modifiées)

```
def soustraction_ligne(matrice): # Soustraction de la valeur minimale de chaque ligne à chaque élément
    index = -1
    for ligne in matrice: # Pour chaque ligne de la matrice
        index+=1
        minimum = min(ligne) # On récupère la valeur minimale de la ligne
        jindex = -1
        for colonne in ligne: # Pour chaque élément de la ligne (donc colonne)
            jindex+=1
            matrice[index][jindex] -= minimum # On soustrait la valeur minimale de la ligne à l'élément

    return matrice

def soustraction_colonne(matrice): # Soustraction de la valeur minimale de chaque colonne à chaque élém
```

```
    nbColonnes = len(matrice[0]) # Nombre de colonnes
    minColonne = []
    for i in range(nbColonnes): # Pour chaque colonne
        value = [min(i) for i in zip(*matrice)][i] # On récupère la valeur minimale de la colonne
        minColonne.append(value) # On ajoute la valeur minimale de la colonne à un array contenant chaq
    for ligne in matrice.T: # Pour chaque ligne de la matrice transposée (donc colonne de la matrice or
        for colonne in ligne: # Pour chaque élément de la ligne (donc élément de la colonne sur la matr
            colonne -= minColonne # On soustrait la valeur minimale de la colonne à l'élément de la col
    return matrice
```

Le programme Python : méthode hongroise partie 2

Dans un deuxième temps, il s'agit d'encadrer les 0 de la matrice qui sont pertinents, et de barrer les 0 qui sont sur des lignes ou colonnes ayant un 0 déjà encadré.

Cette fois encore, nous nous appuyons sur une fonction qui prend en paramètre la matrice, et renvoie deux listes, une contenant les coordonnées des zéros encadrés, et une contenant les coordonnées des zéros barrés.

```
def encadrer_zeros(matrice): # Encadrement des zéros de la matrice
    zero_encadre_indexes = [] # Les zéros encadrés
    zero_barre_indexes = [] # Les zéros barrés
    lcounter = 0 # Compteur de ligne
    ccounter = 0 # Compteur de colonne
    lerror = False # Le zéro trouvé est-il sur une ligne où un zéro est encadré
    cerror = False # Le zéro trouvé est-il sur une colonne où un zéro est encadré
    for ligne in matrice: # Parcours des lignes
        for colonne in ligne: # Pour chaque élément de la ligne (donc colonne)
            if (colonne == 0): # Si l'élément est un zéro
                if (len(zero_encadre_indexes) == 0): # Si c'est le premier zéro à être trouvé
                    zero_encadre_indexes.append([lcounter, ccounter]) # Le zéro peut être directement e
                else: # Pas le premier zéro à être trouvé
                    for i in zero_encadre_indexes: # Pour chacun des zéros déjà encadrés
                        if lcounter == i[0]: # Si le zéro actuel appartient à une ligne déjà encadrée
                            lerror = True # Erreur, le zéro doit être barré
                        if ccounter == i[1]: # Si le zéro actuel appartient à une colonne déjà encadrée
                            cerror = True # Erreur, le zéro doit être barré
                    if not lerror and not cerror: # Pas d'erreur, le zéro peut être encadré (il n'est p
                        zero_encadre_indexes.append([lcounter, ccounter])
                    else: # Erreur, le zéro est sur une ligne ou colonne possédant un zéro déjà encadré
                        zero_barre_indexes.append([lcounter, ccounter])
                    lerror = False # Reset de variable
                    cerror = False

            ccounter += 1 # Colonne suivante (élément suivant)
        ccounter = 0 # Reset de variable
        lcounter += 1 # Ligne suivante
    return zero_encadre_indexes, zero_barre_indexes # Retour des positions des zéros encadrés et zéros
```

Le programme Python : méthode hongroise partie 3

Dans un troisième temps, le programme va "marquer" les lignes et colonnes, dans l'ordre qui suit:

- 1. On marque les lignes n'ayant pas de zéro encadré

```
def marquer_lignes_sans_zero_encadre(matrice, indexes):
    lignes = []
    lignes_zero = []
    for i in range(len(matrice)): # Pour chaque ligne dans la matrice
        lignes.append(i) # On ajoute le numéro de ligne dans la liste des lignes
    for i in indexes: # Pour chaque index de zéro encadré
        lignes_zero.append(i[0]) # On rajoute la ligne de ce zéro encadré
    return list(set(lignes).difference(lignes_zero)) # On retourne la différence entre la liste des lig
```

- 2. On marque les colonnes avec un zéro barré sur une ligne marquée

```
def marquer_colonnes_avec_zero_barre_sur_ligne_marquee(indexes_barres, lignes_marques, colonnes_marquees
    for barre in indexes_barres: # Pour chaque zéro barré
        for ligne in lignes_marques: # Pour chaque ligne déjà marquée
            if (barre[0] == ligne) and (barre[0] not in colonnes_marquees): # Si la ligne du zéro barré
                colonnes_marquees.append(barre[1]) # Ajout colonne
    return colonnes_marquees # Return
```

- 3. On marque les lignes avec un zéro encadré sur une colonne marquée

```
def marquer_lignes_avec_zero_encadre_sur_colonnes_marquee(indexes_encadres, colonnes_marques, lignes_ma
    for encadre in indexes_encadres: # Pour chaque zéro encadré
        for colonne in colonnes_marques: # Pour chaque colonne marquée
            if encadre[1] == colonne and (encadre[0] not in lignes_marquees): # Si un zéro encadré appa
                lignes_marquees.append(encadre[0]) # On marque la ligne à laquelle appartient le zéro e
    return lignes_marquees # Return
```

La méthode hongroise nécessite aussi de répéter les actions `2.` et `3.` tant qu'il est possible de rajouter des lignes ou colonnes marquées. C'est pourquoi dans la boucle du programme principal (partie "boucle principale"), on appelle ces fonctions tant que l'on arrive à rajouter une ligne ou une colonne marquée.

Chacune de ces fonctions renvoie une liste contenant le numéro des lignes ou colonnes étant marquées.

Le programme Python : méthode hongroise partie 4

Dans un quatrième temps, il faut "griser" les lignes et colonnes de la matrice en respectant ce format:

- 1. On grise les lignes qui ne *sont pas* marquées
- 2. On grise les colonnes qui *sont* marquées

La fonction suivante permet de renvoyer deux listes, une contenant le numéro des lignes grisées et l'autre contenant le numéro des colonnes grisées.

```
def griser_les_lignes_et_colonnes(lignes_marquees, colonnes_marquees, matrice): # Grise les lignes et c
    lignes = []
    for ligne in range(len(matrice)): # Pour chaque ligne de la matrice
        lignes.append(ligne) # On ajoute le numéro de ligne dans la liste des lignes
    lignes_grisees = list(set(lignes).difference(lignes_marquees)) # Les lignes grisées sont l'inverse

    colonnes_grisee = colonnes_marquees # Les colonnes grisées sont les colonnes marquées
    return colonnes_grisee, lignes_grisees # Return
```

Nous employons aussi une autre fonction pour déterminer les colonnes non grisées (utile pour la suite du programme).

```
def colonnes_non_grisees(matrice, colonnes_grisees): # Trouve les colonnes non grisées
    colonnes = []
    ccounter = 0
    for ligne in matrice: # Pour chaque ligne de la matrice
        for colonne in ligne: # Pour chaque élément de la ligne
            colonnes.append(ccounter) # On ajoute le numéro de colonne dans la liste des colonnes
            ccounter += 1
    return list(set(colonnes).difference(colonnes_grisees)) # Les lignes non grisées sont l'inverse des
```

Le programme Python : méthode hongroise partie 5

Dans un cinquième temps, il faut trouver le plus petit élément non grisé de la matrice. Pour cela, on utilise la fonction suivante: Cette fonction trouvera la valeur minimale sur toutes les cases non grisées.

```
def find_min(matrice, lignes_marquees, colonnes_grisees): # Trouve le minimum de tous les éléments n'ét
    lcounter = 0 # Compteur de ligne
    ccounter = 0 # Compteur de colonne
    colonnes_non_grisee = colonnes_non_grisees(matrice, colonnes_grisees) # On récupère les colonnes no
    if (len(lignes_marquees) == 0): # Si aucune ligne n'est marquée
        minimum = 0 # Pas de minimum, la matrice est donc déjà optimale. On utilise zéro pour ne pas im
    else: # Sinon
        minimum = matrice[min(lignes_marquees)][min(colonnes_non_grisee)] # On initialise le minimum av
    for ligne in matrice: # Pour chaque ligne de la matrice
        for colonne in ligne: # Pour chaque élément de la ligne
            if (lcounter in lignes_marquees) and (ccounter not in colonnes_grisees): # Si ligne et colo
                if (colonne < minimum): # Et si l'élément actuel est plus petit que le minimum
                    minimum = colonne # On met à jour le minimum
            ccounter += 1 # Colonne suivante (élément suivant)
        ccounter = 0 # Reset de variable
        lcounter += 1 # Ligne suivante
    return minimum # On retourne le minimum
```

Il faut ensuite effectuer 2 opérations:

- 1. On soustrait le minimum trouvé à toutes les cases non grisées
- 2. On ajoute le minimum trouvé à toutes les cases doublement grisées

Nous assurons ces opérations avec la fonction suivante:

```
def soustraire_min_non_grise(matrice, colonnes_grisees, lignes_grisees, lignes_marquees): # Soustrait l
    min = find_min(matrice, lignes_marquees, colonnes_grisees) # On récupère le minimum des cases non g
    lcounter = 0 # Compteur de ligne
    ccounter = 0 # Compteur de colonne
    for ligne in matrice: # Pour chaque ligne de la matrice
        for colonne in ligne: # Pour chaque élément de la ligne
            if (lcounter in lignes_grisees) and (ccounter in colonnes_grisees): # Si ligne et colonnes
                matrice[lcounter][ccounter] += min # On ajoute le minimum
            if (lcounter in lignes_marquees) and (ccounter not in colonnes_grisees): # Si case non gris
```

```
                matrice[lcounter][ccounter] -= min # On soustrait le minimum
            ccounter += 1 # Colonne suivante (élément suivant)
        ccounter = 0 # Reset de variable
        lcounter += 1 # Ligne suivante
    return matrice # On retourne la matrice modifiée
```

Le programme Python : méthode hongroise partie 6

La méthode hongroise est maintenant terminée, il ne reste plus qu'à faire le choix des liens entre un serveur (en ligne) et une tâche (en colonne). Pour cela, on utilise la fonction suivante:

```
def choose_serveurs_taches(matrice): # On attribue les tâches aux serveurs
    links = [] # Liste contenant l'attribution entre serveurs et tâches
    erreur = False
    lcounter = 0 # Compteur de ligne
    ccounter = 0 # Compteur de colonne

    zeros = np.argwhere(matrice == 0) # On récupère les coordonnées des zéros de la matrice

    # Les lignes ayant un seul 0 peuvent directement être séléctionnés (seul choix possible)
    for ligne in matrice:
        if (np.count_nonzero(ligne == 0) == 1): # Chercher les lignes ayant seulement un zéro (ligne ==
            for colonne in ligne: # Itération sur la ligne
                if colonne == 0: # Si le nombre vaut 0
                    erreur = False # Reset de variable
                    for selected in links: # Pour chaque lien déjà établi
                        if (lcounter == selected[0]) or (ccounter == selected[1]): # Vérifier si la lig
                            erreur = True # Si c'est le cas, un 0 a déjà été sélectionné sur une même l
                    if not erreur: # Pas d'erreur
                        links.append([lcounter, ccounter]) # On peut choisir ce 0 comme lien entre serv
                ccounter += 1 # Augmentation compteur colonne
        ccounter = 0 # Reset de la colonne
        lcounter += 1 # Changement de ligne
    lcounter = 0 # Reset du compteur de ligne

    # Pour les lignes ayant plus de un 0, sélectionner celles dont les colonnes ont un seul 0
    for colonne in matrice.T: # Transoposition de la matrice pour en obtenir les colonnes
        if (np.count_nonzero(colonne == 0) == 1): # Chercher les colonnes ayant seulement un seul 0
            for valeur in colonne: # Pour chaque valeur dans la colonne
                if valeur == 0: # Si la valeur est égale à 0 (ne devrait être appelé qu'une fois)
                    erreur = False # Reset de variable
                    for selected in links: # Pour chaque lien déjà établi
                        if (ccounter == selected[0]) or (lcounter == selected[1]): # Vérifier si la lig
                            erreur = True # Si c'est le cas, un 0 a déjà été sélectionné sur une même l
                    if not erreur: # Pas d'erreur
                        links.append([ccounter, lcounter]) # /!\ inversion entre ccounter et lcounter v
                ccounter += 1 # Augmentation compteur colonne
        ccounter = 0 # Reset de la colonne
        lcounter += 1 # Changement de ligne
    lcounter = 0  # Reset du compteur de ligne

    lignes_barres = [] # Liste des lignes ayant déjà un 0 sélectionné (serveur-tâche déjà attribué)
```

```
    colonnes_barres = [] # Liste des colonnes ayant déjà un 0 sélectionné (serveur-tâche déjà attribuée
    
    for selected in links: # Pour chaque lien déjà établi entre serveur et tâche
        lignes_barres.append(selected[0]) # On ajoute la ligne et la colonne à la liste des lignes et c
        colonnes_barres.append(selected[1])
    
    # L'idée du code ci-après est de sélectionner pour chaque 0 restant celui qui a le moins de possibi
    lcounter = 0 # Compteur de ligne
    ccounter = 0 # Compteur de colonne
    for ligne in matrice: # Pour chaque ligne de la matrice
        counter = {}
        for colonne in ligne: # Pour chaque élément de la ligne
            if colonne == 0: # Si le nombre vaut 0
                count = 0
                for position in zeros.tolist(): # Pour chaque zéro de la matrice
                    if (ccounter == position[1]): # Si la colonne du zéro correspond à la colonne actue
                        if (position[1] not in colonnes_barres) and (position[0] not in lignes_barres):
                            if (lcounter != position[0]): # Si la ligne du zéro n'est pas la ligne actu
                                count += 1 # C'est un conflit (un 0 non sélectionné sur la même colonne
                        counter[ccounter] = count # On associe le nombre de conflits à la colonne pour 
            ccounter += 1 # Augmentation compteur colonne
        if len(counter) != 0: # Si il y a des conflits sur ce zéro
            if (lcounter not in lignes_barres) and (min(counter, key = counter.get) not in colonnes_bari
                if ([lcounter, min(counter, key = counter.get)] not in links): # Et si le lien n'est pa
                    links.append([lcounter, min(counter, key = counter.get)]) # On ajoute le lien
                if (lcounter not in lignes_barres): # Si la ligne n'est pas déjà sélectionnée
                    lignes_barres.append(lcounter) # On "sélectionne" la ligne
                if (min(counter, key = counter.get) not in colonnes_barres): # Si la colonne n'est pas 
                    colonnes_barres.append(min(counter, key = counter.get)) # On "sélectionne" la colon
        ccounter = 0 # Reset de la colonne
        lcounter += 1 # Changement de ligne
    
    # Dans ce dernier morceau de code, on sélectionne les valeurs restantes (pas forcément 0, sur les l
    lignes = []
    for i in range(len(matrice)): # Pour chaque ligne de la matrice
        lignes.append(i) # On ajoute le numéro de la ligne à la liste des lignes
    colonnes = [] # De même pour les colonnes
    for i in range(len(matrice[0])):
        colonnes.append(i)
    for couple in links: # Pour chaque lien déjà sélectionné
            if (couple[0] in lignes_barres): # Si la ligne du lien est déjà sélectionnée
                lignes.remove(couple[0]) # On retire la ligne de la liste des lignes à sélectionner
            if (couple[1] in colonnes_barres): # Si la colonne du lien est déjà sélectionnée
                colonnes.remove(couple[1]) # On retire la colonne de la liste des colonnes à sélectionn
    while len(lignes) != 0 and len(colonnes) != 0: # Tant qu'il reste des lignes et des colonnes à séle
        elements = {}
        for ligne in lignes: # On étudie ligne à ligne
            for colonne in colonnes: # Pour chaque élément de la ligne restante
                elements[matrice[ligne][colonne]] = [ligne,colonne] # On enregistre dans un dictionnair
        res = elements[min(elements)] # On sélectionne la valeur la plus faible pour la ligne étudiée
        links.append(res) # On ajoute le lien minimum
        lignes.remove(res[0]) # On retire la ligne et la colonne de la liste des lignes et colonnes à s
        colonnes.remove(res[1])
```

```
    return links # On a fait tous les liens de manière optimale, on retourne la liste des liens
```

Cette fonction paraît longue, mais son fonctionnement est relativement simple et effectue 4 opérations:

- 1. Le programme sélectionne directement tous les 0 uniques sur une ligne (comme ils sont seuls, ce sont d'office le meilleur choix entre un serveur et une tâche)
- 2. Le programme sélectionne ensuite tous les 0 uniques sur une colonne
- 3. Le programme va ensuite chercher parmis les 0 restant, sur chaque ligne, ceux ayant le moins de conflits. C'est à dire ceux qui ont le moins de 0 sur la même colonne sur des lignes qui n'ont pas déjà été sélectionnés.
- 4. Dernièrement, le programme sélectionne les valeurs restantes (pas forcément 0, sur les lignes et colonnes qui n'ont pas encore été sélectionnées), en choisissant les valeurs optimales (les plus faibles) une à une jusqu'à ce que chaque serveur ait une tâche.

Le programme Python : impression des résultats

Maintenant que tous les liens sont établis, il ne reste plus qu'à imprimer les résultats. Pour cela, on utilise la fonction suivante:

```
def print_serveurs_taches(links, size_serveurs, matrice_originale): # Fonction qui affiche les liens se
    cost = 0
    for link in links: # Pour chaque lien
        if (int(link[0]) > size_serveurs): # Comme on a parfois ajouté des serveurs (matrice non carrée)
            cost += int(matrice_originale[int(link[0])][int(link[1])]) # Si c'est le cas, on ajoute le
            print("La serveur "+ str(int(link[0] - size_serveurs)) + " prend la tache " + str(int(link[
        else: # Sinon, on affiche le lien sur un serveur réel
            cost += int(matrice_originale[int(link[0])][int(link[1])])
            print("Le serveur "+ str(int(link[0])) + " prend la tache " + str(int(link[1])) + " avec un
    print("Avec un coût total de " + str(cost)) # On affiche le coût total
```

La seule subtilité ici c'est qu'on vérifie si on a plus de liens que de serveurs réels (cas où on avait plus de tâches que de serveurs), auquel cas on attribue les tâches des serveurs "virtuels" aux serveurs réels.

Le programme Python : boucle principale

Toute la partie précédente regroupait des fonctions python, qui ne font pas grand chose avant d'être appelées. La partie suivante est la boucle principale du programme, qui va appeler les fonctions, permettre à l'utilisateur de choisir ses options, puis d'afficher les résultats.

```
# Sélection de méthode
#type = int(input("Quelle méthode voulez-vous :\n 1) Matrice auto-générée\n 2) Matrice de lien serveurs
type = 1
if type == 1: # Cas ou la matrice est auto-générée aléatoirement en fonction de la taille voulue et de
    #serveur_count = int(input("Quelle est la taille horizontale de la matrice que vous voulez (entrez
    #taches_count = int(input("Quelle est la taille verticale de la matrice que vous voulez (entrez un
    #intervalle = int(input("Quelle est le nombre maximal dans la matrice (entrez un nombre) : "))
    serveur_count = 7
    taches_count = 10
    intervalle = 50
```

```
    matrice_originale = generate_auto_matrix(serveur_count, taches_count, intervalle)
    print("La matrice auto-générée est : \n" + str(matrice_originale))
elif type == 2: # Cas ou la matrice est générée en fonction des valeurs entrées par l'utilisateur pour
    serveurs = []
    serveur_count = int(input("Combien y a-t-il de serveurs (entrez un nombre N >= 1) ? \n"))
    for i in range(0, serveur_count):
       valeur = int(input("Valeur du serveur "+ str(i) +" : "))
       while valeur <= 0:
           print("La valeur doit être comprise entre 1 et N >= 1")
           valeur = int(input("Valeur du serveur "+ str(i) +" : "))
       serveurs.append(valeur)
    taches = []
    taches_count = int(input("Combien y a-t-il de tâches (entrez un nombre N >= 1) ? \n"))
    for i in range(0, taches_count):
        valeur = int(input("Valeur de la tâche "+ str(i) +" : "))
        while valeur <= 0:
            print("La valeur doit être comprise entre 1 et N >= 1")
            valeur = int(input("Valeur de la tâche "+ str(i) +" : "))
        taches.append(valeur)
    for i in range(serveur_count):
        serveurs.append(random.randint(1, 80000))
    for i in range(taches_count):
        taches.append(random.randint(1, 80000))
    matrice_originale = make_matrice(serveurs, taches)
    print("La matrice liée tâches-serveurs est : \n" + str(matrice_originale))
else: # Cas ou la matrice est pré-configurée
    matrice_originale = np.array([
        [100, 300, 140, 250, 120,  90,  40, 120, 130, 170],
        [ 50, 150,  70, 125,  60,  45,  20,  60,  65,  85],
        [ 70, 210,  98, 175,  84,  63,  28,  84,  91, 119],
        [150, 450, 210, 375, 180, 135,  60, 180, 195, 255],
        [ 60, 180,  84, 150,  72,  54,  24,  72,  78, 102],
        [ 90, 270, 126, 225, 108,  81,  36, 108, 117, 153]
    ])

# Sélection de mode d'affichage
#mode = int(input("Voulez-vous le mode :\n 1) Pas à pas (arrêt entre chaque opération)\n 2) Résultat (c
```

```
## Attention, la matrice n'est pas carrée, il y a plus de tâches que de serveurs : on rajoute des valeu
## La matrice auto-générée est :
## [[  2  35  30  15  22  17  23  28  43   5]
##  [ 44  22   4   6  38  34  25  28  42  45]
##  [ 34  28  39  19  19  45  27  29   9  22]
##  [ 34  17  44  42  14  44   1  42   7  32]
##  [ 25  42   8  38  49  49  25  27   1  19]
##  [ 39  12  46  33  36  34  38  26  19  24]
##  [  4  21  14  27  39  38  36  47  36  37]
##  [149 149 149 149 149 149 149 149 149 149]
##  [249 249 249 249 249 249 249 249 249 249]
##  [349 349 349 349 349 349 349 349 349 349]]
```

```
mode = 2
matrice = soustraction_colonne(soustraction_ligne(np.copy(matrice_originale)))
if (mode == 1):
    print("Après soustraction en lignes et colonnes, la matrice est : \n" + str(matrice))
    input("Appuyez sur entrer pour continuer\n\n")

zero_encadres, zero_barres = encadrer_zeros(matrice)
if (mode == 1):
    print("Les zéros encadrés : " + str(zero_encadres))
    print("Les zéros barrés : " + str(zero_barres))
    input("Appuyez sur entrer pour continuer\n\n")


lignes_marquees = marquer_lignes_sans_zero_encadre(matrice, zero_encadres)
change = True
lignes_marquees_before = []
colonnes_marquees_before = []
while change:
    colonnes_marquees = []

    colonnes_marquees = marquer_colonnes_avec_zero_barre_sur_ligne_marquee(zero_barres, lignes_marquees
    if (mode == 1):
        print("Les colonnes marquées : " + str(colonnes_marquees))
        input("Appuyez sur entrer pour continuer\n\n")

    lignes_marquees = marquer_lignes_avec_zero_encadre_sur_colonnes_marquee(zero_encadres, colonnes_marq
    if (mode == 1):
        print("Les lignes marquées : " + str(lignes_marquees))
        input("Appuyez sur entrer pour continuer\n\n")

    if (len(lignes_marquees_before) == len(lignes_marquees)) and (len(colonnes_marquees_before) == len(c
        change = False
    else:
        lignes_marquees_before = lignes_marquees.copy()
        colonnes_marquees_before = colonnes_marquees.copy()

colonnes_grisees, lignes_grisees = griser_les_lignes_et_colonnes(lignes_marquees, colonnes_marquees, ma
if (mode == 1):
    print("Les colonnes grisées : " + str(colonnes_grisees))
    print("Les lignes grisées : " + str(lignes_grisees))
    input("Appuyez sur entrer pour continuer\n\n")


matrice_finale = soustraire_min_non_grise(matrice, colonnes_grisees, lignes_grisees, lignes_marquees)
print("La matrice finale : \n" + str(matrice_finale))
```

```
## La matrice finale :
## [[ 0 30 25 10 17 12 18 23 41  0]
##  [43 18  0  2 34 30 21 24 41 41]
##  [25 16 27  7  7 33 15 17  0 10]
##  [36 16 43 41 13 43  0 41  9 31]
##  [24 38  4 34 45 45 21 23  0 15]
##  [30  0 34 21 24 22 26 14 10 12]
```

```
##  [ 0 14  7 20 32 31 29 40 32 30]
##  [ 3  0  0  0  0  0  0  0  3  0]
##  [ 3  0  0  0  0  0  0  0  3  0]
##  [ 3  0  0  0  0  0  0  0  3  0]]
```

```
links = choose_serveurs_taches(matrice_finale)
print_serveurs_taches(links, serveur_count, matrice_originale)
```

```
## Le serveur 1 prend la tache 2 avec un temps de 4
## Le serveur 2 prend la tache 8 avec un temps de 9
## Le serveur 3 prend la tache 6 avec un temps de 1
## Le serveur 5 prend la tache 1 avec un temps de 12
## Le serveur 6 prend la tache 0 avec un temps de 4
## La serveur 2 prend la tache 9 avec un temps de 349
## La serveur 1 prend la tache 7 avec un temps de 249
## Le serveur 7 prend la tache 5 avec un temps de 149
## Le serveur 0 prend la tache 3 avec un temps de 15
## Le serveur 4 prend la tache 4 avec un temps de 49
## Avec un coût total de 841
```

Ici encore, rien de très compliqué. On demande à l'utilisateur le type de matrice qu'il veut générer (je rappelle : matrice auto-générée, matrice de lien serveurs-tâches, ou matrice pré-configurée).

Le programme se charge de faire les appels aux fonctions utiles et renvoit la matrice.

Ensuite, on demande à l'utilisateur le mode d'affichage qu'il veut (pas à pas ou résultat direct).

Enfin, on lance le programme, qui effectue la méthode hongroise et éventuellement imprime les résultats de chaque étape si le mode `pas à pas` est choisi.

Enfin, peut importe le mode choisi, le programme affiche la matrice finale (après méthode hongroise), et les liens entre serveurs et tâches.

Conclusion

Ce programme permet de résoudre le problème de l'attribution de tâches à des serveurs, en utilisant la méthode hongroise.

Les fonctions ne sont pas tout à fait optimisées, et utilisent nottament des boucles imbriquées, ce qui rallonge considérablement le temps d'execution du programme. Il reste donc un travail d'optimisation à effectuer pour que le programme soit plus rapide.

Cela étant dit, nous avons testé ce programme sur des matrices de 500x500, qui a réussi à produire une solution en 23 secondes, ce qui est tout à fait acceptable dans notre cas.

Afin de mener ce projet, nous avons commencé par modéliser un problème plus simple, avec 5 serveurs et 5 tâches, afin de comprendre le fonctionnement de la méthode hongroise, en écrivant les étapes (fonctions python) une à une et en s'assurant que chaque étape fonctionne correctement.

Lorsque la méthode hongroise était implémentée sur cette version simplifiée, nous avons étendu le champ d'application sur des matrices non carrées, car il n'était pas initialement prévu pour ce type de matrice.

Cependant, avec un peu de réflexion, nous avons pu adapter la méthode hongroise pour qu'elle fonctionne sur des matrices non carrées, et nous avons pu résoudre le problème de l'attribution de tâches à des serveurs.