TP6 – RECHERCHE DU PLUS COURT CHEMIN

OBJECTIFS :

Les applications utilisant les signaux des satellites GPS sont en plein essor. Elles ne se limitent pas aux véhicules automobiles, puisqu'elles sont par exemple intégrées désormais dans certains appareils photographiques. Cependant, dans le cas des véhicules, s'y ajoute le calcul de l'itinéraire optimal entre deux lieux géographiques optimal au regard de critères tels que distance, temps ou coût total.

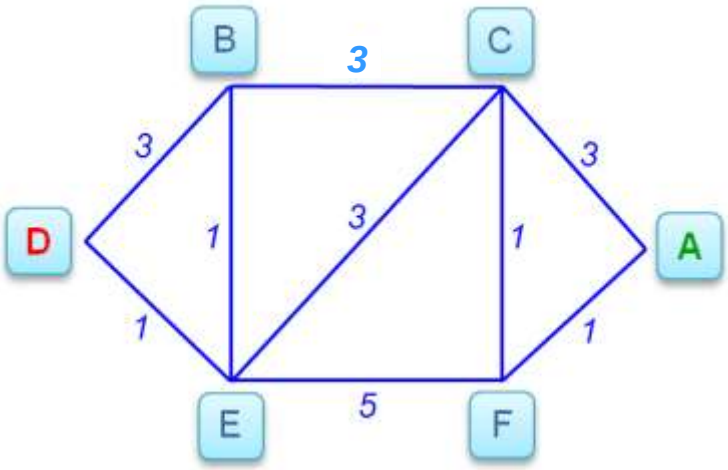
Ce calcul fait appel à la théorie des graphes et utilise différents algorithmes dont celui de Dijkstra, qui est un algorithme du type parcours en largeur ou BFS (Breadth First Search). À la différence d'un algorithme DFS (Depth First Search) où l'on explore un sommet adjacent à celui de départ, puis un autre adjacent au précédent, et ainsi de suite jusqu'à se retrouver bloqué et revenir en arrière, on examine ici dès le départ tous les sommets adjacents au premier.

1. DECOUVERTE DE L'ALGORITHME

Quel est le plus court chemin pour aller de D à A ?

On sélectionne successivement les villes en commençant par celle de départ, D. C'est l'algorithme qui impose l'ordre de sélection. Dans le tableau que l'on va construire, 6(E) dans la colonne F par exemple signifie que provisoirement, la plus courte distance entre D et F est de 6 km au total en passant par E. Lorsque la ville F sera sélectionnée, le provisoire deviendra définitif (case sur fond gris qui n'est plus modifiée par la suite).

Le travail peut être fait sur tableur : une nouvelle ligne du tableau est créée par copier-coller à chaque étape.



Partant de D :	B est a une distance totale de 3 km : on note 3(D) dans la colonne B. E est a une distance totale d' 1 km : on note 1(D) dans la colonne E. Pour les autres villes, pas encore visitées, on note ≪ inf ≫ pour infini.

La prochaine ville sélectionnée est E (plus courte distance : 1 km).

D	B	E	C	F	A	Prochaine ville sélectionnée
	3 (D)	1(D)	inf	inf	inf	E

Partant de D, via E :	B est a une distance totale de 2 km (1+1) : on note 2(E) dans la colonne B, car 2 est plus petit que 3 déjà présent. ***Si le résultat est plus grand ou égal, on ne change rien.*** Idem en l'absence de route directe entre E et B. C est à une distance totale de 4 km (1+3) : on note 4 (E) dans cette colonne. F est à une distance totale de 6 km (1+5) : on note 6 (E) dans la colonne F. A n'est pas directement reliée a E : on ne change rien dans la colonne A.

	E est à une distance totale d' 1 km : on note 1(D) dans la colonne E. Pour les autres villes, pas encore visitées, on note ≪ inf ≫ pour infini.

La prochaine ville sélectionnée est B (plus courte distance : 2 km).

D	B	E	C	F	A	Prochaine ville sélectionnée
	2 (E)	1(D)	4 (E)	6 (E)	inf	B

Q1.Poursuivre l'algorithme tant que la ville d'arrivée n'est pas sélectionnée. (sur feuille ou sur un tableur)

On obtient finalement :

D	B	E	C	F	A	Prochaine ville sélectionnée

Puisqu'on a mémorisé à chaque fois la ville précédente, on remonte à la ville de départ par le plus court chemin. Dans la colonne A, on voit qu'on vient de F. On regarde dans la colonne F : on arrive en F par C. Etc. → On obtient AFCED.

Le plus court chemin est donc DECFA avec une distance totale de 6 km (résultat direct, voir colonne A, puisqu'on mémorise à chaque fois la distance totale).

L'algorithme de Djjkstra est bien un algorithme de parcours en largeur du graphe : les villes sélectionnées sont d'abord D, puis E et B, puis C et F, puis A.

La structure du problème qui peut être retenue pour l'implémentation de l'algorithme est matricielle. On l'appelle matrice d'adjacence.

Dans celle-ci, a_{ij} représente le coefficient du lien reliant le noeud i au noeud j.

Pour deux noeuds non reliés, on choisit $a_{ij}=0$.

« i » est l'indice des lignes, « j » celui des colonnes.

$$\begin{bmatrix} a_{11} & a_{12} & a_{1k} \\ a_{21} & a_{22} & a_{2k} \\ & & \\ a_{k1} & & a_{kk} \end{bmatrix}$$

aij = v(i, j) : valeur du lien de i à j

La matrice est symétrique car les liens ne sont pas orientés.

Elle est carrée, de dimension k×k.

k est le nombre total de noeuds.

Dans l'exemple d'un GPS, aij représente le coefficient (distance, durée, ou coût) de la route reliant la ville i à la ville j. Pour deux villes non reliées entre elles par une route directe, on a choisi aij=0 .

D'un point de vue structure de données, on peut voir cette matrice comme un tableau à deux indices ou comme une liste de listes.

On cherche arbitrairement à déterminer le chemin le plus court du noeud 1 au noeud k (correspondant au nombre total de noeuds).

2. IMPLEMENTATION DE L'ALGORITHME

Q2.Ecrire la matrice correspondant à l'exemple précédent, en considérant que les villes D, B, E, C, F, A correspondent respectivement aux lignes et colonnes 0, 1, 2, 3, 4, 5 de la matrice.

Dans la liste d'adjacence, à la suite de chaque ville, on indique celles qui lui sont adjacentes (villes reliées par une route directe)

D	→	B, E
B	→	D, E, C
E	→	D, B , C, F
C	→	B, E, F, A
F	→	E, C, A
A	→	C, F

Remarque : avec la matrice d'adjacence, la recherche de l'existence d'une route directe entre deux villes est immédiate, alors que lorsqu'on veut trouver si une ville a une voisine, c'est avec la liste d'adjacence que la recherche est la plus rapide.

Q3.Créer la matrice adjacente sous Python et le nombre de villes Nvilles

Q4. Puis, créer la liste de Djikstra DIJ [[]] qui permet de mémoriser les données du tableau

 # Pour chaque ville, la distance totale est préalablement initialisée à une grande valeur (1000000 « ∞ »), la ville précédente à 'X', et non choisi pour l'itinéraire 'N'

Q5.Ecrire l'algorithme de Dijkstra en Python : il permet de modifier la liste DIJ[]

*On utilise les variables **ville_selectionnée et dist_intermédiaire**, initialisées à 0 pour calculer la **distance_totale** entre 2 points.*

*On parcourt alors toutes les villes, **tant que** la ville sélectionnée est différente du nb de la ville finale.*

***A chaque étape**, on affiche pour **chaque ville**, **sauf celle de départ**, la distance totale pour la rejoindre, la ville précédente, et 'O' si ladite ville est sélectionnée ou l'a déjà été, 'N' sinon.*

S'il existe un lien entre la ville sélectionnée et la ville n, et si la distance totale est inférieure à la distance en cours entre le ville n et la ville 1
On réaffecte la distance en cours dans la matrice DIJ
On réaffecte est la ville sélectionnée dans la matrice DIJ

Si la distance en cours entre la ville n et la ville 1 est inférieure au minimum des distances précédemment calculées :
le minimum est actualisé ainsi que le chemin (qui n'a pas besoin de passer par toutes les villes intermédiaires)

*Utiliser une variable **pville_select** pour affecter le numéro de la prochaine ville sélectionnée …*

Rappelons que les villes sont numérotées D, B, E, C, F, A ↔ 0, 1, 2, 3, 4,5. Le dernier affichage correspond donc exactement à ce qu'on obtient en déroulant l'algorithme manuellement.

Q6.Reconstruire enfin la liste correspondant au chemin en partant de la ville d'arrivée et l'afficher

*Vous pouvez l'afficher en sens inverse (puis en sens directe avec **reverse** ou autrement)*

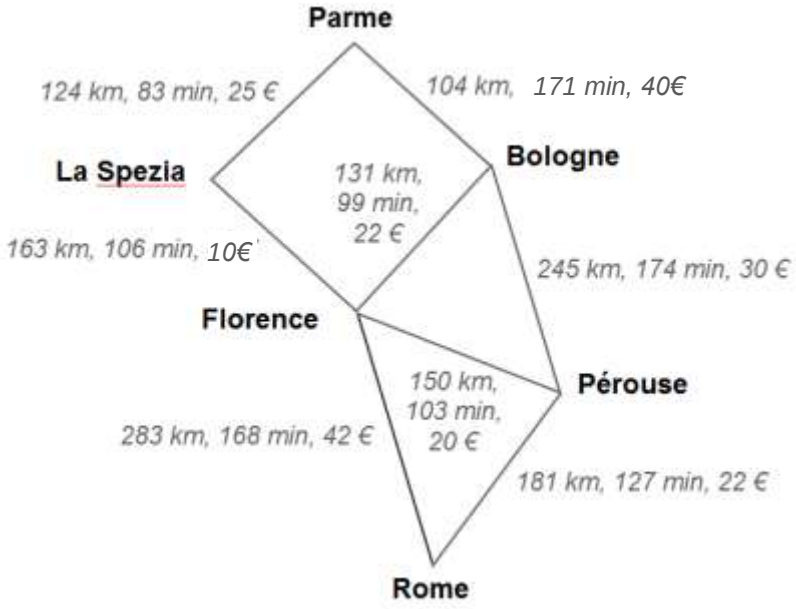
*Q7.Définir cet algorithme (Q4 à Q6) comme une fonction **Djikstra(Nnoeuds, m_adajcente**)*

Q8.Evaluer la complexité de votre algorithme (temporelle et mathématique)

3. MISE EN SITUATION

En général, on recherche le chemin le moins long, le plus rapide, ou le moins cher …

Tous les chemins mènent à Rome, mais pour un parmesan (un habitant de Parme !), quel est le moins long ? Le plus rapide ? Le moins cher ? Est-ce le même chemin dans les trois cas ?



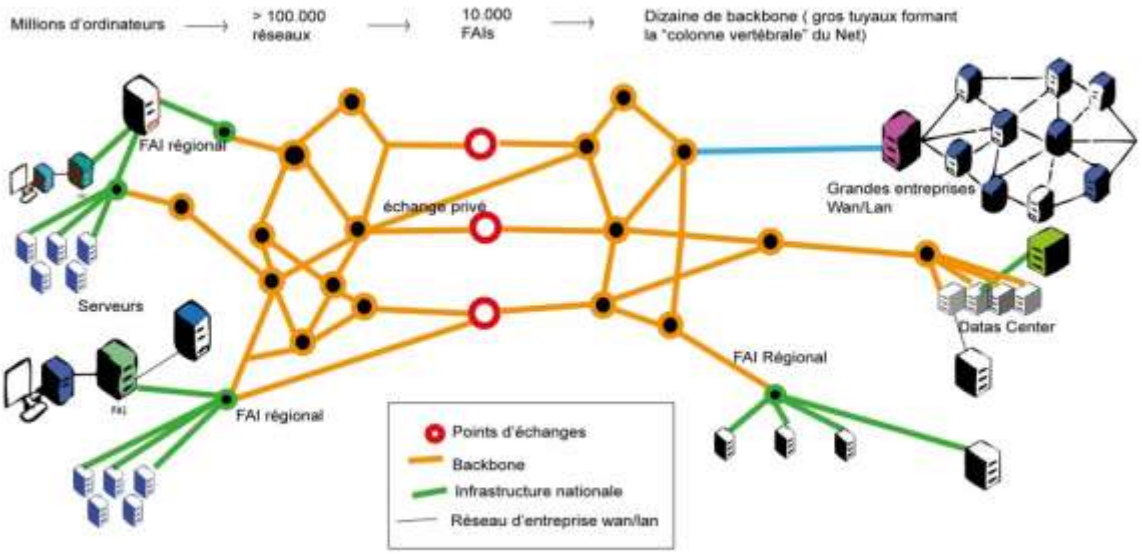
Q9.Mettre en place la liste des données, puis, en utilisant votre algorithme, donner les trois solutions qui permettent :

1. *D'avoir la plus courte distance entre Parme et Rome*
2. *La plus courte durée*
3. *Le moindre cout*

4. AUTRE UTILISATION : LE ROUTAGE

Cet algorithme est également utilisé pour le routage lors des connexions Internet (***voir fichier pdf ressource ISN)***

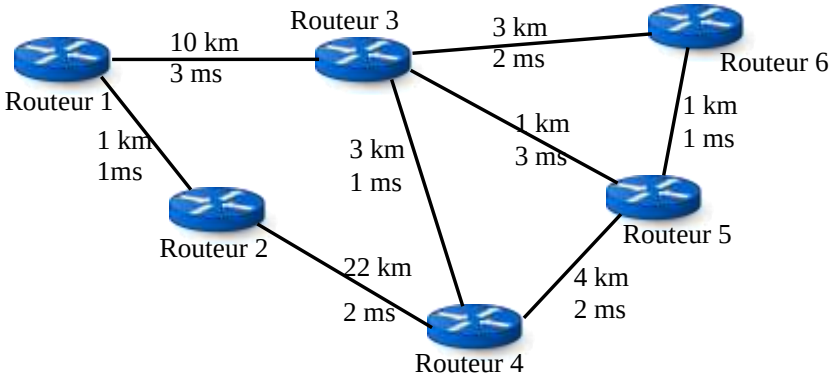
En effet, lorsqu'une connexion point à point est établie sur internet, les routeurs ont pour tâche principale d'aiguiller correctement l'information à travers le réseau.



Les trames passent par une multitude de routeurs avant d'arriver à destination. Ce sont les nœuds du réseau. Il y a donc une multitude de chemins possibles. Si un routeur tombe en panne, il reste plusieurs possibilités d'itinéraires. C'est une des raisons qui rend Internet difficilement contrôlable et presque indestructible.

Un réseau peut se représenter à l'aide d'un graphe.

Par exemple :



Chaque **nœud** (sommet) est un routeur.

Chaque **lien** (arête) est le support qui véhicule l'information. Ici, on les pondère avec la distance à parcourir, cela aurait pu être autre chose. Par exemple, le **poids** du lien entre les routeur 3 et 5 est de 1 km. C'est aussi un **graphe pondéré**.

On considère aussi que les liens ne sont pas orientées. Pour l'algorithme de Djikstra, la pondération doit être positive ou nulle.

Il est aussi possible de représenter les étapes de l'algorithme par un tableau :

1	2	3	4	5	6	Routeurs sélectionnés
(0,_)	(1,1)	(10,1)	(∞,_)	(∞,_)	(∞,_)	**2**
	(1,1)	(10,1)	(23,2)	(∞,_)	(∞,_)	**3**
		(10,1)	(13,3)	(11,3)	(13,3)	**5**
		(10,1)		(11,3)	(12,5)	**6**

Q10. On souhaite trouver le plus court chemin pour aller du routeur 6 au routeur 2. Construire un tableau montrant les différentes étapes de l'algorithme de Djikstra comme ci-dessus.

1	2	3	4	5	6	Routeurs sélectionnés
(∞,_)	(∞,_)	(3,6)	(∞,_)	(1,6)	(0,_)	**5**

Q11. On souhaite trouver le plus court chemin pour aller du routeur 6 au routeur 2 en ne regardant que les temps d'acheminement. Construire un tableau montrant les différentes étapes de l'algorithme de Djikstra.

Q12. Comparer l'algorithme suivant avec votre programme Djikstra puis coder le sur Python. Retrouver le plus court chemin pour aller du routeur 6 au routeur 2 en ne regardant que les temps d'acheminement.

Pseudo code de Dijkstra …

Pseudo Code

```
//question 4
liste Dijkstra ← [] ;    // Création d'une liste vide
Pour i=1 à N_villes    // Déclaration d'itérations permettant d'initialiser la liste
        {
        Dijkstra[i]← [∞,"x","N"] ;// Pour chaque nœud i : distance infinie, prédécesseur non connu,
        }      non choisi pour l'itinéraire
////////////////////////////////////////////////////////////////////

entier Ville_select ← 1 ;  // Premier ville sélectionné
entier d_inter ← 0 ;     // Initialisation de la distance intermédiaire
entier d_totale,d ;    // Déclaration distance totale et distance d'un lien
entier pVille_select ;    // Déclaration du prochain ville à sélectionner
Tant que Ville_select ≠ N_villes                    // Déclaration d'itérations pour rechercher le plus court
        {                                           chemin jusqu'au dernier ville
        entier Minimum← ∞ ;  // Initialisation du minimum
        Pour n=2 à N_villes// Déclaration d'itérations pour procéder ville par ville
 {
                Si Dijkstra[n][3]=="N"// Si la ville n n'est pas encore choisie pour l'itinéraire
                        {
                        d←m_adjac[Ville_select][n]              // d reçoit la distance de ville sélectionné
                                                                a ville n
                        d_totale←d_inter + d                    // d_totale reçoit la distance d + la
                                                                distance intermédiaire
                        Si d≠ 0 et d_totale < Dijkstra[n][1] // S'il existe un lien entre la ville sélectionnée et
                                                                la ville n, et si la distance totale est
                                                                inférieure à la distance
                                {                               en cours entre le ville n et la ville 1
                                Dijkstra[n][1]←d_totale          // La distance en cours reçoit la
                                                                        distance totale
                                Dijkstra[n][2]←Ville_select  // Le prédecesseur est la ville sélectionnée
                                }
                        Si Dijkstra[n][1]<Minimum   // Si la distance en cours entre la ville
                                                                n et la ville 1 est inférieure au
                                {       minimum en cours
                                Minimum←Dijkstra[n][1] // Le minimum est actualisé
                                pVille_select←n     // La prochaine ville à visiter est n
                                }
                        }
                }
        Ville_select←pVille_select     // Le ville sélectionnée devient celle qui  présentait la plus petite distance
        Dijkstra[Ville_select][3]←"O"    // La ville sélectionnée est sur la route recherchée
        d_inter←Dijlstra[Ville_select][1]    // La distance intermédiaire est mise à jour
 }
```

PYTHO N