Structures Intermédiaires
TP04

Exercice 1.)

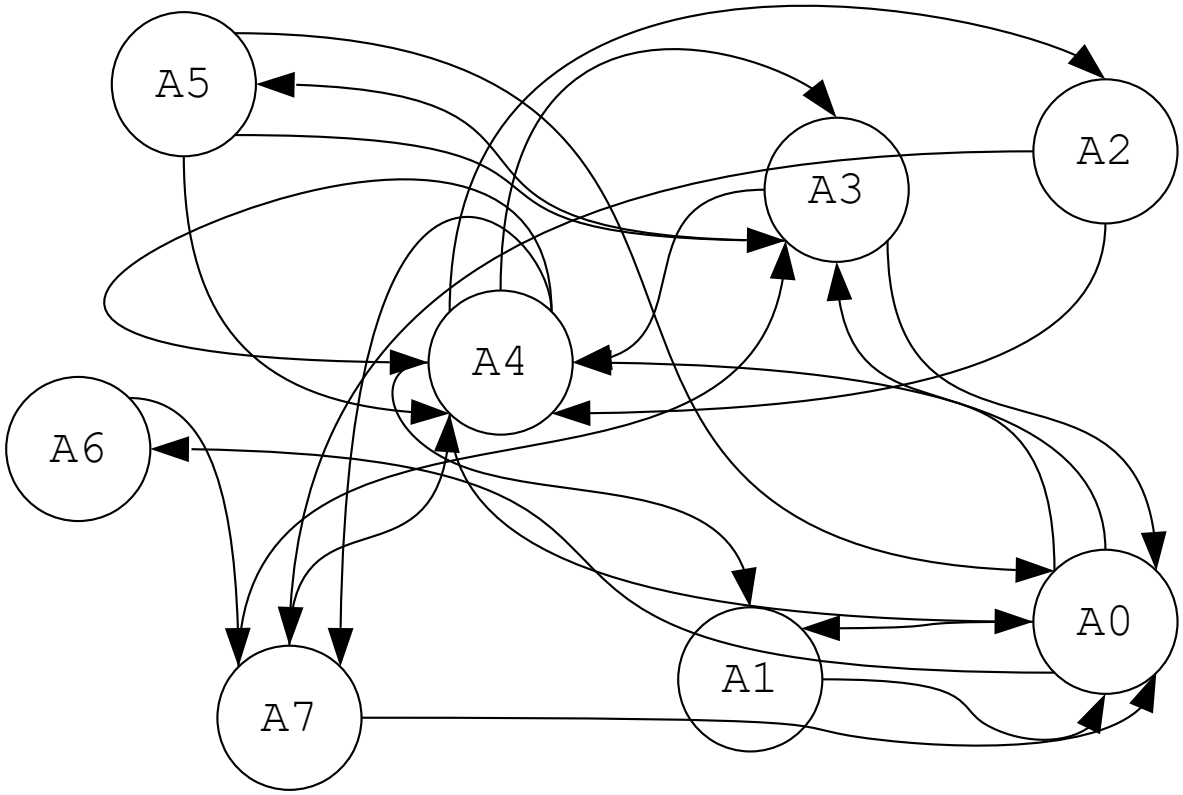
1.1.) Qu'est ce que c'est qu'un aéroport d'un point de vue informatique ? Eh bien, un aéroport possède un code IATA, une ville et un pays (du moins dans le cadre de ce cours). Nous aurons donc une structure de la forme suivante,

```
struct aeroport {
     uint IATA;
     string ville, pays;
}
```

1.2.) Voici le tableau récapitulatif sur lequel se base ce rapport.

Nom du sommet	**Code IATA**	**Ville**	**Pays**	**Liaisons**
A0	LHR	Londres	Royaume-Uni	Paris Tokyo Berlin Madrid
A1	BER	Berlin	Allemagne	Paris
A2	MXP	Milan	Italie	Geneve Paris
A3	NRT	Tokyo	Japon	Montreal Paris Londres
A4	CDG	Paris	France	Paris Londres Berlin Tokyo Geneve
A5	YUL	Montreal	Canada	Paris Tokyo Londres
A6	MAD	Madrid	Espagne	Geneve
A7	GVA	Geneve	Suisse	Tokyo Paris Londres

1.3.) À partir des données ci-dessus, nous avons crée le graphe d'adjacence suivant :



1.4.) Il faut faire attention, les données fournies impliquent que le graphe n'est pas orienté, car certains aéroports ont des vols vers d'autres qui ne sont pas réciproques, on n'aura donc pas une matrice symétrique.

Liste d'adjacence :

```
A0 : A1 A3 A4 A6
A1 : A4
A2 : A4 A7
A3 : A0 A4 A5
A4 : A0 A1 A2 A3 A4 A7
A5 : A0 A3 A4
A6 : A7
A7 : A0 A3 A4
```

Matrice d'adjacence :

	A0	A1	A2	A3	A4	A5	A6	A7
A0	0	1	0	1	1	0	1	0
A1	0	0	0	0	1	0	0	0
A2	0	0	0	0	1	0	0	1
A3	1	0	0	0	1	1	0	0
A4	1	1	1	1	1	0	0	1
A5	1	0	0	1	1	0	0	0
A6	0	0	0	0	0	0	0	1
A7	1	0	0	1	1	0	0	0

1.5.) Pour déterminer le nombre de possibilités pour rejoindre un aéroport à un autre en un nombre donné de vols, il faut utiliser la propriété mathématique suivante : (cf. diapositive 42 Rappel Graphes)

$a^{(p)}_{ij} = 1$ <=> il existe une chaîne de longueur p de v_i à v_j

Donc pour un nombre *p* de vols, il suffit de calculer la *p-ième* puissance de la matrice d'adjacence et regarder quels cases contiennent la valeur 1 pour savoir si ce chemin de longueur p entre les deux nœuds correspondants existe.

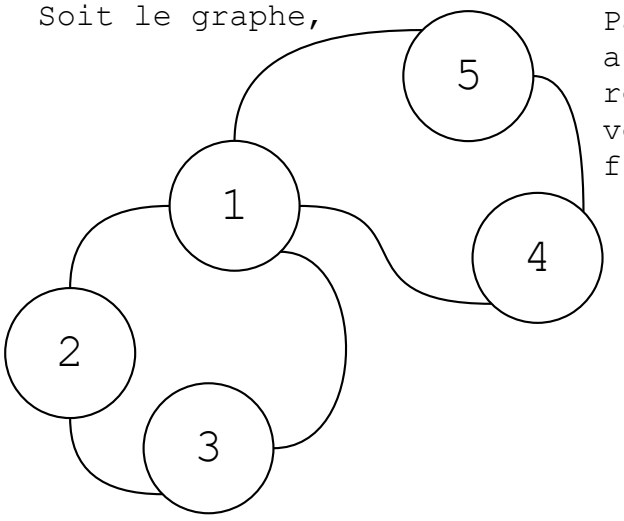
1.6.) Les procédures DFS et BFS sont des algorithmes de parcours de graphe qui fonctionnent les deux de façon récursive mais avec des différences clefs. Nous allons d'abord détailler la procédure DFS, puis la procédure BFS.

DFS (Depth-First Search)

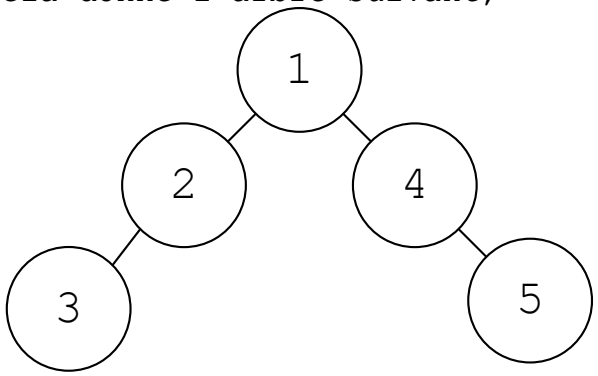
Le principe de fonctionnement de l'algorithme DFS est très simple, on part d'un nœud quelconque et on essaye de s'éloigner le plus possible de ce dernier en suivant ses arcs arbitrairement, ensuite, dés que l'on est dans un cul-de-sac, on rebrousse chemin aussi peu que possible jusqu'à trouver un nouveau chemin à parcourir.

L'intérêt du DFS est qu'il permet de détecter des composantes connexes. C'est à dire, des sous graphes connexes acycliques. En effet, cet algorithme de parcours parcours un graphe comme il parcourrait un arbre, montrons le en un exemple :

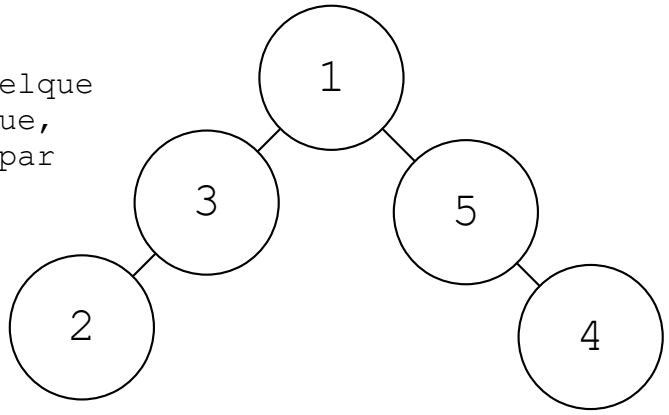
Soit le graphe,



Parcourons le en commençant à 1, puis allant à 2 et 3, et bam, cul-de-sac, on retourne en arrière jusqu'à 1. Allons vers 4 puis 5, bam, cul-de-sac et on a fini, cela donne l'arbre suivant,



Une autre possibilité de parcours, qui est également tout à fait possible serait celui ci-droite.



L'algorithme ainsi présenté est, en quelque sorte non-déterministe, mais en pratique, la direction de parcours sera imposée par l'implémentation.

On voit en tout cas que le parcours renvoie bien un arbre avec deux composantes connexes et permet donc de les détecter dans le graphe original.

Le pseudo-code pour l'algorithme DFS est le suivant,

```
fonction DFS(v)
     if v.categorie <> A { //Si le nœud n'est pas marqué.
          v.categorie <- A; //On le marque.
          pour tout w voisin de v
               DFS(w); //On parcours les voisins récursivement.
}
```

A première vue on pourrait penser que on explore d'un coup tous les voisins d'un nœud via le pour tout, mais c'est récursif ! Donc, dés le premier appel de DFS dans le pour tout on répète l'opération pour le premier voisin allant de plus en plus profondément dans le chemin parcouru jusqu'à que ce ne soit plus possible, puis on « remonte », en parcourant le pour tout du voisin précédent celui pour lequel il n'y a pas de chemin parcourable. C'est un morceau de code très petit mais qui fait quelque chose de très puissant !

BFS (Recherche en largeur)[1]

L'algorithme du BFS suit un principe différent, on commence à un nœud arbitraire, duquel on se rends chez tous les voisins, puis chez tous les voisins des voisins, et ainsi de suite. L'exemple type est celui du labyrinthe : à chaque carrefour on envoie un explorateur dans toutes les directions dont on ne sait pas ou elles mènent, c'est à dire, qui sont de catégorie B, à savoir des sommets adjacents non visités. La catégorie A étant les nœuds visités et C les non adjacents non visités. C'est un algorithme itératif, qui fonctionnera avec un pseudo-code ressemblant :

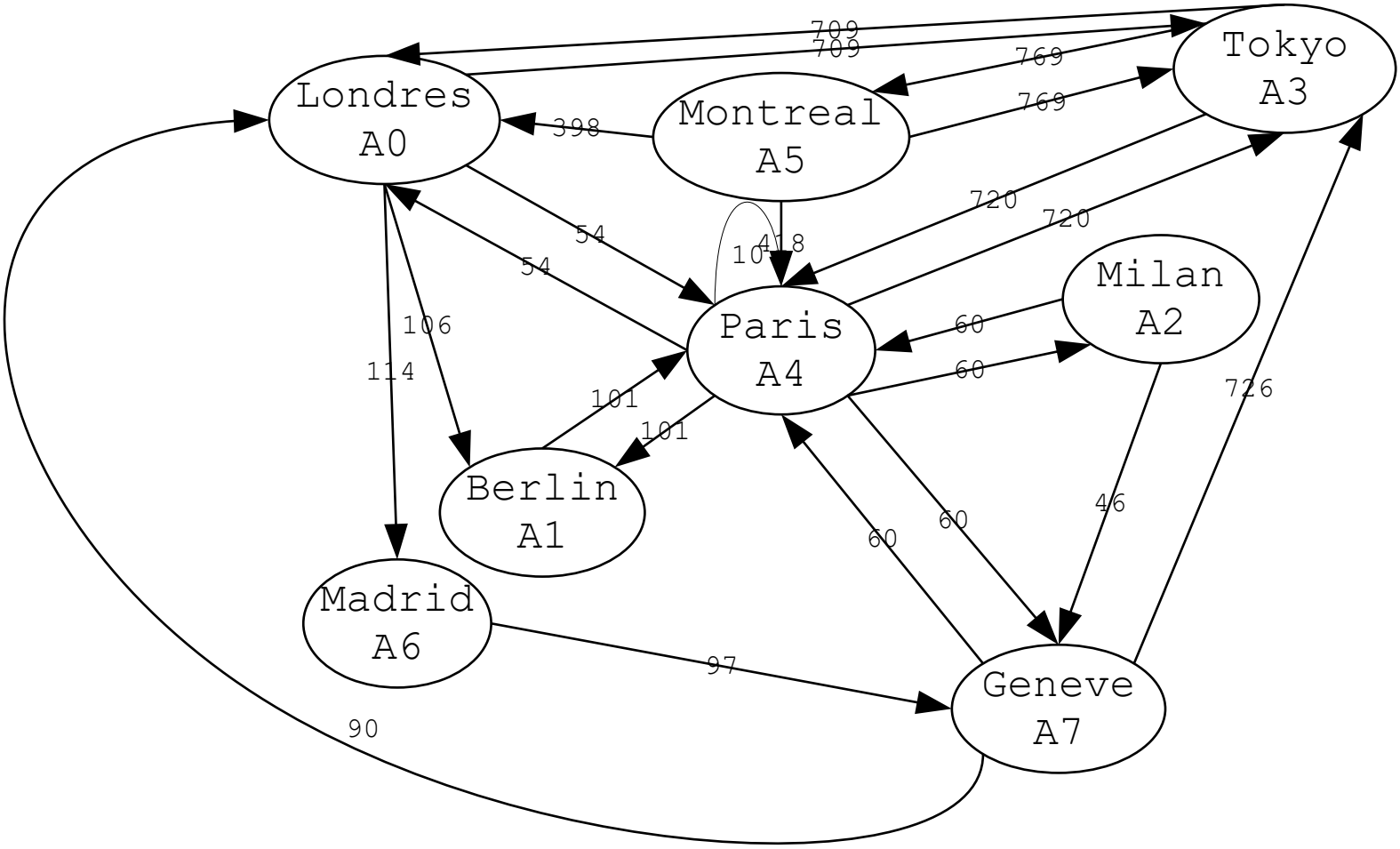
```
procedure BFS(G, start_v)

    Q : FILE; //On utilise une file, donc FIFO.

    start_v.status:= A; //On marque le premier nœud.
    Q.enfiler(start_v); //On le mets en début de file.

    //du moment que la file n'est pas vide.
    tant que Q.length <> 0 do
    v := Q.enleverPremier; //On prends le premier élément.

         pour tout voisin de v, w //Pour chacun de ses voisins
              si w.status <> A //Si le voisin n'est pas marqué.
                   w.status = A; //On le marque.
                   Q.enfiler(w); //On répète la même procédure pour
                                 lui. Cela se fera à la fin du for,
                            donc d'abord on marque tous les
                            voisins, puis les voisins des
                            voisins… Ce qui est la grande
                            différence avec DFS.
```

L'algorithme BFS permet également de récupérer les composantes connexes, il suffit de regarder un exemple pour s'en convaincre, en analysant un parcours de graphe avec ce pseudo-code et en récrivant les nœuds parcourus, on voit que c'est le cas. Une différence clef est que BFS permet de donner le degré de voisinage entre deux nœuds et le chemin le plus court, mais il n'est pas optimisé en tant qu'algorithme de recherche du plus court chemin, car il parcours tout le graphe à chaque fois et ne prends pas en compte le « poids » des arêtes.

1 https://en.wikipedia.org/wiki/Breadth-first_search

Exercice 2.)

1.1.) Je vais me servir des temps de parcours réels pour ces destinations, qui furent fournis par un camarade. Et je vais mettre à jour le graphe précédent, qui est peu lisible. Je vous ait fait une farce, je n'allais pas laisser cela comme ça quand même. Je ne suis pas un psychopathe.



1.2.) Un algorithme du plus court chemin permet, dans un graphe pondéré, de trouver le chemin, c'est à dire, une séquence d'arêtes, entre deux nœuds tels que la somme des poids de chaque arête soit la plus petite possible. Pour ce faire, il existe plusieurs algorithmes possibles, mais le plus fameux est l'algorithme de Dijkstra[2], qui fonctionne selon le principe suivant :

2 https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

Soit un graphe G = (V,E), on définit une fonction poids, P:V x V → R+ qui attribue un poids à chaque arête. Cet algorithme effectue un parcours de graphe et **renvoie un sous-graphe qui est arbre** ou chaque chemin du sommet de départ s_{depart} à un autre sommet est le plus court du graphe de départ en distance, c'est à dire, la somme des poids des arêtes du chemin. La distance est une **propriété d'un sommet relative** au sommet de départ choisi.

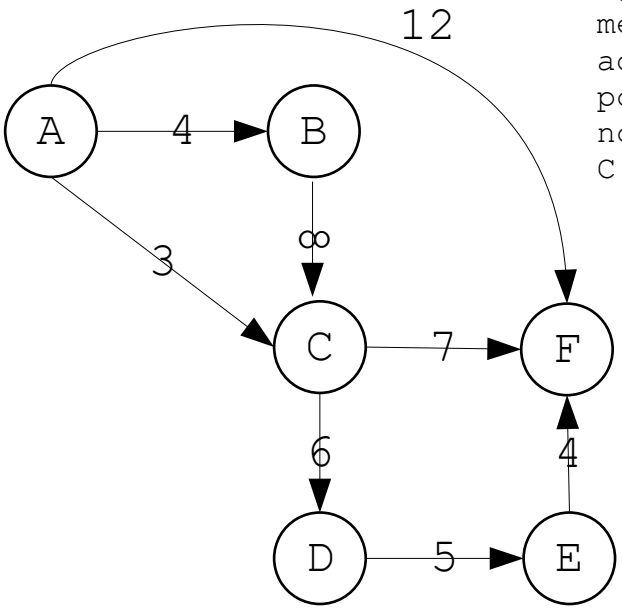
1) On initialise d'abord P qui est l'arbre que l'on vas construire, on y ajoute s, la source. On commence donc avec un graphe à un nœud.

2) On définit les distances de tous les autres sommets comme étant initialement infinies. On définit la distance de s comme étant nulle.

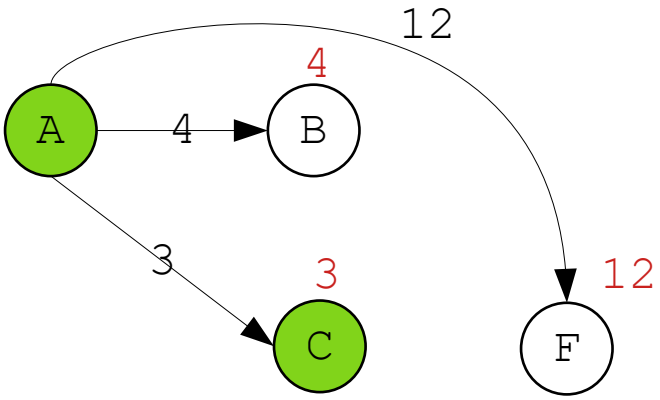
3) Tant que P n'inclue pas tous les nœuds
   a) On choisit un nœud u en dehors de P dont la distance est le minimum parmi les nœuds possibles.
   b) On ajoute u à P
   c) On mets à jour toutes les distances des nœuds adjacents à u. (en prenant le minimum des distances si la distance est déjà assignée.)
   (On recommence le tant que)

C'est un algorithme mal décrit de plusieurs manières semblables en ligne, mais l'article en bas de page le décrivait mieux que le cours ou les encyclopédies que j'ai trouvé, je me suis donc basé la dessus.

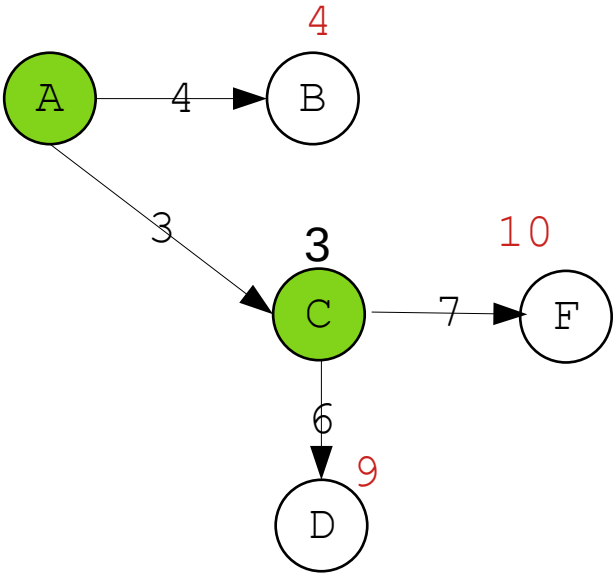
Illustrons le sur un exemple, soit le graphe suivant,



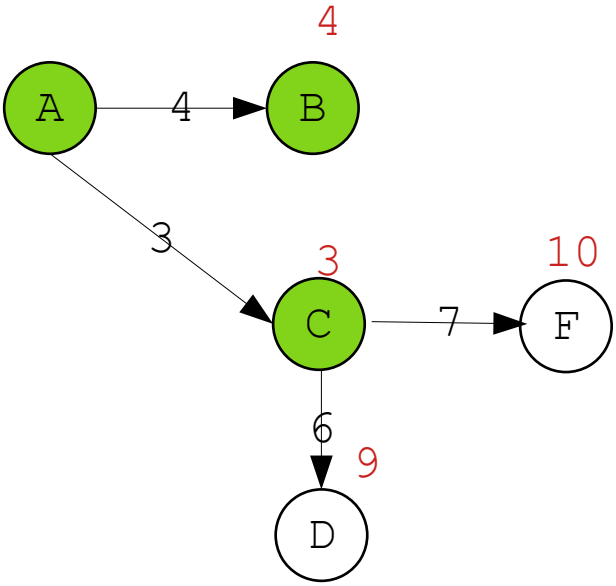
1) On commence avec la source A et on mets à jour les distances des noeuds adjacents. On prends le noeud ayant pour minimum des distances parmis les noeuds adjacents à A non visités, donc, C et on l'ajoute à P (en vert).



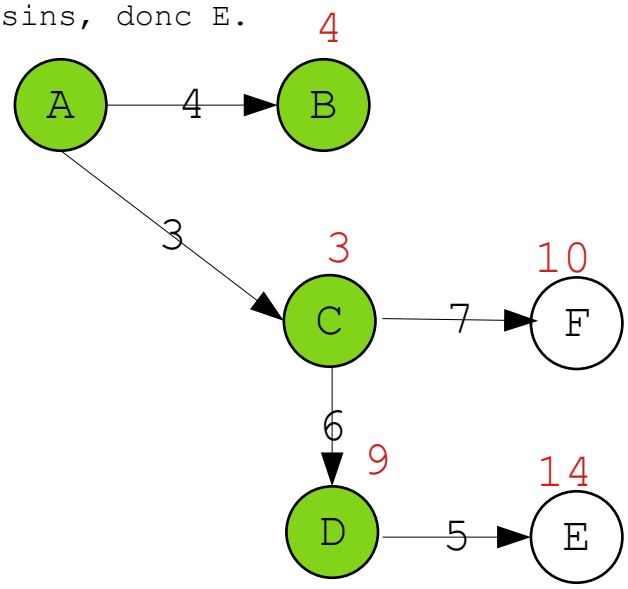
2) On mets à jour les distances des noeuds adjacents à C. (distances en rouge)



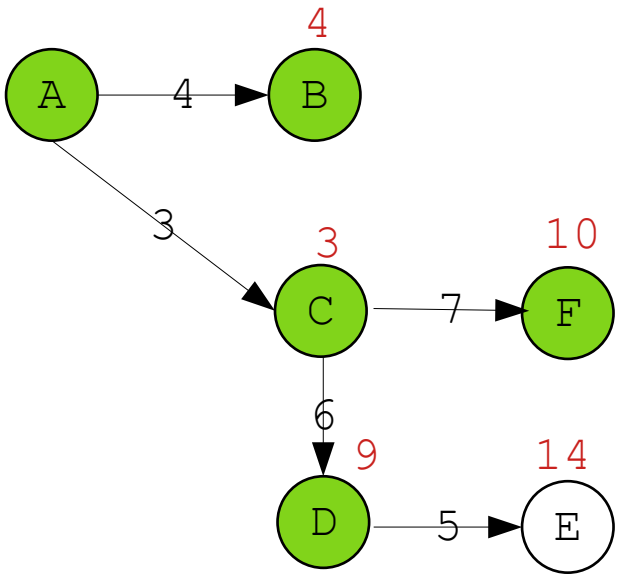
3) On prends le minimum parmi les noeuds adjacents à P, donc B. B n'a pas de noeuds adjacents non visités, donc pas de voisins à mettre à jour.



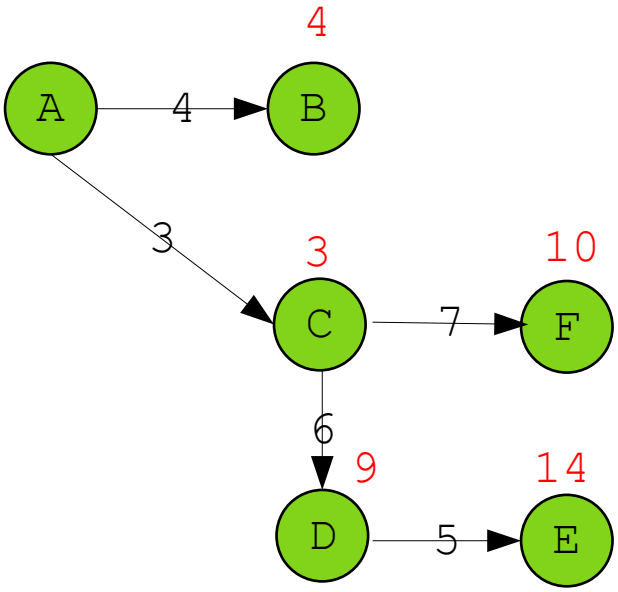
4) On prends le minimum de distance parmi les noeuds adjacents, donc D. Mets à jour les distances de ses voisins, donc E.



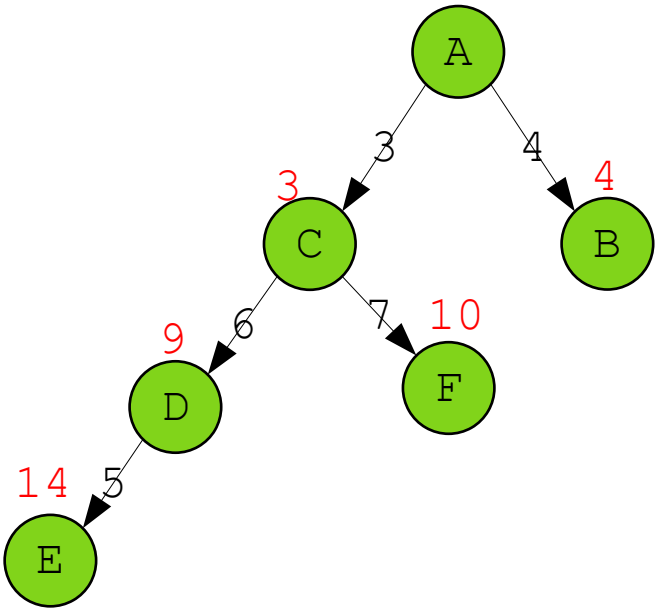
5) On prends le minimum parmis les noeuds adjacents, donc F.



6) On prends le minimum de distance parmi les noeuds adjacents non visités, ici un seul choix, E. Et on a fini.



On remarque que cela nous donne bien un arbre du plus court chemin, comme décrit dans l'algorithme.



Exercice 3.)

3.1.) Implémentation en pièce jointe.

3.2.) Implémentation en pièce jointe.

3.3.) Implémentation en pièce jointe. Pour cet algorithme, je me suis servi d'une matrice d'adjacence et mon algorithme est basé de très près sur celui de geeksforgeeks (note de bas de page N°2), ils sont, par conséquence, très similaires, le code trouvé sur cette source étant tellement clair, je l'ai lu une fois et j'ai refait pratiquement la même chose, car je ne voyais pas comment faire différemment et passer par des listes d'adjacence me semblait rajouter une souffrance inutile.