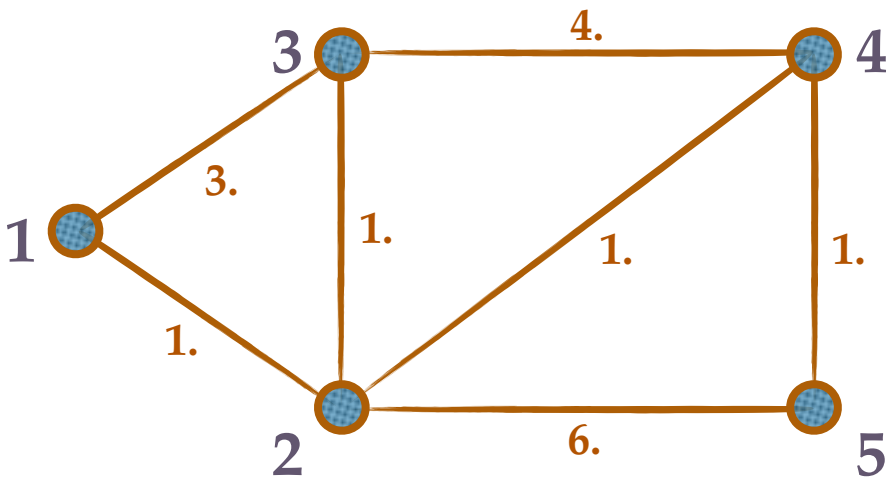
LES GRAPHES PAR LISTE D'ADJACENCES

On se propose d'aller plus loin avec les graphes en considérant cette fois la distance entre les noeuds, comme les distances séparant des villes. Ceci permet de poser des questions algorithmiques plus élaborées. En revanche nos routes sont à double sens : graphe non orienté.



Soit la liste d'adjacences sous la forme d'un dictionnaire { noeud : [(noeud, distance),] }

graphDico = { 1 : [(2, 1.), (3, 3.)],
2 : [(1, 1.), (3, 1.), (4, 1.), (5, 6.)],
3 : [(1, 3.), (2, 1.), (4, 4.)],
4 : [(2, 1.), (3, 4.), (5, 1.)],
5 : [(2, 6.), (4, 1.)] }

I GESTION DES NOEUDS

Les premières questions reprennent le TD introductif, mais attention, les questions sont sensiblement différentes et surtout **les connections sont sous la forme (noeud, distance)**. Vous en aurez besoin dans la suite.

a - Comment obtenir le nombre de noeuds du graphe ?
b - Ecrire une fonction **maxConnect**(graph) qui renvoie le noeud de connectivité maximale et cette connectivité dans un tuple (nCmax, cMax).

c - Ecrire une fonction **supNoeudDico**(dico, p) qui modifie le dictionnaire en supprimant le noeud et toutes ses connections mais ne renvoie rien (None) si le programme fonctionne. Elle renvoie un message d'erreur si le noeud n'existe pas. **Les commandes POP & IN sont autorisées partout.**
Cette fonction peut-être écrite avec une complexité en O(connec) : connectivité du noeud.

d - Ecrire une fonction **addNoeudDico**(dico, p, connec) qui ajoute un noeud et ses connections. Cette fonction renvoie un message d'erreur si le noeud existe déjà sinon rien (None).
Elle doit ajouter le noeud et ses connections mais aussi modifier les connections des noeuds existants. Attention connec est une liste de tuples (noeud, distance). Ex : [(2, 1.), (3, 4.), (5, 1.)].

Vérifier votre code en supprimant et rajoutant le noeud 2 et ses connections d'origine.

II GESTION DES COTÉS

a - Ecrire une fonction **nbCotes**(dico) qui renvoie le nombre de cotés du graphe. Complexité ?

b - Ecrire une fonction **addCoteDico**(dico, p, q, dpq) qui modifie la liste d'adjacences pour que les noeuds p et q soient reliés par un coté de longueur dpq.
La fonction doit créer les noeuds nécessaires si ceux-ci ne sont pas dans le graphe et introduire les connections nécessaires. Si le lien existe déjà un message d'erreur doit être envoyé.

c - Ecrire une fonction **supCoteDico**(dico, p, q) qui supprime le coté entre p et q et renvoie un message d'erreur si le coté n'existe pas.

d - On veut pouvoir construire la liste d'adjacences si l'on nous donne le graphe sous la forme d'une liste de cotés, un coté étant le tuple de 2 noeuds suivi de la longueur du coté (p, q, dpq). Ecrire une fonction **constuireDico**(listeCotes) qui prend en argument la liste des cotés et qui renvoie la liste d'adjacences (dictionnaire). Penser à créer le dictionnaire avec dict().

III PARCOURS DU GRAPHE !

Les deux algorithmes suivants sont à la base de nombreux autres algorithmes sur les graphes.
Exemple : le TomTom réalise un parcours de graphe tout en maintenant pour chaque noeud la connaissance de la distance la plus courte au point de départ et le noeud dont elle provient.
Dans les cas on aura passé en revue tous les noeuds et tous les cotés du graphe.

Le premier algorithme s'appuie sur une structure de queue (FIFO) ce qui induit un balayage des noeuds en couronne concentriques autour du point de départ avant de partir en profondeur.
Le second s'appuie lui sur structure de pile (FILO) ce qui induit une exploration en profondeur des branches avant de remonter et de changer de direction.

L'APPROCHE RETENUE EST NON RÉCURSIVE

a - **Breadth First Search (BFS) : parcours de graphe en largeur**

On veut écrire une fonction **BFS**(dico, start) qui affiche les noeuds visités dans l'ordre **1**2345 ou **5**2413 selon le point de départ en rouge. On s'appuie pour cela sur une **Queue aVoir** des noeuds à voir et sur un tableau (liste) pour marquer les noeuds **dejaVu** qui vaut 1 si le noeud a déjà été visité et 0 sinon. La liste **aVoir** est traitée comme une queue avec des méthodes push/pull, la liste **dejaVu** est un tableau de 0 ou 1. Vous avez le droit à la méthode **.pop**(index) des listes.

A vous d'implémenter cette fonction à partir de la démo sous forme de schémas dans le cours !

Pseudo-code pour vous guider en cas de blocage :

- Initialiser la queue sur start, la liste des déjà vus avec N zéros, et marquer le noeud start.
- Tant qu'il y a des noeuds à voir :
  - Sortir le dernier noeud de la queue
  - Afficher « Je vois le noeud numéro … » #ce sera le résultat du programme
  - Passer en revue toutes les connections de ce noeud:
    - si elles n'ont pas été vues:
      - rentrer dans la queue => par la gauche
      - marquer comme vue dans dejaVu

b **- Deep First Search (DFS) : parcours en profondeur**

On veut écrire une fonction **DFS**(dico, start) qui affiche les noeuds visités dans l'ordre **1**3452 ou **5**4321 selon le point de départ en rouge. On s'appuie cette fois sur une **Pile aVoir** des noeuds à voir et toujours sur le même tableau (liste) pour marquer les noeuds **dejaVu**. Vous avez le droit à la méthode **.pop**(index) des listes.

Pseudo-code pour vous guider en cas de blocage :

- Initialiser la Pile aVoir sur start, la liste des déjà vus avec N zéros.
- Tant qu'il y a des noeuds à voir :
  - Si le noeud a déjà été vu:
    - On le retire de la pile
  - Sinon:
    - On prend le noeud sur la pile.
    - Afficher « Je vois le noeud numéro … » #ce sera le résultat du programme
    - On marque le noeud.
    - Passer en revue toutes les connections de ce noeud:
      - poser toutes ses connections sur la pile.

A vous d'implémenter cette fonction à partir de la démo sous forme de schémas dans le cours !

c - Tester le graphe exemple en vérifiant que chaque noeud est bien vu une fois et une seule ! Essayer de changer le point de départ et comparer les résultats obtenus par BFS et DFS. Introduire dans votre programme le graphe suivant à **convertir sous forme de dictionnaire** :

```
listeCoteHexaGraphe = [(1,2,1.),  (1,6,1.),    (2,3,1.),  (6,5,1. ),  (3,4,1.),  (5,4,1.),
                       (5,7,1.),  (4,10,1.),  (7,8,1.),  (10,9,1.),  (8,9,1.)            ]
```

On le nommera **hexaGraphDico** ! Tracer le graphe sur papier en indiquant les noeuds. Vérifier et comparer à nouveaux vos deux algorithmes en partant de différents noeuds sur le graphe.

QUESTION OUVERTE : APPROCHE RÉCURSIVE

— Trouver la complexité dans le meilleur et le pire des cas pour ces deux algorithmes en fonction de N et M, vous pouvez aussi utiliser une référence nCmax : connectivité maximale des noeuds.
— Vous pouvez tenter de ré-écrire ces deux fonctions de façon récursives.

IV EXPLORATION DU GRAPHE !

Les choses se compliquent nettement dans cette partie. Il est recommandé de revoir la notion d'ensemble dans le cours : **set**() c'est aussi un intérêt de cette partie.
Point technique : on met a jour l'ensemble avec .**update**(uneListe)
et le dictionnaire avec .**update**({clef : uneListe})

Rq : Les personnes durement atteintes d'une phobie des dictionnaires peuvent passer à la partie V.

a - Ecrire une fonction **dicoNCoups**(dico, Ncoups=2) qui renvoie la liste d'adjacence des noeuds atteignables en Ncoups à partir de chaque noeud. ATTENTION tous les coups sont permis : On peut même revenir sur ses pas.

La liste d'adjacence donnera pour tout noeud de départ, la liste de tuples (noeud, distance) où noeud est **atteint après Ncoups** et distance est la **distance accumulée le long du chemin**.

Conseil : vous pouvez commencer par faire un code en 2 coups, vérifier sur les graphes exemples et généraliser votre programme à Ncoups.

L'utilisation des ensembles n'est pas indispensable mais simplifie grandement les choses, car ils ne contiennent jamais deux fois la même connection : tuple —> (noeud, distance).
Attention toutefois : deux chemins différents mais de même longueur n'apparaîtront qu'une fois ! Et c'est une très bonne chose car comme on ne garde pas tout le détail du chemin suivi il ne sert à rien d'avoir plusieurs fois la même information.
De plus dès que le graphe est grand ou dès que Ncoups est grand, La quantité d'information redondante peut devenir gigantesque car on fait du va et vient sur les cotés du graphe !!!

Exemple : hexagraphe 15coups donnerais 190.905 tuples avec des listes en partant du noeud 1, alors qu'il n'y a que 5 tuples distincts….

Pseudo-code pour vous guider en cas de blocage :

- On initialise le dictionnaire sur celui du graphe => correspond à 1 coup.
- On retranche donc 1 coup.
- Pour chaque coup :
  - Passe en revue les noeuds du dictionnaire: **#orange: version 2 coups**
  - On note les connections **(c, d)** qu'ils ont comme étant à visiter
  - Créer un ensemble vide un « sac »
  - Pour les connections **(c, d)** à visiter on visite:
    - toutes les connections **(e, f)** qu'ils ont dans le dictionnaire du graphe:
      On ajoute **(e, d+f)** dans le « sac »
      #le sac n'a pas de doublon (ensemble)
  - On met à jour le dictionnaire pour ce Noeud avec l'ensemble obtenu.
  - on retranche encore un coup
On renvoie le dictionnaire

b - Pour l'hexagraphe que vaut la distance parcourue en fonction de N.
- Observez d'abord que certains noeuds sont in-atteignables à partir d'un point en fonction de la seule parité de N. Comment l'expliquer ?
- Vous pouvez ensuite constater qu'à partir de N = 4, les noeuds atteignables à partir d'un point se divisent en deux partitions et que l'on termine nécessairement sur l'une ou l'autre en fonction de la seule parité de N et de l'appartenance du point de départ à l'une ou l'autre des partitions.
Quel est le lien entre parité et partition ?

c - Quelle est la complexité de l'algorithme dans le pire et meilleur des cas.

IV MINIMUM SPANNING TREE (MST)

Étant donné un graphe de villes connectées par des routes (cotés de longueur déterminée),
On souhaite relier toutes les villes entres-elles par un réseau de voies rapides, tout en minimisant la longueur de route à construire, car elles sont plus couteuses !

Autrement dit :

- on doit pouvoir aller de toute ville (noeud) à une autre par le nouveau réseau
- mais bien choisir pour cela les connections (cotés) à transformer en minimisant la longueur totale de route à transformer.

L'algorithme de Prim permettant cela est illustré par des schémas dans le cours. Il s'appuie sur la notion de coupe d'un graphe et la propriété de coupe suivante :

« Quelle que soit une coupe d'un graphe, le plus petit des cotés sortants de la coupe appartient au MST » .

Rq : il peut y avoir plusieurs cotés sortants qui lui appartiennent.

Algorithme de Prim : Chercher la complexité de cet algorithme

1 - On part d'une coupe à 1 noeud quelconque et ses connections sortantes stockées [**coteAVoir**].

2 - Pour toute coupe [itération] on choisit le noeud extérieur et connecté à la coupe dont le coté sortant est le plus petit parmi ceux de **coteAVoir**. [paradigme Glouton].
On l'ajoute aux noeuds **inclus** et on reconstruit le dictionnaire MST initialement vide.

3 - On met à jour les cotés sortants :
=> ajoute les connexions du nouveau noeud à **coteAVoir**
=> supprime les cotés internes à la coupe.

La version que l'on se propose d'écrire utilise une simple liste pour **coteAVoir**, et on fera une recherche de minimum. L'utilisation d'une queue prioritaire permettrait d'éviter un surcoût.
Nous utiliserons également une liste **inclus** pour les noeuds ajoutés à la coupe. L'ajout d'un coté se fait à l'aide de la fonction **addCoteDico** créée au II. La commande **IN** peut être utilisée pour savoir si un noeud est inclus ou pas. On notera (p, q, dist) un objet coté, et la commande **.remove**() peut être utilisée pour enlever un coté.

a - Ecrire une fonction **getMin**(mesCotés) qui prend une liste d'objets cotés et renvoie le coté dont la distance est la plus courte de la liste.

b - Ecrire une fonction **MST**(dico) qui prend en argument le dictionnaire d'un graphe et renvoie le dictionnaire du MST de ce graphe en appliquant l'algorithme de Prim.

c - Tester votre algorithme sur les deux graphes exemples. Vous pouvez modifier votre fonction MST pour imposer un autre choix du noeud de départ. Le MST obtenu est il différent ? Est-ce possible ? A quelle condition ?

d - Ecrire une fonction rendementMST(dico) qui prend un graphe et renvoie la rapport :
R = longueurRouteGraphe / longueurRouteMST. On pourra faire une fonction intermédiaire **longueurGraphe**(dico) qui calcule la longueur totale du graphe d'un réseau.

FIN