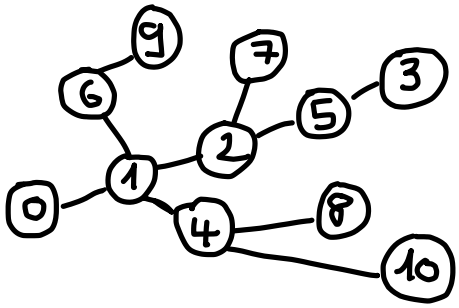
TD 23 – Partie 2 – Parcours dans un graphe et algorithmes simple

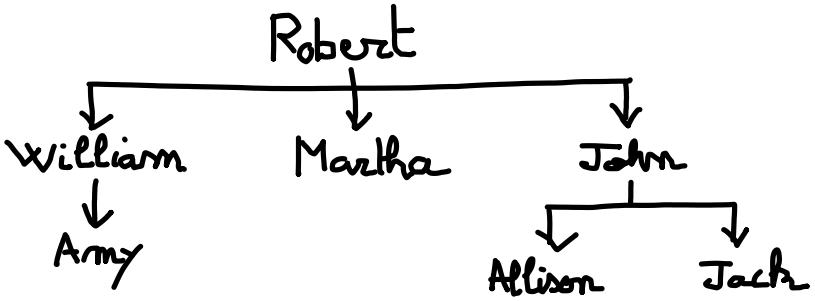
A / Cinq graphes (donnés) de test pour ce sujet

On donne, pour tester les fonctions qui seront codées par la suite, quatre graphes. Ceux-ci sont déjà implémentés dans un fichier *S2-TD23-graphe-2-parcours.py* qui se trouve sur mon site.

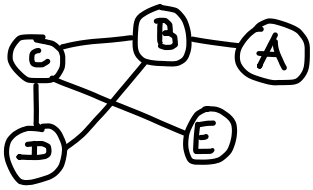
Graphe A – Arbre non orienté



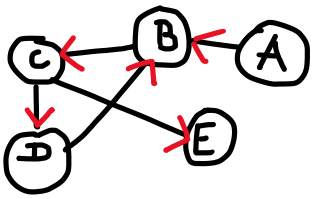
Graphe B – Arbre orienté



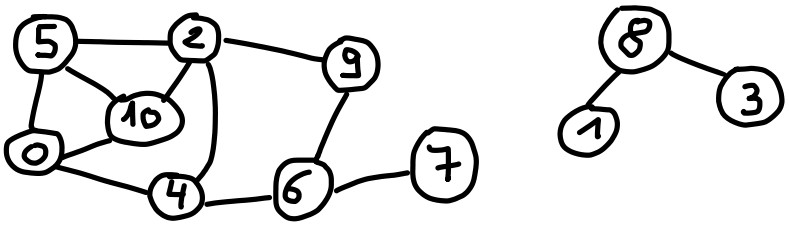
Graphe C – Graphe cyclique connexe non orienté



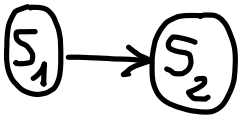
Graphe D – Le même que C, mais orienté



Graphe E – Graphe cyclique non connexe non orienté



B / Convention retenue pour les graphes orientés



Nous avons ici adopté la convention de voisinage orienté sortant, c'est-à-dire que, dans l'exemple illustré ci-contre, **S_2 est le voisin de S_1** (c'est-à-dire qu'il existe un arc partant de S_1 et permettant de relier S_2) alors que **S_1 n'est pas le voisin de S_2** (qui est, ici, une feuille).

En termes de liste d'adjacence, on aurait donc pour ce graphe simpliste : `{ S1:[S2], S2 :[] }`

D'autre part, que les graphes soient orientés ou non, nous n'avons pas représenté de boucle, et supposerons que pour tous les graphes de ce sujet, un sommet S est toujours à une distance 0 de lui-même.

C / Objectifs et questionnaire

On souhaite, dans ce sujet, mettre en évidence que la plupart des algorithmes simples s'appliquant à un graphe utilisent un parcours en largeur ou bien en profondeur de ce graphe. Les versions itératives de ces deux algorithmes sont données *S2-TD23-graphe-2-parcours.py* sous forme de fonctions.

C.1 / Fonction ne s'appuyant pas sur un parcours de graphe

Pour savoir si un graphe est orienté ou non, il faut tester si :

Pour tout nœud S_i de S, quel que soit le voisin S_j de S_i, S_i est aussi voisin de S_j.

Au niveau algorithmique, il est donc plus facile de savoir si un graphe est non orienté, ce qui revient à savoir si :

Il existe un nœud S_i de S, tel que un de ces voisins au moins S_j de S_i, soit tel que S_i n'est pas voisin de S_j.

Q1 – Écrire une fonction `est_oriente(Graphe)` prenant en argument un graphe et renvoyant un booléen : `True` si le graphe est orienté, `False` sinon.

Aide : on rappelle l'existence de la méthode `dico.keys()` qui renvoie toutes les clés d'un dictionnaire `dico`.
Test : tester votre fonction sur les graphes donnés.

Pour la fin de ce sujet, nous allons devoir écrire une fonction qui transforme (momentanément) un graphe orienté en graphe non orienté. Par exemple, si on lui donne le graphe D en entrée, elle devrait renvoyer le graphe C.

- Si le graphe n'est pas orienté, on renverra directement le graphe sans le modifier
- Si le graphe est orienté, alors on souhaite « modifier » le graphe pour transformer les arcs (liaisons non réciproques entre deux sommets) en arêtes (liaisons réciproques).
  L'algorithme à appliquer ressemblera à celui de la **Q1**, à savoir :
  Pour tout nœud S_i de S,quel que soit le voisin S_j de S_i, si S_i n'est pas voisin de S_j , on rajoute S_i à la liste d'adjacence de S_j.

Aide : comme pour les listes, penser à faire une copie du graphe, et travailler sur la copie qui sera renvoyée en fin d'algorithme.

Q2 – Écrire la fonction `sans_orientation(Graphe)` dont le principe est décrit ci-dessus.

C.2 / Fonction s'appuyant sur un parcours en profondeur

Dans cette sous-partie, on souhaite déterminer si un graphe **orienté** est cyclique (càd s'il existe un cycle sur le graphe). Nous allons déjà commencer par suivre l'ensemble des chemins partant d'un nœud de départ `N_d` pour déterminer s'il existe un cycle sur ce chemin.

Nous allons nous inspirer du parcours en profondeur à partir de ce nœud `N_d` et retoucher la fonction. On détecte un cycle s'il existe un chemin qui boucle sur un nœud déjà visité. L'algorithme de détection sera donc le suivant :

- On parcoure le graphe en profondeur
- À chaque nœud qu'on s'apprête à visiter, on teste si ce dernier a déjà été visité auparavant. Si c'est le cas, c'est qu'il y a un cycle, on peut alors quitter la fonction et renvoyer `True`.

Nous estimons ici que la fonction à écrire ne s'applique que dans le cas d'un graphe orienté (la détection de cycle sur un graphe non orienté étant plus difficile). Le test de votre fonction ne se fera donc que sur les graphes B (non cyclique) et D (cyclique).

Q3 – En vous inspirant fortement de la trame du parcours en profondeur (copiez-coller puis ajouter / modifier certaines lignes), écrire une fonction `cycle_sur_chemin(Graphe, N_d)` prenant en argument un graphe **orienté** et un nœud de départ, et renvoyant un booléen : `True` si un cycle est trouvé au départ de `N_d`, `False` sinon. (On souhaite détecter un cycle, qu'il « reboucle » sur `N_d` ou n'importe quel autre nœud du parcours).

Erreur : en tout début de fonction, vous testerez si le graphe est orienté. Si ce n'est pas le cas, vous quitterez la fonction en renvoyant une erreur :

```
assert None, "Erreur, graphe non orienté"
```

Test : pour le graphe D, que se passe-t-il se on part du nœud A ? Et du nœud E ?

Un graphe est dit cyclique s'il existe un parcours présentant un cycle. Nous allons donc tester un algorithme naïf qui consiste à regarder tous les parcours, c'est-à-dire partir successivement de chacun des nœuds du graphe.

Q4 – Écrire alors une fonction `est_cyclique(Graphe)`, clairement pas optimisée, qui essaie de trouver des cycles au départ de chacun des sommets du graphe orienté (on utilise alors la fonction codée en **Q3**), et renvoie un booléen : `True` si un cycle est trouvé, `False` sinon.

➔ Tester votre algorithme sur les deux graphes orientés donnés.

➔ Quel est le point d'optimisation principal qu'il faudrait apporter à cette fonction pour qu'elle consomme moins de temps processeur ?

C.3 / Fonction s'appuyant sur un parcours en largeur

Dans cette seconde partie, nous nous intéressons à des algorithmes de recherche de plus court chemin, qui s'appuient sur des parcours du graphe en largeur.
Une petite différence, toutefois :

- Dans l'algorithme de parcours en largeur, nous avions :
  - Ajouté `Noeud_depart` à `Deja_visites`, à condition qu'il n'ait pas déjà été visité
  - Puis pour chaque voisin, nous l'ajoutions quoi qu'il arrive à `A_visiter`
- Dans les algorithmes qui suivent, je vous impose que vous réfléchissiez à l'inverse :
  - On visite quoi qu'il arrive le `Noeud_depart`
  - Mais pour chaque voisin, on ne l'ajoute que s'il n'a pas déjà été vu auparavant…

  → A vous d'adapter les codes en fonction

Q5 – Ecrire une première fonction `chemins_largeur_ite(Graphe,N_d)` prenant en argument un graphe et un nœud de départ, et qui renvoie un dictionnaire dont les clés sont sommets parcourus (dans l'ordre en largeur), et les valeurs associées sont les distances au nœud `N_d`.

Ex : pour l'arbre A, la fonction doit renvoyer

```
{1: 0, 0: 1, 2: 1, 4: 1, 6: 1, 5: 2, 7: 2, 8: 2, 10: 2, 9: 2, 3: 3}
```

Q6 – Ecrire une seconde fonction `chemins(Graphe,N_d)` prenant en argument un graphe et un nœud de départ, et qui renvoie un dictionnaire dont les clés sont sommets parcourus (dans l'ordre en largeur), et les chemins associés permettant de relier ce sommet au sommet `N_d`.

Remarque: par rapport à la fonction codée en **Q5**, seules deux lignes devraient changer.
Exemple : pour l'arbre A, la fonction doit renvoyer

```
{1: [1],
 0: [1, 0],
 2: [1, 2],
 4: [1, 4],
 6: [1, 6],
 5: [1, 2, 5],
 7: [1, 2, 7],
 8: [1, 4, 8],
 10: [1, 4, 10],
 9: [1, 6, 9],
 3: [1, 2, 5, 3]}
```

Q7 – En vous inspirant du code de la **Q6**, écrire une 3ème fonction `plus_court_chemin(Graphe, N_d, N_a)` prenant en argument un graphe, un nœud de départ `N_d`, et un nœud d'arrivée `N_a`, et qui renvoie le chemin le plus court reliant `N_d` à `N_a`, chemin tel que décrit dans l'exemple précédent.

Remarque: le test, à l'intérieur du while, permettant de détecter si on a atteint le nœud d'arrivée, sera fait de telle sorte que si le nœud de départ est le même que le nœud d'arrivée, le chemin renvoyé soit une liste de taille 1, contenant le nœud en question.

Exemple :

```
      plus_court_chemin(B,"John","John")      devrait renvoyer ["John"]
Sinon plus_court_chemin(C,"A","E")            devrait renvoyer ["A", "B", "C", "E"]
      plus_court_chemin(E,0,7)                devrait renvoyer [0, 4, 6, 7]
```

D.3 / Conclusion : détection d'un arbre sur un graphe orienté

On appelle **arbre** un graphe acyclique et connexe. Dans cette partie, nous allons utiliser les fonctions codées en questions précédentes pour détecter ce type de graphes.

Un graphe est connexe si pour toute paire de sommets (S_i, S_j) de S^2 S_i et S_j peuvent être reliés par au moins un chemin, si l'on ne tient pas compte de l'orientation des arcs (c'est-à-dire en considérant momentanément le graphe comme non-orienté, voir question **Q2**).

Q8 – En utilisant les fonction écrites en **Q2** et **Q7**, écrire une fonction `est_connexe(Graphe)` qui prend en argument un graphe et renvoie un booléen : `True` si le graphe est connexe, `False` sinon.

Remarque : étant donné que vous ne tenez pas compte de l'orientation, j'impose que vous ne testiez pas :

- L'existence de chemin entre S_i et S_i (il en existe forcément un)
- L'existence de chemin entre S_i et S_j si vous avez déjà testé le chemin entre S_j et S_i

Pour ce faire : vous allez avoir une structure d'algorithme avec deux boucles *for* imbriquées, il suffit de bien jouer sur les bornes de la boucle *for* interne.

Test : tester sur les graphes donnés, seul le graphe E devrait ne pas être connexe.

Finalement, nous allons nous servir des fonctions codées aux questions **Q1**, **Q4** (où nous avons utilisé un parcours en profondeur) et **Q8** (qui appelle la **Q7** où nous avons fait un parcours en largeur) pour déterminer si un graphe est un arbre orienté.

Q9 – une fonction `est_arbre_oriente(Graphe)` qui prenne en argument un graphe, et renvoie un booléen : `True` si le graphe est un arbre orienté, `False` sinon.

Test : Seul le graphe B doit renvoyer `True`.
J'impose que `est_arbre_oriente(A)` (idem pour C et E, càd des graphes non orientés) renvoie `False`, et pas une erreur.