Applications

Recherche de connexité

Ressources de Nicolas Courrier

Exercice 1 – Recherche de connexité

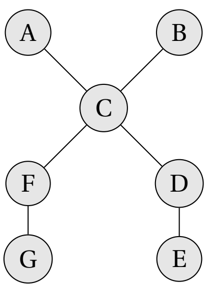
D'après ressources Nicolas COURRIER, UPSTI.

Introduction

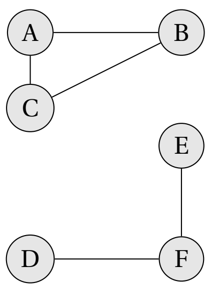
Définitions

Définition Un graphe non orienté $G = (S, A)$ est dit connexe si pour tout couple de sommets (u, v), il existe une chaîne reliant u et v.

Exemples :



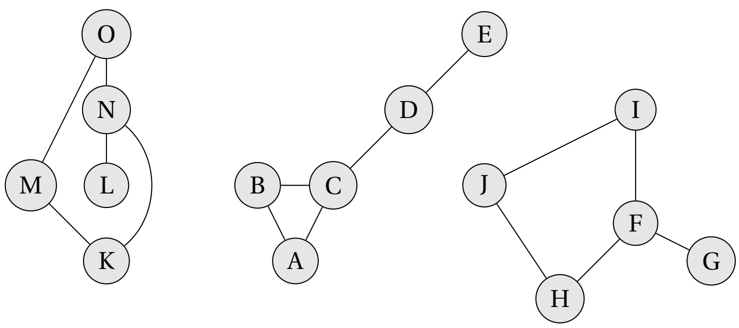
Graphe 1 : graphe connexe



Graphe 2 : graphe non connexe

Définition Un graphe qui n'est pas connexe est l'union de plusieurs sous-graphes connexes. Chacune des ces parties étant indépendantes des autres. C'est-à-dire qu'elles n'ont pas de nœuds en commun. Les sous-graphes connexes disjoints sont des **composantes connexes** du graphe.

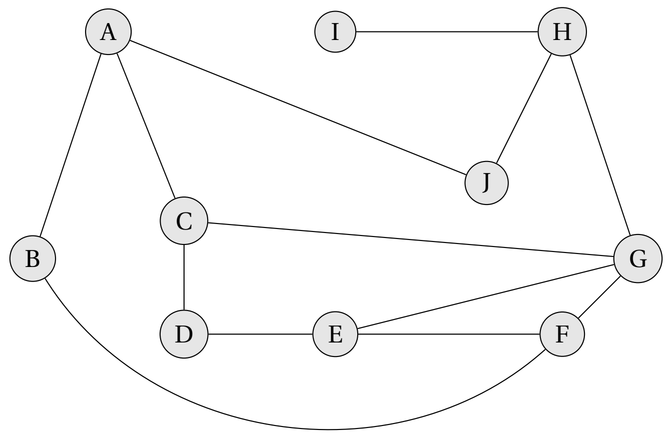
Exemple



Graphe 3 : graphe non connexe avec trois composantes connexes

Préparation aux codes

Soit le graphe suivant :



Graphe 4

Question 1 *Ce graphe est-il connexe ?*

Question 2 *Renseigner la matrice d'adjacence* `mat_adj` *permettant de modéliser le graphe précédent.*

Question 3 *Créer une fonction* `verif_symetrie(matrice)` *retournant le booléen* `True` *si la matrice* `matrice` *est effectivement symétrique et* `False` *sinon. Vérifier ainsi que la matrice* `mat_adj` *renseignée à la question précédente est bien symétrique (cela permettra de trouver d'éventuelles erreurs de rentrées d'informations).*

Dans le but de parcourir un graphe, il est beaucoup plus simple d'identifier les sommets par des numéros (plus facile pour réaliser des boucles par exemple ou pour récupérer une ligne ou colonne de la matrice d'adjacence). Ainsi dans le cas traité, A sera le nœud 0, B sera le nœud 1 etc.

On dispose de la fonction `determination_numero(etiquettes,nom_noeud)` prenant comme argument la liste `etiquettes` et une chaîne de caractères `nom_noeud`. Cette fonction retourne le numéro de nœud associé au nom du nœud étudié. On supposera que `nom_noeud` est bien présent dans la liste `etiquettes`.

Dans le but de réaliser le parcours de graphe (que ce soit en largeur ou profondeur), il est nécessaire de déterminer les voisins d'un sommet qui n'aura pas été pas été déjà découvert. On donne la liste `noeuds_visites` contenant la liste des nœuds déjà découverts (ou visités).

Question 4 *Créer une fonction* `recherche_voisins(mat_adj,noeuds_visites,noeud)` *ayant comme arguments la matrice d'adjacence, la liste des nœuds déjà découverts et le nœud* `noeud` *dont on cherche les voisins encore non visités. Cette fonction renverra ainsi la liste des nœuds voisins non visités. On précise que dans cette fonction les sommets (ou nœuds) sont identifiés par leur numéro. Ainsi, par exemple* `recherche_voisins(mat_adj,[0,1,3],2)` *doit renvoyer [6]. En effet, on cherche ici les sommets voisins de C parmi ceux non visités. Les nœuds déjà visités dans cet exemple sont A, B et D. Le sommet G est le seul voisin de C non visité.*

Parcours en largeur

Question 5 *Réaliser « à la main » le parcours en largeur du graphe proposé en partant du nœud A et dessinez l'arbre ainsi obtenu.*

> **Définition** Lors d'un parcours en largeur ou profondeur d'un graphe, si l'arbre de parcours obtenu est couvrant – c'est-à-dire que l'arbre contient l'ensemble des nœuds du graphe – alors le graphe est connexe.

Question 6 *Réaliser une fonction* `parcours_largeur_graphe(mat_adj,depart,etiquettes)` *prenant en argument la matrice d'adjacence du graphe, la liste* `etiquettes` *et le nœud de départ (avec son nom et non pas son numéro par exemple* `parcours_largeur_graphe(mat_adj,'A',etiquettes)`*. Cette fonction devrait renvoyer la* *liste des nœuds suite au parcours en largeur du graphe.*

Question 7 *Modifier le code précédent pour que la fonction* `parcours_largeur_graphe` *retourne en plus de la liste des nœuds visités :*

- *la matrice d'adjacence du graphe de parcours obtenu ;*
- *la liste* `provenance` *contenant le numéro du nœud « père » ou « origine » dont est issu un nœud découvert. L'origine du sommet de départ est lui-même. Par exemple en partant du nœud A, on doit obtenir la liste* `provenance`*=[0,0,0,2,5,1,2,9,7,0]. En effet, lors du processus du parcours, le nœud B a pour origine le nœud A, le nœud C également, le nœud D a pour nœud père C etc. Si un nœud n'a pas été découvert alors son origine sera* `None`.

Question 8 *D'après la **Définition 3**, comment peut-on savoir si le graphe est connexe ?*

La ligne de code suivante a été tapée, avec `depart` un nœud du graphe :

```
liste_noeuds,mat_arbre,provenance= \
parcours_largeur_graphe(mat_adj,depart,etiquettes)
```

Question 9 *À l'aide d'une fonction* `restitution_parcours(provenance,etiquettes,depart,arrivee` *établir la chaîne permettant d'aller du nœud* `depart` *au nœud arrivee.*

Définition Le parcours en largeur permet de trouver le chemin le plus court (si les arêtes ne sont pas pondérées) entre deux sommets.

Parcours en profondeur

Question 10 *Réaliser le codage « à la main » du parcours en profondeur du graphe étudié en partant du point A. Quand plusieurs possibilités se présentent à vous, vous choisirez d'explorer le nœud le mieux classé dans l'ordre alphabétique.*

Question 11 *Réaliser une fonction* `parcours_profondeur_graphe(mat_adj,depart,etiquettes)` *prenant en argument la matrice d'adjacence du graphe, la liste* `etiquettes` *et le nœud de départ (avec son nom et non pas son numéro par exemple* `parcours_largeur_graphe(mat_adj,'A',etiquettes)`*. Cette fonction devrait renvoyer la liste des nœuds suite au parcours en profondeur du graphe.*

Question 12 *Modifier la fonction* `visiter` *et parcours_profondeur_graphe de façon à retourner également :*

- *la matrice d'adjacence du graphe de parcours obtenu ;*
- *la liste* `provenance` *contenant le numéro du nœud « père » ou « origine » dont est issu un nœud découvert. L'origine du sommet de départ est lui-même. Par exemple en partant du nœud A, on doit obtenir la liste* `provenance`*=[0, 0, 3, 4, 5, 1, 2, 6, 7, 7]. En effet, lors du processus du parcours, le nœud B a pour origine le nœud A, le nœud C a pour nœud père D, etc. Si un nœud n'a pas été découvert alors son origine sera* `None`.

Optimisation en temps

Parcours en largeur : Utilisation des files

Dans cette section, on va chercher à optimiser un code de parcours en largeur d'un graphe. Le premier graphe proposé est composé de 2000 nœuds, le second de 5000. Pour réaliser cette partie, un fichier « codes_parcours_graphe.py » et deux fichiers contenant les matrices d'adjacence des graphes sont à votre disposition : « graphe_2000.txt » et « graphe_5000.txt ».

Question 13 *Dans un fichier principal « main.py » importer via la commande* `from ... import`*, l'ensemble des fonctions du fichier « codes_parcours_graphe.py »*

Question 14 *Dans le fichier principal, coder l'ouverture et la lecture du fichier « graphe_2000.txt ». Stocker la matrice d'adjacence ainsi obtenue dans une variable* `mat_adj`*. Si besoin consulter le fonctionnement de la commande* `eval` *(*`help(eval)`*).*

Sachant que la fonction `time()` de la bibliothèque `time` permet de prendre la mesure du temps à l'instant t, pour estimer la durée d'une action, il faut donc prendre la différence entre deux points de mesures, comme sur l'exemple ci-dessous :

```
t1=time.time()
Action quelconque
...
t2=time.time()
duree=t2-t1
```

Remarque : sur les versions plus récentes de Python, la fonction `time()` a été supprimée au profit de `process_time()`.

Question 15 *À l'aide de cette explication, de la spécification et des commentaires présents dans le code* `parcours_largeur_gra` *expliquer le rôle de celui-ci et les informations que l'on a en retour.*

Question 16 *Dans le fichier principal, importer le module* `time` *et réaliser le parcours de graphe à l'aide de la fonction* `parcours_largeur_graphe1`*. Mesurer le temps total d'exécution ainsi que le temps d'exécution des différentes opérations effectuées lors de l'appel à cette fonction. Afficher ces temps à l'écran.*

La durée totale du code est de 32.68 s, la gestion de la file prend 0.01 s, la gestion de la liste `noeuds_visites` 0.001 s et la recherche de voisins non visités 32.65 s.

Question 17 *Sur quel paramètre peut-on essayer de jouer pour gagner du temps ?*

Question 18 *Quelle est la différence principale entre le code* `parcours_largeur_graphe1` *et le code* `parcours_largeur_gr`

Question 19 *Dans le fichier principal, réaliser maintenant le parcours de graphe à l'aide de la fonction* `parcours_largeur_gr` *. Mesurer le temps total d'exécution ainsi que le temps d'exécution des différentes opérations effectuées lors de l'appel à cette fonction. Afficher ces temps à l'écran. Entre la phase de recherche des voisins d'un nœud et la phase de test pour savoir si un voisin de ce nœud a déjà été découvert, quelle est l'étape la plus longue ?*

Question 20 *Justifier alors le fonctionnement du code* `parcours_largeur_graphe3`.

Question 21 *Dans votre fichier principal, réaliser maintenant le parcours de graphe à l'aide de la fonction*`parcours_largeur_` *. Conclure sur la stratégie adoptée.*

On a pu constater que l'utilisation d'une liste de taille connue et définie à l'avance était plus efficace que la manipulation de celle-ci (ajout d'élément ou recherche dans celle-ci). Fort de cette constatation, on peut remarquer que dans les codes proposés, la liste `file` subit des modifications de taille au fur et à mesure des boucles. Le principe de fonctionnement d'une file s'appuie sur le principe du premier arrivé, premier sorti (FIFO : First in, First out). C'est-à-dire que les nouveaux éléments à traiter sont stocker en fin de file et ceux qui vont être traités sont en début de file. Exactement comme une file d'attente à la caisse d'un supermarché.

Le principe d'une file est de stocker les nouveaux éléments en fin de file et de stocker un élément dans une variable tout en le retirant du début de file pour poursuivre l'algorithme.

Question 22 *On peut encore chercher à améliorer l'efficacité du code en utilisant la bibliothèque* `collections` *et la fonction* `deque`*. En effet, celle-ci est spécialement dédiée à la construction de piles et de files. Réaliser le code suivant :*
- *créer une liste de nombres aléatoires de 100000 éléments ;*
- *créer à l'aide de cette liste, la file correspondante à l'aide de la fonction* `deque` *;*
- *dans deux boucles* `for` *distinctes de 100000 itérations chacune, simuler la récupération du premier élément de la liste ou de la file ainsi que la suppression de celui-ci dans cette même liste ou file. (Ce qui est fait lors du parcours de graphe). Utiliser pour cela l'annexe proposée sur le module* `collections.deque` *;*
- *mesurer les temps d'exécution de ces deux boucles.*

Conclure.

Question 23 *Proposer alors un code optimisé (en partie en réalité) du parcours de graphe en largeur.*

Question 24 *Le graphe de 200 sommets proposés est-il connexe ?*

Question 25 *Si votre ordinateur le permet, comparer les durées d'exécution des codes* `parcours_largeur_graphe1` *et celui que vous avez proposé avec le graphe composé de* 5000 *nœuds.*

Parcours en profondeur : Utilisation des piles

Vous venez de réaliser l'optimisation d'un code de parcours en largeur d'un graphe relativement "simpliste" (au sens où il ne renseigne que sur les nœuds qui ont été découverts lors du parcours). L'utilisation d'une file avec le module `collections.deque` et une amélioration de la structure du code ont permis d'optimiser en bonne partie le temps de calcul.

Question 26 *Reprendre votre code de parcours en profondeur d'un graphe et l'optimiser. Bien entendu comme pour la partie précédente, il ne renverra que l'information des nœuds découverts.*

Question 27 *Mesurer le temps nécessaire pour parcourir le graphe de* 2000 *nœuds proposé dans le fichier « graphe_2000.txt ». Que constate-t-on ?*

Question 28 *Coder cet algorithme à l'aide du module* `collections.deque` *et mesurer le temps d'exécution nécessaire pour le graphe à* 2000 *nœuds.*