17.1 Parcours de graphe

Chemins dans un graphe On parle de *chemin dans le graphe* G lorsqu'on considère une suite de sommets de G $v_0, \dots, v_p$ telle que deux éléments successifs sont reliés par une arête, autrement dit $(v_i, v_{i+1}) \in E$ pour tout i. figure 17.1 donne un exemple de chemin. La longueur d'un chemin est p, c'est-à-dire le nombre d'arêtes parcourues. La détection de chemins dans un graphe, en particulier de plus court chemin, forme tout un pan de la théorie des graphes.

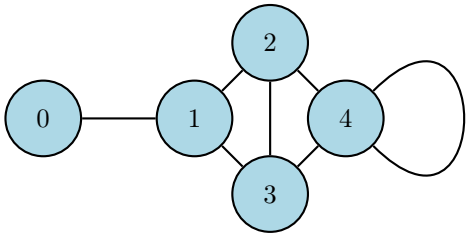
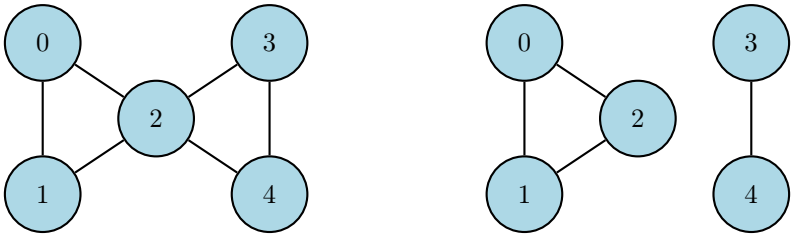


FIGURE 17.1 – $(1, 3, 4, 4, 4, 3, 2, 1, 0)$ est un chemin de longueur 8 du graphe G

Connexité Un graphe non orienté est *connexe* si de tout sommet v il existe un chemin à tout autre sommet w. Moins formellement, un graphe est connexe s'il n'est pas en plusieurs morceaux, voir figure 17.2.



(a) Un graphe connexe (b) Un graphe non connexe

FIGURE 17.2 – Connexité d'un graphe

On souhaite parcourir le graphe de proche en proche, c'est-à-dire à partir d'un sommet v fixé de déterminer tous les sommets que l'on peut atteindre par un chemin partant de v. L'idée est de parcourir v, puis tous ses voisins, puis tous les voisins de ses voisins si on ne les a pas déjà visités, puis les voisins des voisins des voisins de v si on ne les a pas déjà visités, etc. Pour ce faire, on maintient deux listes

— *àVisiter* qui contient les sommets que l'on doit visiter prochainement, car ce sont les voisins de noeuds que l'on a visités

— *déjaVus* qui contient la liste des sommets déjà visités.

La pseudo code de la fonction ressemble donc à celui-ci :

Function *Parcours(v, G)*
    àVisiter ← $\{v\}$
    déjàVus ← $\{v\}$
    while *àVisiter* $\neq \emptyset$ **do**
        prendre $u \in$ àVisiter
        for *w voisin de u* **do**
            if $w \notin$ *déjàVus* **then**
                rajouter w à àVisiter
                rajouter w à déjàVus
            end
        end
    end

Dans la suite, on considère qu'un graphe G est représenté par sa matrice d'adjacence M (voir le TP 15). Le pseudo-code ci-dessus laisse le choix sur la façon de choisir u dans àVisiter. En fait, selon la façon qu'on choisit, on effectue soit un *parcours en profondeur*, soit un *parcours en largeur*.

1. En utilisant le pseudo-code précédent, écrire une fonction `parcoursEnProfondeur(v, M)`. Lorsque l'on choisit u dans àVisiter, on prend le sommet qui a été ajouté le plus récemment. On dit que àVisiter à une structure de *pile* (penser à une pile d'assiette), car elle vérifie la propriété LIFO *Last In First Out*, en français *dernier arrivé premier servi*.

2. Écrire cette fois une fonction `parcoursEnLargeur(v, M)`, pour laquelle on choisit u dans àVisiter comme celui qui y a été ajouté en premier. On dit alors que àVisiter à une structure de file (penser à une file d'attente), car elle vérifie la propriéte FIFO *First In First Out*, en français *premier arrivé premier servi*.

3. Tester les fonctions sur le graphe figure 17.3 en partant du sommet 0. Ajouter un `print(w)` au moment où vous visitez un noeud. Comprenez vous pourquoi on parle de parcours en *profondeur* et en *largeur* ?

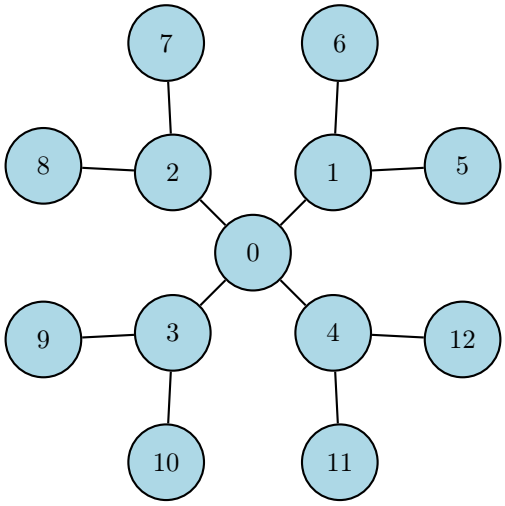


FIGURE 17.3 – Un graphe en flocon de neige

4. Déduire des fonctions précédentes une fonction `connexe(M)` qui détermine si le graphe représenté par la matrice d'adjacence `M` est connexe.

17.2 Parcours de graphe, version récursive

Rappel sur la récursivité

Une fonction Python f a le droit de *faire appel à elle-même*. L'exemple classique est la fonction `factoriel(n)` qui peut s'écrire

```
def factoriel(n):
    if n == 0:
        return 1
    else:
        return n * factoriel(n-1)
```

en utilisant le fait que $n! = n \cdot (n-1)!$.

De manière générale, lorsque l'on écrit une fonction récursive, il y a une partie qui assure la *terminaison*, c'est-à-dire qui s'assure que la fonction va finir par arêter de s'appeler elle-même, et une partie qui "fait le travail". Ici, le test `if n == 0` est là pour assurer la terminaison. Pensez aux similarités avec le raisonnement par récurrence qui demande d'écrire le cas d'initialisation et l'hérédité.

5. Écrire de façon récursive une fonction `fibonacci(n)` qui renvoie le n-ième terme de la suite de Fibonacci $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$.

Parcours récursif d'un graphe

On va maintenant écrire une fonction `parcours(v, M, dejaVus)` qui parcourt tous le graphe et va s'appeler récursivement sur les voisins des sommets visités. `v` est le noeud en train d'être visité, `M` est la matrice d'adjacence du graphe, et `dejaVus` est un tableau de booléens de taille `n`, qui indique quel sommet a déjà été visité.

Pour parcourir un graphe depuis v, il suffit de parcourir le graphe depuis tous les voisins de v qui n'ont pas déjà été vus (en les ayant préalablement marqués comme vus). C'est cette vision récursive d'un parcours de graphe qui permet d'écrire la fonction `parcours`,

6. a) Écrire la fonction `parcours(v, M, dejaVus)`.

b) La tester sur les graphes figure 17.2.

7. La version récursive de l'algorithme effectue-t-elle un parours en largeur ou un parcours en profondeur ?

Pour les plus rapides

8. Écrire une fonction `chemins(M, v, n)` qui obtient la liste de tous les chemins de longueurs inférieur ou égal à `n` partant de `v` dans le graphe représenté par sa matrice d'adjacence `M`. Bonus : faire en sorte que la liste de ces chemins soient triés par ordre lexicographique.

Un cycle est une chemin de longueur $\geqslant 1$ qui commence et termine au même noeud.

9. Écrire une fonction `aUnCycle(M)` qui détermine si le graphe *non-orienté* G, représenté par sa matrice d'adjacence `M`, possède un cycle. Vérifier en particulier que `aUnCycle` donne le résultat attendu sur le graphe figure 17.4.

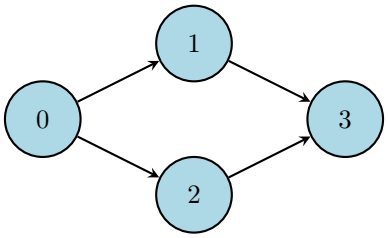


FIGURE 17.4 – Le graphe "diamant"