Chapitre n°5

Graphes : intro et parcours

Table des matières

1 Notion de graphe **1**
1.1 Introduction 1
1.2 Un peu de vocabulaire lié aux graphes 2

2 Implémentation(s) d'un graphe **4**
2.1 Implémentation à l'aide d'un dictionnaire 4
2.2 Implémentation à l'aide d'une liste d'adjacence 5
2.3 Implémentation à l'aide d'une matrice d'adjacence 6

3 Identification de chaîne / de chemin dans un graphe **7**
3.1 Chaîne, chemin 7
3.2 Connexité 8

4 Intermède : deque, pile, file **9**
4.1 Les deques 9
4.2 Structure de file (d'attente) 10
4.3 Structure de pile 11

5 Structures de données pour le parcours d'un graphe **11**

6 Parcours d'un graphe **12**
6.1 Notion d'ordre de parcours 12
6.2 Parcours en largeur d'un graphe 13
6.3 Parcours en profondeur d'un graphe 15

À l'issue de ce cours **18**

1. Notion de graphe

1.1. Introduction

Les graphes constituent une classe d'objets qu'on retrouve un peu partout en Informatique :

- structure des réseaux de communication (fibre optique, lignes ADSL, etc.) ;
- structure des réseaux sociaux ;
- arborescence de sites Web ;
- etc.

mais également dans bien d'autres domaines :

- structure des réseaux routiers, ferroviaires, de distribution d'eau, d'électricité...
- relations entre auteurs en publication scientifique ;
- représentation de molécule en Chimie organique ;
- structure cristalline en cristallographie ;
- séquençage en Biologie ;
- etc.

On ne va considérer, dans ce cours, qu'un sous-ensemble limité de graphes présentant des propriétés relativement simples.

1.2. Un peu de vocabulaire lié aux graphes

Définition : graphe non-orienté

Un graphe non-orienté G est déterminé par un couple d'ensembles (S, A) où :

- **S désigne un ensemble fini de sommets s_i (ou nœuds) du graphe (notés ⓪, ①, ...).**
- **A désigne un ensemble A de couples non-ordonnés $\{s_i, s_j\} \subset S \times S$ appelés arêtes du graphe (notées —).**

Deux sommets s_i et s_j sont dits adjacents si $\{s_i, s_j\} \in A$.

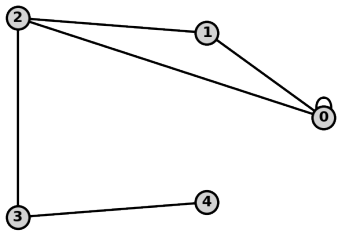


Fig. 1 – exemple de graphe non-orienté

Ex : le graphe $G = (S, A)$ ci-dessus est un exemple de graphe non-orienté. L'ensemble S de ses sommets est $\{0, 1, 2, 3, 4\}$. L'ensemble A de ses arêtes est $\{\{0, 0\}, \{0, 1\}, \{0, 2\}, \{1, 2\}, \{2, 3\}, \{3, 4\}\}$.

Définition : graphe orienté

Un graphe orienté G est déterminé par un couple d'ensembles (S, A) où :

- **S désigne un ensemble fini de sommets s_i (ou nœuds) du graphe (notés ⓪, ①, ...).**
- **A désigne un ensemble de couples ordonnés $(s_i, s_j) \in S^2$ appelés arêtes orientées (ou arcs) du graphe (notées $\longrightarrow$).**

L'arc (s_i, s_j) est dit sortant en s_i et entrant en s_j.

Si $(s_i, s_j) \in A$:
- le sommet s_i est appelé un prédécesseur de s_j
- le sommet s_j est appelé un successeur de s_i.

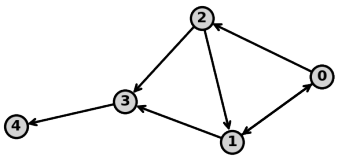


Fig. 2 – exemple de graphe orienté

Ex : le graphe $G = (S, A)$ ci-dessus est un exemple de graphe orienté. L'ensemble S de ses sommets est $\{0, 1, 2, 3, 4\}$. L'ensemble A de ses arcs est $\{(0, 1), (0, 2), (1, 0), (1, 3), (2, 1), (2, 3), (3, 4)\}$.

Définition : graphe simple

Un graphe multiple est un graphe pouvant présenter plusieurs liens (arêtes ou arcs) entre deux sommets.

Une boucle est un lien (arête ou arc) reliant un sommet à lui-même.

Un graphe simple est un graphe ne présentant ni boucle, ni lien multiple.

Nota : dans le cadre de ce cours, on se limitera principalement à l'étude des graphes simples.

Définition : ordre et taille d'un graphe

L'ordre d'un graphe est le nombre de sommets de ce graphe, soit $Card(S)$.

La taille d'un graphe est le nombre d'arêtes de ce graphe, soit $Card(A)$.

Définition : degré(s) des sommets d'un graphe

- **dans un graphe non-orienté, le degré $d(s_i)$ d'un sommet s_i est égal au nombre de sommets s_j adjacents au sommet s_i ;**
- **dans un graphe orienté, le degré sortant $d^+(s_i)$ d'un sommet s_i est égal au nombre d'arcs du type (s_i, s_j) ;**
- **dans un graphe orienté, le degré entrant $d^-(s_i)$ d'un sommet s_i est égal au nombre d'arcs du type (s_j, s_i).**

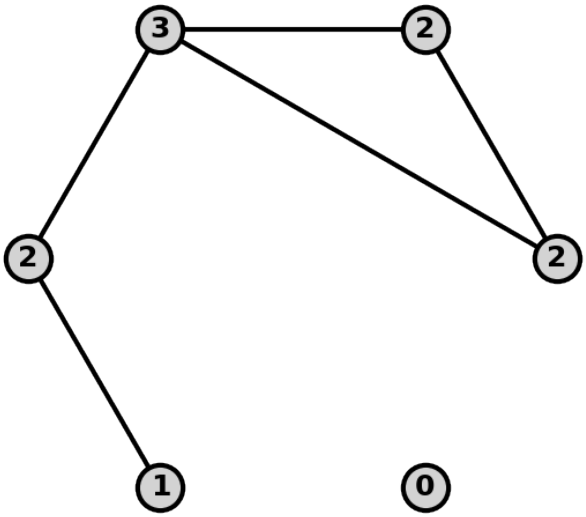


Fig. 3 – graphe simple non-orienté d'ordre 6 où chaque sommet s_i est étiqueté en fonction de son degré $d(s_i)$

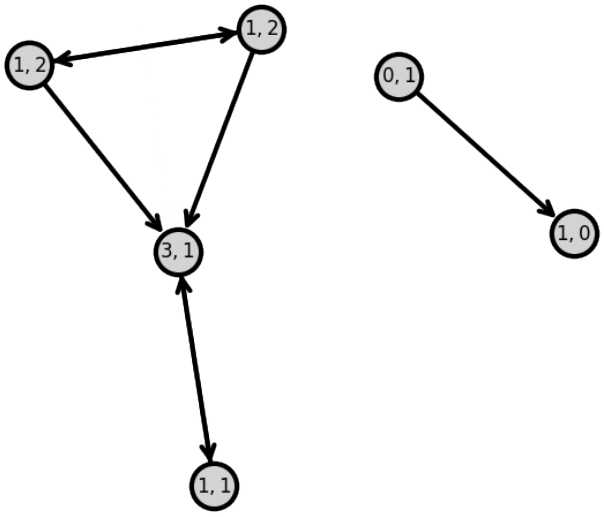


Fig. 4 – graphe simple orienté d'ordre 6 où chaque sommet s_i est étiqueté par un couple (m, n) avec $m = d^-(s_i)$ et $n = d^+(s_i)$

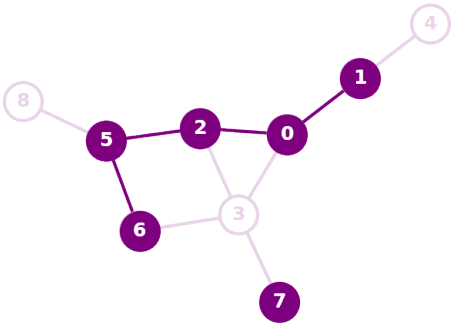
Définition : sous-graphe

Un sous-graphe (induit) du graphe $G = (S, A)$ est un graphe $H = (T, B)$ tel que :

$$T \subset S \qquad \text{et} \qquad B = \{(s_i, s_j) \in A \,|,\, s_i \in T\,,\, s_j \in T\}$$

Autrement dit, on prend un sous-ensemble T de nœuds contenu dans S, et on ne garde dans B que les arêtes du graphe qui relient deux points de T.

Ex : le graphe $H = (T, B)$ (en gras) ci-contre est le sous-graphe du graphe $G = (S, A)$ (en clair) ci-contre obtenu en choisissant $T = \{0, 1, 2, 5, 6, 7\} \subset S$:



2. Implémentation(s) d'un graphe

2.1. Implémentation à l'aide d'un dictionnaire

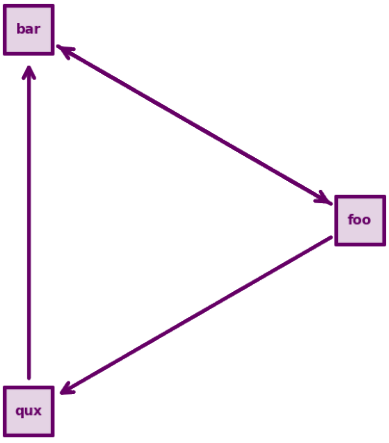
L'implémentation la plus générale d'un graphe consiste à stocker toutes les informations (sommets et arêtes ou arcs) sous la forme d'un **dictionnaire**. Chaque clef du dictionnaire correspond à un sommet ; elle est associée à une liste contenant les clefs correspondant aux sommets adjacents à celui-ci.

Ex : le dictionnaire suivant :

Code Python

```
>>> G = {
...       "foo" : ["bar", "qux"],
...       "bar" : ["foo"],
...       "qux" : ["bar"]
...     }
```

code ainsi pour le graphe orienté ci-contre :



Nota : avec ce type d'implémentation, il est délicat de manipuler des graphes non-orientés car l'information « s_i et s_j sont adjacents » correspond à une seule arête, mais à deux voisins dans les listes associées respectivement aux sommets s_i et s_j.

Ce type de représentation présente un certain nombre d'avantages et d'inconvénients :

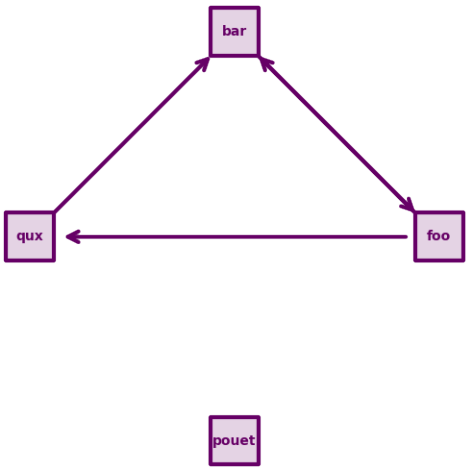
- il est facile ajouter un sommet à un graphe déjà existant en ajoutant une nouvelle clef au dictionnaire :

Ex : par exemple, l'instruction :

Code Python

```
>>> G["pouet"] = []
```

conduit à la mise à jour du graphe précédent pour conduire au graphe ci-contre :



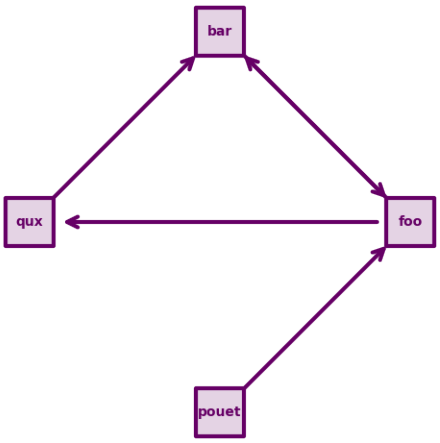
- il est facile d'ajouter un arc à un graphe déjà existant en ajoutant une nouvelle valeur dans la liste associée à une des clefs du dictionnaire :

Ex : par exemple, l'instruction :

Code Python

```
>>> G["pouet"] = G["pouet"] + ["foo"]
```

conduit à la mise à jour du graphe précédent pour conduire au graphe ci-contre :



- en pratique, ce type d'implémentation est très peu efficace – en effet, il est nécessaire de parcourir la liste complète des clefs du dictionnaire lorsqu'on cherche un sommet en particulier. On réservera donc ce type d'implémentation aux cas où le nombre de sommets du graphe n'est pas statique.

2.2. Implémentation à l'aide d'une liste d'adjacence

Lorsque le nombre de sommets du graphe est fixé, il est possible de mettre en place des structures beaucoup plus efficaces pour représenter et manipuler un graphe. On suppose par la suite que l'ordre du graphe $G = (S, A)$ étudié est de $n = Card(S)$, et que les sommets sont étiquetés de 0 à $n - 1$.

Définition : liste d'adjacence

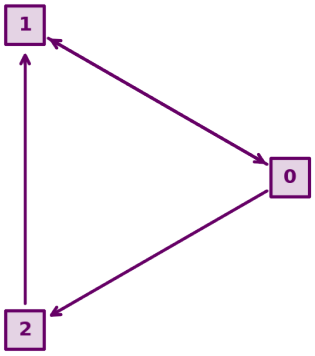
Une liste d'adjacence `L` d'un graphe est une liste telle que le i-ième élément, donc `L[i]`, est (encore) une liste qui contient tous les voisins du sommet s_i.

Ex : la liste suivante :

Code Python

```
>>> G = [[1, 2], [0], [1]]
```

code ainsi pour le graphe orienté ci-contre :



2.3. Implémentation à l'aide d'une matrice d'adjacence

Les listes d'adjacence sont particulièrement adaptées à l'implémentation de graphes « creux », c'est-à-dire de graphes où il y a peu de liens (arêtes ou arcs) par rapport au nombre de sommets. Cependant, dans le cas de graphes « denses », c'est-à-dire de graphes présentant un nombre important d'arêtes ou d'arcs par rapport au nombre de sommets, l'utilisation de listes d'adjacence peut devenir particulièrement laborieuse. On peut alors utiliser une **matrice d'adjacence** pour représenter le graphe étudié.

Définition : matrice d'adjacence

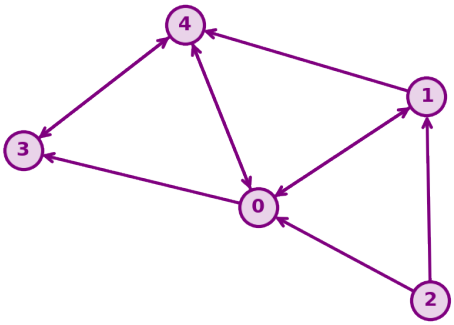
Une matrice d'adjacence M est une implémentation d'un graphe $G = (S, A)$ dans laquelle l'élément de matrice M_{ij} est défini par :

$$M_{ij} = \begin{cases} 1 & \text{si } (s_i, s_j) \in A \\ 0 & \text{sinon} \end{cases}$$

Ex : la matrice suivante :

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

code ainsi pour le graphe orienté ci-contre :



Nota : les termes diagonaux d'une matrice d'adjacence sont – par construction – nuls pour un graphe simple, soit :

$$\forall i \in Card(S) \quad M_{ii} = 0$$

Nota : la matrice d'adjacence d'un graphe non-orienté est – par construction – symétrique, soit :

$$\forall (i,j) \in (Card(S))^2 \quad M_{ij} = M_{ji}$$

Pour des exercices sur les listes et matrices d'adjacence, cf TP 13 d'introduction aux graphes.

3. Identification de chaîne / de chemin dans un graphe

3.1. Chaîne, chemin

Afin de parcourir un graphe, il est nécessaire de pouvoir établir l'existence de **chaîne** (pour un graphe non-orienté) ou de **chemin** (pour un graphe orienté) reliant deux sommets d'un graphe :

Définition : chaîne

Dans un graphe non-orienté, une chaîne $\mathcal{C}_{s_i—s_j}$ allant du sommet s_i au sommet s_j est une suite finie d'arêtes définie par :

$$\begin{cases} u_0 = s_i \\ u_\ell = s_j \\ \forall k \in [\![0, \ell-1]\!] \, , \, \{u_k \, , \, u_{k+1}\} \in A \end{cases}$$

- **Une chaîne est dite de longueur ℓ s'il passe par ℓ arêtes.**
- **Une chaîne est élémentaire si tous les sommets $u_0, u_1, \cdots, u_\ell$ sont distincts deux à deux.**
- **Une chaîne est simple si toutes les arêtes $\{u_k \, , \, u_{k+1}\}$ empruntées sont distinctes deux à deux.**
- **Une chaîne pour laquelle $u_0 = u_\ell$ est un cycle.**
- **On parle de cycle élémentaire si tous les sommets $u_1, u_2, \cdots, u_{\ell-1}$ sont distincts deux à deux (mais on a bien entendu $u_0 = u_\ell$).**
- **On peut également parler de cycle simple : c'est une chaîne simple qui est aussi un cycle.**

Définition : chemin

Dans un graphe orienté, un chemin $\mathcal{C}_{s_i \longrightarrow s_j}$ allant du sommet s_i au sommet s_j est une suite finie d'arcs définie par :

$$\begin{cases} u_0 = s_i \\ u_\ell = s_j \\ \forall k \in [\![0, \ell-1]\!] \, , \, (u_k \, , \, u_{k+1}) \in A \end{cases}$$

- **Un chemin est dit de longueur ℓ s'il passe par ℓ arêtes.**
- **Un chemin est élémentaire si tous les sommets $u_0, u_1, \cdots, u_\ell$ sont distincts deux à deux.**
- **Un chemin est simple si toutes les arêtes $(u_k \, , \, u_{k+1})$ empruntées sont distinctes deux à deux.**
- **Un chemin pour lequel $u_0 = u_\ell$ est un cycle (ou circuit).**
- **On parle de chemin élémentaire si tous les sommets $u_1, u_2, \cdots, u_{\ell-1}$ sont distincts deux à deux (mais on a bien entendu $u_0 = u_\ell$).**
- **On peut également parler de cycle simple : c'est un chemin simple qui est aussi un cycle.**

Dès lors qu'il existe une chaîne / un chemin entre deux sommets, il est possible de trouver une chaîne / un chemin élémentaire d'après le lemme de KÖNIG :

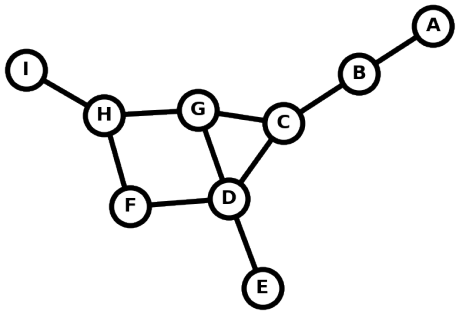
LOI : LEMME DE KÖNIG

Dans un graphe, s'il existe une chaîne / un chemin du sommet s_i vers le sommet s_j alors il existe une chaîne / un chemin élémentaire du sommet s_i vers le sommet s_j.

APPLICATION

On considère le graphe ci-contre :

- Identifier un chemin non-élémentaire du sommet H au sommet B.
- Déterminer un chemin élémentaire du sommet H au sommet B.
- Déterminer un cycle simple élémentaire de longueur 5 à partir de H.



3.2. Connexité

DÉFINITION : CONNEXITÉ (GRAPHES NON-ORIENTÉS)

Un graphe non-orienté $G = (S, A)$ est connexe si pour tout couple de points (s_i, s_j), il existe une chaîne qui relie s_i à s_j :

$$\forall (s_i, s_j) \in S^2 \quad \exists \mathcal{C}_{s_i \text{—} s_j}$$

DÉFINITION : FORTE CONNEXITÉ (GRAPHES ORIENTÉS)

Un graphe orienté $G = (S, A)$ est fortement connexe si pour tout couple de points (s_i, s_j), il existe un chemin qui relie s_i à s_j :

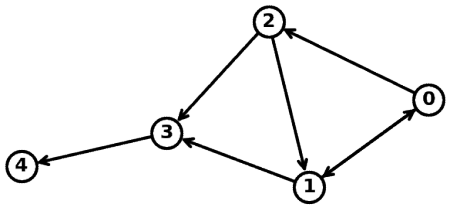
$$\forall (s_i, s_j) \in S^2 \quad \exists \mathcal{C}_{s_i \longrightarrow s_j}$$

Nota : on peut également parler de connexité dans un graphe orienté. Dans ce cas, on considère le graphe non-orienté correspondant.

Application

On considère le graphe orienté ci-contre :

- Ce graphe est-il connexe ?
- Ce graphe est-il fortement connexe ?



- Proposer une manière d'obtenir un graphe fortement connexe à partir de ce graphe en ajoutant une seule arête.

4. Intermède : deque, pile, file

4.1. Les deques

En Python, on peut stocker des données de plusieurs manières : la plus courante est d'utiliser une liste `L`, et d'accéder aux éléments avec l'instruction `L[i]`. Ainsi, pour une liste, on doit garder en mémoire la place de chaque élément dans la liste (il y a un ordre qui doit être conservé).
Cela est en général bien pratique, mais cela peut entrainer un coût, notamment lorsqu'on veut ajouter un élément `u` au début d'une liste `L`, par exemple avec l'instruction

```
M = [u] + L
```

Si `len(L)=n` Cette opération a en fait une complexité en $O(n)$: en effet pour mettre en place une liste, il faut faire pointer `M[0]` sur `u`, puis `M[1]` sur `L[0]`, etc. soit $n+1$ opérations de pointage.
L'objet **deque** (Double End QUEue) permet de stocker les données sans pointage, de sorte qu'on puisse ajouter ou enlever les éléments au début ou à la fin avec un coût constant en $O(1)$:

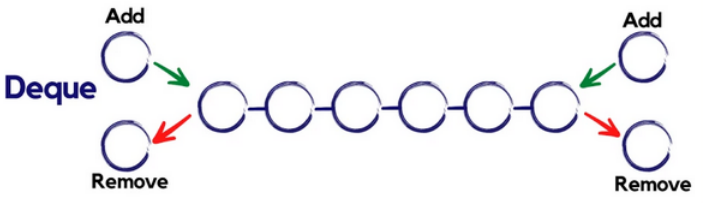


Fig. 5 – exemple de fonctionnement d'un deque

Cependant, les deques ont un désavantage : toute opération qui se situe "vers le milieu" est plus longue. Par exemple si `D` est un deque de longueur n paire, alors l'évaluation `D[n//2])` aura une complexité en $O(n)$. On verra plus d'informations sur le TP d'introduction aux graphes.

Les **files** et **piles** sont des cas particuliers de deques, avec moins d'opérations disponibles :

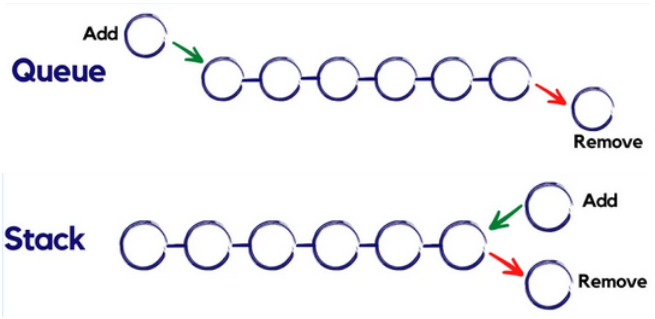


Fig. 6 – exemple de fonctionnement d'une pile (stack) et d'une file (queue)

Détaillons les structures de files et de piles :

4.2. Structure de file (d'attente)

Définition : file d'attente

Une file d'attente (ou file, *queue*) est une structure de données reposant sur le principe du « premier entré, premier sorti » (FIFO pour *first in, first out*). Cette structure de données admet un nombre très limité d'opérations élémentaires :

- **déterminer si la file est vide ou non ;**
- **ajouter un élément en queue de file (« enfiler » ou *enqueue*) ;**
- **retirer l'élément en tête de file (« défiler » ou *dequeue*) si celle-ci est non-vide.**

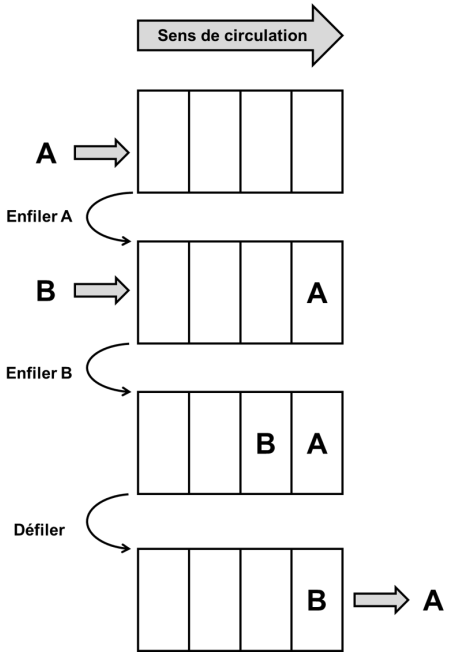


Fig. 7 – exemple de fonctionnement d'une file d'attente

Ce type de structure est celui correspondant intuitivement à la file d'attente à la caisse d'une boutique – les premiers clients faisant la queue sont les premiers à être servis. Les files sont utilisées dans de nombreux domaines en Informatique : file d'attente d'impression d'une imprimante en réseau, exécution de tâches dans l'ordre où elles ont été demandées, etc.

4.3. Structure de pile

Définition : pile

Une pile (*stack*) est une structure de données reposant sur le principe du « dernier entré, premier sorti » (LIFO pour *last in, first out*). Cette structure de données admet un nombre très limité d'opérations élémentaires :

- **déterminer si la pile est vide ou non ;**
- **ajouter un élément sur le dessus de la pile (« empiler » ou *push*) ;**
- **retirer l'élément du dessus de la pile (« dépiler » ou *pop*) si celle-ci est non-vide.**

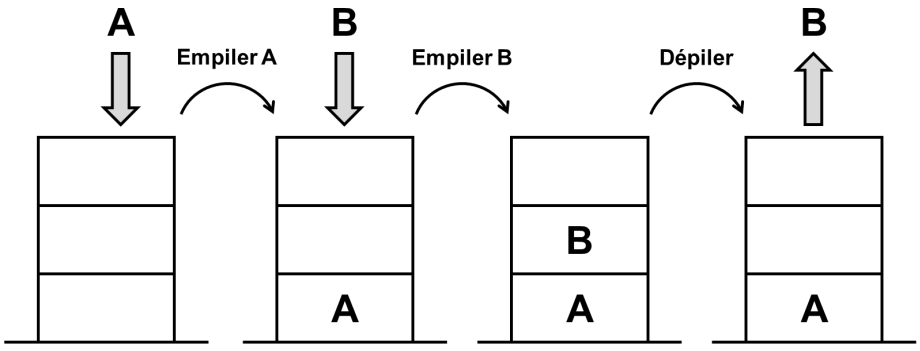


Fig. 8 – exemple de fonctionnement d'une pile

Ce type de structure est celui correspondant intuitivement à celui d'une pile de crêpes – la dernière crêpe cuite est celle qui atterrit sur le dessus de la pile ; c'est également la première à être mangée. Ce type de structure est extrêmement courante en Informatique : les boutons « page précédente » et « page suivante » en navigation web, les commandes « annuler » et « répéter » dans un éditeur de textes, etc. Une fonction récursive va également traiter les différents appels selon une structure de pile où chaque nouvel appel vient prendre la priorité sur les appels qui n'ont pas encore été traités.

5. Structures de données pour le parcours d'un graphe

Le parcours d'un graphe permet de répondre à différentes questions comme :

- quels sont les sommets accessibles à partir d'un sommet d'origine donné ?
- quel est le plus court chemin entre deux sommets d'un graphe ?
- un graphe est-il connexe ?
- etc.

Un parcours de graphe est défini par le choix d'un sommet d'origine, et d'une méthode pour déterminer comment le graphe va être exploré.

On va voir deux algorithmes de parcours. Dans chacun, les sommets sont divisés en trois catégories :

- Les sommets « cachés » n'ont pas encore été atteint par l'algorithme. Au tout début de l'algorithme, tous les sommets sont cachés, sauf le sommet de départ qui est classé dans la catégorie "à visiter".
- Les sommets « à visiter » ont été identifiés par l'algorithme mais pas encore visités (ils le seront ultérieurement). Tant qu'il reste des sommets à visiter, l'algorithme va continuer de les explorer.
- Les sommets « (déjà) visités » ont été explorés par l'algorithme. Il n'est plus nécessaire de les revisiter à nouveau.

⋆ Une quatrième catégorie concerne les sommets « découverts » : ce sont les sommets à visiter ou déjà visités. On peut donc atteindre ces sommets en partant du sommet de départ.

Chaque sommet accessible depuis le sommet d'origine sera successivement caché, découvert, puis visité par l'algorithme. Une fois que l'algorithme aura tout exploré, les sommets découverts seront ceux qui sont accessibles depuis le sommet de départ.

6. Parcours d'un graphe

6.1. Notion d'ordre de parcours

On va voir deux algorithmes de parcours : le parcours en largeur et le parcours en profondeur. La différence entre les deux tient en l'ordre dont on va visiter les sommets « découverts » :

- Avec le **parcours en largeur** du graphe (BFS pour *breadth-first search*), l'ensemble des sommets à visiter est une file. Dès qu'un nouveau sommet est découvert, il est enfilé sur la file d'attente des sommets à visiter : il ne sera visité que lorsque tous les sommets qui le précèdent dans la file auront été visités.
- Avec le **parcours en profondeur** du graphe (DFS pour *depth-first search*), l'ensemble des sommets à visiter est une pile. Dès qu'un nouveau sommet est découvert, il est empilé sur le dessus de la pile des sommets à visiter : il sera alors visité immédiatement (sauf si on découvre plusieurs sommets en même temps, auquel cas c'est le dernier empilé qui sera visité ensuite).

6.2. Parcours en largeur d'un graphe

Le parcours en largeur d'un graphe repose sur l'utilisation d'une file d'attente. On présente ci-dessous un exemple de programme en pseudo-code. Présentons le rôle de quelques variables de ce programme :

- `D` : liste de booléens. `D(i)` (ou `D[i]` en Python) vaut `VRAI` si le sommet `i` a été découvert.
- `F` : file d'attente qui correspond aux sommets à visiter.
- `L` : liste des sommets visités, dans l'ordre où ils ont été visités.

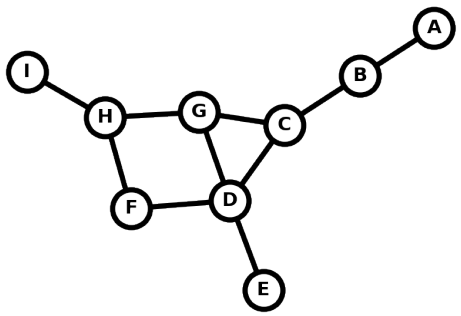
Algorithme : parcours en largeur d'un graphe

Entrée : graphe `G`, sommet de départ `start`
Sortie : `L`

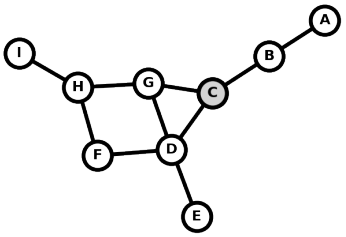
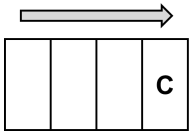
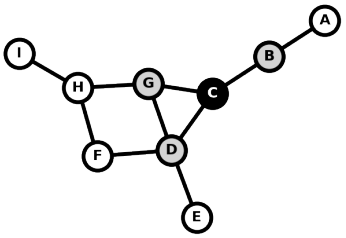
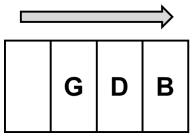
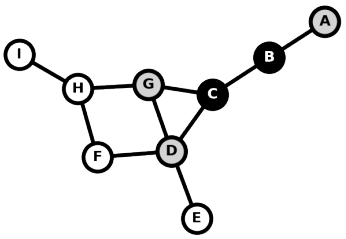
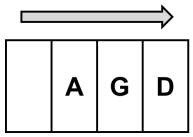
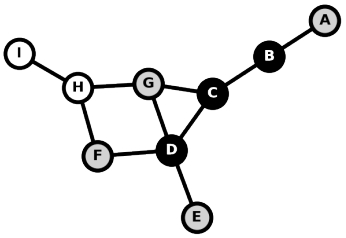
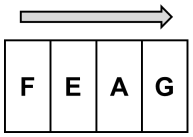
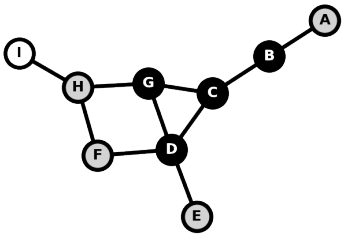
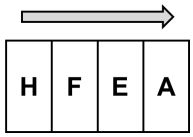
```
1  n ⟵ ordre de G        (i.e. nombre de sommets de G)
2  D ⟵ liste contenant n fois FAUX
3  L ⟵ liste vide
4  F ⟵ file vide
5  enfiler start sur F
6  D(start) ⟵ VRAI
7  Tant que F ≠ file vide faire :
8      i ⟵ défiler F
9      ajouter l'élément i à la liste L
10     V ⟵ liste des indices des voisins du sommet d'indice i du graphe G
11     Pour j dans V faire :
12         Si D(j) = FAUX alors :
13             enfiler j sur F
14             D(j) ⟵ VRAI
15 Retourner L
```

Expliquons les lignes 7 à 14. Tant qu'il y a des points à visiter (`F` non vide), on défile un sommet (noté `i`) de `F` : on visite ce sommet `i` donc on l'ajoute à `L`.
Une fois le sommet `i` visité, on récupère la liste de ces voisins. Pour chaque sommet voisin `j`, s'il n'a pas déjà été découvert (i.e. visité ou placé dans la file "à visiter"), alors il faudra l'explorer : on l'enfile dans `F` et on notifie qu'on l'a découvert.

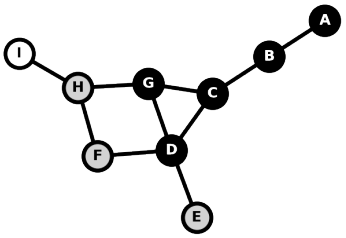
Afin d'illustrer le parcours en largeur d'un graphe, on considère le graphe ci-contre :

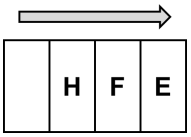


On cherche à parcourir en largeur le graphe en partant du sommet C. Lorsqu'un sommet est découvert, il est colorié en gris jusqu'à ce qu'il soit visité – il est alors colorié en noir :

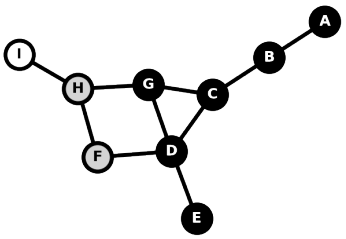
Description de l'étape	Représentation du graphe	Représentation de la file
▶ initialisation : le sommet C est enfilé dans la file `F` ;		
▶ le sommet C est visité : il est défilé de la file `F` et est placé dans la liste `L` (rang 0). Ses voisins sont les sommets B, D et G (découverts selon l'ordre alphabétique) : ceux-ci sont enfilés (dans cet ordre) dans la file `F` ;		
▶ le sommet B est visité : il est défilé de la file `F` et est placé dans la liste `L` (rang 1). Son unique voisin est le sommet A : celui-ci est enfilé dans la file `F` ;		
▶ le sommet D est visité : il est défilé de la file `F` et est placé dans la liste `L` (rang 2). En regardant ses voisins, on découvre (dans cet ordre) les sommets E et F : ceux-ci sont enfilés (dans cet ordre) dans la file `F`. C et G ont déjà été découverts et ne sont donc pas enfilés ;		
▶ le sommet G est visité : il est défilé de la file `F` et est placé dans la liste `L` (rang 3). Son unique voisin non découvert est le sommet H : celui-ci est enfilé dans la file `F` ;		

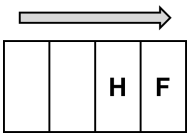
- le sommet A est visité : il est défilé de la file `F` et est placé dans la liste `L` (rang 4). Il n'a pas de voisin n'ayant pas encore été découvert : on ne modifie donc pas la file `F`. ;



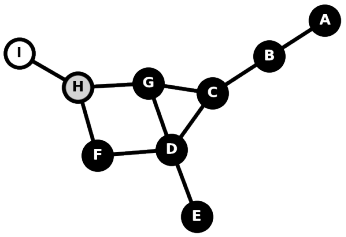


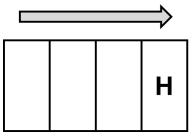
- le sommet E est visité : il est défilé de la file `F` et est placé dans la liste `L` (rang 5). Il n'a pas de voisin n'ayant pas encore été découvert ;



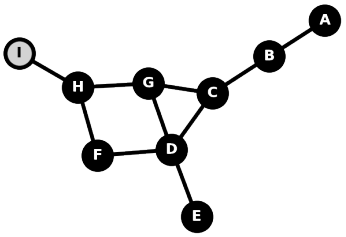


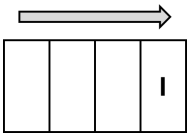
- le sommet F est visité : il est défilé de la file `F` et est placé dans la liste `L` (rang 6). Il n'a pas de voisin n'ayant pas encore été découvert ;



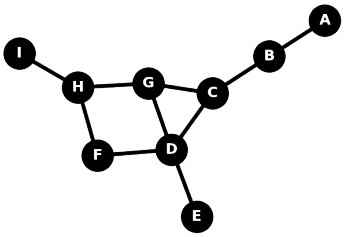


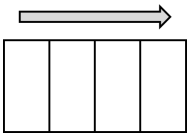
- le sommet H est visité : il est défilé de la file `F` et est placé dans la liste `L` (rang 7). Son unique voisin non découvert est le sommet I : celui-ci est enfilé dans la file `F` ;





- le sommet I est visité : il est défilé de la file `F` et est placé dans la liste `L` (rang 8). La file `F` est vide et l'algorithme se termine.





6.3. Parcours en profondeur d'un graphe

Le parcours en profondeur d'un graphe repose sur l'utilisation d'une pile. On présente ci-dessous un exemple de programme en pseudo-code. Présentons le rôle de quelques variables de ce programme :

- `T` : liste de booléens. `T(i)` (ou `T[i]` en Python) vaut `VRAI` si le sommet `i` a été visité.
- `P` : pile qui correspond aux sommets à visiter.
- `L` : liste des sommets visités, dans l'ordre où ils ont été visités.

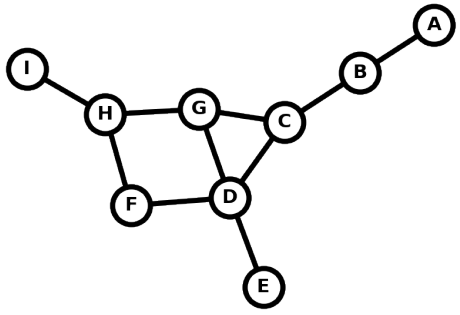
ALGORITHME : parcours en profondeur d'un graphe

Entrée : graphe `G`, sommet de départ `start`
Sortie : `L`

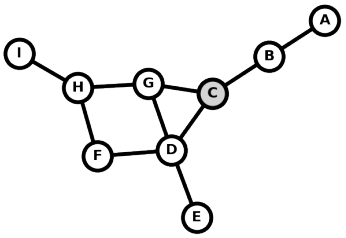
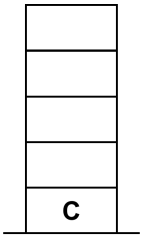
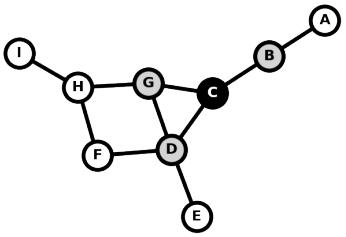
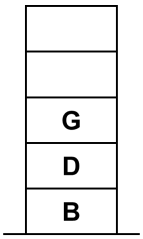
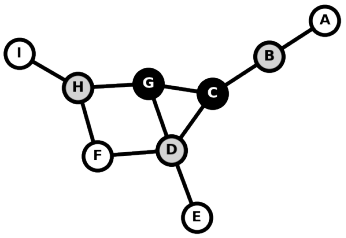
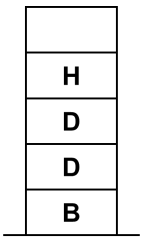
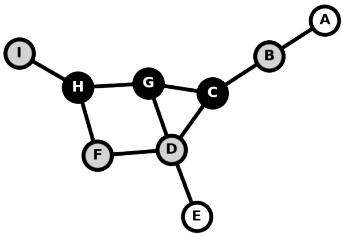
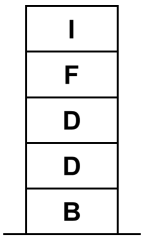
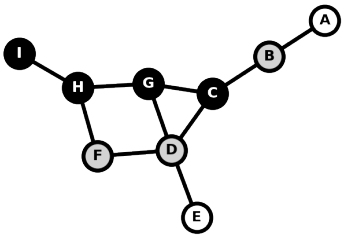
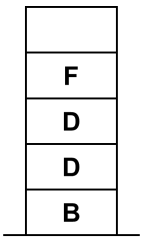
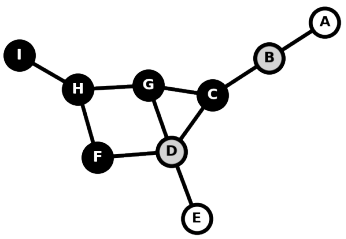
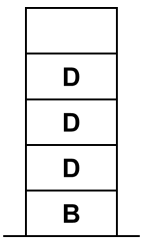
```
1  n ⟵ ordre de G        (i.e. nombre de sommets de G)
2  T ⟵ liste avec n fois FAUX
3  L ⟵ liste vide
4  P ⟵ pile vide
5  empiler start sur P
6  Tant que P ≠ pile vide faire :
7      i ⟵ dépiler P
8      Si T(i) = FAUX alors :
9          T(i) ⟵ VRAI
10         ajouter l'élément i à la liste L
11         V ⟵ liste des indices des voisins du sommet d'indice i du graphe G
12         Pour j dans V faire :
13             Si T(j) = FAUX alors :
14                 empiler j sur P
15 Retourner L
```

Le parcours en profondeur se démarque du parcours en largeur de plusieurs façons : il y a bien sûr le fait que l'ensemble des sommets à visiter est maintenant une pile et non une file. Or, du fait que ce soit une pile, un même sommet qui n'a pas encore été visité peut se retrouver présent plusieurs fois dans la pile (cf exemple ci-dessous).
Ainsi, quand on dépile la pile, le sommet `i` peut avoir déjà été visité, auquel cas on ne fait rien. Par contre, si le sommet `i` n'a pas été visité, alors on le visite en l'ajoutant à `L`. Ensuite, on ajoute à la pile `P` les voisins de `i` qui n'ont pas déjà été visités.

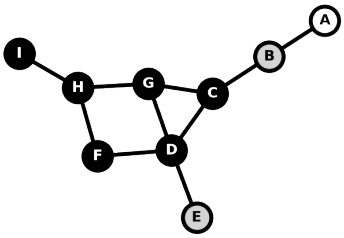
Afin d'illustrer le parcours en profondeur d'un graphe, on considère le même graphe que précédemment :

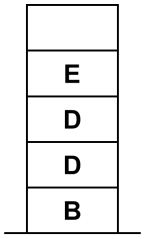


On cherche à parcourir en profondeur le graphe en partant du sommet C. Lorsqu'un sommet est découvert, il est colorié en gris jusqu'à ce qu'il soit visité – il est alors colorié en noir :

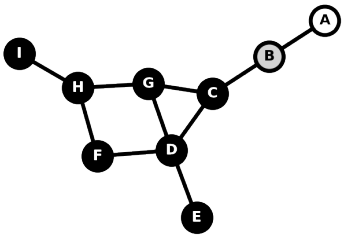
Description de l'étape	Représentation du graphe	Représentation de la pile
▶ initialisation : le sommet C est empilé sur la pile `P` ;		
▶ le sommet C est visité : il est dépilé de la pile `P` et est placé dans la liste `L` (rang 0). Ses voisins sont les sommets B, D et G (découverts dans cet ordre) : ceux-ci sont empilés (dans cet ordre) sur la pile `P` ;		
▶ le sommet G est visité : il est dépilé de la pile `P` et est placé dans la liste `L` (rang 1). Le sommet C a déjà été visité : il est donc ignoré. Ses deux autres voisins sont les sommets D et H : ceux-ci sont empilés (dans cet ordre) sur la pile `P` ;		
▶ le sommet H est visité : il est dépilé de la pile `P` et est placé dans la liste `L` (rang 2). Le sommet G a déjà été visité : il est donc ignoré. Ses deux autres voisins sont les sommets F et I : ceux-ci sont empilés (dans cet ordre) sur la pile `P` ;		
▶ le sommet I est visité : il est dépilé de la pile `P` et est placé dans la liste `L` (rang 3). Il n'a pas de voisin n'ayant pas encore été visité ;		
▶ le sommet F est visité : il est dépilé de la pile `P` et est placé dans la liste `L` (rang 4). Son unique voisin n'ayant pas encore été visité est le sommet D : celui-ci est empilé sur la pile `P` ;		

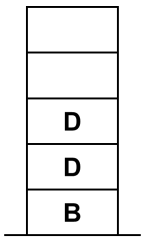
- le sommet D est visité : il est dépilé de la pile `P` et est placé dans la liste `L` (rang 5). Son unique voisin n'ayant pas encore été visité est le sommet E : celui-ci est empilé dans la pile `P` ;



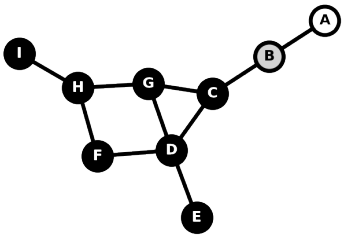


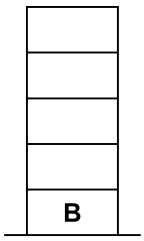
- le sommet E est visité : il est dépilé de la pile `P` et est placé dans la liste `L` (rang 6). Il n'a pas de voisin n'ayant pas encore été visité ;



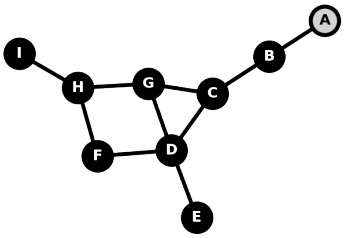


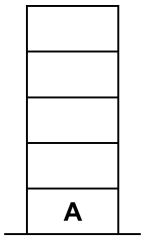
- le sommet D a déjà été visité : il est dépilé de la pile `P` et ignoré (deux fois) ;



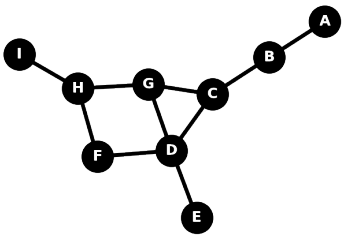


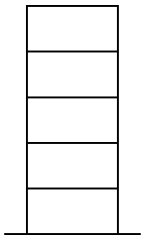
- le sommet B est visité : il est dépilé de la pile `P` et est placé dans la liste `L` (rang 7). Son unique voisin restant est le sommet A : celui-ci est empilé dans la pile `P` ;





- le sommet A est visité : il est dépilé de la pile `P` et est placé dans la liste `L` (rang 8). La pile `P` est vide et l'algorithme se termine.





À l'issue de ce cours

À l'issue de ce cours, je suis capable :

- ☐ de définir les notions de chaîne (graphe non-orienté) et de chemin (graphe orienté) ;
- ☐ d'identifier un cycle (graphe non-orienté) ou un circuit (graphe orienté) dans un graphe ;
- ☐ de connaitre les notions de file d'attente, pile et deque ;
- ☐ de parcourir en largeur un graphe en s'appuyant sur la structure de file d'attente ;
- ☐ de parcourir en profondeur un graphe en s'appuyant sur la structure de pile ;