DIU Enseigner l'informatique au lycée

Notes de cours

Jean-Christophe Filliâtre

Jean-Christophe.Filliatre@lri.fr

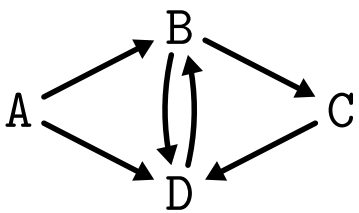
Graphes

définition

Un graphe est un ensemble de sommets

reliés entre eux par des arcs.

Exemple :

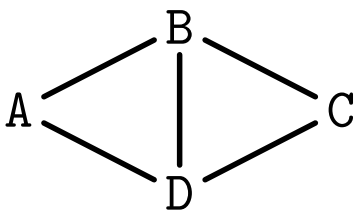


Il y a un arc du sommet `A` vers le sommet `B`, mais pas du sommet `B` vers le sommet `A`. On parle de graphe orienté.

graphe non orienté

Lorsqu'en revanche le sens des arcs n'est pas significatif, on parle de graphe non orienté.

Exemple :



graphes simples

On se limite ici à des graphes simples, c'est-à-dire

- au plus un arc entre deux sommets ;
- pas d'arc reliant un sommet à lui-même.

exemples de graphes

réseau routier

villes / routes entre ces villes

exemples de graphes

réseau routier

villes / routes entre ces villes

réseau informatique

machines / connections entre ces machines

exemples de graphes

réseau routier

villes / routes entre ces villes

réseau informatique

machines / connections entre ces machines

réseau social

personnes / contacts

mathématiciens / coauteurs

exemples de graphes

réseau routier

villes / routes entre ces villes

réseau informatique

machines / connections entre ces machines

réseau social

personnes / contacts

mathématiciens / coauteurs

le Web

pages / liens entres les pages

exemples de graphes

réseau routier
: villes / routes entre ces villes

réseau informatique
: machines / connections entre ces machines

réseau social
: personnes / contacts
: mathématiciens / coauteurs

le Web
: pages / liens entres les pages

labyrinthe
: salles / portes

exemples de graphes

réseau routier

villes / routes entre ces villes

réseau informatique

machines / connections entre ces machines

réseau social

personnes / contacts

mathématiciens / coauteurs

le Web

pages / liens entres les pages

labyrinthe

salles / portes

jeu

configurations / coups valides

(un ou plusieurs joueurs)

(graphe possiblement infini)

exemples de graphes

réseau routier
: villes / routes entre ces villes

réseau informatique
: machines / connections entre ces machines

réseau social
: personnes / contacts
: mathématiciens / coauteurs

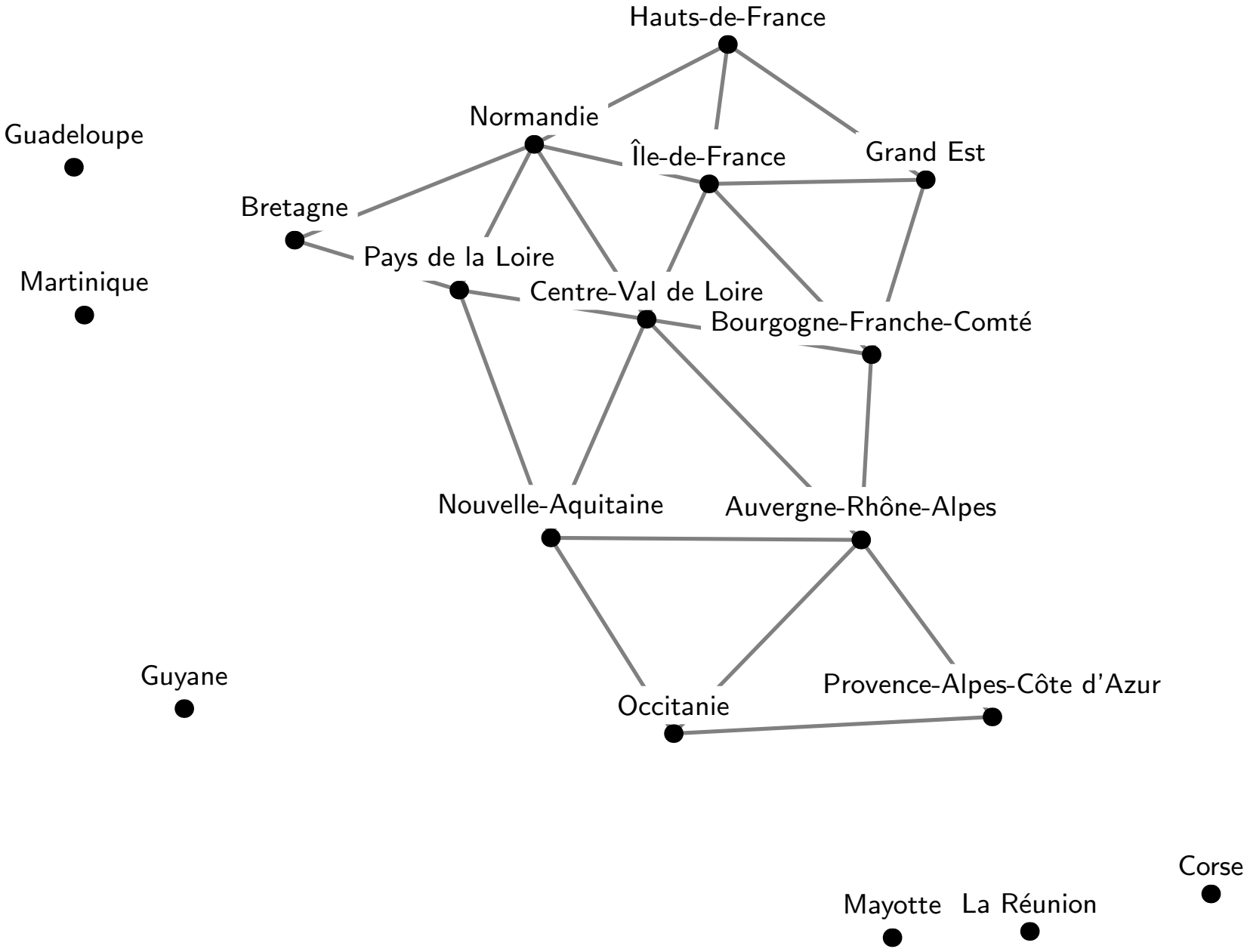
le Web
: pages / liens entres les pages

labyrinthe
: salles / portes

jeu
: configurations / coups valides
: (un ou plusieurs joueurs)
: (graphe possiblement infini)

carte
: régions, pays, etc. / contiguïté

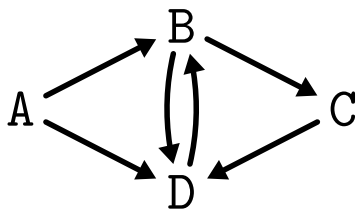
carte des régions françaises



vocabulaire : voisinage

Lorsqu'il y a un arc d'un sommet s vers un sommet t, on dit que t est adjacent à s. Les sommets adjacents à s sont également appelés les voisins de s.

Exemple : dans le graphe

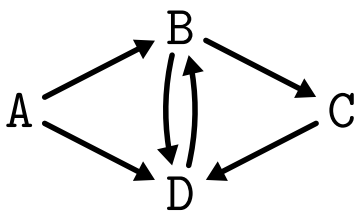


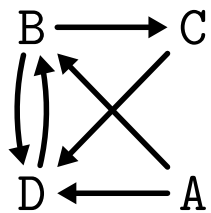
le voisinage de `A` est l'ensemble $\{\texttt{B}, \texttt{D}\}$ et le voisinage de `D` est le singleton $\{\texttt{B}\}$.

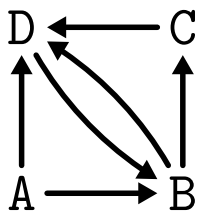
remarque

C'est uniquement l'ensemble des sommets et l'ensemble des arcs qui définissent le graphe, et non pas la façon dont on le dessine.

Ainsi, les trois graphes suivants





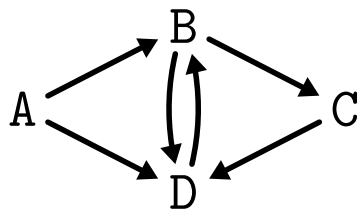


sont identiques, bien que dessinés différemment.

définition : chemin

Un chemin reliant un sommet u à un sommet v est une séquence finie de sommets reliés deux à deux par des arcs et menant de u à v.

Ainsi, dans le graphe



il existe un chemin reliant `A` à `C`, à savoir $\texttt{A} \to \texttt{B} \to \texttt{C}$,

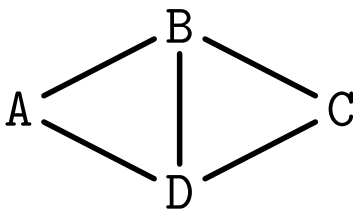
mais aussi $\texttt{A} \to \texttt{D} \to \texttt{B} \to \texttt{C}$

ou encore $\texttt{A} \to \texttt{B} \to \texttt{C} \to \texttt{D} \to \texttt{B} \to \texttt{C}$,

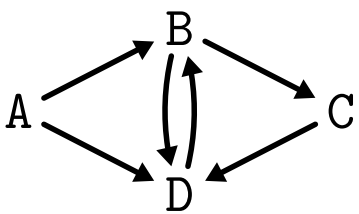
etc.

remarque

Pour un graphe non orienté, on a un chemin reliant u à v si et seulement si on a un chemin reliant v à u.



Mais pour un graphe orienté, on peut avoir un chemin reliant u à v mais pas de chemin reliant v à u.



définition : cycle

Un chemin est dit simple s'il n'emprunte pas deux fois le même arc, et élémentaire s'il ne passe pas deux fois par le même sommet.

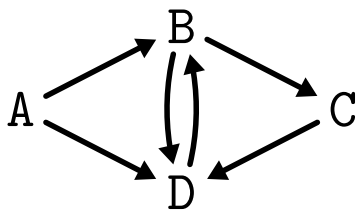
Un chemin simple reliant un sommet à lui-même et contenant au moins un arc est appelé un cycle.

définition : distance

La longueur d'un chemin est définie comme le nombre d'arcs qui constituent ce chemin.

La distance entre deux sommets est la longueur du plus court chemin reliant ces deux sommets, le cas échéant.

Exemple :



Le chemin $\mathtt{A} \to \mathtt{D} \to \mathtt{B} \to \mathtt{C}$ a pour longueur 3, mais la distance entre $\mathtt{A}$ et $\mathtt{C}$ est 2, obtenue pour le chemin plus court $\mathtt{A} \to \mathtt{B} \to \mathtt{C}$.

La distance d'un sommet s à lui-même est 0, obtenue pour le chemin allant de s à s en n'empruntant aucun arc.

représentation en Python

Il existe de très nombreuses solutions.

Quelle que soit la solution, on veut pouvoir

- construire des graphes ;
- les parcourir, c'est-à-dire a minima
  - parcourir l'ensemble des sommets du graphe,
  - parcourir l'ensemble des voisins d'un sommet donné.

solution 1 : la matrice d'adjacence

- Les sommets sont les entiers $0, 1, \ldots, N-1$;
- Le voisinage est représenté par une matrice de booléens `adj`, de taille $N \times N$. Le booléen `adj[i][j]` indique la présence d'un arc entre les sommets `i` et `j`.

Construire un graphe sans arcs :

```
adj = [[False] * N for _ in range(N)]
```

Ajouter un arc entre le sommet 2 et le sommet 7 :

```
adj[2][7] = True
```

solution 1 : parcours

Parcourir tous les sommets du graphe :

```
for s in range(N):
    ...
```

Parcourir tous les voisins du sommet `s` :

```
for v in range(N):
    if adj[s][v]:
        ...
```

encapsulation dans une classe

```
class Graphe:
    def __init__(self, n):
        self.n = n
        self.adj = [[False] * n for _ in range(n)]

    def ajouter_arc(self, s1, s2):
        self.adj[s1][s2] = True

    def arc(self, s1, s2):
        return self.adj[s1][s2]

    def voisins(self, s):
        v = []
        for i in range(self.n):
            if self.adj[s][i]:
                v.append(i)
        return v
```

efficacité

L'efficacité de la matrice d'adjacence est mitigée :

+ c'est compact en mémoire ;

+ ajouter un arc / tester la présence d'un arc est immédiat ;

- parcourir les voisins de s coûte N opérations (même s'il n'y a que 2 voisins, par exemple).

Par ailleurs, il peut être contraignant

- que les sommets soient les entiers $0, 1, \ldots, N$;
- que N soit déterminé à l'avance.

solution 2 : avec un dictionnaire

Un graphe est un dictionnaire, qui associe à chaque sommet l'ensemble de ses voisins.

- les sommets ne sont plus limités à des entiers ;
- l'ensemble des sommets peut évoluer avec le temps (c'est l'ensemble des clés du dictionnaire).

exemple

Construire un graphe sans sommets ni arcs :

```
g = {}
```

Ajouter un sommet "A" et un sommet "B" :

```
g["A"] = set()
g["B"] = set()
```

Ajouter un arc entre le sommet "A" et le sommet "B" :

```
g["A"].add("B")
```

Tester la présence d'un arc entre "A" et "B" :

```
"B" in g["A"]
```

efficacité

Les dictionnaires et les ensembles de Python sont réalisés par des tables de hachage, où l'on peut considérer que les opérations d'ajout et de recherche se dont en temps constant.

Cette solution est donc très efficace.

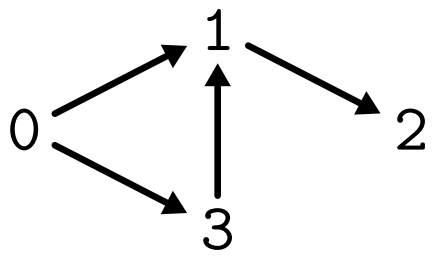
Le seul intérêt de la matrice d'adjacence peut être une occupation mémoire moindre.

encapsulation dans une classe

```
class Graphe:
    def __init__(self):
        self.adj = {}
    def ajouter_sommet(self, s):
        if s not in self.adj:
            self.adj[s] = set()
    def ajouter_arc(self, s1, s2):
        self.ajouter_sommet(s1)
        self.ajouter_sommet(s2)
        self.adj[s1].add(s2)
    def arc(self, s1, s2):
        return s2 in self.adj[s1]
    def sommets(self):
        return list(self.adj)
    def voisins(self, s):
        return self.adj[s]
```

utilisation

```
g = Graphe()
g.ajouter_arc(0, 1)
g.ajouter_arc(0, 3)
g.ajouter_arc(1, 2)
g.ajouter_arc(3, 1)
```



représenter un graphe non orienté

La solution la plus simple : on le représente exactement comme un graphe orienté, avec l'invariant suivant :

il y a un arc reliant le sommet u au sommet v
si et seulement si
il y a un arc reliant le sommet v au sommet u.

Pour maintenir cet invariant, il suffit que l'opération `ajouter_arc` ajoute systématiquement la paire d'arcs.

Exercices

Parcours

en profondeur et en largeur

Parcours en profondeur

labyrinthe

Pour sortir d'un labyrinthe, il suffit de toujours tourner à droite.

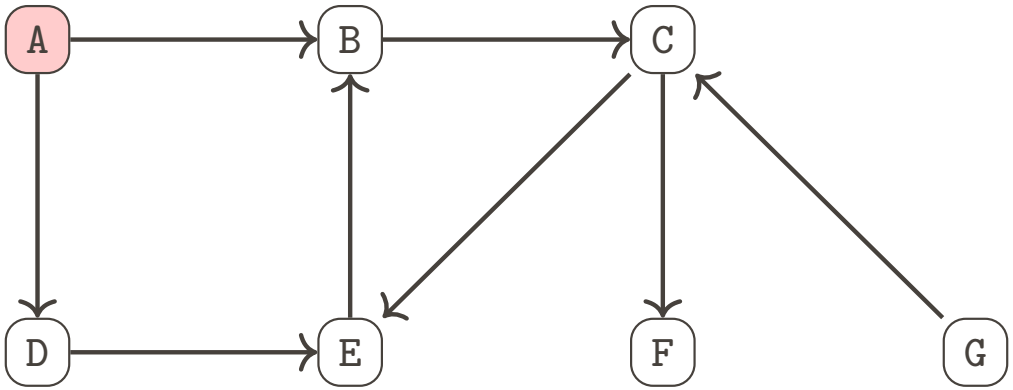
labyrinthe

Pour sortir d'un labyrinthe, il suffit de toujours tourner à droite.

- On peut se retrouver à tourner en rond $\Rightarrow$ il faut marquer les endroits par lesquels on est déjà passé.
- Tourner à droite, ou à gauche, ou faire encore un choix arbitraire dans chaque salle n'a pas d'importance.

exemple

Partons du sommet A sur le graphe suivant :

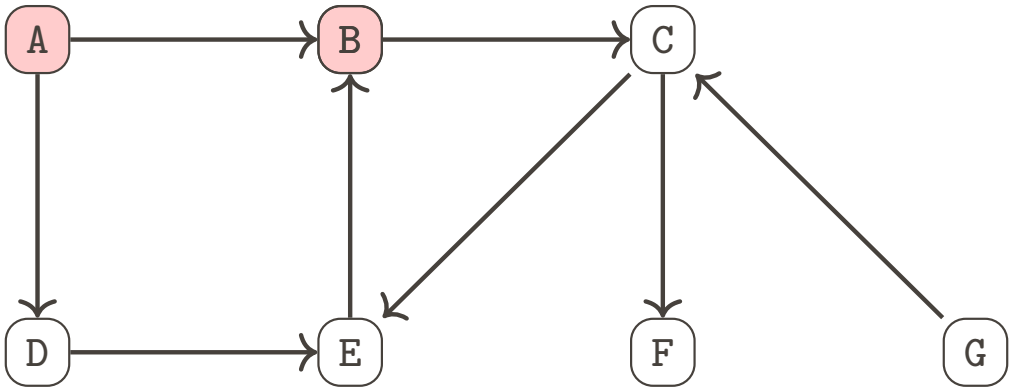


on visite : A

on a vu : A

exemple

Partons du sommet A sur le graphe suivant :

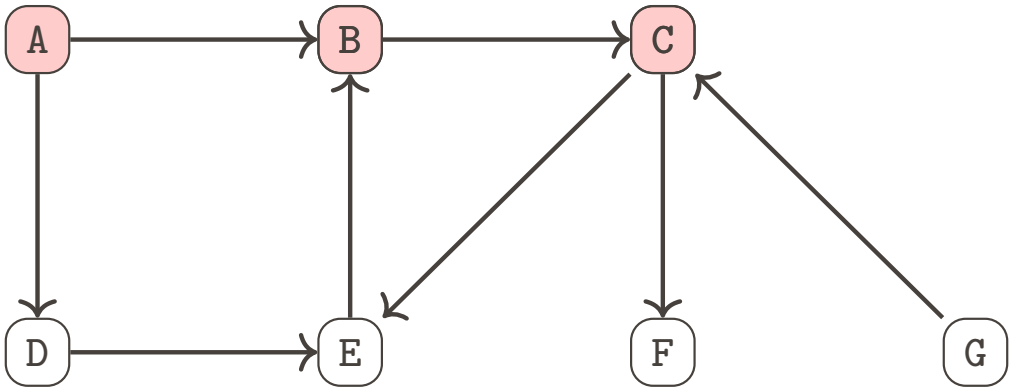


on visite : B

on a vu : A B

exemple

Partons du sommet A sur le graphe suivant :

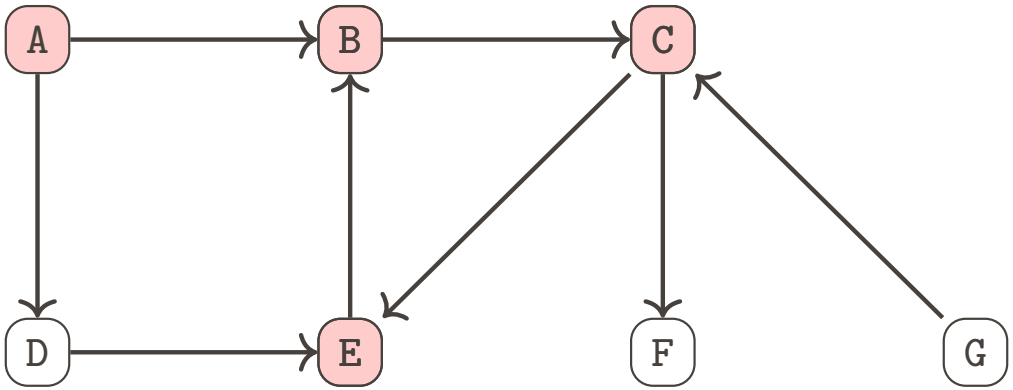


on visite : C

on a vu : A B C

exemple

Partons du sommet A sur le graphe suivant :

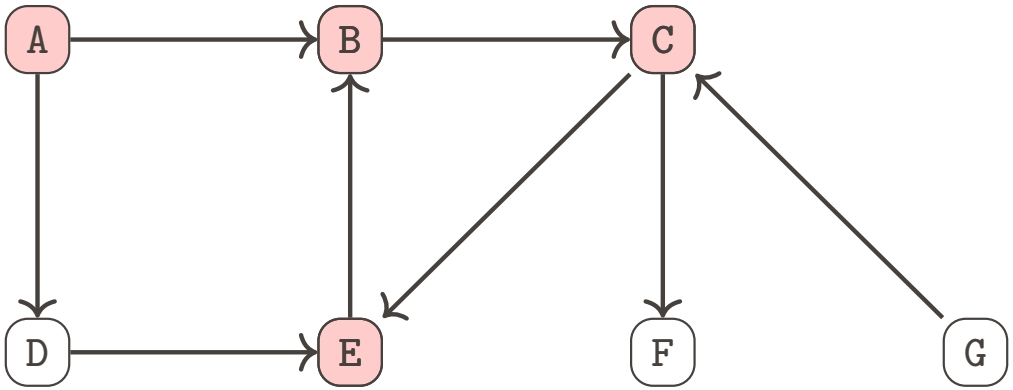


on visite : E

on a vu : A B C E

exemple

Partons du sommet A sur le graphe suivant :

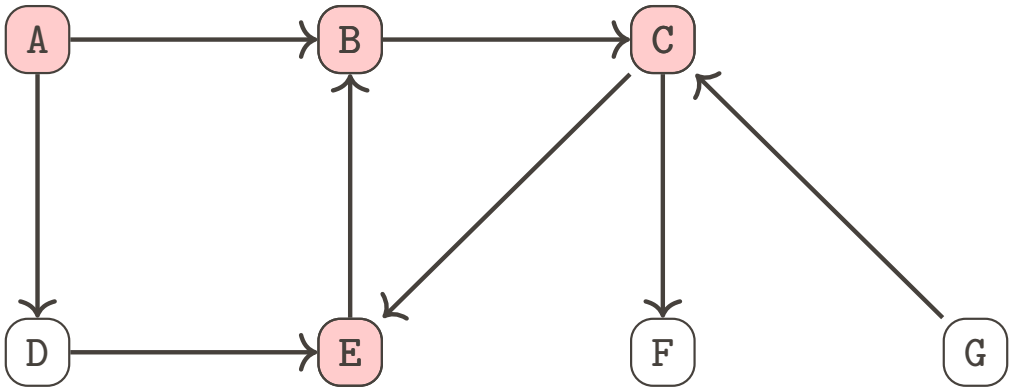


on visite : B, déjà vu, on s'arrête

on a vu : A B C E

exemple

Partons du sommet A sur le graphe suivant :

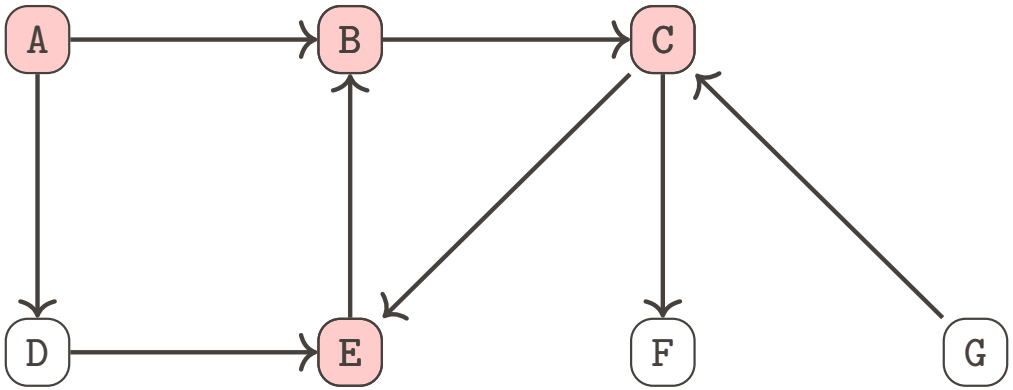


on visite : E, terminé

on a vu : A B C E

exemple

Partons du sommet A sur le graphe suivant :

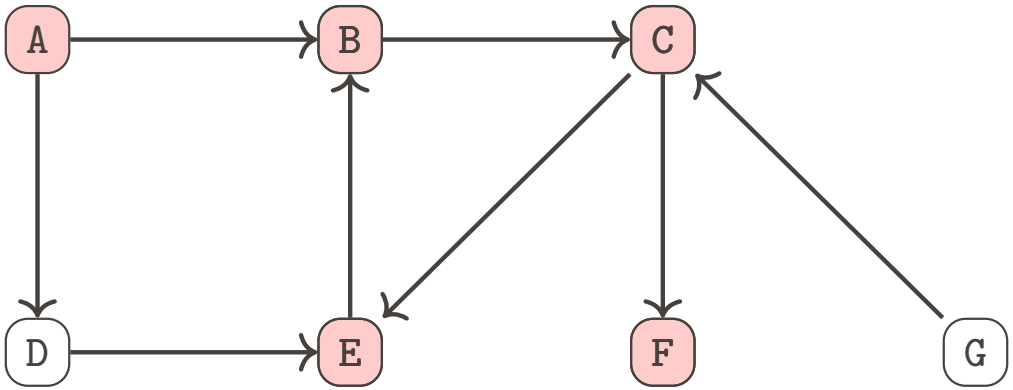


on visite : C

on a vu : A B C E

exemple

Partons du sommet A sur le graphe suivant :

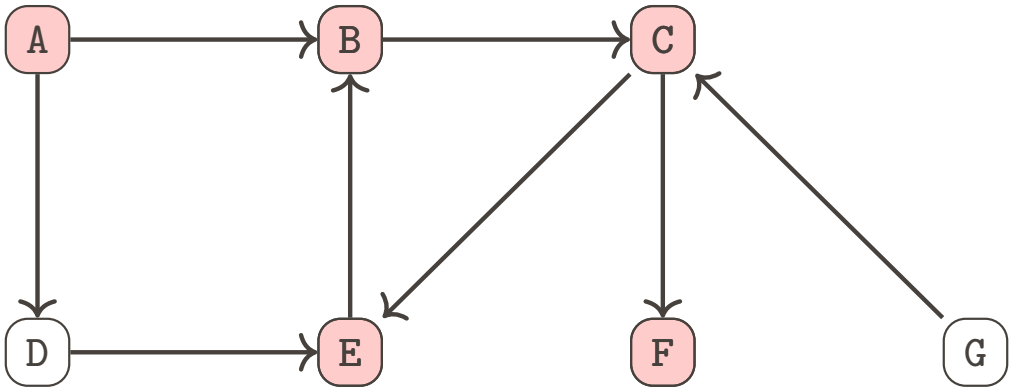


on visite : F, terminé

on a vu : A B C E F

exemple

Partons du sommet A sur le graphe suivant :

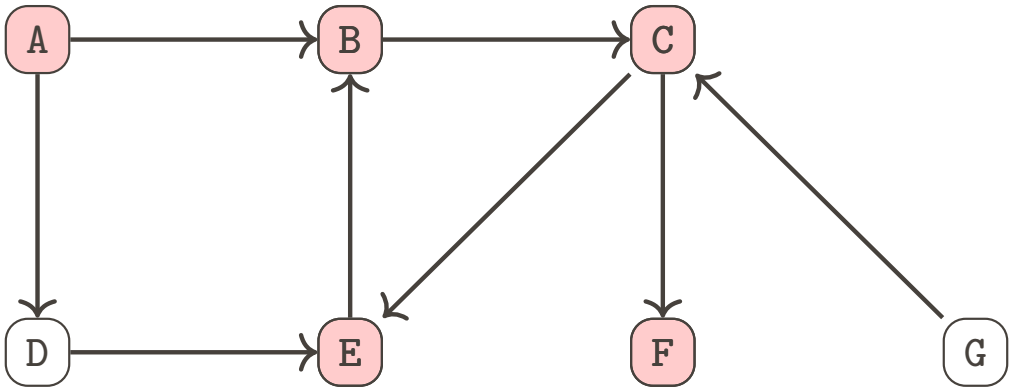


on visite : C, terminé

on a vu : A B C E F

exemple

Partons du sommet A sur le graphe suivant :

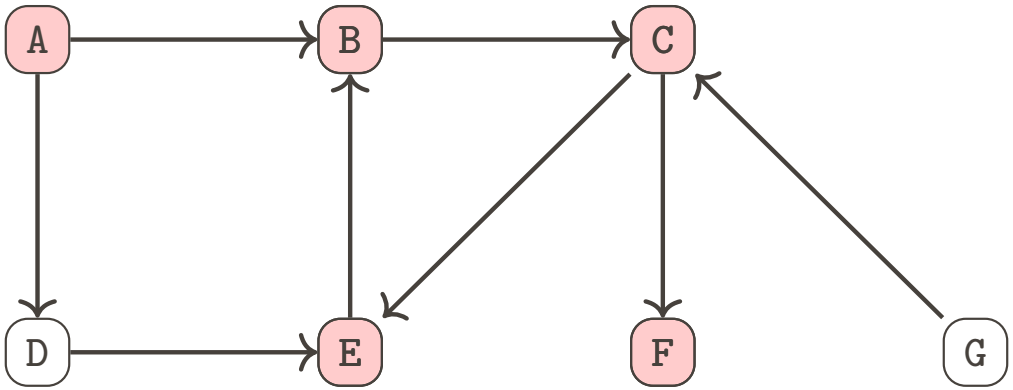


on visite : B, terminé

on a vu : A B C E F

exemple

Partons du sommet A sur le graphe suivant :

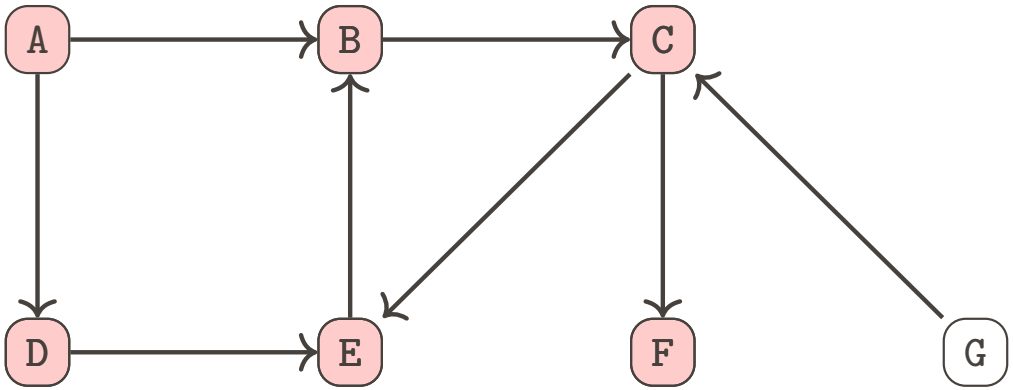


on visite : A

on a vu : A B C E F

exemple

Partons du sommet A sur le graphe suivant :

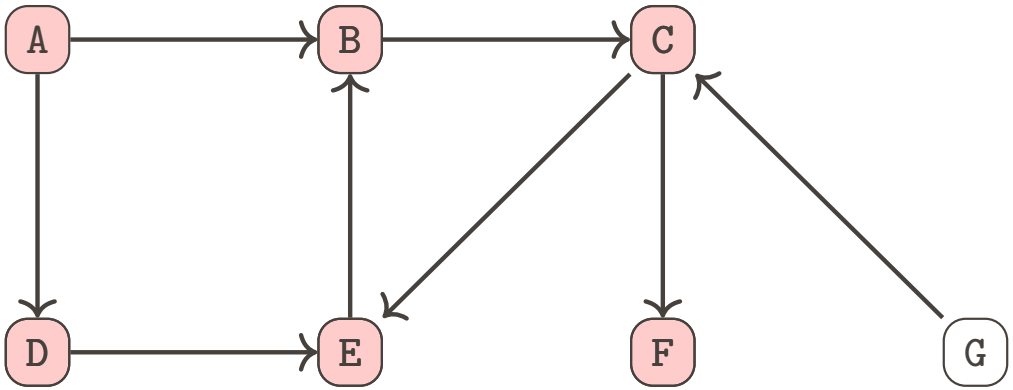


on visite : D

on a vu : A B C E F D

exemple

Partons du sommet A sur le graphe suivant :

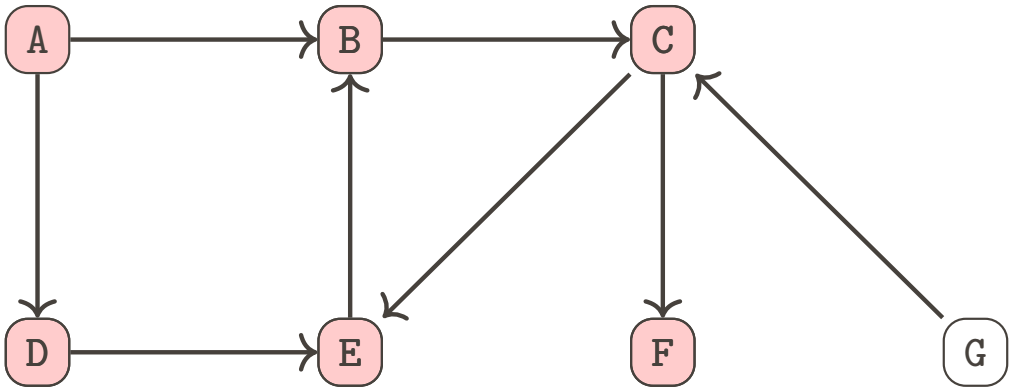


on visite : E, déjà vu, on s'arrête

on a vu : A B C E F D

exemple

Partons du sommet A sur le graphe suivant :

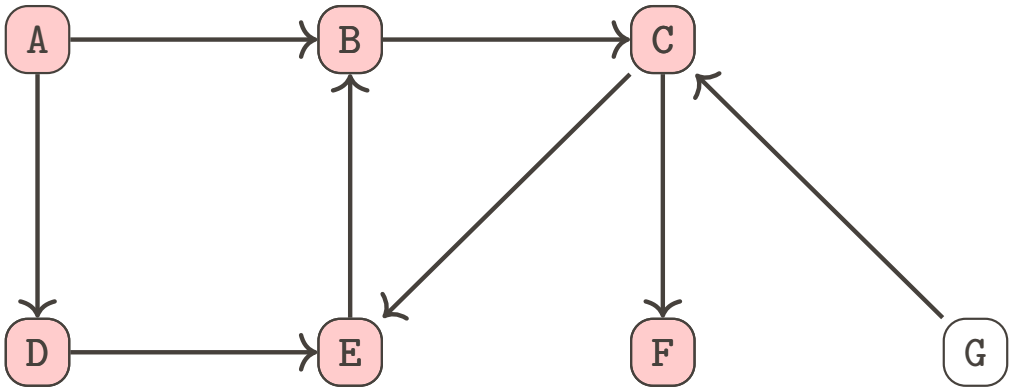


on visite : D, terminé

on a vu : A B C E F D

exemple

Partons du sommet A sur le graphe suivant :



on visite : A, terminé

on a vu : A B C E F D

propriété

Une fois le parcours terminé, tous les sommets atteignables à partir du sommet de départ ont été visités.

(`A`, `B`, `C`, `D`, `E` et `F` sur notre exemple)

Inversement, tout sommet qui n'est pas atteignable n'est pas visité.

(`G` sur notre exemple)

C'est une propriété fondamentale du parcours en profondeur.

programmation

Ingrédients pour le parcours en profondeur :

- Une fonction récursive pour l'écrire.
- Un graphe `g`, avec une méthode `voisins`.
- Un ensemble `vus` pour stocker les sommets déjà visités.

parcours en profondeur

```
def parcours(g, vus, s):
    """parcours en profondeur depuis le sommet s"""
    if s not in vus:
        vus.add(s)
        for v in g.voisins(s):
            parcours(g, vus, v)
```

et c'est tout !

application

Existe-t-il un chemin entre u et v ?

```
def existe_chemin(g, u, v):
    """existe-t-il un chemin de u à v ?"""
    vus = set()
    parcours(g, vus, u)
    return v in vus
```

Exemple :

```
assert existe_chemin(regions, "Bretagne", "Occitanie")
assert not existe_chemin(regions,"Bretagne","Mayotte")
```

efficacité

Le parcours en profondeur est un algorithme très efficace, dont le coût est directement proportionnel au nombre d'arcs qui sont examinés pendant ce parcours.

Dans le pire des cas, tous les sommets sont atteignables et tout le graphe est donc parcouru.

Le coût en mémoire est celui de l'ensemble `vus`, qui contient au final tous les sommets atteignables.

remarques

- Si on veut construire le chemin, il faut se fatiguer un peu plus. Au lieu d'un ensemble `vus`, on prend un dictionnaire qui associe à chaque sommet le sommet qui a permis de l'atteindre (la première fois) pendant le parcours en profondeur.

- On peut provoquer `RecursionError` sur un gros graphe.
  - soit on se sert de `sys.setrecursionlimit` ;
  - soit on utilise une pile.

parcours en profondeur avec une pile

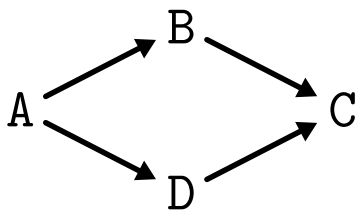
La pile contient les sommets à visiter.

```
def parcours(g, vus, s):
    """parcours en profondeur depuis le sommet s"""
    pile = Pile()
    pile.empiler(s)
    while not pile.est_vide():
        s = pile.depiler()
        if s in vus:
            continue
        vus.add(s)
        for v in g.voisins(s):
            pile.empiler(v)
```

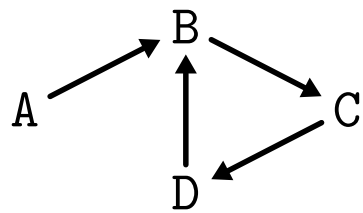
détecter la présence d'un cycle

Idée : s'il y a un cycle dans le graphe, on va retomber sur un sommet déjà rencontré (et donc dans `vus`).

Il y a une subtilité néanmoins :



ici, on retombe sur C,
mais uniquement à cause
d'un chemin parallèle



ici, on retombe sur C,
mais cette fois bien
à cause d'un cycle

détecter la présence d'un cycle

Pour y remédier, on va distinguer dans l'ensemble `vus` deux sortes de sommets :

- ceux pour lesquels le parcours n'est pas encore terminé ;
- et ceux pour lesquels il est au contraire terminé.

On se sert de trois couleurs pour cela :

- `BLANC` : pas encore visité
- `GRIS` : en cours de visite
- `NOIR` : visite terminée

```
BLANC, GRIS, NOIR = 1, 2, 3
```

détection de cycle 1/2

La fonction de parcours en profondeur renvoie `True` dès qu'un cycle est détecté.

```
def parcours_cy(g, couleur, s):
    """parcours en profondeur depuis le sommet s"""
    if couleur[s] == GRIS:
        return True
    if couleur[s] == NOIR:
        return False
    couleur[s] = GRIS
    for v in g.voisins(s):
        if parcours_cy(g, couleur, v):
            return True
    couleur[s] = NOIR
    return False
```

détection de cycle 2/2

```
def cycle(g):
    couleur = {}
    for s in g.sommets():
        couleur[s] = BLANC
    for s in g.sommets():
        if parcours_cy(g, couleur, s):
            return True
    return False
```

Parcours en largeur

point de départ

Le parcours en profondeur nous permet de déterminer l'existence d'un chemin entre deux sommets, mais il ne détermine pas pour autant la distance entre deux sommets, c'est-à-dire la longueur minimale d'un chemin entre ces deux sommets.

exemple

Sur le graphe des régions, le parcours en profondeur va trouver le chemin

$$\text{Bretagne} \to \text{Normandie} \to \text{Hauts-de-France} \to \text{Île-de-France} \to \text{Bourgogne-Franche-Comté} \to \text{Grand Est}$$

de longueur 5, alors qu'il existe un chemin de longueur 3, à savoir

$$\text{Bretagne} \to \text{Normandie} \to \text{Île-de-France} \to \text{Grand Est}$$

parcours en largeur

Pour déterminer la distance entre deux sommets, c'est-à-dire un plus court chemin entre ces deux sommets, on va utiliser un autre algorithme, à savoir le parcours en largeur.

Comme pour le parcours en profondeur,

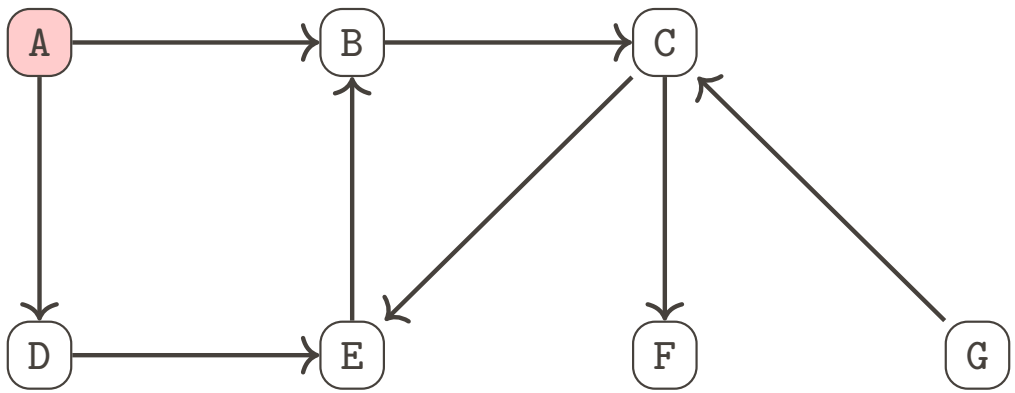
- on se donne un sommet de départ,
- on détermine tous les sommets atteignables.

Contrairement au parcours en profondeur,

- on détermine d'abord les sommets à distance 1,
- puis les sommets à distance 2,
- etc.

exemple

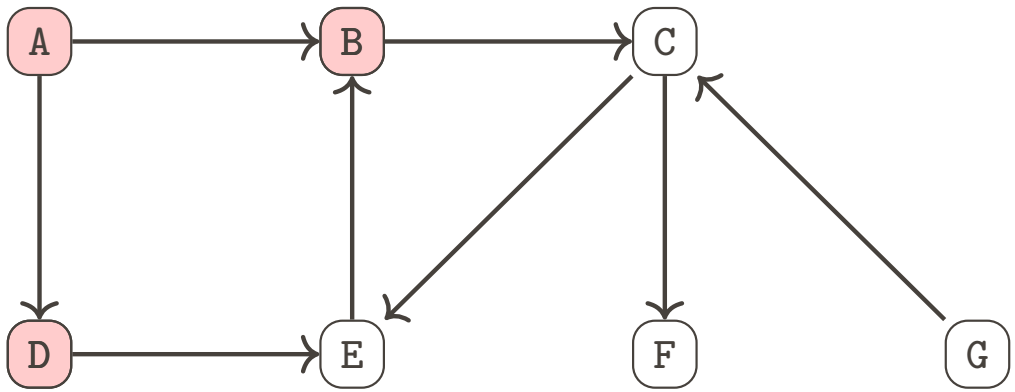
Partons du sommet A sur le graphe suivant :



on visite : A (distance 0)

exemple

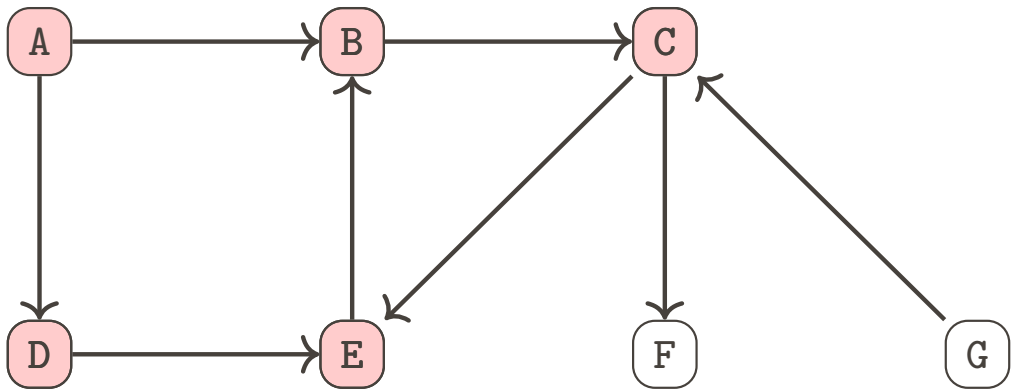
Partons du sommet A sur le graphe suivant :



on visite : A (distance 0) puis B et D (distance 1)

exemple

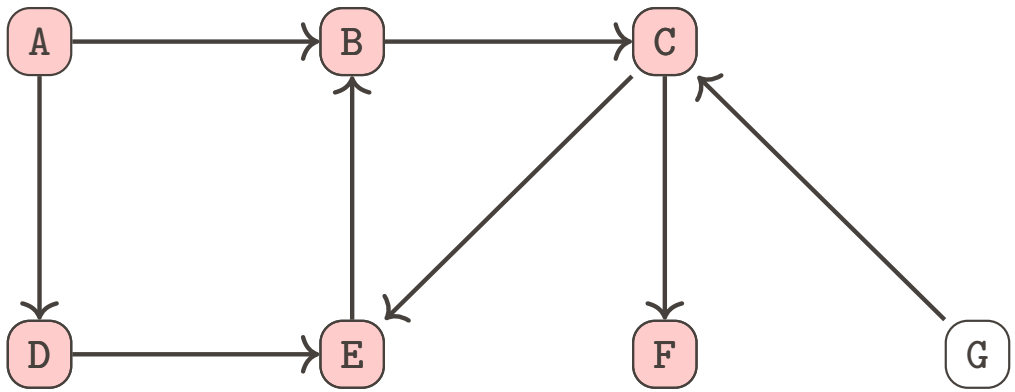
Partons du sommet A sur le graphe suivant :



on visite : A (distance 0) puis B et D (distance 1) puis C et E (distance 2) puis F (distance 3)

exemple

Partons du sommet A sur le graphe suivant :



on visite : A (distance 0) puis B et D (distance 1) puis C et E (distance 2) puis F (distance 3)

mise en œuvre

On se sert de deux ensembles :

- un ensemble `courant` de sommets situés à distance d de la source ;
- un ensemble `suivant` de sommets situés à distance $d+1$ de la source.

et d'un dictionnaire `dist` donnant, pour chaque sommet déjà visité, sa distance à la source.

algorithme

1. initialement, la source est placée dans l'ensemble `courant` et sa distance est fixée à 0 ;
2. tant que l'ensemble `courant` n'est pas vide,
   - 2.1 on en retire un sommet `s`,
   - 2.2 pour chaque voisin `v` de `s` qui n'est pas encore dans `dist`
     - on ajoute `v` à l'ensemble `suivant`,
     - on fixe `dist[v]` à `dist[s] + 1`
   - 2.3 si l'ensemble `courant` est vide, on l'échange avec l'ensemble `suivant`.

(`dist` joue en particulier le rôle de `vus` dans le parcours en profondeur)

code Python

```
def parcours_largeur(g, source):
    """parcours en largeur depuis le sommet source"""
    dist = {source: 0}
    courant = {source}
    suivant = set()
    while len(courant) > 0:
        s = courant.pop()
        for v in g.voisins(s):
            if v not in dist:
                suivant.add(v)
                dist[v] = dist[s] + 1
        if len(courant) == 0:
            courant, suivant = suivant, set()
    return dist
```

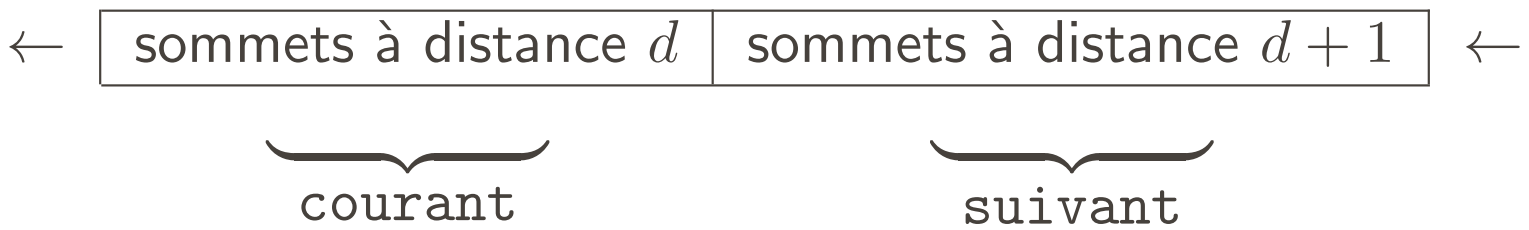
application : distance entre deux sommets

```
def distance(g, u, v):
    """distance de u à v (et None si pas de chemin)"""
    dist = parcours_largeur(g, u)
    return dist[v] if v in dist else None
```

(Comme pour le parcours en profondeur, on pourrait se fatiguer pour exhiber un chemin de longueur minimale.)

remarque

On peut aussi se servir d'une file, qui a la forme suivante :



efficacité

Comme pour le parcours en profondeur, le parcours en largeur a un coût directement proportionnel à la taille de la partie du graphe qui a été explorée.

Exercices