Algorithmique des Graphes

I Définition

Les *graphes* sont des modèles théoriques utiles pour représenter des relations entre les objets d'un espace. Les objets sont les sommets (ou nœuds) du graphe et les relations seront les arcs ou arêtes.

Nos graphes seront toujours finis, *i.e.* avec un nombre fini de sommets, et donc d'arc.

1.1 Graphe Orienté

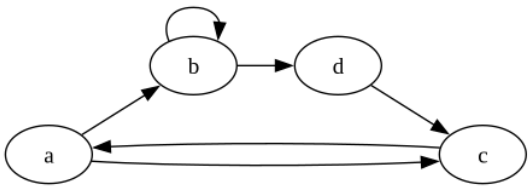
❦ Définition. *Graphe orientés*

Un *graphe orienté* est un couple (S, A) tel que S est un ensemble fini et A est un sous-ensemble de S^2, l'ensemble des couples de deux éléments de S.

Pour représenter un graphe orienté, on place les éléments de S dans le plan et on relie par une flèche toute paire de sommets qui constitue un arc. La flèche par du premier élément et se termine au deuxième.

⚛ Exemples.

Soit $G = (\{a, b, c, d\}, \{(a, b), (a, c), (c, a), (b, d), (d, c), (b, b)\})$ un graphe orienté. On peut le représenter par :



G

❧ ❧ ❧

On remarquera que le nombre d'arcs d'un graphe orienté est compris entre 0 et le carré du nombre de sommets. Il est aussi possible de relier, par un arc, un sommet à lui-même.

On appellera *boucle* un arc d'un sommet vers lui-même.

❦ Définition. *Successeurs et Prédécesseurs*

Soit $G = (S, A)$ un graphe orienté et soit $u \in S$. $v \in S$ est un :

- *successeur* de u si et seulement si $(u, v) \in A$
- *prédécesseur* de u si et seulement si $(v, u) \in A$

❦ Définition. *Degrés*

Soit G = (S, A) un graphe orienté et soit $s \in S$.

— Le degré entrant de s dans G, noté $d_G^-(s)$, est le nombre d'arcs *arrivant* en s, *i.e* le nombre de prédécesseurs de s, $d_G^-(s) = |\{(x, s) \in A, x \in S\}|$

— Le degré sortant de s dans G, noté $d_G^+(s)$, est le nombre d'arcs *partant* de s, *i.e* le nombre de successeurs de s, $d_G^+(s) = |\{(s, x) \in A, x \in S\}|$

On parle parfois, abusivement, du *degré* de s comme étant la somme du degré entrant et sortant de s dans G.

Dans un contexte où le graphe dans lequel on travaille est clair, on omet de préciser G dans la notation.

❦ Proposition. *Somme des degrés*

Soit G = (S, A) un graphe orienté.

$$\sum_{s \in S} d^+(s) = \sum_{s \in S} d^-(s) = |A|$$

1.2 Graphe Non Orienté

❦ Définition. *Graphe non orientés*

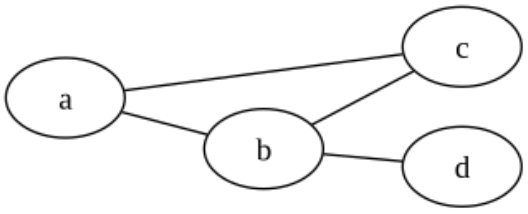
Un *graphe non orienté* est un graphe orienté (S, A) ne contenant aucune boucle et uniquement des *paires non orientées*, *i.e* vérifiant $\forall (u, v) \in A, u \neq v$ et $(v, u) \in A$.

On appelle *arête* et on note $\{u, v\}$ une telle paire non orientée. Le nombre d'arêtes d'un graphe est donc $\frac{|A|}{2}$.

Pour représenter un graphe non orienté, on place les éléments de S dans le plan et on relie par un trait toute paire de sommets qui constitue un arc. La flèche par du premier élément et se termine au deuxième.

⚛ Exemples.

Soit $G = (\{a, b, c, d\}, \{\{a, b\}, \{a, c\}, \{b, d\}, \{b, c\}\})$ un graphe non orienté. On peut le représenter par :



G

🐿 🐿 🐿

On remarquera que le nombre d'arêtes d'un graphe non orienté est compris entre 0 et $\binom{n}{2}$.

❦ Définition. *Voisin*

Soit G = (S, A) un graphe orienté et soit $u \in S$. $v \in S$ est un voisin de u si et seulement si $\{u, v\} \in A$, *i.e.* v est un successeur et prédécesseur du u.

❦ Définition. *Degré*

Soit G = (S, A) un graphe orienté et soit $s \in S$. Le degré de s dans G, noté $d_G(s)$, est le nombre d'arêtes attachées à s, *i.e* le nombre de voisins, $d_G(s) = d_G^+(s) = d_G^-(s)$.

Dans un contexte où le graphe dans lequel on travaille est clair, on omet de préciser G dans la notation.

❦ Proposition. *Somme des degrés*

Soit G = (S, A) un graphe orienté à m arêtes.

$$\sum_{s \in S} d(s) = 2m = |A|$$

1.3 Chemins

❦ Définition. *Chemin (finis)*

Soit G = (S, A) un graphe orienté. Un chemin (fini) de G est une séquence finie de sommets de s, $s_0\, s_1\, \cdots\, s_n$, telle que pour tout $0 < i \leq n$, s_i est un successeur de s_{i+1} dans G.

s_0 s'appelle alors la *source* du chemin, s_n la *destination* et n est la *longueur* du chemin.

En particulier dans les graphes non orientés, le sommet suivant dans la séquence doit être un *voisin*.

❦ Définition. *Chemin simple*

Soit G = (S, A) un graphe orienté. Un chemin de G est *simple* si et seulement si tous les sommets de la séquence sont différents deux à deux.

❦ Définition. *Cycles et cycles simple*

Soit G = (S, A) un graphe orienté. Un chemin de G est un *cycle* si et seulement s'il est de longueur

non nulle et sa source est identique à sa destination.

Le cycle est dit *simple* si et seulement si la source et la destination sont les deux seuls sommets identiques de la séquence.

❦ Définition. *Cycles et cycles simple*

Soit G = (S, A) un graphe orienté. Un chemin de G est un *cycle* si et seulement s'il est de longueur non nulle et sa source est identique à sa destination.

Le cycle est dit *simple* si et seulement si la source et la destination sont les deux seuls sommets identiques de la séquence.

❦ Définition. *Accessibilité*

Soit G = (S, A) un graphe orienté. Soit u et v deux sommets de G. On dit que v est accessible depuis u si et seulement s'il existe un chemin de G dont la source est u et la destination est v.

❦ Définition. *Connexité dans un graphe non orienté*

Soit G = (S, A) un graphe *non* orienté. G est connexe si et seulement si pour toute paire de sommet (u, v), v est accessible depuis u.

❦ Proposition.

Soit G = (S, A) un graphe *non* orienté. G est connexe si et seulement si il existe un sommet u de G tel que tout sommet v de G est accessible depuis u.

⚠ Bon à savoir (HP).

Il existe deux notions de connexité pour les graphes non orientés :

— La connexité *faible* demande à ce que le graphe non orienté obtenu en complétant chaque arc du graphe en une arête, soit connexe.
— La connexité *forte* reprend la définition par l'accessibilité. Pour toute paire de sommet (u, v), v doit être accessible depuis u.

❦ ❦ ❦

❦ Définition. *Distance*

Soit G = (S, A) un graphe orienté. Soit u et v deux sommets de G. La *distance* de u à v, notée

$d_G(u, v)$, est la plus petite longueur parmi les chemins de source u et de destination v, *i.e.*

$$d_G(u, v) = \min\{n \in \mathbb{N}, \exists c, \text{ un chemin de longueur } n \text{ de source } u \text{ et de destination } v\}$$

Si v n'est pas accessible depuis u, la distance est par convention *infinie*, ce qui est cohérent avec la convention $\min \varnothing = +\infty$.

1.4 Sous graphes

❦ Définition. *Graphe orientés*

Soit G = (S, A) un graphe, orienté ou non, un *sous graphe* de G est un graphe G′ = (S′, A′) tel que S′ ⊆ S et A′ ⊆ S.

❦ Définition. *Composante Connexe*

Soit G = (S, A) un graphe, orienté ou non, une *composante connexe* de G est un sous graphe G′ = (S′, A′) tel que G′ est connexe et pour tout sommet s' de S′, S′ contient tous les sommets accessibles depuis s' dans G.

❦ Proposition. *Equivalence par accessibilité*

Soit G = (S, A) un graphe, orienté ou non, tout sommet s de G appartient à une et une seule composante connexe.

1.5 Graphe Pondérés ou Étiquetés

❦ Définition. *Graphe orienté étiquetés*

Un graphe orienté *étiqueté* est un couple (G, λ) tel que G est un graphe orienté (S, A) et λ est une fonction de A dans un ensemble quelconque d'étiquettes.

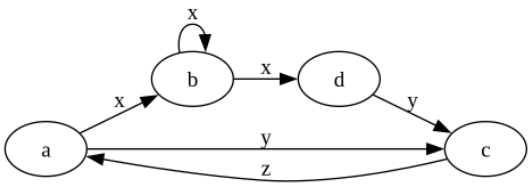
Pour représenter un graphe orienté étiqueté, on représente le graphe orienté et on surmonte chaque arc de son étiquette.

⚛ Exemples.

Soit G = $(\{a, b, c, d\}, \{(a, b), (a, c), (c, a), (b, d), (d, c), (b, b)\})$ un graphe orienté, E = $\{x, y, z\}$ un ensemble de trois étiquettes et λ définie par :

- $\lambda((a, b)) = \lambda((b, b)) = \lambda((b, d)) = x$
- $\lambda((a, c)) = \lambda((d, c)) = y$
- $\lambda((c, a)) = z$

. On peut le représenter par :



G

❧ ❧ ❧

❦ Définition. *Graphe non orienté étiquetés*

Un graphe non orienté *étiqueté* est un couple (G, λ) tel que G est un graphe non orienté (S, A) et λ est une fonction de A dans un ensemble quelconque d'étiquettes vérifiant pour toute arête $\{u, v\}$, $\lambda((u, v)) = \lambda((v, u))$ qu'on note $\lambda(\{u, v\})$.

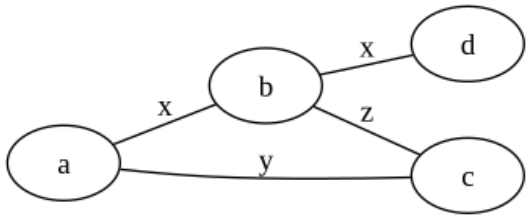
Pour représenter un graphe non orienté étiqueté, on représente le graphe non orienté et on surmonte chaque arête de son étiquette.

⚛ Exemples.

Soit $G = (\{a, b, c, d\}, \{\{a, b\}, \{a, c\}, \{b, d\}, \{b, c\}\})$ un graphe non orienté, $E = \{x, y, z\}$ un ensemble de trois étiquettes et λ définie par :

— $\lambda(\{a, b\}) = \lambda(\{b, d\}) = x$
— $\lambda(\{a, c\}) = y$
— $\lambda(\{b, c\}) = z$.

On peut le représenté par :



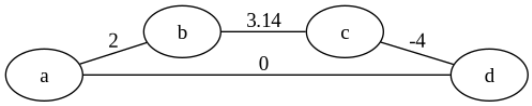
G

❧ ❧ ❧

❦ Définition. *Graphe pondéré*

Un graphe, orienté ou non, *pondéré* est graphe étiqueté (G, ρ) avec ρ à valeurs dans $\mathbb{R}$.

Les étiquettes sont alors appelées les *poids* de l'arc ou de l'arête en question.



G

❀ Exemples.

❦ ❦ ❦

II Représentation

Les représentations sont données pour des graphes orientés. On en déduit à chaque fois la représentation d'un graphe non orienté comme un cas particulier de graphe orienté.

2.1 Matrice d'adjacence

❦ Définition. *Matrice d'adjacence*

Soit un graphe orienté $G = (S, A)$ avec $n = |S|$ et $S = \{s_0, s_1, \cdots, s_{n-1}\}$. La *matrice d'adjacence* de G est la matrice $M = (m_{i,j})_{0 \leq i,j < n}$ de taille $n \times n$ telle que

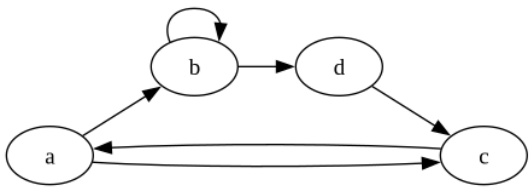
$$m_{i,j} = 1_A((s_i, s_j)) = \begin{cases} 0 & \text{si } (s_i, s_j) \in A \\ 1 & \text{sinon} \end{cases}$$

La matrice d'adjacence caractérise de manière unique un graphe et s'implémente en `Python` comme toutes les matrices par une liste de liste. Il est nécessaire que les sommets soient numérotés. Le plus simple, quitte à renommer étant de prendre $S = [\![0, n-1]\!]$.

On remarquera que la matrice d'adjacence d'un graphe *non* orientée est *symétrique à diagonale nulle*.

❀ Exemples.

Le graphe suivant, en numérotant dans l'ordre $a(0), b(1), c(2)$ et $d(3)$,



G

a pour matrice d'adjacence, écrite en `Python` :

```
[[0,1,1,0],
 [0,1,0,1],
 [1,0,0,0],
```

```
[0,0,1,0]]
```

🐞🐞🐞

🐞 Définition. *Matrice d'adjacence avec etiquettes*

Soit un graphe orienté étiqueté $G = ((S, A), \lambda)$ avec $n = |S|$ et $S = \{s_0, s_1, \cdots, s_{n-1}\}$. On étend λ en $\overline{\lambda}$ sur S^2 avec un symbole par défaut choisi selon la situation, noté ici Ω. La *matrice d'adjacence* de G est la matrice $M = (m_{i,j})_{0 \leq i,j < n}$ de taille $n \times n$ telle que

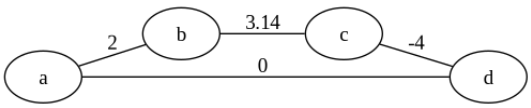
$$m_{i,j} = \overline{\lambda}((s_i, s_j)) = \begin{cases} \lambda((i,j)) & \text{si } (i,j) \in A \\ \Omega & \text{sinon} \end{cases}$$

`None` est généralement un bon choix pour Ω.

Dans le cadre des graphes pondérés, avec les poids considérés comme le *coût* du passage d'un sommet à un autre, on choisit classiquement $\Omega = +\infty$.

⚛ Exemples.

Le graphe suivant, en numérotant dans l'ordre $a(0), b(1), c(2)$ et $d(3)$,



`G`

a pour matrice d'adjacence, écrite en `Python` :

```
I = float("inf")
[[I   ,2    ,I    ,0 ],
 [2   ,I    ,3.14 ,I ],
 [I   ,3.14 ,I    ,-4],
 [0   ,I    ,-4   ,I ]]
```

🐞🐞🐞

2.2 Listes d'adjacence

🐞 Définition. *Listes d'adjacence*

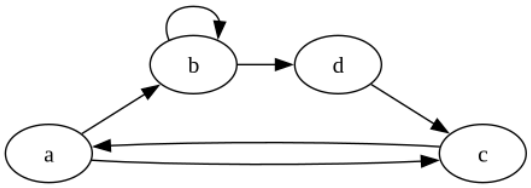
Soit un graphe orienté $G = (S, A)$ avec $n = |S|$ et $S = \{s_0, s_1, \cdots, s_{n-1}\}$. Soit $s \in S$, la *liste d'adjacence* de s dans G est la liste des numéros de ses *successeurs*, *i.e.* la liste des $0 \leq i < n$ tels que $(s, s_i) \in A$.

La représentation par listes d'adjacences de G est la liste `L` de longueur n telle que pour tout $0 \leq i < n$, `L[i]` est la liste d'adjacence de s_i.

Les listes d'adjacence caractérisent de manière unique un graphe et s'implémentent en `Python` par une liste de liste.

⚛ Exemples.

Le graphe suivant, en numérotant dans l'ordre $a(0), b(1), c(2)$ et $d(3)$,



`G`

a comme listes d'adjacence, écrites en `Python` :

```
[[1,2],
 [1,3],
 [0],
 [3]]
```

🐾 🐾 🐾

On peut éviter la phase de numérotation des sommets en utilisant, en Python, un dictionnaire de liste.

⚛ Exemples.

On peut représenter l'exemple précédent avec le dictionnaire suivant.

```
{'a': [1,2],
 'b': [1,3],
 'c': [0],
 'd': [3]}
```

🐾 🐾 🐾

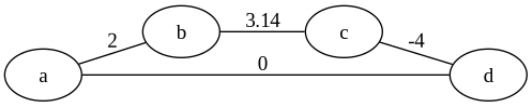
❦ Définition. *Listes d'adjacence avec etiquettes*

Soit un graphe orienté $G = ((S, A), \lambda)$ avec $n = |S|$ et $S = \{s_0, s_1, \cdots, s_{n-1}\}$. Soit $s \in S$, la *liste d'adjacence* de s dans G est la liste des numéros de ses *successeurs* associés à leurs étiquettes, *i.e.* la liste des couples $(i, \lambda((s, s_i))$ tels que $0 \leq i < n$ et $(s, s_i) \in A$.

La représentation par listes d'adjacences de G est la liste `L` de longueur n telle que pour tout $0 \leq i < n$, `L[i]` est la liste d'adjacence de s_i.

⚛ Exemples.

Le graphe suivant, en numérotant dans l'ordre $a(0), b(1), c(2)$ et $d(3)$,



G

a pour matrice d'adjacence, écrite en `Python` :

```
I = float("inf")
[[(1,2), (3, 0)],
 [(0,2), (2, 3.14)],
 [(1,3.14), (3, -4)],
 [(0,0), (2,-4)]]
```

❧ ❧ ❧

III Parcours

Comme pour les séquences, beaucoup d'algorithmes s'appuie sur un *parcours* de la structure de donnée, ici le graphe.

❦ Définition. *Parcours de graphe*

> Un parcours de graphe est un algorithme permettant de visiter *chaque* sommet *et* chaque arc (ou arête) du graphe *exactement une fois*.

On remarque aussi que les représentations des graphes utilisant des listes et les sommets étant numérotés, de simples parcours de liste permettent de parcourir les sommets du graphe dans l'ordre numérique et les arcs qui leur sont liés.

On s'intéresse à des parcours qui visitent les éléments du graphe en *respectant sa structure*, *i.e.* en suivant les relations modélisées par les arcs.

❦ Définition. *Parcours de graphe au départ d'un sommet*

> Un parcours de graphe, G, au départ d'un sommet, s, de G est un algorithme permettant de visiter, exactement une fois, chaque sommet *et* chaque arc (ou arête) *accessibles* depuis s dans G .

Lors d'un parcourt les sommets (et les arcs ou arêtes) peuvent être dans trois états, souvent associés à des couleurs :

1. Inconnus (blancs)
2. Découverts (gris)
3. Visités (noirs)

La découverte est l'étape qui permet à la fois de savoir que le sommet est *accessible* et, dans un graphe non orienté, fait donc bien partie de la composante connexe du sommet de départ, mais aussi de définir son ordre dans le parcours.

❦ Définition. *Marquage*

Soit G = (S, A) un graphe, orienté ou non. Un marquage est une fonction de l'ensemble des sommets dans un ensemble d'états, par exemple, pour les parcours suivants, dans l'ensemble des couleurs *blanc*, *gris*, *noir*.

Pour représenter un marquage, on peut utiliser une liste (si les sommets sont numérotés) ou un dictionnaire (si les sommets sont des objets immuables).

3.1 Parcours en largeur

❦ Définition. *Parcours en largeur*

Un parcours *en largeur* d'un graphe, G, au départ d'un sommet, *s*, de G est défini par l'ordre des opérations suivantes :

1. La visite d'un sommet est directement suivie de la découverte de tous les arcs sortants et de ses successeurs.
2. Un arc est visité dès qu'il est découvert.
3. Les sommets sont visités dans l'ordre de leur découverte.

De manière équivalente on peut définir le parcours en largeur comme un parcours qui permet de visiter chaque sommet par *ordre de distance à s*, au sens de la définition donnée plus haut.

Lors d'un parcours en largeur il est donc important de se souvenir de l'ordre de la découverte des sommets. En effet, la visite doit se faire dans l'ordre *FIFO* (First In First Out), premier découvert, premier visité. Pour se faire, on utilise une *file*.

❦ Définition. *File*

Une *file* est un conteneur permettant les opérations suivantes :

— vérifier si la file est vide
— ajouter un élément dans la file (*enfiler*)
— enlever l'élément le plus vieux (ajouté en premier) présent dans la file (*défiler*)

Algorithme

Voici un algorithme possible pour le parcours en largeur.

Algorithme Parcours en largeur au départ d'un sommet

entrées : G = (S, A) un graphe orienté, $s \in S$

```
1 . μ ← un marquage constant à INCONNU
2 . ϕ ← une file vide
3 . μ(s) = DECOUVERT
4 . Enfiler s dans ϕ
5 . tant que ϕ n'est pas vide faire
6 .    Défiler u de ϕ
7 .    μ(u) = VISITE
8 .    pour v successeur de u faire
9 .       /* Ici on découvre et visite l'arc (v, u) */
10 .       si μ(v) = INCONNU alors
11 .          μ(v) = DECOUVERT
12 .          Enfiler v dans ϕ
13 .       fin si
14 .    fin pour
15 . fin tant que
16 . renvoyer μ
```

Voici les analyses importantes concernant cet algorithme :

1. L'algorithme termine. Le variant de la boucle est le nombre de sommet du marquage non visité.
2. Les invariants les plus importants sont :
   (a) Les sommets découverts et visités sont accessibles depuis le sommet de départ s.
   (b) L'ensemble des successeurs des sommets visités est exactement l'union des ensembles des sommets découverts et visités.
   (c) Les sommets inconnus sont à distance strictement plus grande que tous les sommets visités.
   (d) La file contient exactement les sommets découverts (et non visités), dans l'ordre de distance à s.
3. En supposant que toutes les opérations de file soient en O(1), la complexité de l'algorithme est en $O(|S| + |A|)$. Si toutefois on considère une complexité σ pour la visite et la découverte d'un sommet et α la complexité de la découverte et la visite d'un arc, alors la complexité est en $O(\sigma|S| + \alpha|A|)$.

On notera qu'il s'agit d'un squelette d'algorithme permettant de visiter chaque sommet et arête de la composante connexe du sommet de départ. Un algorithme *utilisant* un parcours devra ajouter les instructions à faire lors de la visite ou de la découverte des sommets et des arcs. La valeur de retour changera aussi en fonction de la spécification de l'algorithme utilisant le parcours.

Pour parcourir *tous* les sommets et les arcs d'un graphe, et pas seulement les sommets accessibles, il faut lancer ce parcours sur chaque composante connexe.

Algorithme Parcours en largeur

entrées : G = (S, A) un graphe orienté
1. $\mu \leftarrow$ un marquage constant à `INCONNU`
2. **tant que** $\exists s \in S, \mu(s) =$ `INCONNU` **faire**
3. Faire un parcours en largeur au départ de s
4. Récupérer μ' le marquage renvoyé par ce parcours.
5. Mettre à jour μ avec les sommets visités dans μ'
6. **fin tant que**

Implémentation

Les sommets du graphe sont représentés par des entiers (correspondant à leur numéro). On choisit de représenter les graphes par listes d'adjacences. Les états des sommets sont représentés par les entiers 0 (blanc), 1 (gris), 2 (noir). Le marquage sera alors une liste d'entiers, tel que l'élément à l'indice `i` du marquage soit l'état du sommet s_i. Les complexités des opérations de marquage sont alors toutes en O(1).

La file peut être implémentée par une liste, `F`. On aurait alors comme implémentation des opérations de file :

— `len(F) > 0` pour le test à vide, en O(1)
— `F.append(s)` pour enfiler `s`, en O(1)
— `s, F = F[0], F[1:]` pour défiler, en O($n-1$) avec n la longueur de la file.

Il existe d'autres implémentations d'une file, plus efficaces. Le module `deque` donne accès à une structure, nommée aussi `deque`, pour laquelle les trois opérations précédentes sont en O(1).

```
from deque import deque

def parcours_largeur(G, s):
    n = len(G)
    mu = [0]*n
    fi = deque()
    mu[s] = 1
    fi.append(s)
    while len(fi) > 0:
        u = fi.popleft()
        # Visiter u ici
        mu[u] = 2
        for v in G[u]:
            # Decouvrir et visiter (u,v) si besoin
            if mu[v] == 0:
                mu[v] = 1
                fi.append(v)
    return mu
```

Dans le cas de la représentation des listes d'adjacences par un dictionnaire, le marquage sera représenté lui aussi par un dictionnaire.

3.2 Parcours en profondeur

❦ Définition. *Parcours en profondeur*

Un parcours *en profondeur* d'un graphe, G, au départ d'un sommet, s, de G est défini par l'ordre des opérations suivantes :

1. La visite d'un sommet est directement suivie d'un parcours en profondeur dans le graphe des sommets non visités, partant de chacun de ses successeurs.
2. Un arc est visité dès qu'il est découvert.
3. Un sommet est visité dès qu'il est découvert.

Le parcours est donc naturellement *récursif* et le marquage doit être partagé par tous les appels récursifs.

Algorithme

Voici un algorithme possible pour le parcours en profondeur.

Algorithme Parcours en profondeur au départ d'un sommet

entrées : G = (S, A) un graphe orienté, $s \in S$, μ un marquage de G

```
1. μ(s) = VISITE
2. pour v successeur de u faire
3.    /* Ici on découvre et visite l'arc (v, u) */
4.    si μ(v) ≠ VISITE alors
5.       μ(v) = DECOUVERT
6.       Appel récursif sur G, v et μ
7.    fin si
8. fin pour
9. renvoyer μ
```

Voici les analyses importantes concernant cet algorithme :

1. L'algorithme termine. Il suffit de prendre comme taille de l'entrée le nombre de sommets *inconnus* dans le marquage.
2. On peut montrer la post-condition : μ marque comme visités exactement les sommets accessibles depuis s. Les autres sommets ne changent pas d'état.
3. La complexité de l'algorithme est en $O(|S| + |A|)$. Si toutefois on considère une complexité σ pour la visite et la découverte d'un sommet et α la complexité de la découverte et la visite d'un arc, alors la complexité est en $O(\sigma|S| + \alpha|A|)$.

On notera qu'il s'agit d'un squelette d'algorithme permettant de visiter chaque sommet et arête de la composante connexe du sommet de départ, étant donné un marquage. Un algorithme *utilisant* un parcours devra ajouter les instructions à faire lors de la visite ou de la découverte des sommets et des arcs. La valeur de retour changera aussi en fonction de la spécification de l'algorithme utilisant le parcours.

Pour parcourir *tous* les sommets et les arcs d'un graphe, et pas seulement ceux d'une composante connexe, il faut lancer ce parcours sur chaque composante connexe.

Algorithme Parcours en profondeur

entrées : G = (S, A) un graphe orienté
1. $\mu \leftarrow$ un marquage constant à `INCONNU`
2. **tant que** $\exists s \in S, \mu(s) =$ `INCONNU` **faire**
3. Faire un parcours en profondeur au départ de s avec le marquage μ
4. **fin tant que**

Implémentation

Les sommets du graphe sont représentés par des entiers (correspondant à leur numéro). On choisit de représenter les graphes par listes d'adjacences. Les états des sommets sont représentés par les entiers 0 (blanc), 1 (gris), 2 (noir). Le marquage sera alors une liste d'entiers, tel que l'élément à l'indice `i` du marquage soit l'état du sommet s_i. Les complexités des opérations de marquage sont alors toutes en O(1).

```
def parcours_profondeur_s(G, s, mu):
    mu[s] = 2
    for v in G[s]:
        mu[v] = 1
        parcours_profondeur_s(G, v, mu)

def parcours_profondeur_G(G):
    n = len(G)
    mu = [0] * n
    for s in range(n):
        if mu[s] == 0:
            parcours_profondeur_s(G, s, mu)
```

Dans le cas de la représentation des listes d'adjacences par un dictionnaire, le marquage sera représenté lui aussi par un dictionnaire.

Variantes itératives

Les variantes itératives de l'algorithme utilisent une *pile* pour simuler (ou se rapprocher) de la pile des appels récursifs.

Une pile permet de faire la visite dans l'ordre *LIFO* (Last In First Out), dernier découvert, premier visité.

❦ Définition. *File*

Une *pile* est un conteneur permettant les opérations suivantes :
- vérifier si la pile est vide
- ajouter un élément au-dessus de la pile (*empiler*)
- enlever l'élément du dessus de la pile (ajouté en dernier) (*dépiler*)

La seule manière de reproduire *exactement* le parcours en profondeur récursif par un programme itératif est de simuler la *pile des appels récursifs* pour se souvenir du prochain arc à visiter. On pré-

sente plutôt une variante, dans laquelle l'ordre de visite des sommets est identique à l'algorithme récursif.

La pile peut être implémentée par une liste, `P`. On aurait alors comme implémentation des opérations de file :

- `len(F) > 0` pour le test à vide, en O(1)
- `F.append(s)` pour enfiler `s`, en O(1)
- `s = F.pop()` pour défiler, en O(1).

La structure `deque` donne aussi accès aux trois opérations précédentes en O(1).

```
from deque import deque

def parcours_profondeur(G, s):
    n = len(G)
    mu = [0]*n
    pi = []
    mu[s] = 1
    pi.append(s)
    while len(pi) > 0:
        u = pi.pop()
        if mu[u] != 2:
            # Visiter u ici
            mu[u] = 2
            for v in G[u]:
                # Decouvrir et visiter (u,v) si besoin
                if mu[v] != 2:
                    mu[v] = 1
                    fi.append(v)
    return mu
```

⚠ Bon à savoir (HP).

C'est donc l'ordre de visite des arcs et de découverte des sommets qui diffère. La visite ne suivant plus directement la découverte, un sommet peut être empilé plusieurs fois pour maintenir l'ordre de visite final des sommets. D'où le test, après avoir dépilé `u`, pour savoir si le sommet n'a pas par hasard déjà été visité.

IV Recherche de plus court chemin

4.1 Graphe orienté

Pour calculer la distance à un sommet, s, un parcours en largeur au départ de s suffit.

Il faut, après la visite d'un arc (u, v), si v est inconnu, lui attribuer la distance de u plus 1. On garde par ailleurs en mémoire le prédécesseur u de v qui lui a donné cette distance : le plus court chemin de s à v passant nécessairement pas u, on pourra reconstruire le plus court chemin de s à v.

Algorithme Distance des sommets

entrées : G = (S, A) un graphe orienté, $s \in S$

```
1 . δ ← une fonction de distance constante sur S égale à +∞.
2 . ρ ← une fonction de parenté constante sur S égale à −1.
3 . μ ← un marquage constant à INCONNU
4 . ϕ ← une file vide
5 . μ(s) = DECOUVERT
6 . Enfiler s dans ϕ
7 . δ(s) = 0
8 . tant que ϕ n'est pas vide faire
9 .     Défiler u de ϕ
10 .    μ(u) = VISITE
11 .    pour v successeur de u faire
12 .        si  μ(v) = INCONNU alors
13 .            μ(v) = DECOUVERT
14 .            Enfiler v dans ϕ
15 .            δ(v) = δ(u) + 1
16 .            ρ(v) = u
17 .        fin si
18 .    fin pour
19 . fin tant que
20 . renvoyer  δ, ρ
```

En utilisant l'analyse faîte précédemment pour le parcours en largeur, on peut en conclure que l'algorithme termine et a une complexité égale à $O(|S| + |A|)$. De plus, les invariants du parcours en largeur permettent facilement de prouver l'invariant suivant, pour tout sommet découvert ou visité, δ lui associe sa distance à s et ρ lui associe son prédécesseur dans un chemin minimal au départ de s.

4.2 Algorithme de Dijkstra

Dans le cadre des graphes pondérés, on peut définir un autre notion de distance, liée cette fois-ci aux poids des arcs traversés. Chaque poids est vu comme un coût, et le coût d'un chemin est la somme des coûts de chaque arc. Toutefois, la distance, vue comme le *coût minimal*, ne peut être définie que si ce minimum existe. Il faut pour cela interdire les cycles de poids négatif.

On se place pour cet algorithme dans le cas, plus restrictif encore, de graphes pondérés avec poids *positifs*, *i.e.* avec une fonction d'étiquetage ρ à valeurs dans $\mathbb{R}^+$.

❦ Définition. *Poids d'un chemin*

Soit G = ((S, A), ρ) un graphe orienté pondéré, avec ρ à valeurs dans $\mathbb{R}^+$. Le poids d'un chemin de G, $c = s_0\, s_1 \cdots s_n$, est la somme des poids de chaque arc qui compose c. On note alors

$$\rho(c) = \sum_{i=1}^{n} \rho((s_{i-1}, s_i))$$

❦ Définition. *Distance dans un graphe pondéré à poids positifs.*

Soit $G = ((S, A), \rho)$ un graphe orienté pondéré, avec ρ à valeurs dans $\mathbb{R}^+$. Soit u et v deux sommets de G. La *distance* de u à v, notée $d_G(u, v)$, est le plus petit poids parmi les poids des chemins de source u et de destination v, *i.e.*

$$d_G(u, v) = \min\{\rho(c), c, \text{ un chemin de source } u \text{ et de destination } v\}$$

Si v n'est pas accessible depuis u, la distance est par convention *infinie*, ce qui est cohérent avec la convention $\min \varnothing = +\infty$.

Algorithme

Un simple parcours en largeur ne suffit plus. C'est l'invariant suivant qui devient faux et invalide l'algorithme : les éléments dans la file ne sont plus par ordre de distance. Il suffit donc de rétablir sa validité en changeant la structure de file utilisée dans le parcours en largeur. On introduit une *file de priorité* qui permet donc d'organiser les éléments, non plus par leur ancienneté, mais par ordre de priorité, la priorité étant spécifiée à l'ajout.

❦ Définition. *File de priorité*

Une *file de priorité* est un conteneur permettant les opérations suivantes :

- vérifier si la file est vide
- ajouter un élément dans la file avec une priorité p (*enfiler*)
- mettre à jour la priorité d'un élément à p (*mise à jour*)
- enlever l'élément de priorité minimale présent dans la file (*défiler*)

Algorithme Dijkstra

entrées : G = (S, A) un graphe orienté, $s \in S$

```
1 . δ ← une fonction de distance constante sur S égale à +∞.
2 . ρ ← une fonction de parenté constante sur S égale à −1.
3 . μ ← un marquage constant à INCONNU
4 . ϕ ← une file de priorité vide
5 . μ(s) = DECOUVERT
6 . δ(s) = 0
7 . Enfiler s dans ϕ avec priorité δ(s)
8 . tant que ϕ n'est pas vide faire
9 .    Défiler u de ϕ
10 .    μ(u) = VISITE
11 .    pour v successeur de u faire
12 .       /* Relâchement de l'arc (u, v) */
13 .       si δ(v) > δ(u) + ρ((u, v)) alors
14 .          δ(v) = δ(u) + ρ((u, v))
15 .          ρ(v) = u
16 .       fin si
17 .       si  μ(v) = INCONNU alors
18 .          μ(v) = DECOUVERT
19 .          Enfiler v dans ϕ avec priorité δ(v).
20 .       sinon si  v est dans ϕ avec une priorité p < δ(v)  alors
21 .          Mettre à jour la priorité de v dans ϕ à δ(v).
22 .       fin si
23 .    fin pour
24 . fin tant que
25 . renvoyer  δ, ρ
```

Voici les analyses importantes concernant cet algorithme.

1. L'algorithme termine. Le variant de la boucle est le nombre de sommets du marquage non visité.
2. L'invariant le plus important est : δ associe à chaque sommet u le poids du chemin minimal de s à u, *ne passant que* par les sommets marqués comme visités. On ajoutera que la file de priorité contient exactement les éléments découverts avec priorité $\delta(u)$ et que les éléments inconnus sont tous à distance infinie.
3. La complexité dépend des opérations de la file de priorité. On supposera que le test à vide est en O(1). On note α, β, γ les complexités des opérations défiler, enfiler et mettre à jour. La complexité finale est en $O((\alpha|S| + (\beta + \gamma)|A|)$. On n'en fera pas l'implementation, mais on peut assurer une complexité logarithmique pour les trois opérations ce qui donne une complexité en $O(\log(|S|)(|S| + |A|))$

Implémentation

Une implémentation naïve de la file de priorité peut reposer sur une simple liste de sommets et une fonction pour les priorités. La priorité étant directement obtenue dans ce cas précis dans la fonction de distance.

Les opérations enfiler et mettre à jour serait alors en O(1) et la fonction défiler en $O(|\phi|)$ pour la recherche du minimum.

```python
def defiler(fi, delta):
    im, m = -1, float("inf")
    for i in fi:
        if delta[i] < m:
            im, m = i, delta[i]
    return im

def dijkstra(G, s):
    n = len(G)
    mu = [0]*n
    delta = [float("inf")]*n
    rho = [-1]*n
    fi = []
    mu[s] = 1
    fi.append(s)
    delta[s] = 0
    while len(fi) > 0:
        u = defiler(fi, delta)
        mu[u] = 2
        for v, p in G[u]:
            d = delta[u] + p
            if d < delta[v]:
                delta[v] = d
                rho[v] = u
                if mu[v] == 0:
                    mu[v] = 1
                    fi.append(v)
    return delta, rho
```

On obtient une complexité en $O(|S|^2 + |A|) = O(|S|^2)$.

4.3 A*

Dans le cas où on s'intéresse à la distance d'un sommet destination particulier, on peut utiliser une variante de l'algorithme de Dijkstra dans lequel la priorité n'est pas la distance à la source, mais la distance *estimée* à la destination depuis la source en passant pas le sommet.

Cette estimation se fait sur la base d'une *heuristique*. Il ne s'agit plus forcément ici d'avoir un algorithme correct, mais un algorithme plus efficace et qui donne des résultats qu'on espère proche de l'optimal, sans pouvoir l'assurer.

Définition. *Heuristique pour* A*

Dans le cadre de l'algorithme A*, une heuristique sera une fonction qui prend en entrée deux sommets et renvoie une estimation de la distance entre ces deux sommets.

Il faut que le calcul de cette distance soit rapide (classiquement en $O(1)$).

On reprend l'algorithme de Dijkstra, avec en entrée un sommet source s et un sommet de destination

d. La priorité d'un sommet n'est maintenant plus $\delta(v)$, mais $\delta(v) + h(v, d)$. On s'arrête par ailleurs dès qu'on atteint le sommet d.

Algorithme A*

ENTRÉES : G = (S, A) un graphe orienté, $s \in S$, $d \in S$

```
1 . δ ← une fonction de distance constante sur S égale à +∞.
2 . ρ ← une fonction de parenté constante sur S égale à −1.
3 . μ ← un marquage constant à INCONNU
4 . ϕ ← une file de priorité vide
5 . μ(s) = DECOUVERT
6 . δ(s) = 0
7 . Enfiler s dans ϕ avec priorité δ(s) + h(s, d)
8 . tant que ϕ n'est pas vide faire
9 .    Défiler u de ϕ
10 .   μ(u) = VISITE
11 .   si u = d alors
12 .      Renvoyer δ(d) et ρ
13 .   fin si
14 .   pour v successeur de u faire
15 .      /* Relâchement de l'arc (u, v) */
16 .      si δ(v) > δ(u) + ρ((u, v)) alors
17 .         δ(v) = δ(u) + ρ((u, v))
18 .         ρ(v) = u
19 .      fin si
20 .      si  μ(v) = INCONNU alors
21 .         μ(v) = DECOUVERT
22 .         Enfiler v dans ϕ avec priorité δ(v) + h(v, d).
23 .      sinon si  v est dans ϕ avec une priorité p < δ(v) + h(v, d)  alors
24 .         Mettre à jour la priorité de v dans ϕ à δ(v) + h(v, d).
25 .      fin si
26 .   fin pour
27 . fin tant que
28 . renvoyer  d n'est pas accessible
```

1. Si h est nulle, on retombe sur l'algorithme de Dijkstra avec un arrêt sur d.
2. Si h ne surestime jamais la distance d'un sommet à d, *i.e* $\forall u \in S, h(u, d) \leq d_G(u, d)$, alors A* trouve bien le chemin minimal. Il est en général plus rapide que Dijkstra.
3. Si h ne peut surestimer la distance d'un sommet à d alors A* ne trouve pas assurément le chemin minimal. Il est en général plus rapide que dans le cas précédent.

V Arbres (Hors Programme)

❦ DÉFINITION. *ARBRES*

Un *arbre* est un graphe non orienté connexe et acyclique.

❦ Définition. *Arbres Enracinés*

Un *arbre enraciné* est la donnée d'un arbre et d'un sommet de l'arbre qui sera nommée la racine.

On peut organiser les sommets par distance à la racine, appelés niveaux.

- Le seul sommet au niveau 0 est la racine.
- La distance des sommets au dernier niveau est appelée la *hauteur*.
- un sommet a un unique voisin dans le niveau précédent, appelé sommet *parent*, *ascendant* ou *prédécesseur*.
- les autres voisins sont au niveau suivants et sont appelés sommets *enfant*, *descendants* ou *successeurs*.
- les sommets ne possédant aucun successeurs (de degré 1 et différent de la racine), sont appelés les *feuilles* de l'arbre.
- les autres sommets sont appelés des sommets *internes*.

❦ Théorème. *Chemin depuis la racine*

Dans un *arbre enraciné* il existe un unique chemin allant de la racine à tout autre sommet.

❦ Définition. *Arbres Binaire*

Un *arbre binaire* est un arbre enraciné tel que tout sommet possède au plus deux successeurs.

On dit que l'arbre est *parfait* si toutes les feuilles sont au dernier niveau, *i.e.* tous les niveaux sont remplis. Dans ce cas, si h est la hauteur de l'arbre, l'arbre contient $2^{h+1} - 1$ sommets.