Graphes, révisions

Fabrice Lembrez - PSI*

Lycée Pierre de Fermat

Plan

Deux notions de graphe

Déf - Graphe orienté, non orienté

Donnée de deux ensembles $G = (S, A)$
- l'ensemble G des sommets (ou nœuds)
- cas non orienté : ensemble S des arêtes = paires $\{x, y\}$ de sommets
- cas orienté : ensemble S des arcs = couples (x, y) de sommets
  dont évent. des boucles (x, x).

Deux notions de graphe

Déf - Graphe orienté, non orienté

Donnée de deux ensembles $G = (S, A)$
- l'ensemble G des sommets (ou nœuds)
- cas non orienté : ensemble S des arêtes = paires $\{x, y\}$ de sommets
- cas orienté : ensemble S des arcs = couples (x, y) de sommets
  dont évent. des boucles (x, x).

Représentation et enrichissement :
- des nœuds joints par des "traits" ou par des "flèches".
- ajout éventuel d'information sur les sommets ou sur les arêtes : "étiquettes", "poids".

Plan

Comptage global

Deux mesures de base :

Déf - Ordre et taille

$|G|$ (cardinal) = nombre de sommets = ordre du graphe,
$||G||$ = nombre d'arcs ou arêtes = taille du graphe ($=|S|$).

Par exemple dans le cas orienté

$$0 \leq ||G|| \leq |G|^2$$

Vocabulaire de l'adjacence

Incidence : contact arête-sommet

Adjacence : contact entre sommets par une arête

Le plus souvent, on donne la priorité aux sommets et à la relation d'adjacence.

- Cas non orienté : un sommet a des voisins formant son voisinage

$$V_G(s) = \{t, \{s,t\} \in A\}$$

- Cas orienté : un sommet a des prédécesseurs et des successeurs

$$V_G^-(s) = \{t, (t,s) \in A\} \qquad V_G^+(s) = \{t, (s,t) \in A\}$$

On définit alors le degré entrant, le degré sortant, et le degré du sommet.

Arithmétique de base

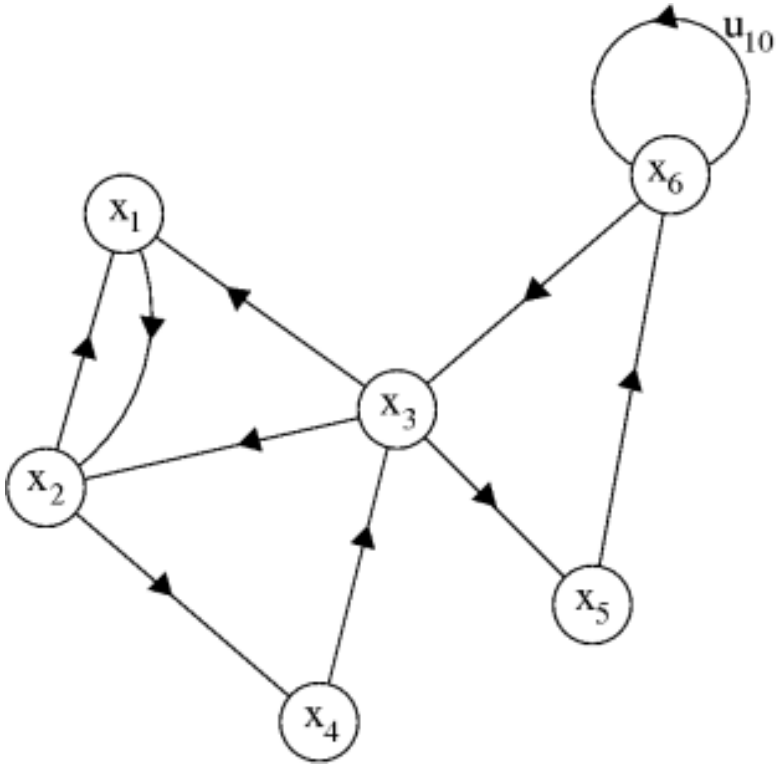
Cas orienté : somme des degrés sortants = somme des entrants
Cas non orienté : somme des degrés $= 2||G||$ donc paire
Sur un graphe non orienté, il y a un nombre pair de sommets de degré impair.

Les boucles ne perturbent pas ce compte : elles comptent pour un degré entrant et un degré sortant.

Plan

Choix de représentation

On le fait pour un graphe orienté (pour un graphe non orienté, il suffit de mettre des flèches dans les deux sens).



	x_1	x_2	x_3	x_4	x_5	x_6
x_1	0	1	0	0	0	0
x_2	1	0	0	1	0	0
x_3	1	1	0	0	1	0
x_4	0	0	1	0	0	0
x_5	0	0	0	0	0	1
x_6	0	0	1	0	0	1

Un graphe orienté et sa matrice d'adjacence

Choix de représentation

• **Liste d'adjacence :** successeurs de chaque sommet

- si les sommets sont numérotés : sous forme de liste de listes

[[],[2],[1,4],[1,2,5],[3],[6],[3,6]]

- s'ils sont étiquetés : sous forme de dictionnaire de listes

{'x1':['x2'], 'x2':['x1','x4'], 'x3':['x1','x2','x5'], 'x4':['x3'], 'x5':['x6'], 'x6':['x3','x6']}

• **Matrice d'adjacence :** $M_{i,j} = 1$ s'il y a une arête de i vers j, 0 sinon. Cette matrice est symétrique si on a affaire à un graphe orienté.

• **Matrice d'adjacence pour graphe pondéré :** on utilise des poids comme coefficients, $+\infty$ quand il n'y a pas d'arête de i vers j.

Complexité du parcours de base

On reverra des parcours "géographiques" en suivant le sens des flèches, il s'agit ici d'un parcours complet pour déterminer des propriétés générales du graphe, par exemple le nombre d'arêtes.

Complexité du parcours complet

Parcours de la matrice d'adjacence en $O(|G|^2)$, parcours de la liste d'adjacence en $O(|G| + ||G||)$.

Quel choix de représentation ?

- Avantage de la liste d'adjacence : compacité

Graphes peu denses

S'il y a peu d'arêtes, privilégier la liste d'adjacence (ex : carte routière).

Peu d'arêtes $=$ un degré moyen $\frac{||G||}{|G|}$ assez limité ($<< |G|$).

- Avantage de la matrice d'adjacence

Quel choix de représentation ?

- Avantage de la liste d'adjacence : compacité

Graphes peu denses

S'il y a peu d'arêtes, privilégier la liste d'adjacence (ex : carte routière).

Peu d'arêtes $=$ un degré moyen $\frac{||G||}{|G|}$ assez limité ($<< |G|$).

- Avantage de la matrice d'adjacence

Dans une graphe orienté, si on doit "remonter" le sens des flèches, pas besoin de recherche, juste une interrogation directe.

Plan

Chemins, cycles

Chemin = succession finie d'arêtes (ou de sommets) compatible avec la relation d'incidence.

Longueur = nombre d'arêtes empruntées (sauf s'il y a un poids).

Cycle = chemin qui termine à son point de départ $s_n = s_0$

Vous entendrez parfois "chaîne" pour chemin et "circuit" pour cycle (dans le cas orienté).

Compter les chemins avec la matrice d'adjacence

Le coefficient $[M^k]_{ij}$ compte les chemins de longueur k de s_i à s_j.

Ce qui donne une autre problématique où travailler avec la matrice d'adjacence est intéressant.

Connexité (cas non orienté)

De façon rapide, connexe = d'un seul tenant. Formellement,

Déf - Connexité, composantes connexes [non orienté]

Un graphe non orienté est connexe quand entre deux sommets quelconque il existe un chemin.
Tout graphe non orienté se partitionne en sous-graphes connexes les plus grands possibles, les composantes connexes.

Avantage : complexité

Si le graphe est connexe $|G| \leq ||G||$. Ainsi parcourir la liste d'adjacence est en $O(||G||)$.

Forte connexité dans le cas orienté

Pour un graphe non orienté, la connexité est une hypothèse courante et naturelle.

Déf - Forte connexité [orienté]

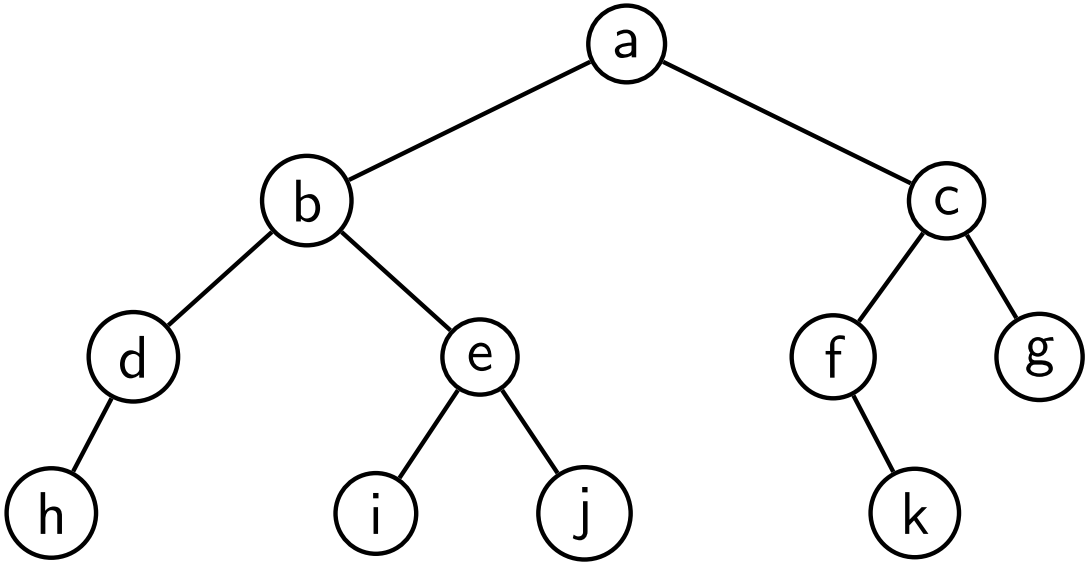
Dans le cas orienté, on parlera de forte connexité quand on peut aller d'un sommet à l'autre et inversement.

Plan

Parcourir un arbre (vers le bas !)

On commence par une situation plus simple que le graphe et fréquemment utilisée.

Arbre (enraciné) : un nœud "racine" qui a des "fils", chaque fils qui a des fils, etc jusqu'aux nœuds terminaux, les "feuilles".

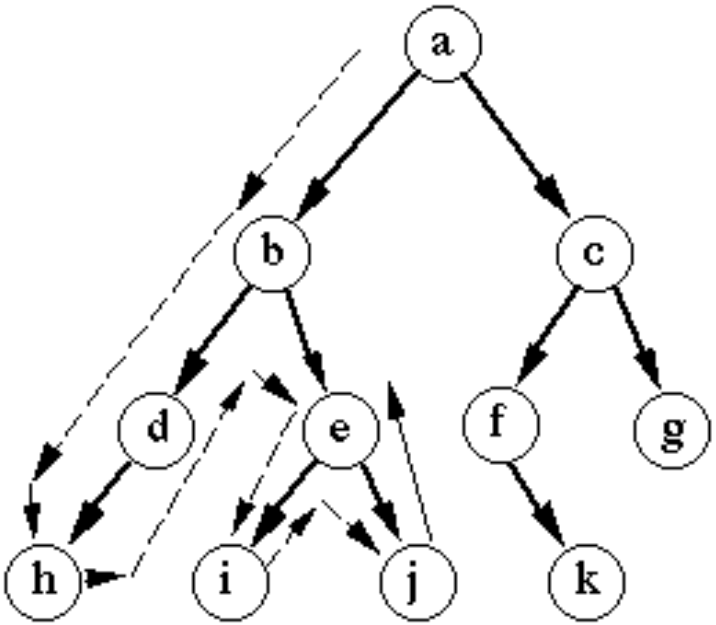


Racine : a
Feuilles : h,i,j,k, et g

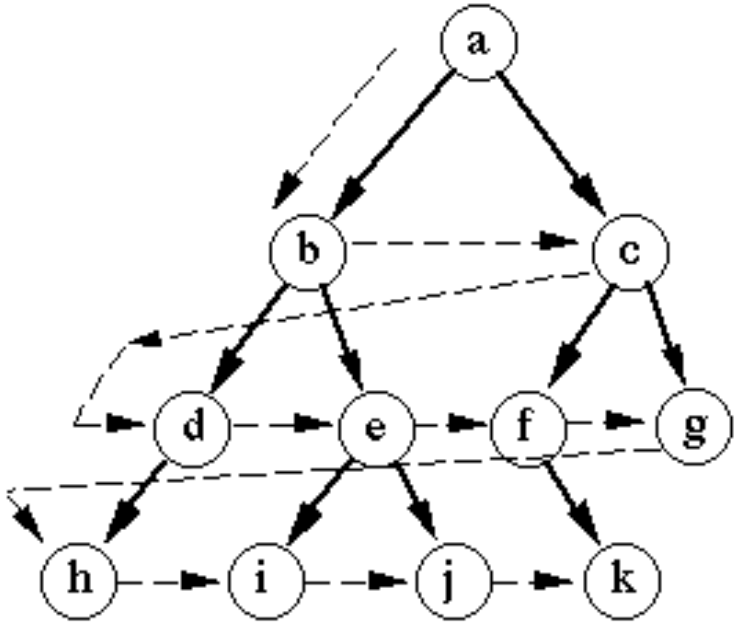
Parcourir un arbre (vers le bas !)

On commence par une situation plus simple que le graphe et fréquemment utilisée.

Arbre (enraciné) : un nœud "racine" qui a des "fils", chaque fils qui a des fils, etc jusqu'aux nœuds terminaux, les "feuilles".
Il y a alors deux modes de parcours naturels.



Depth-first search



Breadth-first search

Parcours en profondeur d'abord

Le squelette du parcours est simplissime :

```
def parcoursArbreProf(listeAdj,noeud):
    for fils in listeAdj[noeud]:
        parcoursArbreProf(listeAdj,fils)
```

- La programmation récursive est très adaptée.
- Il y a arborescence des appels, mais c'est voulu
  on verra la complexité plus loin (cf complexité plus loin).
- Principe : toujours descendre au 1er fils d'abord.
  Après retour, les autres fils...
- S'il n'y a pas de fils (=feuille), c'est un cas terminal.

Parcours + action + mémorisation

- Mémoriser où on en est ? pas besoin la récursivité le fait pour nous
  en interne : une "pile des appels en cours" (call stack)
- Faire passer une info / Constituer une info lors du parcours :
  Initialiser avant la récursivité, puis passer cela en argument
- Mener une action lors de la visite : choix avant la descente / après la remontée

```
def parcoursArbreProf(listeAdj,noeud,memoires):
    #-----ici action en arrivant sur le noeud-----
    for fils in listeAdj[noeud]:
        parcoursArbreProf(listeAdj,fils,memoires)
    #-----ici action en quittant le noeud-----

def initialiseParcours(listeAdj,depart):
    #initialise memoires
    parcoursArbreProf(listeAdj,depart,memoires)
```

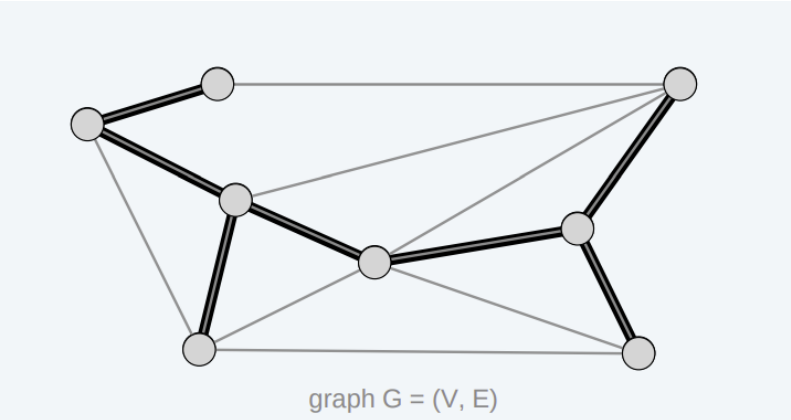
Plan

De l'arbre au graphe

Déf - L'arbre est un cas particulier de graphe

Un arbre est un graphe connexe, sans cycle. Un arbre défini ainsi n'a pas de sommet privilégié (pas de "racine" naturelle).

A partir d'un graphe connexe, on montre qu'il est possible de construire un "arbre couvrant" occupant tous les sommets et certaines arêtes. Il permet de parcourir le graphe et c'est ce qui explique qu'on retrouve les parcours en profondeur d'abord et en largeur d'abord.



Exemple d'arbre couvrant(spanning tree) pour un graphe.

Conséquence sur le parcours d'un graphe

Parenté et différence entre parcourir un arbre et un graphe

Les démarches de parcours d'arbre peuvent fonctionner sur un graphe, puisqu'il existe des arbres couvrants. Il faut cependant éviter les cycles.

Pour adapter les algorithmes de parcours, on devra mémoriser les points déjà visités, soit en les "marquant", soit sous forme d'un dictionnaire de sites visités.

Il peut y avoir une bonne raison de privilégier l'approche "dictionnaire" :
- si les nœuds sont étiquetés par autre chose que des nombres
- si on visite une portion seulement de l'arbre (le dictionnaire sera plus léger)

Plan

Profondeur d'abord

```
def grapheProf(listeAdj,noeud,noeudsVisites):
    noeudsVisites[noeud]=True #valeur importe peu
    for successeur in listeAdj[noeud]:
        if successeur not in noeudsVisites:
            grapheProf(listeAdj,successeur)

noeudsVisites=dict()
parcoursGrapheProf(listeAdj,x0,noeudsVisites)
```

Profondeur d'abord

```
def grapheProf(listeAdj,noeud,noeudsVisites):
    noeudsVisites[noeud]=True #valeur importe peu
    for successeur in listeAdj[noeud]:
        if successeur not in noeudsVisites:
            grapheProf(listeAdj,successeur)

noeudsVisites=dict()
parcoursGrapheProf(listeAdj,x0,noeudsVisites)
```

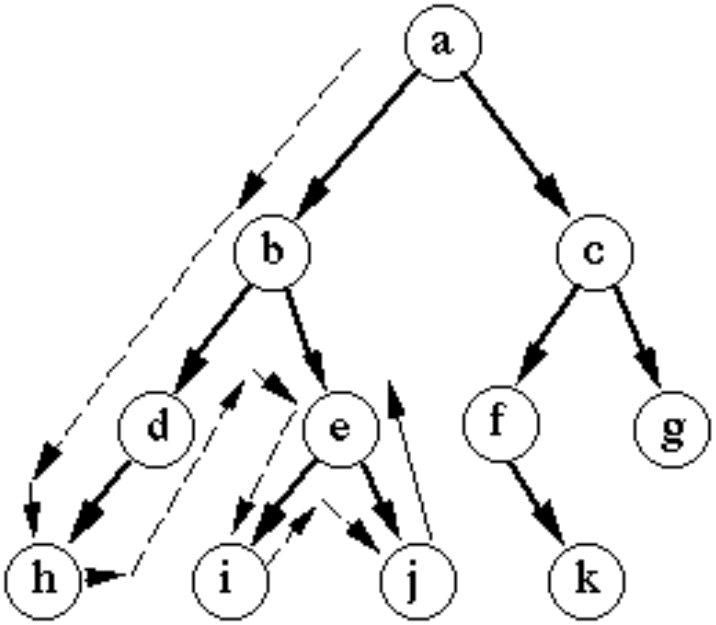
Complexité temporelle : il y a à la fois la récursivité et une boucle
- on appelle la fonction uniquement pour un nœud non encore visité
  donc nombre d'appels inférieur au nombre de nœuds
- nombre de tours de boucle : inférieur à $d^+(\texttt{noeud})$. Donc complexité totale de toutes les boucles inférieure à $||G||$.

Tenir compte des deux (ex : nœuds n'ayant aucun successeur).

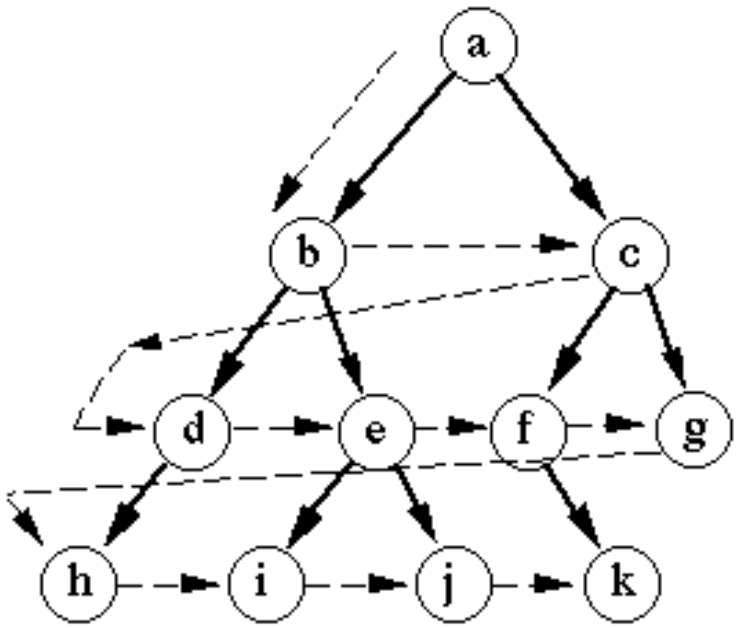
Complexité en $O(|G| + ||G||)$, donc $O(||G||)$ si connexe.

Plan

Parcours en largeur d'abord, sur un arbre



Depth-first search



Breadth-first search

Parcours en largeur d'abord

On visite successivement
- tous les fils
- puis tous les petit-fils...
- toute une génération avant de passer à la suivante.

Intérêt : ce parcours respecte la notion de proximité à la racine. Par exemple on peut se demander à quelle génération une certaine propriété apparaît, sans avoir besoin de parcourir tout l'arbre.

Mémorisation des positions à visiter dans une file

Le parcours en largeur demande de mémoriser les nœuds vus et non terminés
- en arrivant sur un nœud on stocke ses fils dans un conteneur
- en quittant le nœud on le retire du conteneur
- et on passe au prochain nœud dans le conteneur.

Le conteneur adapté est donc une file, variable `maFile` (j'utilise un dèque Python ici).

Parcours en largeur d'abord (arbre)

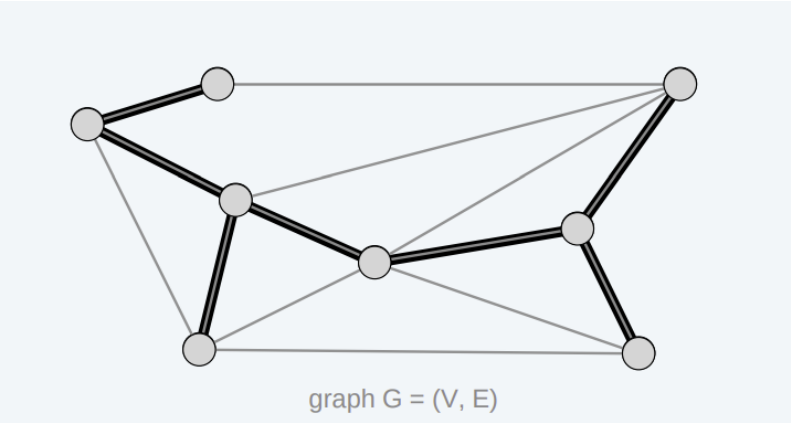
```
from collections import deque

def parcoursArbreLargeur(listeAdj):
    maFile=deque[0]
    while len(maFile)>0:
        #---insérer une action sur le noeud ici ---
        pere=popleft(maFile)
        for fils in listeAdj[pere]:
            maFile.append(fils)
```

Plan

Rappel : de l'arbre au graphe

A partir d'un graphe connexe, il est possible de construire un "arbre couvrant" occupant tous les sommets et certaines arêtes. Il permet de parcourir le graphe et c'est ce qui explique qu'on retrouve les parcours en profondeur d'abord et en largeur d'abord.



Exemple d'arbre couvrant(spanning tree) pour un graphe.

Pour adapter les algorithmes de parcours, on devra mémoriser les points déjà visités, sous forme d'un dictionnaire de sites visités.

Largeur d'abord : deux conteneurs

- la file des nœuds à visiter
- le dictionnaire des nœuds visités

peuvent sembler redondants.

En fait non :

- la file est ordonnée : détermination rapide du prochain,
- le dictionnaire permet une réponse rapide à la question "as-tu déjà vu...."

On ne quitte pas ces conteneurs au même moment. En revanche on y entre au même moment : lors de la découverte du nœud.

Largeur d'abord

```
from collections import deque

def parcoursGrapheLargeur(listeAdj):
    noeudsVisites=dict()
    maFile=deque[0]
    while len(maFile)>0:
        #---insérer une action sur le noeud ici ---
        noeud=popleft(maFile)
        for succ in listeAdj[noeud]:
            if succ not in noeudsVisites:
                maFile.append(succ)
                noeudsVisites[succ]=True
```

Complexité

- la longueur de la file varie, en montant et en descendant. Mais un élément s'en va définitivement à chaque étape et ne revient jamais. Le nombre de tours de boucle dans le while est donc majoré par $|G|$.
- le nombre de tours de boucle d'un for est inférieur au degré sortant du nœud $d^+(\texttt{noeud})$. Donc au total, la complexité de l'ensemble des boucles for est inférieure à $||G||$.

Tenir compte des deux (ex : nœuds n'ayant aucun successeur).

Complexité en $O(|G| + ||G||)$, donc $O(||G||)$ si connexe.

Complexité

Complexité d'un parcours

De façon générale, quand on décrit (tous) les sommets et (toutes) les arêtes issues de chaque sommet en ne revenant jamais sur un sommet déjà visité, on attend une complexité en $O(|G| + ||G||)$.

Dans la plupart des graphes, il y a plus d'arêtes que de sommets (notamment c'est le cas quand on n'a pas de sommet isolé) et on peut même dire que cette complexité dépend de la taille : $O(||G||)$.

Plan

Sommets visités

Ensemble des sommets visités

Lors d'un des deux parcours précédents, à partir d'un sommet s_0 on passe en revue
- dans le cas non orienté : toute la composante connexe de s_0, avec toutes ses arêtes même si on emprunte certaines et pas d'autres
- dans le cas orienté : seulement la "descendance" de s_0

En conséquence, dans le cas NON orienté, un certain nombre d'informations se déduisent directement des parcours précédents (éventuellement en rajoutant une mémorisation)

Application (cas non orienté)

Connexité, recherche d'une composante connexe : la réponse est le dictionnaire des nœuds visités

Construction d'un arbre couvrant : ce sont les arêtes père-fils utilisées pour la visite (on peut les mémoriser)

Recherche de cycles : dès qu'un des "if" sonne False, on a un cycle (ce sont les arêtes qu'on a refusé de suivre).

Distance (de base) à un point donné : il faut utiliser le parcours en largeur d'abord pour visiter tous ceux qui sont à distance 1, puis 2, etc. On peut stocker la distance dans le dictionnaire : elle augmente de 1 en passant du père au fils. On peut aussi mémoriser les arêtes de façon à pouvoir reconstruire le chemin optimal. Il y a une variante dans un graphe à poids : Dijkstra.

Besoin d'algorithmes plus fins pour le cas orienté

La connexité, la recherche de cycles ne se généralisent pas directement aux graphes orientés.

Plan

Dijkstra

Dans un graphe à poids positifs, on a une autre notion de "longueur" : la somme des poids des arêtes.

Dijkstra = plus court chemin depuis s0

Chercher les plus courts chemins depuis une origine s0 donnée sur un graphe (évent. orienté) à poids positifs. Ce peut être tous les plus courts chemins depuis s0, ou alors en arrêtant l'algorithme, le plus court de s0 à s1.

Dijkstra, variante de la recherche en profondeur

- on utilise le dictionnaire des sommets visités pour mémoriser la meilleure distance connue
- le conteneur n'est plus une "file" mais une file de priorité : on sort de la file celui qui est le plus proche de s0
- et en enfilant ses successeurs, on met à jour l'information sur la meilleure distance connue.

Dijkstra : correction

Invariant de boucle : cela revient à décrire le contenu du dictionnaire `MDC` des meilleures distances connues.

- pour les sommets qui n'ont jamais été dans la file `MDC[s]` non défini
- pour les sommets s sortis de la file `MDC[s]=distmin(s0,s)`
- pour ceux qui sont encore dans la file

`MDC[s]=min Hs`

`Hs={distmin(s0,s')+dist(s's), s' sommet sorti de la file }`

Dijkstra

```
def Dijkstra(origine,listAdjPoids):
    distMin={origine:0}
    filePrio=[origine]
    while len(filePrio)>0:
        point=recupereEtEnleveMin(filePrio,distMin)
        dpoint=distMin[point]
        for voisin,pds in listAdjPoids[point]:
            dist=dpoint+pds
            if voisin not in distMin:
                distMin[voisin]=dist
                ajouteAFile(filePrio,voisin)
            elif dist<distMin[voisin]:
                distMin[voisin]=dist
    return distMin
```

Dijkstra

Attention différentes versions !

Dans cette version de l'algorithme j'ai fait les choix suivants
- ne mettre le sommet qu'une fois dans la file
- garder la mémoire de la meilleure distance connue dans `distMin`

Elle ressemble à la version blanc/gris/noir qu'on rencontre parfois mais le dictionnaire m'évite de colorer les sommets ou faire des initialisations à la valeur ∞. En effet
- les sommets blancs (non rencontrés) sont ici non présents dans le dictionnaire `distMin`
- les gris (rencontrés non traités) sont dans le dictionnaire et dans la file de priorité
- les noirs (traitement terminé) sont dans le dictionnaire et sortis de la file de priorité.

Djijkstra : complexité

C(parcours profondeur) x C(AjoutSuppr ds la file de priorité)

Car à chaque étape du parcours en largeur il faut récupérer l'élément prioritaire

- donc un facteur supplémentaire $O(||G||)$ si on utilise une recherche ordinaire

- on pourrait faire mieux si on utilise une structure de file de priorité adaptée, comme `heappush, heappop` qui existent dans le module `heapq`. On peut les utiliser mais il faut faire évoluer dans le code ci-dessus. On montre que le facteur multiplicatif est alors plutôt $O(\ln ||G||)$.

Plan

Objectif de A étoile

Dijkstra fonctionne de façon "concentrique" en garantissant des distances minimum dans toutes les directions à partir de s0. Quand on a un objectif, par exemple dans le cas d'une carte routière, on sait qu'il y a des directions de départ moins pertinentes que d'autres : il semble peu pertinent a priori de partir à l'opposé de l'objectif... quoi que, quelquefois...

A*

Dans les mêmes conditions que Dijkstra, obtenir rapidement le plus court chemin entre deux sommets donnés s0 et s1.
On suppose disposer d'une heuristique, fonction d'estimation de la distance restante.

A étoile, variante de Dijkstra

Algorithme A étoile, variante de Dijkstra

On utilise deux dictionnaires :
- un qui contient les meilleures distances connues (MDC) depuis s0,
- l'autre contient les évaluations MDC[s]+évaluation du chemin qui reste

Il faut les mettre à jour convenablement. La liste de priorité utilise la distance évaluée.

A étoile

```
def Astar(origine,but,listAdjPoids,coordSommets):
    distMin={origine:0}
    evalDist={origine:0} #0 simple valeur d'initialisation
    filePrio=[origine]
    point=0
    while point!=but:
        point=recupereEtEnleveMin(filePrio,evalDist)
        dpoint=distMin[point]
        for voisin,pds in listAdjPoids[point]:
            dist=dpoint+pds
            dEstim=dist+dHeurist(coordSommets,voisin,but)
            if voisin not in distMin:
                distMin[voisin]=dist
                evalDist[voisin]=dEstim
                ajouteAFile(filePrio,voisin)
            elif dist<distMin[voisin]:
                distMin[voisin]=dist
                evalDist[voisin]=dEstim
    return distMin
```

Plan

Floyd Warshall

Floyd-Warshall (plus d'origine privilégiée)

Trouver l'ensemble des plus courts chemins entre deux points quelconques du graphe.

Idée : utiliser la notion de programmation dynamique. Voir TP.

Comparaison des parcours

Algorithme	Prochain sommet	Structure adaptée
Profondeur d'abord	Le fils de l'actuel d'abord	Pile (cachée)
Largeur d'abord	Les plus anciens d'abord	File
Dijkstra	Le plus proche d'abord	File de priorité