

XX – Parcours de graphes

1/ Introduction

Les graphes sont utilisés pour modéliser les réseaux routiers ou de télécommunication, et les algorithmes permettant de les manipuler sont fondamentaux. De nombreux problèmes peuvent ainsi être résolus à l'aide des graphes : le guidage par satellite, la stabilité des réseaux de communication, les allocations de fréquences, etc...

Notre programme se focalise sur les parcours de graphes et les recherches de plus court chemin dans des graphes pondérés (représentés à l'aide de listes d'adjacence). D'autres problèmes classiques serviront de prétexte aux épreuves de concours.

2/ Les parcours de graphes

2.1) Le principe

Le parcours d'un graphe consiste à explorer les diverses arêtes en partant d'un sommet donné s afin de découvrir le reste du graphe. Au cours de cette recherche, on accumule des informations qui pourront servir ultérieurement.

Lors du parcours, on explore toutes les arêtes issues d'un sous-ensemble de sommets initialement réduit à $\{s\}$. On ne visitera ainsi que les sommets v pour lesquels il existe un chemin reliant s à v : ce sont les **descendants** de s .

Dans le cas d'un graphe non-orienté, on explore ainsi toute la composante connexe du sommet initialement choisi. On obtiendra les autres composantes en relançant l'exploration.

Quand on effectue l'exploration du graphe « à la main », chaque sommet accessible va passer par trois états, ce qui permet d'illustrer graphiquement le parcours :

- blanc s'il n'a pas encore été découvert ;
- gris s'il a été découvert mais que son traitement n'est pas terminé ;
- noir lorsque le traitement est terminé.

Par contre, lors d'une exploration algorithmique, on utilisera simplement une liste *visites* pour recenser les sommets déjà visités.

2.2) Parcours en profondeur d'un graphe

L'idée est de chercher à descendre systématiquement le plus loin possible dans le graphe. Pour cela, on part d'un sommet que l'on marque comme *visité* (il devient gris), puis on choisit un sommet voisin non visité (donc blanc) que l'on marque à son tour, et l'on recommence. Quand tous les voisins du sommet courant ont déjà été visités, son traitement est terminé et il devient noir. On remonte alors au sommet précédent et l'on réitère le processus.

Exemple : appliquer ceci aux graphes n°1 et n°2 des exercices du chapitre précédent.

Pour implémenter cet algorithme, on va utiliser les deux procédures suivantes.

- ❶ Une sous-procédure récursive *decouvre*, qui permet de traiter un sommet non visité en le marquant comme visité et en appelant récursivement la procédure sur tous les éléments **non visités** de sa liste d'adjacence.
- ❷ Une procédure principale *parcoursPR* qui initialise les variables utilisées et qui lance la procédure récursive à partir du sommet initialement choisi.

On affiche un message à chaque fin de traitement, mais on peut imaginer d'autres opérations. En vue d'applications ultérieures, on emmagasine quelques informations supplémentaires :

- on crée un compteur *date* incrémenté à chaque étape de l'algorithme ;
- on stocke la date de début de traitement, la date de fin de traitement et l'unique antécédent de chaque sommet dans les listes *debut*, *fin* et *peres* ;
- la procédure principale renvoie alors ces trois listes (ne pas hésiter à les supprimer du programme si elles s'avèrent inutiles).

```
def parcoursPR(L,s):
    # Initialisation des variables
    n=len(L) ; date=1 ; visites=[]
    debut=[-1]*n ; fin=[-1]*n
    peres=[None for i in range(n)]

    # Sous-procédure récursive
    def decouvre(i):
        nonlocal date
        debut[i]=date
        visites.append(i)
        date=date+1
        for j in L[i]:
            if j not in visites:
                print('Découverte du sommet',j)
                peres[j]=i
                decouvre(j)
        fin[i]=date
        print("Fin de traitement du sommet",i)
        date=date+1

    # On lance l'exploration à partir du sommet s
    decouvre(s)

    return debut, fin, peres
```

Remarque : la récursivité permet ici de traiter élégamment et simplement le problème de la remontée au sommet précédent lors de l'exploration. Comme l'algorithme comporte peu d'étapes, l'utilisation de fonctions récursives ne pose aucun souci.

2.3) Parcours en largeur d'un graphe

Le parcours en largeur d'un graphe consiste à visiter tous les voisins du sommet courant et à les traiter avant de passer à leurs descendants. Pour cela, on part d'un sommet que l'on marque comme *visit  * (il devient gris) et que l'on ajoute    une file d'attente. Ensuite, on d  file le sommet de t  te, on enfile tous les sommets non marqu  s (donc blancs) de sa liste d'adjacence et on les marque    leur tour. Le traitement du sommet d  fil   est alors termin   et il devient noir. On r  it  re le proc  d   jusqu'     puisement de la file.

Rappels : dans une *structure de pile*, c'est le dernier   l  ment ajout   qui est retir   en premier (comme dans une pile d'assiettes) alors que dans une *structure de file*, c'est le premier   l  ment entr   qui est retir   en premier (comme dans une file d'attente, si l'on a affaire    des gens civilis  s). Les listes doublement chain  es (type deque, dans la biblioth  que collections) permettent de manipuler facilement les piles comme les files en Python :

- on ajoute des   l  ments    la fin ou au d  but gr  ce    `append()` et `appendleft()` ;
- on retire le dernier ou le premier   l  ment gr  ce    `pop()` et `popleft()`.

On va comme pr  c  demment emmagasiner quelques informations suppl  mentaires :

- ❶ la liste `peres` contient l'unique ant  c  dent de chaque sommet lors du parcours ;
- ❷ la liste `dist` contient la distance, en nombre d'arcs, qui s  pare chaque sommet du sommet initial. Elle vaut `np.inf` si le sommet est inaccessible.

```
def parcoursL(L,s):
    # Initialisation des variables
    n=len(L)
    visites=[s]
    F=deque([s])
    peres=[None for i in range(n)]
    dist=[np.inf for i in range(n)]
    dist[s]=0

    # Exploration it  rative
    while len(F)>0:
        i=F.popleft()
        for j in L[i]:
            if j not in visites:
                print('D  couverte du sommet',j)
                dist[j]=dist[i]+1
                peres[j]=i
                visites.append(j)
                F.append(j)
            print("Fin de traitement du sommet",i)

    return peres,dist
```

Remarque : on supprimera   videmment les listes `peres` et `dist` si l'on n'en a pas besoin.

2.4) Parcours en profondeur, version itérative

On peut effectuer les parcours en profondeur de manière itérative en utilisant une pile pour recenser les sommets en cours de traitement. On empile les sommets rencontrés au cours de l'exploration, en commençant par le sommet initial, et on les dépile lorsque leur traitement est fini, c'est-à-dire quand tous les descendants ont été visités.

Remarque : on utilisera une boucle `while` portant sur `Ls`, la liste des sommets non visités adjacents au sommet en cours de traitement, que l'on met à jour à chaque étape pour descendre dans le graphe.

```
def parcoursP(L,s):
    # Initialisation des variables
    n=len(L) ; date=1
    debut=[-1]*n ; fin=[-1]*n
    peres=[None for i in range(n)]
    visites=[s] ; P=[s]
    j=0

    # Exploration itérative
    debut[s]=date
    while len(P)>0:
        Ls=[i for i in L[s] if i not in visites]

        while len(Ls)>0:
            j=Ls[0]
            peres[j]=s
            P.append(j)
            visites.append(j)
            date=date+1
            debut[j]=date
            s=j
            Ls=[i for i in L[s] if i not in visites]

        j=P.pop()
        print("Fin de traitement du sommet",j)
        date=date+1
        fin[j]=date
        if len(P)>0:
            s=P[-1]

    return debut, fin, peres
```

3/ Propriétés et applications des parcours de graphes

Dans la suite, on considère un graphe $G = (S, A)$. On notera $|S|$ son nombre de sommets et $|A|$ son nombre d'arêtes. Soit enfin s le sommet à partir duquel on lance les parcours en largeur et en profondeur.

3.1) Propriétés du parcours en profondeur

Proposition

La complexité du parcours en profondeur est linéaire.

PREUVE – la fonction `decouvre` s'applique au plus une fois par sommet car seuls les sommets blancs peuvent être découverts et une fois découverts, ils sont marqués et ne redeviennent plus jamais blancs. Examinons maintenant le coût de cette fonction :

- En dehors de la boucle, elle effectue un nombre fini et fixé d'opérations. Leur coût total est alors en $\mathcal{O}(|S|)$.
- Elle comporte une boucle `for` de même longueur que la liste d'adjacence du sommet découvert et qui contient un nombre fini et fixé d'opérations. Comme la somme des longueurs des listes d'adjacence est $|A|$, le coût total de ces boucles est en $\mathcal{O}(|A|)$.

L'initialisation des variables ayant un coût en $\mathcal{O}(|S|)$, la complexité totale de notre algorithme est donc en $\mathcal{O}(|S| + |A|)$: on a bien affaire à une complexité linéaire.

Proposition

Le parcours en profondeur découvre tous les descendants du sommet s . Parmi ceux-ci, v est un descendant de u si, et seulement si, v est découvert pendant que u est gris.

Théorème (des parenthèses)

Reprenons les listes `debut` et `fin` renvoyées par le parcours en profondeur. Soient u et v deux des sommets découverts. Une et une seule des trois propositions suivantes est alors vérifiée :

- ❶ *Les intervalles $[debut[u], fin[u]]$ et $[debut[v], fin[v]]$ sont disjoints.
Dans ce cas, ni u ni v n'est un descendant de l'autre.*
- ❷ *L'intervalle $[debut[u], fin[u]]$ est inclus dans l'intervalle $[debut[v], fin[v]]$.
Dans ce cas, u est un descendant de v .*
- ❸ *L'intervalle $[debut[v], fin[v]]$ est inclus dans l'intervalle $[debut[u], fin[u]]$.
Dans ce cas, v est un descendant de u .*

PREUVE – on peut supposer sans perte de généralité que $debut[u] < debut[v]$.

- Si $debut[v] < fin[u]$, cela signifie que v a été découvert alors que u était encore gris : c'est de ce fait un descendant de u . Comme l'appel récursif `decouvre(v)` a été lancé après `decouvre(u)`, il se termine avant lui donc on a $fin[v] < fin[u]$. Ainsi, l'intervalle $[debut[v], fin[v]]$ est bien inclus dans $[debut[u], fin[u]]$.
- Si $debut[v] > fin[u]$, les intervalles $[debut[v], fin[v]]$ et $[debut[u], fin[u]]$ sont disjoints. Ainsi, aucun des deux sommets n'a été découvert tandis que l'autre était gris. D'après la proposition précédente, ni u ni v n'est un descendant de l'autre.

Remarque : il en découle que pour deux sommets découverts, v est un descendant de u si, et seulement si, $debut[u] < debut[v] < fin[v] < fin[u]$. On a même mieux :

Théorème (du chemin blanc)

Le sommet v est un descendant du sommet u si, et seulement si, au moment où u est découvert, il existe un chemin de u vers v composé uniquement de sommets blancs.

PREUVE – on raisonne par double implication.

- Supposons que v soit un descendant de u : il existe donc un chemin qui relie v à u . Si le sommet w est sur ce chemin, c'est également un descendant de u de sorte que $\text{debut}[u] < \text{debut}[w]$ d'après le théorème des parenthèses. Par conséquent, le sommet w est blanc quand u est découvert.
- Pour l'implication réciproque, on raisonne par l'absurde. Soit v un sommet tel qu'il existe un chemin blanc de u vers v au moment où u est découvert. Supposons que v n'est pas un descendant de u . Sans perte de généralité, on peut supposer que v est le sommet le plus proche de u vérifiant cette propriété. Notons alors w le père de v sur le chemin entre u et v . Par hypothèse sur v , le sommet w est un descendant de u . Ainsi, $\text{fin}[w] < \text{fin}[u]$. Par ailleurs :
 - À l'instant $\text{debut}[u]$, v est blanc donc $\text{debut}[u] < \text{debut}[v]$.
 - Si v était toujours blanc à l'instant $\text{debut}[w]$, alors v serait découvert pendant le traitement de w et il serait un descendant de w , donc de u . C'est exclu par hypothèse. De ce fait, on a $\text{debut}[v] < \text{debut}[w] < \text{fin}[w] < \text{fin}[u]$.

Ainsi, $\text{debut}[v]$ appartient à $[\text{debut}[u], \text{fin}[u]]$. Le théorème des parenthèses assure que v est descendant de u , d'où la contradiction recherchée.

3.2) Applications du parcours en profondeur**(a) Chemins d'un sommet à un autre**

Soient a et b deux sommets d'un graphe. En lançant un parcours en profondeur depuis le sommet a , on peut déterminer s'il existe un chemin le reliant au sommet b et le reconstituer à l'aide de la liste *peres*. C'est l'objet de l'exercice 3.

Remarque : on verra que le parcours en largeur permet d'obtenir de la même manière le plus court chemin d'un sommet à un autre.

(b) Composantes connexes

Voici comment déterminer la liste des composantes connexes d'un graphe non-orienté.

- ❶ On écrit une procédure auxiliaire *origine(L, s, visites)* qui utilise le parcours en profondeur pour déterminer tous les descendants du sommet s . On renvoie leur liste (c'est la composante connexe de s) et on les ajoute à une liste *visites*, qui aura été créée en dehors de la procédure, pour les marquer.
- ❷ On écrit une procédure principale *composantes(L)* qui initialise la liste *visites* et qui lance la procédure précédente sur tous les sommets non marqués.

On obtient de la sorte les différentes composantes connexes du graphe.

On se propose d'écrire et tester ces procédures dans l'exercice 4.

(c) Composantes fortement connexes

C'est plus délicat car la composante fortement connexe d'un sommet s est constituée de tous les descendants du sommet s qui ont aussi s pour descendant. Pour déterminer toutes ces composantes, on aura besoin d'utiliser la notion de *graphe transposé*.

Définition

Soit $G = (S, A)$ un graphe orienté. Le **graphe transposé** de G est le graphe orienté obtenu en inversant les sens de parcours de tous ses arcs. On le note ${}^tG$.

Autrement dit, on a ${}^tG = (S, {}^tA)$ où ${}^tA = \{(v, u) \mid (u, v) \in A\}$.

❶ Méthode naïve

Considérons un sommet s du graphe et notons $\mathcal{F}_s$ sa composante fortement connexe.

- À l'aide d'un parcours en profondeur du graphe G , on trouve tous les descendants du sommet s . Ils forment une liste *chemin* contenant entre autres tous les sommets de $\mathcal{F}_s$.
- Avec un parcours en profondeur de ${}^tG$, on trouve tous les sommets depuis lesquels on peut rejoindre s en passant uniquement par des points de *chemin*. On montre en effet qu'un chemin reliant deux sommets de $\mathcal{F}_s$ ne passe que par des sommets de $\mathcal{F}_s$.
- Les sommets obtenus sont ainsi tous ceux pour lesquels il existe un chemin allant à s et un chemin provenant de s , donc sa composante fortement connexe.

Pour ne pas effectuer d'explorations inutiles, on mettra les sommets des composantes déjà déterminées dans une liste *visitesF*, qui sera la valeur initiale de la liste *visites* lors des divers parcours en profondeur.

❷ Méthode experte (algorithme de Kosaraju)

Cette amélioration de la méthode précédente utilise les dates de fin de traitement.

- On effectue un parcours en profondeur complet du graphe, qui renvoie la liste *finis* des sommets rangés par ordre décroissant de date de fin de traitement.
- On effectue un parcours en profondeur complet du graphe transposé, en partant toujours du sommet u de date de fin de traitement maximale pour le premier parcours.
- Ceci nous fournit les composantes fortement connexes du graphe initial.

PREUVE – lors de la première étape du second parcours, on visite tous les sommets de la composante fortement connexe de u puisque G et ${}^tG$ ont les mêmes composantes. Mais peut-on en découvrir d'autres ? Soit v un des sommets découverts. Déjà, $\text{fin}[v] < \text{fin}[u]$.

- Le sommet v est accessible depuis u dans ${}^tG$, donc u est un descendant de v dans G .
- Si u est découvert avant v dans G , alors $\text{debut}[u] < \text{debut}[v]$ et l'on déduit du théorème des parenthèses que v est un descendant de u dans G .
- Sinon, $\text{debut}[u] > \text{debut}[v]$ et les intervalles $[\text{debut}[u], \text{fin}[u]]$ et $[\text{debut}[v], \text{fin}[v]]$ sont disjoints d'après le théorème des parenthèses. Dans ce cas, ni u ni v n'est un descendant de l'autre, ce qui est contradictoire. On se situe donc dans le premier cas.

Ainsi, v appartient à la composante fortement connexe de u . La première étape du second parcours découvre donc tous les sommets de cette composante, et uniquement eux.

De même, on découvre à l'étape suivante tous les sommets de la composante fortement connexe de u – puisqu'aucun d'entre eux n'a été découvert à l'étape précédente – et rien qu'eux. Et ainsi de suite, ce qui conduit au résultat.

3.3) Propriétés du parcours en largeur

Proposition

La complexité du parcours en largeur est linéaire.

PREUVE – lors du parcours, chaque sommet entre au plus une fois dans la file d'attente car seuls les sommets blancs peuvent être enfilés et une fois enfilés, ils sont marqués et ne redeviennent jamais plus blancs. Par conséquent, le nombre d'étapes de la boucle `while` est majoré par $|S|$. Examinons le contenu de cette boucle :

- on défile un sommet et affiche un message à chaque passage, ce qui donne un nombre d'opérations en $\mathcal{O}(|S|)$;
- on a une boucle `for` de même longueur que la liste d'adjacence du sommet défilé, qui contient un nombre fini et fixé d'opérations. Comme la somme des longueurs des listes d'adjacence est $|A|$, le coût total de ces boucles est ainsi en $\mathcal{O}(|A|)$.

L'initialisation ayant un coût en $\mathcal{O}(|S|)$, la complexité totale de notre algorithme est donc en $\mathcal{O}(|S| + |A|)$: on a bien affaire à une complexité linéaire.

Théorème

Reprenons les listes `peres` et `dist` renvoyées par le parcours en largeur.

- L'algorithme découvre tous les sommets du graphe accessibles depuis s .
- Pour un tel sommet v de S :
 - le nombre `dist[v]` est la longueur d'arc minimale d'un chemin de s vers v
 - le sommet `peres[v]` est l'antécédent de v sur un plus court chemin de s vers v

PREUVE – le premier point découle directement de la définition du parcours en largeur. La démonstration complète et rigoureuse du second point nécessite de nombreux lemmes, donc je vous l'épargne. Pour faire simple : lors du parcours en largeur, on explore tous les voisins d'un sommet avant de passer aux descendants de ceux-ci, si bien que les distances depuis s croissent en cours d'algorithme. De ce fait, lorsque l'on découvre un nouveau sommet v , c'est nécessairement à l'aide d'un plus court chemin de s vers v .

3.4) Applications du parcours en largeur

(a) Plus court chemin d'un sommet à un autre

Il suffit de procéder comme avec le parcours en profondeur : en effet, les chemins reliant deux sommets découverts lors du parcours en largeur sont nécessairement des plus courts chemins du point de vue de la longueur d'arc (cf. théorème ci-dessus).

On va ainsi recycler dans l'exercice 7 les procédures écrites dans l'exercice 3.

(b) Composantes connexes

C'est le même principe qu'avec le parcours en profondeur : un parcours d'un graphe non-orienté depuis le sommet s , qu'il soit effectué en largeur ou en profondeur, permet d'explorer toute la composante connexe de celui-ci.

On va donc adapter dans l'exercice 8 les procédures écrites lors de l'exercice 4.

4/ Plus courts chemins sur un graphe pondéré**4.1) Problématique**

Un individu doit se déplacer par un réseau de rues ou de routes pour aller d'un point à un autre : par où doit-il passer pour effectuer le trajet le plus court ou le plus rapide ?

Un tel problème se modélise très bien à l'aide d'un graphe pondéré : les routes sont représentées par les arêtes du graphe, les sommets correspondent à leurs points d'intersection et les distances ou les temps de parcours entre ceux-ci font office de pondération. De plus, le poids d'un chemin donné est défini naturellement comme la somme de tous les poids des arêtes qui le constituent.

Le problème précédent va alors s'énoncer comme ceci : étant donnés deux sommets du graphe, on cherche à déterminer un chemin de poids minimal les reliant.

Pour des raisons de complexité, il est bien entendu hors de question de calculer les poids de tous les chemins possibles entre les deux sommets. On a vu par contre qu'un parcours en largeur d'un graphe permet d'obtenir efficacement la longueur d'arc minimale vers tout sommet accessible. Malheureusement, cette information n'a plus aucun intérêt dès lors que les arêtes ont des poids différents, puisque le chemin le plus court n'est pas forcément celui comportant le moins d'arêtes.

4.2) Solutions algorithmiques

Cette année, nous allons voir comment calculer un plus court chemin d'un sommet donné vers chaque sommet du graphe. On peut citer deux algorithmes efficaces pour cela : l'algorithme de Bellman-Ford et celui de Dijkstra. Le premier a le mérite de fonctionner avec des poids négatifs contrairement au second, qui a en revanche une meilleure complexité.

Attention en présence de poids négatifs : rajouter une même constante à tous les poids pour les rendre positifs n'est pas une solution, car les longueurs d'arcs vont venir s'ajouter aux longueurs des chemins et fausser ainsi la recherche des plus courts chemins !!!

Dans les deux cas, on commence par attribuer une valeur par défaut aux distances entre le sommet d'origine s et les autres sommets, nulle pour s et infinie pour les autres. Ces valeurs forment la liste `dist` et sont mises à jour au cours du déroulement de l'algorithme. Elles ne font que décroître et sont égales aux distances minimales en fin d'exécution.

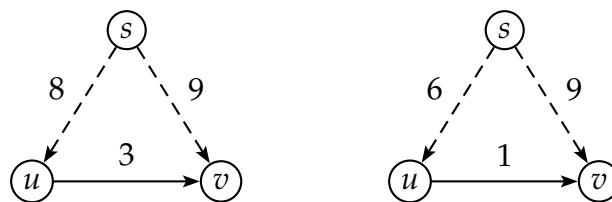
En parallèle, on établit une liste `peres` de relations père / fils comme pour les parcours en profondeur et en largeur afin de reconstituer les plus courts chemins après l'exploration.

La technique utilisée pour la mise à jour des distances porte le nom de *relâchement*. On parle aussi de *relaxation*. Concrètement, il s'agit pour un arc (u, v) d'examiner si l'on peut améliorer la distance actuelle entre s et v en passant par u . On note pour cela $d(\cdot, \cdot)$ la distance entre deux sommets et $p(\cdot, \cdot)$ la longueur d'un arc. Si l'on a

$$d(s, u) + p(u, v) < d(s, v)$$

on remplace le chemin $s \rightarrow v$ par le chemin $s \rightarrow u \rightarrow v$. Le relâchement de l'arc (u, v) consiste ainsi à remplacer $d(s, v)$ par $d(s, u) + p(u, v)$ et le père de v par u .

Exemple : considérons les deux situations ci-dessous, où les lignes en pointillés désignent des chemins et pas seulement des arcs.



Dans le cas de droite, on améliore le chemin de s à v en passant par u car le nouveau chemin $s \rightarrow u \rightarrow v$ est de poids 7, contre 9 pour le chemin $s \rightarrow v$.

La différence essentielle entre les algorithmes de Bellman-Ford et de Dijkstra est le nombre de relâchements effectués : le premier relâche n fois chaque arc (n étant le nombre de sommets) tandis que le second relâche une seule fois chaque arc, dans un ordre judicieux. Nous allons maintenant étudier en détail cet algorithme, qui est le seul au programme.

4.3) Algorithme de Dijkstra

Son fonctionnement est assez simple. Soit s le sommet d'origine. On va considérer un ensemble E de sommets dont les distances minimales à s ne sont pas encore calculées.

- ❶ Initialement, cet ensemble est égal à l'ensemble de tous les sommets.
- ❷ A chaque étape, on extrait de E le sommet u dont la distance à s est minimale, puis on relâche tous les arcs issus de u et à destination dans E .
- ❸ On s'arrête lorsque E est vide.

Remarque : pour le second point, les autres arcs issus de u mènent à des sommets plus proches de s que u puisqu'ils ont déjà été extraits de E . Par conséquent, ils ne feront pas partie d'un plus court chemin issu de s et passant par u , donc il est inutile de les examiner.

On illustrera graphiquement cet algorithme de la manière suivante :

- les sommets sont coloriés en noir au fur et à mesure de l'extraction
- on colorie en gris le sommet réalisant la distance minimale parmi les sommets de E
- les arcs relâchés sont hachurés, et l'on surligne les arcs donnant les relations père / fils

Ceci permet de visualiser les plus courts chemins issus de s une fois l'exploration terminée.

Pour l'implémentation en Python, on va écrire trois procédures :

- ❶ la procédure auxiliaire `extraction(E,dist)` renvoie le sommet `u` de `E` réalisant la distance minimale au sommet `s` et l'extrait de la liste `E` ;
- ❷ la procédure auxiliaire `relaxation(u,v,p,dist,peres)` relâche l'arc pondéré `(u,v,p)` et effectue les mises à jour nécessaire des listes `dist` et `peres` ;
- ❸ la procédure principale `Dijkstra(L,s)` initialise les variables `E`, `dist` et `peres`, puis lance la boucle d'extractions / relaxations successives.

```
I=numpy.inf

def extraction(E,dist):
    u=E[0]
    for i in E:
        if dist[i]<dist[u]:
            u=i
    E.remove(u)
    return u

def relaxation(u,v,p,dist,peres):
    d=dist[u]+p
    if d<dist[v]:
        dist[v]=d
        peres[v]=u

def Dijkstra(L,s):
    n=len(L)
    E=list(range(n))
    dist=[I for i in range(n)]
    dist[s]=0
    peres=[None for i in range(n)]

    while E!=[]:
        u=extraction(E,dist)
        if dist[u]==I:
            break
        for v,p in L[u]:
            relaxation(u,v,p,dist,peres)
    return dist, peres
```

Application : recherche d'un plus court chemin

Après exécution de l'algorithme de Dijkstra, on reconstitue facilement les plus courts chemins reliant `s` aux sommets atteints à l'aide la liste `peres`. Leurs longueurs respectives seront données par la liste `dist`. C'est l'objet de l'exercice 10.

4.4) Propriétés de l'algorithme de Dijkstra

Soient $G = (S, A)$ un graphe pondéré et $s \in S$ le sommet à partir duquel on applique cet algorithme. On note $n = |S|$ et $\delta(s, u)$ le poids minimal d'un chemin de s à $u \in S$.

Proposition

La complexité de l'algorithme de Dijkstra est quadratique.

PREUVE – la phase d'initialisation est de coût $\mathcal{O}(n)$. Examinons ensuite la boucle principale : à chaque étape, on effectue une extraction suivie du relâchement de tous les arcs issus du sommet extrait. Chaque sommet étant extrait une et une seule fois, le nombre total de relâchement est égal au nombre d'arcs du graphe. Ensuite :

- la procédure **extraction** a un coût en $\mathcal{O}(|E|)$ à cause de la boucle **for** et de la fonction **remove**, donc le coût total des extractions est $\mathcal{O}(n + n - 1 + n - 2 + \dots + 1)$ soit $\mathcal{O}(n^2)$;
- la procédure **relaxation** contient un nombre fini et fixé d'opérations, donc elle a un coût constant et le coût total des relâchements est $\mathcal{O}(|A|)$.

La complexité globale de l'algorithme est ainsi de $\mathcal{O}(n^2 + |A|)$, soit $\mathcal{O}(n^2)$ puisque le nombre d'arcs $|A|$ est majoré par $n(n - 1)$: elle est bien quadratique.

Proposition

Pour tout arc $(u, v) \in A$ de poids $p(u, v)$, on a

$$\delta(s, v) \leq \delta(s, u) + p(u, v)$$

avec égalité ssi il existe un plus court chemin de s vers v passant par u et l'arc (u, v) .

Théorème

Lors du déroulement de l'algorithme de Dijkstra, les propriétés suivantes sont vérifiées.

- ❶ Pour tout sommet u , on a $\text{dist}[u] \geq \delta(s, u)$.
En cas d'égalité, la valeur de $\text{dist}[u]$ ne change plus.
- ❷ S'il n'y a pas de chemin reliant s au sommet u , alors $\text{dist}[u] = +\infty$ en permanence.
- ❸ Supposons qu'il existe un plus court chemin de s vers v finissant par l'arc (u, v) . Si l'on a $\text{dist}[u] = \delta(s, u)$ à un moment antérieur au relâchement de l'arc (u, v) , alors $\text{dist}[v] = \delta(s, v)$ à tout moment postérieur à ce relâchement.
- ❹ Soit un plus court chemin de $s = v_0$ à $v = v_k$ constitué des arcs $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$. Si les arcs sont relâchés dans cet ordre, alors $\text{dist}[v] = \delta(s, v)$ après ces relâchements, indépendamment de tous les autres relâchements susceptibles de se produire.

PREUVE – on va justifier les propriétés les unes après les autres.

- 1) Par définition de $\delta(s, u)$, on a $\text{dist}[u] \geq \delta(s, u)$ à tout instant. De plus, les relâchements ne pouvant que faire baisser $\text{dist}[u]$, celle-ci reste constante dès qu'elle atteint $\delta(s, u)$.
- 2) C'est une conséquence du premier point, puisqu'alors on a $\delta(s, u) = +\infty$.

- 3) Supposons que $\text{dist}[u] = \delta(s, u)$ à un moment antérieur au relâchement de l'arc (u, v) : d'après la propriété ❶, on a $\text{dist}[u] = \delta(s, u)$ à partir de ce moment, et notamment lors du relâchement de l'arc (u, v) . Comme il existe un plus court chemin de s vers v passant par u , il commence par un plus court chemin de s vers u et l'on obtient alors

$$\text{dist}[u] + p(u, v) = \delta(s, u) + p(u, v) = \delta(s, v)$$

si bien que $\text{dist}[v] = \delta(s, v)$ à partir du relâchement.

- 4) On procède par récurrence sur l'entier $k \in \mathbb{N}^*$. La propriété est évidente pour $k = 1$. Supposons qu'elle est vraie au rang k et considérons un plus court chemin de v_0 à v_{k+1} constitué des arcs (v_i, v_{i+1}) avec $i \in \llbracket 0; k \rrbracket$. Les k premiers arcs constituent un plus court chemin de s vers v_k donc $\text{dist}[v_k] = \delta(s, v_k)$ après le relâchement de l'arc (v_{k-1}, v_k) , par hypothèse de récurrence. On déduit de la propriété ❸ que $\text{dist}[v_{k+1}] = \delta(s, v_{k+1})$ après le relâchement de l'arc (v_k, v_{k+1}) , ce qui prouve que la propriété est vraie au rang $k + 1$.

Théorème

À la fin de l'exécution de l'algorithme de Dijkstra, on a $\text{dist}[u] = \delta(s, u)$ pour tout sommet u .

PREUVE – la boucle **while** possède l'**invariant** suivant

« Au début de chaque itération, $\text{dist}[u] = \delta(s, u)$ pour tout sommet $u \notin E$ »

Voici une justification rapide : au moment où l'on extrait un sommet u de E , le réel $\text{dist}[u]$ est la longueur d'un plus court chemin de s vers u passant par des sommets n'appartenant plus à E . Comme u est le sommet de E à distance minimale de s , les chemins passant par d'autres sommets de E sont encore plus longs, donc on a bien $\text{dist}[u] = \delta(s, u)$.

4.5) L'algorithme A*

Idée générale

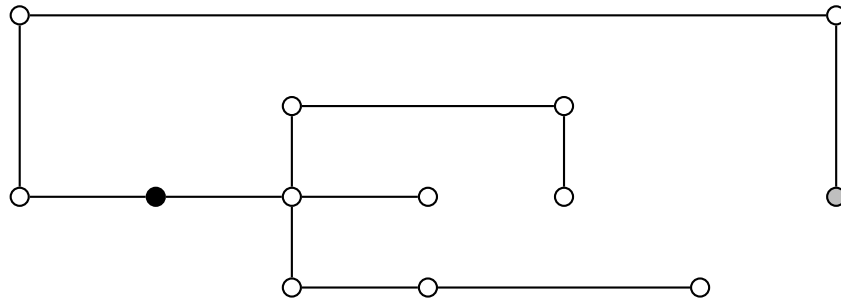
L'algorithme de Dijkstra permet de trouver le plus court chemin entre deux sommets s_1 et s_2 en déterminant les plus courts chemins de s_1 vers tous les autres sommets du graphe. On peut noter qu'il effectue une recherche similaire à un parcours en largeur : il calcule les plus courtes distances vers les sommets les plus proches en premier temps, puis de proche en proche jusqu'aux sommets les plus éloignés.

En pratique, le calcul de la plus courte distance vers un sommet qui n'est pas sur un chemin menant à s_2 est une perte de temps. Le problème est bien entendu que l'on ne sait pas à l'avance quels sommets sont sur un chemin qui conduit à s_2 . Une idée pour améliorer l'algorithme consiste alors à privilégier les sommets susceptibles d'être « dans la bonne direction » lors de la recherche du plus court chemin.

Prenons l'exemple du déplacement sur une carte routière : si le sommet s_2 est situé au nord du sommet s_1 , on va examiner en priorité les sommets qui se situent au nord de s_1 . Même si certains d'entre eux peuvent mener à une impasse, on peut espérer que l'on trouvera de la sorte plus rapidement un chemin vers s_2 , voire bien souvent un itinéraire optimal.

On appelle ainsi **heuristique** une méthode de calcul qui fournit rapidement une solution à un problème, mais sans pour autant garantir son optimalité.

Bien évidemment, l'efficacité de l'heuristique dépend fortement du graphe. Considérons par exemple le graphe ci-dessous. Si l'on effectue une recherche de plus court chemin du sommet noir vers le sommet gris, tous les chemins qui démarrent par un déplacement dans la bonne direction mènent à une impasse. Il faut contraire partir dans la direction opposée pour trouver le seul chemin, et l'heuristique proposée ne produit pas un meilleur résultat.



Définition

L'algorithme A^* est une amélioration de l'algorithme de Dijkstra dans laquelle on explore en priorité les sommets pour lesquels la somme de la distance déjà parcourue depuis s_1 et de la distance les séparant (à vol d'oiseau) de s_2 est minimale.

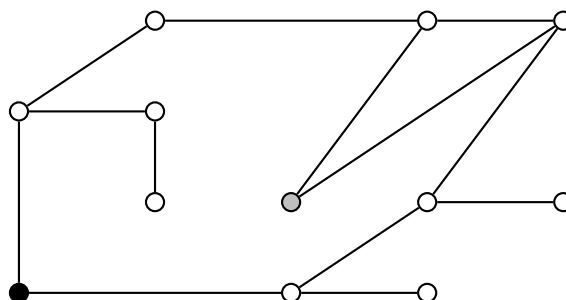
Implémentation

On va appliquer l'algorithme A^* à un graphe dont les sommets sont des points de $\mathbb{R}^2$. Pour cette raison, il vaut mieux implémenter la liste d'adjacence à l'aide d'un dictionnaire, comme on l'a vu à la fin du chapitre précédent. Le poids d'une arête sera simplement la distance euclidienne entre ses deux extrémités.

Le programme principal prend en argument le graphe D , le sommet origine s et le sommet destination c . Il utilise encore la liste E des sommets dont les distances minimales à s ne sont pas encore calculée, ainsi que trois dictionnaires :

- `dist` donne les distances temporaires au sommet s ;
- `peres` donne le père de chaque élément sur un plus court chemin depuis le sommet s ;
- `prio` donne la priorité associée à chaque sommet `sommet` ; `prio[u]` est simplement la somme de `dist[u]` et de la distance entre u et c .

Il suffit ensuite de réécrire les trois procédures `extraction`, `relaxation` et `Dijkstra`, en les adaptant à des dictionnaires et en tenant compte de `prio`.



Exercices

Exercice 1.

Soient a et b deux sommets appartenant à la même composante fortement connexe d'un graphe orienté G .

Montrer que tous les sommets d'un chemin reliant le sommet a au sommet b font encore partie de la même composante fortement connexe du graphe G .

Exercice 2.

Même exercice avec les composantes connexes d'un graphe non orienté.

Exercice 3.

Créer une fonction `chemin(L, a, b)` qui utilise le parcours en profondeur (et en particulier la liste `peres`) depuis un sommet a pour déterminer un chemin le reliant au sommet b .

Penser aux listes de type `deque` et à la fonction `appendleft`

Pour les tests, on utilisera les graphes n°1 à n°4 du chapitre précédent.

Exercice 4.

Modifier le parcours en profondeur pour détecter la présence de cycles dans un graphe non orienté. On en affichera le plus possible.

Voici la démarche : on lance un parcours depuis le sommet s ; si l'on peut atteindre s via un sommet qui n'est pas un de ses descendants, on tient un cycle. On peut alors le reconstituer en s'inspirant de l'exercice précédent.

Exercice 5.

Utiliser le parcours en profondeur pour déterminer les composantes connexes d'un graphe non-orienté. On écrira les deux procédures `origine(L, s, visites)` et `composantes(L)` qui sont décrites dans le cours.

Tester les deux dernières procédures sur les graphes n°1 et n°3.

Exercice 6.

Utiliser la « méthode naïve » décrite en cours pour déterminer les composantes fortement connexes d'un graphe orienté. On écrira pour cela quatre procédures :

- `origine(L, s, visitesF)` qui réalise la première étape et renvoie la liste `chemin`
- `transpose(L)` qui renvoie le graphe transposé
- `retour(LT, s, visitesF, chemin)` qui réalise la seconde étape
- `composantesF(L)` qui initialise la liste `visitesF` et lance l'exploration sur les sommets non marqués pour obtenir une à une les composantes fortement connexes. C'est la procédure principale.

Pour les tests, on utilisera les graphes n°2 et n°4.

Exercice 7.

Même chose avec la « méthode experte ». On écrit quatre procédures :

- `aller(L)` qui effectue un parcours en profondeur complet du graphe et qui ajoute les sommets au début d'une deque nommée `finis` lorsque leur traitement est terminé. On obtient la liste des sommets rangés par ordre décroissant de date de fin de traitement
- `transpose(L)` qui renvoie le graphe transposé
- `retour(LT,s,visites)` qui réalise un parcours en profondeur du graphe transposé depuis le sommet `s` et renvoie la liste des sommets croisés
- `composantesF(L)` qui initialise la liste `visites`, lance la procédure `aller` et récupère la liste `finis`, puis calcule le graphe transposé et lance la procédure `retour` sur les sommets non marqués de `finis` pour obtenir les composantes fortement connexes. C'est la procédure principale.

Exercice 8.

Créer une fonction `pluscourtchemin(L,a,b)` qui utilise le parcours en largeur depuis un sommet `a` pour déterminer un plus court chemin le reliant au sommet `b`.

Pour les tests, on utilisera les graphes n°1 à n°4 du chapitre précédent.

Exercice 9.

Modifier le parcours en largeur pour détecter la présence de cycles dans un graphe non orienté. On en affichera le plus possible.

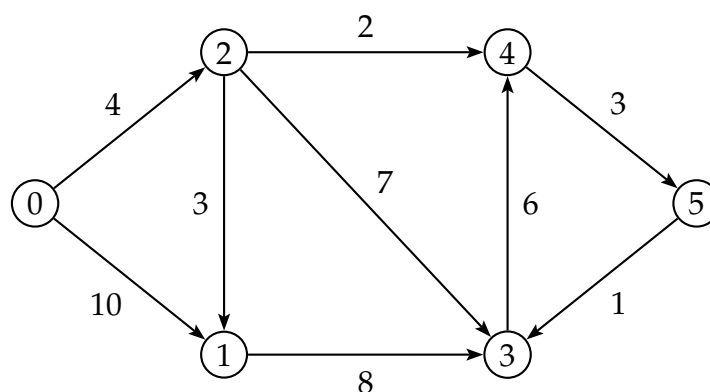
Exercice 10.

Utiliser le parcours en largeur pour déterminer les composantes connexes d'un graphe.

Tester ces deux dernières procédures sur les graphes n°1 et n°3.

Exercice 11.

Appliquer l'algorithme de Dijkstra à la main sur le graphe pondéré suivant :

**Exercice 12.**

Créer une fonction `pluscourtchemin_D(L,a,b)` qui utilise l'algorithme de Dijkstra pour déterminer le plus court chemin reliant deux sommets `a` et `b` d'un graphe pondéré `L`.

Pour les tests, on utilisera les graphes n°5 et n°6 du chapitre précédent.