Thème info 5 – Quelques algorithmes sur les graphes

La théorie des graphes a toujours été un domaine essentiel de l'informatique théorique et elle a pris une importance considérable depuis la multiplication des réseaux de toutes sortes dans le monde "moderne". Outre les réseaux numériques, les graphes servent aussi à modéliser les réseaux de transport (problèmes de plus court chemin, du voyageur de commerce...), l'ordonnancement de tâches (programmation, productique, logistique, recette de cuisine...), etc.

I - Définitions et notations

1) Graphes non orientés

Un *graphe non orienté* est un couple $G = (S, A)$ où S est un ensemble fini non vide, *l'ensemble des sommets de* G, et A un ensemble de **paires** d'éléments de S, *l'ensemble des arêtes de* G.
Si $\{u, v\} \in A$ est une arête de G, on dit que *les sommets* u *et* v *sont adjacents* (nécessairement $u \neq v$). L'arête $\{u, v\}$ est souvent notée u — v.

Un *chaîne* de *longueur* ℓ reliant deux sommets u et v dans G est une séquence de sommets $[v_0, \dots v_\ell]$ telle que $v_0 = u$, $v_\ell = v$ et, pour tout $k \in [\![1, \ell]\!]$, v_{k-1} — v_k est une arête de G. La longueur ℓ est ainsi le nombre d'arêtes dans la chaîne.

Un graphe non orienté est dit *connexe* si et seulement si deux sommets quelconques sont reliés par une chaîne.

2) Graphes orientés

Un *graphe orienté* est un couple $G = (S, A)$ où S est un ensemble fini non vide, *l'ensemble des sommets de* G, et A un ensemble de **couples** d'éléments de S, *l'ensemble des arcs de* G.
Si $(u, v) \in A$ est un arc de G, on dit qu'*il part du sommet* u et qu'*il arrive au sommet* v. On dit alors que *le sommet* v *est adjacent au sommet* u (la relation d'adjacence n'est *a priori* pas symétrique ici). L'arc (u, v) est souvent noté $u \to v$.

On remarquera qu'il est possible d'avoir des arcs de la forme (u, u), appelés *boucles*.

NB : par définition d'un ensemble, un couple (u, v) donné d'éléments de S apparaît au plus une fois dans A. On peut définir des structures de données similaires avec la possibilité d'avoir plusieurs arcs de u vers v, ce sont les *multigraphes*, que nous n'étudierons pas ici.

Un *chemin* de *longueur* ℓ d'un sommet u à un sommet v dans G est une séquence de sommets $[v_0, \dots v_\ell]$ telle que $v_0 = u$, $v_\ell = v$ et, pour tout $k \in [\![1, \ell]\!]$, $v_{k-1} \to v_k$ est un arc de G. La longueur ℓ est ainsi le nombre d'arcs dans le chemin. S'il existe un chemin de u à v on dit que v *est accessible à partir de* u.
Un graphe orienté est dit *fortement connexe* si et seulement si tout sommet est accessible à partir de n'importe quel autre.

3) Graphes valués

Il est entendu que chaque sommet d'un graphe peut-être accompagné d'une information, ne serait-ce que son nom (nom d'une ville dans un réseau routier, adresse IP d'une machine sur Internet)... On peut aussi utiliser une application de S dans un ensemble quelconque pour associer des données à chaque sommet.

De même, on parle de *graphe valué* lorsqu'on dispose d'une application de A dans un ensemble E, qui à chaque arête (ou arc) associe une valeur (distance entre deux villes, temps du trajet d'une ville vers une autre...).

II - Représentations d'un graphe

Pour simplifier les schémas et faciliter l'implémentation en Python, nous considérerons ici que $S = \{0, \dots, n-1\}$, ce qui permettra de représenter une application de S dans E par un simple tableau (ou une "liste Python") de n éléments de E.

1) Représentations graphiques

Comme pour les arbres (qui sont des cas particuliers de graphes !), on utilise très souvent un dessin représentant les sommets du graphes reliés ou pas par un trait (*resp.* une flèche) pour un graphe non orienté (*resp.* orienté) selon que l'arête (*resp.* l'arc) existe ou pas dans A. C'est un support précieux pour visualiser les propriétés du graphe, la mise en œuvre d'un algorithme...

2) Représentations en mémoire

On commence par numéroter les sommets du graphe, disons de 0 à $n-1$. Désormais chaque sommet sera identifié à son numéro. Il reste à stocker en mémoire les arêtes (ou les arcs). On pourrait bien sûr créer une liste de paires (ou de couples), mais on utilise de préférence l'une des deux méthodes suivantes, qui permettent un parcours rapide de l'ensemble des sommets adjacents à un sommet donné (tâche omniprésente comme nous le verrons).

a) Matrice d'adjacence

On stocke ici les données contenues dans A dans un tableau M de type (n, n) selon la convention suivante :

- $M[u, v] = 0$ si l'arête u — v (*resp.* l'arc $u \to v$) est absente de A ;
- $M[u, v] = 1$ si l'arête u — v (*resp.* l'arc $u \to v$) est présente dans A.

NB : pour un graphe non orienté, la matrice est symétrique ! On peut aussi utiliser une matrice de booléens, mais l'affichage prend plus de place...

L'ensemble des sommets adjacents au sommet i s'obtient en parcourant la ligne i de M.

Inconvénient : l'encombrement en mémoire, lorsque le cardinal de A est "très inférieur" à n^2.

Avantages : test en temps constant de l'adjacence de deux sommets donnés ; pour un graphe valué, possibilité de mémoriser dans $M[u, v]$ une valeur associée à l'arête (ou l'arc) correspondant (dans ce cas, prendre garde à choisir une valeur "impossible" pour indiquer l'absence de l'arête ou de l'arc ! Par exemple -1 si les valeurs sont positives, ou la valeur `inf` de Python, correspondant à $+\infty$, à importer du module `numpy`).

b) Listes d'adjacence

On stocke ici une liste L de n listes, indexées par les numéros de 0 à $n-1$ des sommets de G, $L[u]$ étant une liste contenant les numéros des sommets adjacents à u. Les avantages et inconvénients sont symétriques des précédents.

Avantage : l'encombrement en mémoire, lorsque le cardinal de A est "très inférieur" à n^2 !

Inconvénients : pour tester l'adjacence de deux sommets donnés, il faut parcourir une liste (l'utilisation avec Python du test `v in L[u]` est certes simple et efficace, mais ne s'exécute pas en temps constant !) ; pour un graphe valué, il faut compliquer la structure afin de mémoriser la valeur associée à une arête (ou un arc) ; on peut par exemple stocker dans $L[u]$ des couples de la forme (v, V) où v est un sommet adjacent à u et V la valeur associée à l'arête (ou l'arc) correspondant. Dans ce cas, L est une liste de listes de couples... On peut aussi stocker les valeurs dans une autre structure (tableau, dictionnaire...).

3) Création et conversion

Nous gardons $S = \{0, \dots, n-1\}$, ainsi le graphe G est entièrement connu par la donnée de n et de A, que nous supposerons fourni sous forme d'une liste de couples (u, v) (ou de listes $[u, v]$).

Rappel : dans Python, contrairement aux **listes**, les **couples** sont *non-mutables* (une fois qu'on les a créés, on ne peut pas modifier les valeurs qu'ils contiennent) ; dans ce contexte ce n'est pas gênant, au contraire...

Programmer les fonctions suivantes :

1) `GtoM(n,A)` qui renvoie la matrice d'adjacence M (tableau `numpy`) décrivant G

2) `GtoL(n,A)` qui renvoie la liste L des listes d'adjacence décrivant G

3) `MtoL(M)` qui renvoie la liste des listes d'adjacence à partir de la matrice d'adjacence M

4) `LtoM(L)` qui renvoie la matrice d'adjacence à partir de la liste des listes d'adjacence.

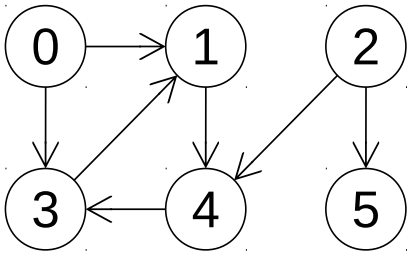
(On notera que n s'obtient par `len(M)` *ou* `len(L)`*.)*

On peut adapter si besoin ces fonctions au cas des graphes valués, mais ce ne sera pas utile ici.

4) Premiers exemples

A se lit sur la représentation graphique, on donne au-dessous M et L.

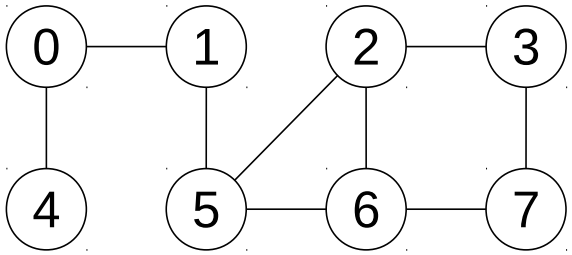
$G1$ (orienté)



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

$$L = \left[[1,3]\,,[4]\,,[4,5]\,,[1]\,,[3]\,,[\,]\right]$$

$G2$ (non orienté)



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

$$L = \left[[1,4]\,,[0,5]\,,[3,5,6]\,,[2,7]\,,[0]\,,[1,2,6]\,,[2,5,7]\,,[3,6]\right]$$

5) Parcours d'un graphe

Il est assez simple de visiter (récursivement !) tous les nœuds d'un arbre. Le parcours de tous les sommets d'un graphe (en restant sur des chemins !) est une question plus délicate, dans la mesure ou il n'est pas évident de choisir un "sens de parcours".

Les sections suivantes détaillent les deux types de parcours les plus usuels, *en profondeur* et *en largeur*.

III - Parcours en profondeur – Tri topologique

1) Parcours en profondeur d'un graphe

L'idée du parcours en profondeur est de descendre, à partir de chaque sommet non encore visité, le plus profondément possible en enchaînant les arcs autant que l'on peut. Nous traiterons ici l'exemple d'un graphe orienté, mais l'algorithme fonctionne également pour un graphe non orienté.

a) Principe de l'algorithme

Nous allons utiliser une variable *instant* qui permettra de dater les événements successifs se produisant durant le parcours.

Chaque sommet u aura une *couleur*, stockée dans un tableau c : $c[u]$ sera la couleur du sommet u.

Nous associerons deux autres valeurs à chaque sommet u : $d[u]$, instant de début du traitement de u, et $f[u]$, instant de fin de traitement de u.

- Au début *instant* vaut 1 et tous les sommets sont *blancs*.
- Lorsqu'un sommet est *découvert* (la première fois qu'on explore un arc partant de u), il devient *gris*, $d[u]$ prend la valeur *instant* (et *instant* est incrémenté).
- Lorsque le traitement d'un sommet est terminé (quand on a exploré tous les arcs partant de u), il devient *noir* et $f[u]$ prend la valeur *instant* (et *instant* est incrémenté).

Après l'initialisation, nous balayons l'ensemble des sommets et nous *visitons* les sommets qui sont encore blancs, la visite du sommet u consistant à visiter récursivement tous les sommets adjacents à u.

b) Formalisation en pseudo-code

Nous donnons une version à partir de la liste L des listes d'adjacence, l'adaptation au cas où c'est la matrice d'adjacence qui est donnée est simple (même sans utiliser la fonction `LtoM` !).

PP(L)
  initialiser les couleurs à *blanc*, d et f à 0
  $instant \leftarrow 1$
  pour chaque sommet u
    si $c[u] = blanc$ alors Visiter(u)

où la fonction Visiter est définie par :

Visiter(u)
  $c[u] \leftarrow gris$
  $d[u] \leftarrow instant : Inc(instant)$
  pour chaque v adjacent à u
    si $c[v] = blanc$ alors Visiter(v)
  $c[u] \leftarrow noir$
  $f[u] \leftarrow instant : Inc(instant)$

c) Programmation en Python

Traduire en Python l'algorithme décrit ci-dessus. Penser à définir `Visiter` **à l'intérieur** de `PP`, de sorte que les tableaux c, d, f soient visibles depuis la fonction `Visiter`. Pour la variable `instant`, comme elle doit être modifiée au sein de la fonction `Visiter`, il faut la déclarer `nonlocal` dans la fonction `Visiter` (sinon on déclare tout au niveau global, mais c'est moins élégant). Pour cela, insérer comme première ligne de la fonction `Visiter` la commande `nonlocal instant`.

On pourrait penser à déclarer aussi `c`, `d`, `f` comme `nonlocal`, mais ce n'est pas nécessaire car ce ne sont pas les variabes `c`, `d`, `f` qui sont modifiées (qui contiennent les **adresses** des listes), seulement certaines des valeurs contenues dans lesdites listes.

En guise de sortie, on pourra imprimer les tableaux d et f.

Un tel parcours sert souvent à appliquer un traitement donné à chaque sommet, soit au moment où il devient gris, soit au moment où il devient noir (cf. les parcours préfixes et postfixes d'un arbre).

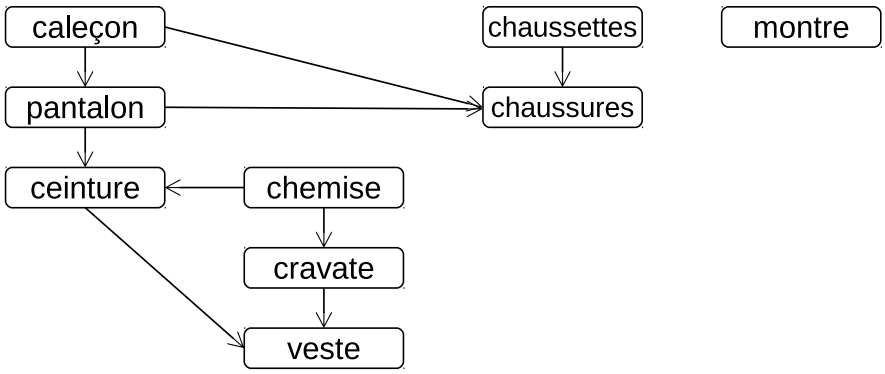
2) Tri topologique d'un graphe orienté

Une application classique du parcours en profondeur est le *tri topologique* de l'ensemble des sommets d'un graphe orienté, qui consiste à lister les sommets, disons de gauche à droite, de sorte que, pour tout sommet u, les sommets accessibles depuis u par un chemin dans le graphe se trouvent à droite de u. Cet algorithme est fondamental pour le problème de l'ordonnancement des tâches : si l'existence d'un arc $u \to v$ dans le graphe implique que la tâche associée à v soit effectuée **après** celle affectée à u, le tri topologique fournira un ordre chronologique possible d'exécution des tâches.

Une condition nécessaire pour qu'il existe une solution est que le graphe soit *acyclique* (*i.e.* qu'il n'existe pas de chemin fermé). On peut montrer que cette condition est suffisante et que le parcours en profondeur fournit une solution : il suffit de classer les sommets selon les instants de fin de traitement décroissants.

Adapter le programme précédent pour afficher une solution au tri topologique. Pour éviter un tri *a posteriori* du tableau f, il suffit, en partant d'une pile vide, d'empiler u au moment où il devient noir et d'afficher le contenu de la pile à la fin du parcours. En effet chaque sommet est "visité" exactement une fois et *instant* va croissant.

Tester sur l'exemple de l'habillage du savant Cosinus :



IV - Parcours en largeur d'un graphe

L'idée est ici de partir d'un sommet donné s et de visiter les sommets du graphe en commençant par les sommets adjacents à s, puis en s'éloignant progressivement de s (sommets accessibles via un chemin de longueur 2, puis 3, etc.). Par essence même, ce parcours n'atteindra que les sommets accessibles à partir de s.

1) Principe de l'algorithme

Nous allons encore colorier les sommets, successivement en *blanc* (sommet non encore "découvert"), *gris* (sommet découvert en cours de traitement, c'est-à-dire que ses sommets adjacents n'ont pas tous été traités) puis *noir* (traitement terminé). Nous tiendrons donc à jour un tableau c comme précédemment.

Nous associerons également deux valeurs à chaque sommet u (accessible depuis s !) : $d[u]$ contiendra *la distance de s à u* (*i.e.* la longueur d'un plus court chemin de s à u) et $p[u]$ contiendra *un père de u* (un sommet précédant u sur un plus court chemin de s à u). Ainsi la remontée "de père en père" à partir de u permettra de remonter jusqu'à s le long d'un plus court chemin.

L'idée même du parcours en largeur étant que les sommets sont découverts dans l'ordre de leur distance à s, il n'est pas question d'appels récursifs comme dans le parcours en profondeur. Dès qu'un sommet v est découvert à partir d'un sommet u pour lequel on connaît $d[u]$ (par hypothèses !), on connaît $d[v] = d[u] + 1$ et v devient gris. Pour ne pas oublier de revenir à v pour explorer sa liste d'ajacence, on le stocke dans une *file d'attente* F (cf. T.D. 3) qui contiendra en permanence les sommets gris. **Il faut** utiliser une structure FIFO pour être sûr de repartir d'abord des sommets les moins éloignés, qui auront été ajoutés les premiers dans la file d'attente.

- Tous les $c[u]$ sont initialisés à *blanc*, les $d[u]$ à ∞ et les $p[u]$ à -1 ; F est initialisée à la file vide.
- Au début du traitement, $c[s]$ devient *gris*, s est ajouté à F, $d[s]$ prend la valeur 0 ($p[s]$ reste à -1 !)
- On traite les sommets gris présents dans F en explorant leurs sommets adjacents.

2) Formalisation en pseudo-code

Là encore, donnons une version à partir de la liste L des listes d'adjacence.

PL(L, s)
  initialiser les couleurs à *blanc*, d à ∞, p à -1 et F à la file vide
  $c[s] \leftarrow gris$: $d[s] \leftarrow 0$: ajouter s à F
  tant que F n'est pas vide
    $u \leftarrow suivant(F)$
    pour chaque v adjacent à u
      si $c[v] = blanc$ alors
        $c[v] \leftarrow gris$: ajouter v à F
        $d[v] \leftarrow d[u] + 1$: $p[v] \leftarrow u$
    $c[u] \leftarrow noir$

3) Programmation en Python

Traduire en Python l'algorithme décrit ci-dessus par une fonction `PP(L,s)`.

Penser à :

- `from numpy import inf` pour pouvoir utiliser le ∞ de `numpy`
- `from files import *` pour importer les primitives du fichier `files.py` (copiez-le du dossier commun dans votre dossier de travail, ou bien réutilisez votre travail du T.D. 3).

En guise de sortie, on pourra imprimer les tableaux d et p.

Dans le cas d'un graphe non orienté non connexe, les valeurs finies dans d correspondent à la composante connexe de s. On peut adapter le programme pour déterminer si besoin les autres composantes connexes.

V - Algorithme de Dijkstra (prononcer "deil-kstra")

On se donne ici un **graphe orienté valué**, c'est-à-dire qu'à chaque arc $u \to v$ est associé une valeur $V(u, v)$ (typiquement une durée, une distance, un prix...). Le *coût* d'un chemin dans le graphe est alors par définition la somme des valeurs des arcs constituant ledit chemin.

Dans le cas où les valeurs sont positives, l'algorithme de Dijkstra (publié en 1959 par l'informaticien néerlandais du même nom) fournit tous les chemins de coût minimal partant d'un sommet donné s et rejoignant les différents sommets du graphe (en tout cas ceux qui sont accessibles depuis s !).

1) Principe de l'algorithme

Comme dans le parcours en largeur, nous allons étendre progressivement une "arborescence" reliant les sommets du graphe, en partant du sommet initial s et en absorbant progressivement les autres sommets. Il s'agit d'un algorithme *glouton* au sens où, à chaque étape, on sélectionne parmi les sommets non encore traités celui qui "coûte le moins cher".

Pour cela, nous allons maintenir une liste E des sommets non encore traités et, pour chaque sommet u de cette liste, une *estimation de coût minimal* (sous-entendu pour atteindre u via un chemin partant de s), notée $e[u]$. Enfin, comme pour le parcours en largeur, nous mémoriserons pour tout sommet u le *père* $p[u]$ qui permettra d'arriver de s à u via un chemin de coût minimal.

- Tous les $e[u]$ sont initialisés à ∞, les $p[u]$ à -1 ; E est initialisée à $[0, \dots, n-1]$.
- $e[s]$ prend la valeur 0 ($p[s]$ reste à -1 !)
- Tant qu'elle n'est pas vide, on extrait de E un élément u pour lequel e est minimale (au premier passage on doit trouver s !) et l'on met à jour les $e[v]$ pour les sommets v adjacents à u : si $e[u] + V(u, v) < e[v]$, c'est que le choix du chemin actuellement prévu de s à u suivi de l'arc $u \to v$, est préférable au chemin actuellement prévu de s à v (correspondant à l'estimation actuelle $e[v]$) ; dans ce cas $e[v]$ prend la valeur $e[u] + V(u, v)$ et $p[v]$ prend la valeur u.

2) Formalisation en pseudo-code

Là encore, nous donnons une version à partir de la liste L des listes d'adjacence, mais pour pouvoir prendre en compte la valeur associée à chaque arc, nous supposerons que, pour tout sommet u, $L[u]$ contient les couples (v, V) où v est adjacent à u et V est le coût de l'arc $u \to v$.

Dijkstra(L, s)
  initialiser e à ∞, p à -1 et E à $[0, \ldots, n-1]$
  $e[s] \leftarrow 0$
  tant que E n'est pas vide
    $u \leftarrow Extraire\_min(E)$
    pour chaque (v, V) de la liste d'adjacence de u
      si $e[u] + V < e[v]$ alors
        $e[v] \leftarrow e[u] + V : p[v] \leftarrow u$

où la fonction $Extraire\_min(E)$ renvoie un élément u de E tel que $e[u]$ soit minimal **et le supprime de** E !

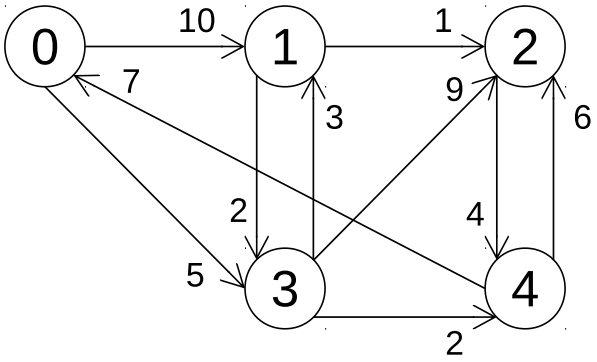
3) Programmation en Python

Traduire en Python l'algorithme décrit ci-dessus par une fonction `Dijkstra(L,s)`.

Penser à :

- `from numpy import inf` pour pouvoir utiliser le ∞ de `numpy`
- définir la fonction `Extraire_min(E)` **à l'intérieur de** `Dijkstra` afin qu'elle "connaisse" e et qu'elle puisse modifier E, cela toujours dans le but d'éviter les variables globales.

Tester avec l'exemple suivant



pour lequel

$$L = \left[[(1,10),(3,5)], [(2,1),(3,2)], [(4,4)], [(1,3),(2,9),(4,2)], [(0,7),(2,6)] \right].$$

NB : dans ce contexte, on remarquera que l'instruction Python `for (v,V) in L[u]` fonctionne.

4) Complément culturel : amélioration à l'aide d'un tas

Pour de grands graphes, la complexité linéaire de la version naïve de la fonction `Extraire_min` n'est guère satisfaisante. On peut gérer plus efficacement l'ensemble des sommets non encore traités en les stockant dans un *tas* (cf. le thème 1), avec comme clés de comparaison les valeurs contenues dans e.