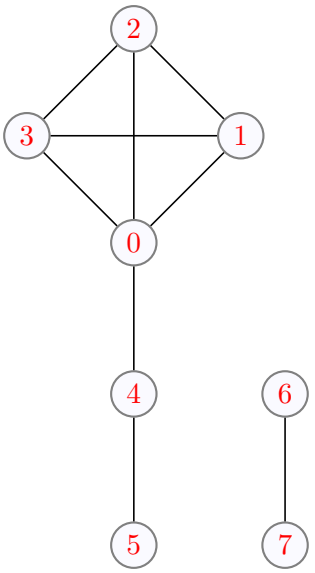
TP : Parcours de graphes

I) Notions élémentaires

Exercice 1



1) Donner la liste d'adjacence L correspondant au graphe ci-contre. On rappelle que, dans le cas d'un graphe non orienté, s'il y a une arête $\{i, j\}$, alors la liste $L[i]$ contient j, et la liste $L[j]$ contient i.

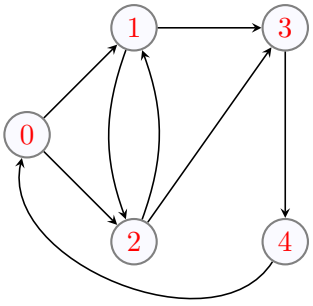
2) Donner les degrés des sommets 2 et 5.

3) Donner un chemin de 3 à 5 (il y a plusieurs possibilités). On précisera sa longueur.

4) Trouver un cycle de longueur 4 et un cycle de longueur 3.

5) Déterminer les composantes connexes.

Exercice 2



1) Donner la liste d'adjacence correspondant au graphe ci-contre.

2) Donner les degrés entrants et sortants des sommets 2 et 3.

3) Donner un chemin de 0 à 4 (il y a plusieurs possibilités). On précisera sa longueur.

4) Trouver un cycle de longueur 2 et un cycle de longueur 4.

Exercice 3

On donne la liste d'adjacence suivante :

```
L=[[4, 5], [3, 5], [3, 4, 5], [], [0, 1], [0, 1, 2]]
```

1) Le graphe est-il orienté ?

2) Tracer le graphe correspondant.

3) Donner la matrice d'adjacence correspondante.

II) Parcours en largeur

Vous trouverez ici :

`http://mpechaud.fr/scripts/parcours/index.html`

une très belle animation de plusieurs algorithmes de parcours de graphe.

Exercice 4

Pour parcourir en largeur un graphe G à n sommets, on utilisera les structures de données suivantes :

- `G` : une liste de liste de longueur n, qui contient la liste d'adjacence du graphe à parcourir.
- `c` : une liste de caractères `'b'`, `'g'` ou `'n'`, de longueur n, où `c[i]` contient la couleur – blanche, grise ou noire – du sommet `i`.

- `f` : une collection (voir question 1) qui contiendra la liste des sommets gris.
- `p` : la liste des pères.
- `d` : la liste des distances.

1) Le module `collections` propose une structure de « double ended queue » : des listes qui ont deux bouts, c'est-à-dire
   - On peut ajouter et enlever des éléments à droite avec `append` et `pop`, comme pour une liste habituelle.
   - On peut ajouter et enlever des éléments à gauche avec `appendleft` et `popleft`, à coût constant ($O(1)$).
   - La commande **len** fonctionne comme pour une liste.

   Importer le module sous le nom `co`, et poser `f = co.deque()`. Ajouter (à droite), successivement, 0, 3 et 1, et donner le contenu de `f`. Retirer un élément à droite, et donner le contenu de `f`. Retirer un éléments à gauche, et donner le contenu de `f`.

2) Écrire la liste d'adjacence `G` des sommets du graphe orienté du cours.

3) Initialiser les variables `c`, `f`, `p`, `d` comme indiqué dans le cours (première vignette).

4) Comment obtient-on la liste des sommets a voisins de $u = 0$? C'est-à-dire, tel qu'il y ait un arc $(0, a)$.

5) Écrire les instructions (les plus génériques possibles) pour que les différentes variables passent à l'état de la seconde vignette. On affichera le contenu des variables à l'aide d'un **print**.

6) À l'aide d'une boucle **while**, écrire l'algorithme de parcours en largeur. On affichera l'état des variables à chaque étape.

7) Écrire une fonction `parcours_en_largeur(G, s)` qui effectue un parcours en largeur du graphe `G` depuis le sommet `s`. Elle renvoie `p` et `d`.

8) Donner la complexité de cet algorithme. On note $|S| = n$ le nombre de sommets, et $|A|$ le nombre d'arêtes, et on comptera les accès aux données dans les listes, en notant qu'un sommet ne passe qu'une fois de blanc à gris.

III) Parcours en profondeur

Exercice 5

Pour parcourir en profondeur un graphe G à n sommets, on utilisera une partie des structures de données de l'exercice 4):

- `G` : une liste de liste de longueur n, qui contient la liste d'adjacence du graphe à parcourir.
- `c` : une liste de caractères `'b'`, `'g'` ou `'n'`, de longueur n, où `c[i]` contient la couleur – blanche, grise ou noire – du sommet `i`.
- `p` : la liste des pères.

On utilise à nouveau le graphe orienté `G` du cours. *Toute fonction doit être testée ! Une fonction non testée est bugguée.*

1) Initialiser les variables `c` et `p` comme indiqué dans le cours (première vignette).

2) Écrire une fonction `visiter_sommet(u)` qui prend en argument un sommet `u` blanc et ne retourne rien : elle modifie les variables `c` et `p` par effet de bord :
   - Elle colorie en gris le sommet `u`,
   - Elle parcourt les sommets `a`, blancs et voisins de `u`, note que le père de `a` est `u`, puis appelle `visiter_sommet(a)`.
   - Elle colorie en noir le sommet `u`

3) Écrire une fonction `parcours_en_profondeur(G, s)` qui effectue un parcours en profondeur du graphe `G` depuis le sommet `s`. Elle renvoie `p`.

4) Donner la complexité de cet algorithme. On note $|S| = n$ le nombre de sommets, et $|A|$ le nombre d'arêtes, et on comptera les accès aux données dans les listes, en notant qu'un sommet ne passe qu'une fois de blanc à gris.

Exercice 6
Dans cet exercice, on cherche à reconstituer un chemin depuis le sommet `s` de départ vers un sommet `u` à l'aide de la liste `p` des pères construite par l'algorithme de parcours.

1) Sur un exemple, reconstituer le chemin de `s = 0` vers `u = 4` à l'aide de `p`.

2) Écrire une fonction `chemin(s, u, p)` qui prend en arguments deux sommets `s` et `u` du graphe, et la liste `p` des pères construite par l'algorithme précédent, et retourne une liste de sommets représentant le chemin de `s` vers `u`.

3) Tester pour `p` donnée par l'algorithme de parcourt en largeur et en profondeur. Quel est le plus court chemin, en terme de nombre d'arêtes ?

IV) Plus court chemin

Dans cette partie, on s'intéresse aux graphes pondérés : on cherche un plus court chemin, c'est-à-dire un chemin de poids minimal, depuis un sommet `s`.
Un graphe pondéré sera codé par une liste d'adjacence `G`. Désormais, `G[i]` contient une liste de couples `(j, p)`, où l'arête (i, j) est de poids p.
Le premier exercice porte sur l'algorithme de **Dijkstra**, qui calcule les plus courts chemins depuis un sommet `s` vers tous les autres sommets, dans un graphe pondéré par des poids positifs. *On peut interrompre l'algorithme si on cherche le chemin vers un sommet `sc` cible fixé : lorsque `sc` devient* noir, *le plus court chemin est trouvé.*

Exercice 7 (Dijkstra)
Pour coder l'algorithme de Dijkstra, on utilisera les structures de données suivantes :

- `G` : une liste de liste de longueur n, qui contient la liste d'adjacence du graphe à parcourir.
- `c` : une liste de caractères `'b'`, `'g'` ou `'n'`, de longueur n, où `c[i]` contient la couleur – blanche, grise ou noire – du sommet `i`.
- `d` : la liste des distances.
- `p` : la liste des pères.
- `E` : la liste des sommets gris.

1) Écrire la liste d'adjacence du graphe du premier exemple de la présentation de l'algorithme de Dijkstra. On notera les sommets 0, 1, etc.

2) Initialisation :

- `c` : tous les sommets sont blancs,
- `d` : tous les sommets sont à l'infini, codé `'inf'`,
- `p` : tous les pères sont à `-1`,
- `E` : vide

Puis mettre à jour $s = 0$: il est gris et à distance nulle de lui-même.

3) On suppose désormais les listes en cours de remplissage. Le cœur de l'algorithme consiste à comparer

- la distance entre `s` et `a` stockée dans `d[a]`,
- la distance entre `s` et `a` en passant par le sommet actif `u` : `d[u] + poids`, où `poids` est le poids de l'arête (u, a).

Si la seconde distance est strictement meilleure, on met à jour la distance `d[a]` et le père `p[a]`.
Écrire une fonction `relacher(u, a, poids, d, p)` qui relâche l'arête (u, a). On prendra soin de traiter le cas où `d[a] = 'inf'`. Cette fonction ne renvoie rien, mais modifie les listes `d` et `p`, éventuellement.

4) À chaque étape, le sommet actif choisi est celui, gris, de valeur de `d` minimale.
Écrire une fonction `choix_sommet(d, E)` qui retourne le sommet `s0` gris tel que `d[s0]` soit minimale. Elle met aussi à jour `E`.

5) Écrire une fonction `dijkstra(G, s)` qui retourne les plus courts chemins depuis `s`. On peut s'inspirer et adapter la fonction écrite pour le parcours en largeur.

6) Modifier votre fonction pour ne plus utiliser la liste `c`. La liste `E` doit suffire.

7) Donner la complexité de votre algorithme. On admet que l'on peut, avec des structures de données bien choisie, avoir une fonction de choix du sommet actif en $O(\log |S|)$. Donner la complexité de l'algorithme dans ce cas.

Le second exercice porte sur l'algorithme $\mathbf{A}^\star$, qui calcule le plus court chemin entre deux sommets `s` et `sc`, dans un graphe pondéré muni d'une heuristique. Ici, on suppose que la position des sommets dans le dessin du graphe est une information pertinente, et l'heuristique choisie sera la distance euclidienne au sommet cible. C'est une heuristique plausible si on cherche le plus court chemin sur une carte routière : la distance « à vol d'oiseau » n'est pas forcément le chemin le plus rapide, ni même le plus court (montagnes, fleuves), mais elle indique la direction à tester en priorité.

Exercice 8
Les sommets du graphe sont désormais des points de $\mathbb{R}^2$, donc codés par des couples de flottants. Au lieu d'une liste, `G` est désormais un dictionnaire `{(x, y): L}`. La liste `L` contient des arêtes partant du sommet `(x, y)`, codées sous la forme `(xc, yc)`. Le poids entre `(x, y)` et `(xc, yc)` est la distance euclidienne entre ces deux sommets.

1) Donner un exemple de graphe et son dictionnaire associé.

2) Adaptation de l'algorithme de Dijkstra vu à l'exercice précédent.

- **a)** Coder une fonction `dist(a, b)` donnant la distance euclidienne entre deux points `a` et `b` de $\mathbb{R}^2$: `a` et `b` sont des couples de flottants.
- **b)** Adapter la fonction `dijkstra` au nouveau type de graphe pondéré.
- **c)** Modifier la fonction `dijkstra(G, s)` en une fonction `dijkstra_cible(G, s, sc)`

3) Modifier la fonction `dijkstra_cible(G, s, sc)` pour la transformer en `Aetoile(G, s, sc)` : le sommet suivant ne sera pas simplement celui de plus petit poids, mais celui qui minimise `prio` pour choisir le sommet actif.

4) Donner un exemple de graphe où l'algorithme A* est inefficace.