Parcours dans un graphe

On présente ici le principe de coloriage qui permet d'identifier la progression d'un parcours dans un graphe... c'est une notion délicate mais elle permet de faciliter la programmation des algorithmes de plus court chemin.

Quand on parcourt les sommets d'un graphe, on réalise simplement un chemin entre des sommets : on parle de **chemin reliant deux sommets i et j**, et on peut facilement distinguer sur un tel chemin les sommets **ascendants** ou **descendants** en fonction d'une position donnée.

Dans les problèmes d'optimisation, on peut alors chercher des **chemins élémentaires** minimisant les poids rencontrés ou le nombre de sommets parcourus. Mais avant cela, on fera quand même attention parce que les algorithmes de parcours sur les graphes sont souvent très lourds et pour définir un chemin entre deux sommets i et j, si celui-ci existe, il nous faudra tester tous les chemins partant de i pour trouver ceux qui mènent à j.

Concrètement, on distinguera le **parcours en largeur** et le **parcours en profondeur** :

- le **parcours en largeur** consiste, à partir d'un sommet donné, à visiter tous les sommets successeurs. On répète l'opération tant qu'il existe des sommets non visités. On explore donc tous les sommets qui sont directement accessibles puis ceux qui sont accessibles en passant par un sommet, puis deux, puis trois... et ainsi de suite en organisant les chemins par nombre de sommets.
- le **parcours en profondeur** lui ne fonctionne pas de la même manière, en effet, il explore un sommet et essaie d'aller le plus loin possible. Quand ce n'est pas possible, on revient en arrière et on essaie de parcourir en profondeur en prenant un sommet qui n'a pas encore été visité.

1 Parcours par coloriage des sommets

Les algorithmes précédents procèdent généralement par **coloriage des sommets**.

Initialement, tous les sommets sont coloriés en **blanc**, traduisant le fait que ces sommets n'ont pas été "découverts".

Lorsqu'un sommet est découvert (autrement dit, quand on arrive pour la première fois sur ce sommet), il est colorié en **gris** et doit être exploré. Le sommet reste gris tant qu'il reste des successeurs de ce sommet qui sont blancs, c'est à dire qui n'ont pas encore été découverts.
Un tel sommet est alors colorié en **noir** lorsque tous ses successeurs sont gris ou noirs, autrement dit, lorsqu'ils ont tous été à leur tour découverts.

Pour cela, on utilise tout au long du parcours, une **"liste d'attente au coloriage en noir"** dans laquelle on va stocker tous les sommets gris : un sommet est mis dans la liste d'attente dès qu'il est découvert. Un sommet gris dans la liste d'attente peut donc faire rentrer dans la liste ses successeurs qui sont encore blancs (en les coloriant en gris). Et quand tous les successeurs d'un sommet gris de la liste d'attente sont soit gris soit noirs, il est colorié en noir et il sort de la liste d'attente.

On itère alors le processus jusqu'à obtenir ce qu'on souhaite.

Remarques

1. Le parcours en largeur utilise une **file d'attente**, pour laquelle le premier sommet arrivé est aussi le premier à en sortir en fin de liste : on parle de liste **FIFO** pour *first in first out*.
   Par exemple, dans le langage Python, il y a des méthodes pratiques pour enfiler les données par devant et faire cette extraction :

```
In : F=[]; F.insert(0,1); F.insert(0,2); F;
                                        [2,1]
```
python

```
In : F.pop(); F;
                                        1; [2]
```
python

2. Le parcours en profondeur utilise une **pile d'attente**, pour laquelle le dernier sommet arrivé dans la pile est le premier à en sortir en fin de liste : on parle de pile **LIFO** pour *last in first out*.
   Par exemple, dans le langage Python, il y a des méthodes pratiques pour empiler les données par derrière et faire cette extraction :

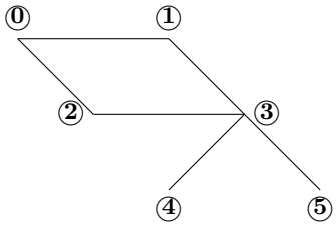
```
In : P=[]; P.append(1); P.append(2); P;
                                        [1,2]
```
python

```
In : P.pop(); P;
                                                    2; [1]
```
python

Pour finir, on essaiera une fois familiarisé avec les graphes, de mettre en place ces algorithmes de parcours pour tester l'**accessibilité d'un sommet par rapport à un autre**, c'est à dire s'il est possible de trouver un chemin permettant de relier le sommet i au sommet j, puis de tester si un graphe est **connexe**, c'est à dire on vérifiera si pour tout couple $(i,j) \in S^2$, il existe au moins un chemin permettant de relier les sommets i et j.

Exercice 1 (test d'accessibilité et connexité dans un graphe non pondéré)**.** [ ]

On considère un graphe $G = (S, A)$ à n sommets, qu'on suppose non orienté, non valué et sans boucle. On suppose que celui-ci est représenté par une liste d'adjacence telle que pour tout $i \in [\![0, n-1]\!]$, $L[i]$ contient la liste des voisins du sommet i. Par exemple, on pourra considérer le graphe de la semaine dernière :



Pour tout couple de sommets $(i,j) \in [\![0, n-1]\!]^2$, on rappelle ici comment on parcourt le graphe en profondeur pour déterminer s'il existe un chemin de i vers j :

On considère la donnée de (L, i, j), puis on initialise une liste *couleur* qui colore à 0 tous les sommets, c'est à dire pour n sommets :

$$couleur = [0, \dots, 0]$$

On définit une liste L_a des sommets gris qui sont à explorer, et donc en attente de coloriage au noir, en plaçant le premier sommet i, c'est à dire :

$$L_a = [i]$$

Puis, tant que la liste L_a n'est pas vide,

- on extrait le dernier sommet s de L_a, et on le colorie en noir : $couleur[s] = 2$
- si $s = j$, c'est fini et on retourne `True`
- sinon, on parcourt les voisins $v \in L[s]$ du sommet s :
  - si leur couleur est blanche, on les ajoute naturellement à la liste L_a sous la forme d'une pile, et on les colorie en gris : $couleur[v] = 1$.
  - sinon, on ne fait rien.

Enfin, si la boucle s'interrompt, c'est qu'il n'y a plus de sommet à traiter, et on renvoie simplement `False` : le sommet j n'était finalement pas accessible.

1. Dans le langage Python, construire la fonction `accessibilite(L:list,i:int,j:int)->bool` qui pour tout graphe donné par sa liste d'adjacence L et tout couple (i,j) de sommets, teste s'il existe un chemin reliant i à j. On pourra ajouter une pré-condition pour vérifier si i et j sont bien des sommets possibles.
2. En utilisant le programme précédent, construire la fonction `acces(L:list)->array` qui pour tout graphe donné par sa liste d'adjacence L, renvoie le tableau des accessibilités, autrement dit le programme renvoie une matrice dans laquelle le coefficient $m_{ij} \in \{\texttt{True}, \texttt{False}\}$ en fonction de l'accessibilité de i vers j.
3. De la même façon, construire la fonction `connexe(L:list)->bool` qui teste si un graphe donné par sa liste d'adjacence est connexe ou non.

On peut aussi adapter le programme d'accessibilité en suivant un parcours en largeur : pour cela, il suffirait de traiter la liste L_a sous la forme d'une file d'attente...

2 Cas particulier de l'algorithme de Dijkstra

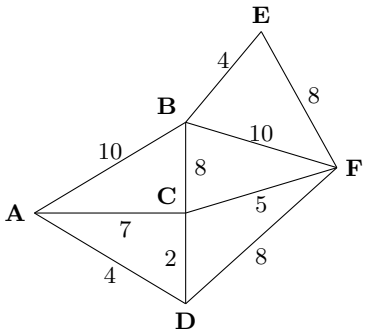
On considère un graphe valué à n sommets qu'on peut supposer orienté et dont on connait le poids de toutes les arêtes. En particulier, on peut définir M sa matrice d'adjacence de sorte que pour tout $(i,j) \in [\![0, n-1]\!]$:

$$M_{i,j} = \omega_{ij} \geq 0$$

On cherche alors à déterminer **le plus court chemin entre deux sommets**, c'est à dire que pour deux sommets connexes, on cherche le chemin par lequel la somme des poids rencontrés sera minimale.

Pour cela, on présente l'**algorithme de Dijkstra** qui consiste à parcourir le graphe en largeur, c'est à dire que les sommets à explorer sont traités dans une file d'attente, mais **en faisant aussi attention à la distance cumulée depuis le sommet initial** et qui devra être minimale... en réalité, c'est cette condition qui définit la priorité du traitement des sommets.

Par exemple, on définit le graphe suivant pour lequel on cherche à déterminer le plus court chemin entre A et F :



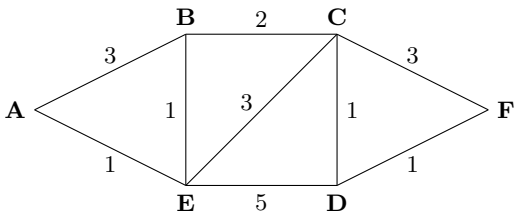
On construit alors un tableau dans lequel on va ajouter les poids rencontrés au fur et à mesure, et ceci de la façon suivante :

1. on place les sommets de A à F, puis on initialise le sommet initial à 0 et les autres à la valeur ∞.
2. Puis à chaque étape :
   - on sélectionne le sommet réalisant la distance cumulée $d(n)$ la plus petite,
   - et on complète la ligne suivante en ajoutant les distances parcourues entre deux sommets, et ceci <u>à condition</u> que la distance obtenue soit strictement plus petite. Sinon, on recopie la distance précédente.

   On fera quand même attention à bloquer le sommet quand celui-ci a été traité, et à indexer les distances cumulées par le sommet précédent : c'est pour cela, qu'on placera plus tard dans la file d'attente un sommet à traiter, auquel on associera la distance en cours.
3. enfin, quand la distance minimale est obtenue dans la dernière colonne, le sommet cible est atteint et on peut lire la distance minimale entre les sommets. Il suffit alors de remonter les sommets prédécesseurs, colonne par colonne, pour identifier le plus court chemin entre A et F.

Exercice 2 (algorithme de Dijkstra dans un graphe pondéré). [ ]

1. On considère le graphe suivant :



   (a) Donner la matrice d'adjacence M associée à ce graphe.

   (b) En appliquant l'algorithme de Dijkstra, déterminer le plus court chemin du sommet A au sommet F.

2. Plus généralement, on considère un graphe $G = (S, A)$ à n sommets, qu'on suppose orienté et valué. On suppose que celui-ci est représenté par une matrice d'adjacence $M = (m_{ij})$ telle que pour tout $(i, j) \in [\![0, n-1]\!]$,

$$m_{ij} \text{ représente le poids de l'arc } i \longrightarrow j.$$

   De plus, et avant de construire l'algorithme dans le langage Python, on détaille ici comment on parcourt le graphe en largeur pour déterminer, s'il existe, le plus court chemin de i vers j :

On considère la donnée de (M, i, j), puis on initialise une liste *couleur* qui colore à 0 tous les sommets, c'est à dire pour n sommets :

$$couleur = [0, \dots, 0]$$

On définit une liste L_a des sommets gris qui sont à explorer, et donc en attente de coloriage au noir, en plaçant le premier sommet i :

$$L_a = [i]$$

On définit enfin L la liste d'adjacence associée à M, D la liste des distances cumulées qu'on initialise à :

$$D = [float('inf'), \dots, \underbrace{0}_{i}, \dots, float('inf')]$$

et P une liste de prédecesseurs qu'on initialise par défaut à :

$$P = [-1, \dots, -1]$$

Puis, tant que la liste L_a n'est pas vide,

- on extrait le sommet s, associé à une distance minimale, inclus dans L_a puis on l'enlève et on le colorie en noir : $couleur[s] = 2$
- si $s = j$, c'est fini et on retourne `True`, et le chemin obtenu à partir de la liste des prédecesseurs de j, ainsi que la distance totale $D[s]$.
- sinon, on parcourt les voisins $v \in L[s]$ du sommet s :
  - si le sommet v est blanc, on l'ajoute naturellement à la liste L_a sous la forme d'une file, et on le colorie en gris : $couleur[v] = 1$. De plus,
    - on calcule la nouvelle distance : $D[v] = D[s] + m_{sv}$,
    - on place alors le sommet s comme prédecesseur dans $P[v]$.
  - si le sommet v est déjà gris. Dans ce cas, on compare les distances $D[v]$ et $D[s] + m_{sv}$ et si $D[s] + m_{sv} < D[v]$, on pose $D[v] = D[s] + m_{sv}$, et on place alors le sommet s comme prédecesseur dans $P[v]$.

Enfin, si la boucle s'interrompt, c'est qu'il n'y a plus de sommet à traiter, et on renvoie simplement `False` : le sommet j n'était finalement pas accessible.

Pour finir, on aura besoin de fonctions auxiliaires : l'une qui nous permettra d'extraire de la file d'attente les sommets prioritaires (ici, ceux associés à une distance minimale) et l'autre qui nous permettra de remonter l'ensemble des sommets parcourus pour en exhiber le plus court chemin.

(a) On considère un graphe à n sommets, et on admet avoir construit une liste $L = [s_1, \dots, s_p]$ de sommets à traiter. On note $D = [d_1, \dots, d_n]$ les distances associées à chacun des sommets du graphe. En particulier, $p \leq n$ et donc le sommet s_1 est associé à la distance d_{s1}...
Dans le langage Python, construire la fonction `sommet(L:list,D:list)->int` qui parcourt les distances incluses dans D et renvoie le sommet s_{i_0} de L associé à la distance minimale.
Par exemple, avec $L = [0, 1, 4]$ et $D = [15, 12, float('inf'), 7, 18]$, il vient :

```
In : sommet(L,D)
                                    1
                                                                    python
```

(b) On appelle liste des prédecesseurs toute liste P à n sommets pour laquelle $P[i]$ renvoie le sommet qui a permis d'obtenir i, avec la convention $P[i] = -1$ si le sommet i n'a pas de successeur. Par exemple, le chemin : $0 \longrightarrow 2 \longrightarrow 4$ aurait pour liste de prédecesseurs dans un graphe à 6 sommets :

$$P = [-1, -1, 0, -1, 2, -1]$$

Dans le langage Python, construire alors la fonction `chemin(P:list,j)->list` qui renvoie la liste des prédécesseurs du sommet j, c'est à dire qu'on remonte les sommets jusqu'à obtenir -1, ce qui signifie qu'il n'y a plus de prédecesseur.
Par exemple :

```
In : chemin(P,4)
                                  [0,2,4]
                                                                    python
```

(c) Construire alors le programme `dijkstra` qui pour tout graphe donné par sa matrice d'adjacence, et pour tout couple $(i, j) \in S^2$ renvoie le plus court chemin éventuel.
On testera évidemment notre programme avec le graphe proposé à la question 1.

Remarques

1. La vitesse d'excution de l'algorithme de Dijkstra dépend surtout de l'implémentation et de la gestion de la file d'attente. Autrement dit, dans le langage Python, les files sont souvent gérées sous la forme de liste et les méthodes associées (`insert(0,.)` ou encore `pop()`) fonctionnent en un temps quasi-linéaire. C'est très efficace !

2. Attention, l'algorithme de Dijkstra ne convient pas pour des graphes à poids négatifs... et on devra adapter nos graphes à la situation pour obtenir des poids positifs : par exemple, au lieu de travailler avec des dénivelés, on travaille en altitude au dessus de la mer.

3. Il existe d'autres algorithmes de calcul du plus court chemin dans un graphe valué et orienté. Certains permettent même d'imposer une **heuristique** h, c'est à dire qu'on modifie le choix des sommets dans la file d'attente au cours du programme.
Concrètement, au lieu de considérer la distance minimale $d(n)$ à chaque étape, on peut par exemple choisir de minimiser:

$$d'(n) = d(n) + h(n)$$

où $h(n)$ représente la distance du sommet courant jusqu'au point d'arrivée. C'est notamment le cas de l'algorithme A^*, on dit alors que l'algorithme est **guidé** ou **informé**.