

## TP 12

## Piles, Files et Introduction aux graphes

## Exercice 1 – Évaluation d'une expression postfixée

Jean-Pierre Becirspahic

La notation postfixée d'une expression algébrique consiste à placer les opérateurs après son ou ses opérandes. Par exemple, l'addition de  $a$  et de  $b$  sera écrite  $a \ b \ +$  en notation postfixée, la racine carrée de  $a$  sera écrite  $a \ \sqrt{\phantom{x}}$ .

L'intérêt majeur de cette notation est qu'une expression postfixe n'est jamais ambiguë : alors que l'expression infixe  $1 + 2 \times 3$  peut avoir deux significations :  $(1 + 2) \times 3$  ou  $1 + (2 \times 3)$ , ce n'est jamais le cas d'une expression postfixe, ce qui rend l'usage des parenthèses superflu :  $1 \ 2 \ + \ 3 \ \times$  ne peut être compris que de cette façon :  $(1 \ 2 \ +) \ 3 \ \times$  et  $1 \ 2 \ 3 \ \times \ +$  de cette façon :  $1 \ (2 \ 3 \ \times) \ +$ . Nous allons montrer comment, à l'aide d'une pile, on peut évaluer une expression algébrique postfixe. Dans cet exercice, les expressions algébriques seront représentées par les listes qui pourront contenir des nombres (de type `int` ou `float`) ou des chaînes de caractères représentant des opérateurs unaires ou binaires (comme par exemple `sqrt` ou `+`). Par exemple, l'expression  $\frac{1+2\sqrt{3}}{4}$  sera représentée par la liste `[1, 2, 3, 'sqrt', '*', '+', 4, '/']`. On suppose donné deux dictionnaires répertoriant pour l'un les opérateurs unaires, pour l'autre les opérateurs binaires, et qui associent à chaque chaîne de caractère la fonction correspondante. On peut par exemple définir ces deux dictionnaires à l'aide du script suivant, et les compléter en suivant le même modèle.

```
from numpy import sqrt, exp, log
op_uni = {'sqrt': sqrt, 'exp': exp, 'ln': log}
def add(x, y):
    return x + y

def sous(x, y):
    return x - y

def mult(x, y):
    return x * y

def div(x, y):
    return x / y

op_bin = {'+': add, '-': sous, '*': mult, '/': div}
```

L'évaluation d'une expression postfixe consiste à utiliser une pile initialement vide et à parcourir les éléments de la liste représentant l'expression à évaluer en appliquant les règles suivantes :

- si l'élément est un nombre, il est empilé ;
- si l'élément est un opérateur unaire, le sommet de la pile est dépilé, l'opérateur lui est appliqué et le résultat ré-empilé ;
- si l'élément est un opérateur binaire, deux éléments de la pile sont dépilés, l'opérateur leur est appliqué et le résultat ré-empilé.

Si l'expression postfixe est correcte sur le plan syntaxique (et mathématique), à la fin du traitement de la liste la pile ne contient plus qu'un seul élément égal au résultat de l'évaluation de l'expression. On suppose donnés les deux dictionnaires `op_uni` et `op_bin`.

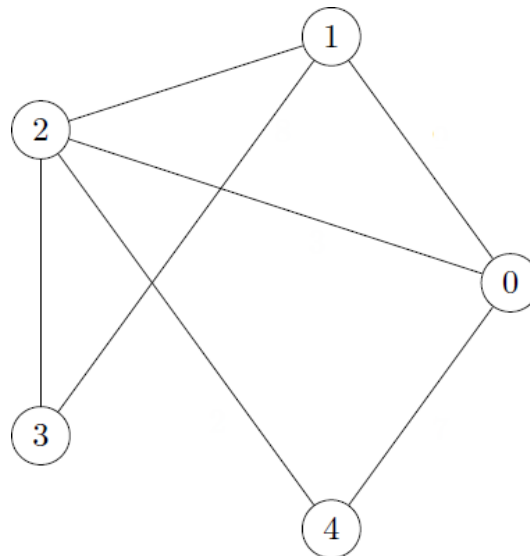
**Question 1** Rédiger une fonction qui évalue une expression postfixe donnée sous forme de liste `evalue(lst:liste) -> float`. Dans un premier temps, on pourra supposer que l'expression est syntaxiquement correcte.

**Question 2** Rédiger une seconde fonction d'évaluation qui détecte les erreurs de syntaxe.

On la notera `evaluer2(lst:liste)->float`

## Exercice 2 – Implémentation des graphes par une liste d'adjacence

On considère le graphe G suivant, où le nombre situé sur l'arête joignant deux sommets est leur distance, supposée entière.



Pour implémenter le graphe, on utilise une liste G1 qui a pour taille le nombre de sommets. Chaque élément G1[i] est la liste des voisins de i.

Dans ce cas, G1[0]=[1, 2, 4] car Les sommets 1, 2 et 4 sont des voisins de 0.

**Question 3** Construire la liste d'adjacence G1 en utilisant la méthode énoncée ci-dessus.

**Question 4** Écrire une fonction `voisins_l(G:list, i:int) -> list`, d'argument la liste d'adjacence G et un sommet i, renvoyant la liste des voisins du sommet i.

**Question 5** Écrire une fonction `arretes_l(G:list) -> list`, renvoyant la liste des arêtes. Les arêtes seront constitués de couples de sommets (l'arête entre les sommets 0 et 1 sera donnée par (0, 1)).

Les instructions suivantes permettent de tracer un graphe.

```

import networkx as nx

def plot_graphe_l(G):
    Gx = nx.Graph()
    edges = arretes_l(G)
    Gx.add_edges_from(edges)
    nx.draw(Gx, with_labels = True)
    plt.show()
plot_graphe(M)

```

**Question 6** Écrire et tester la fonction `plot_graphe_l(G)`.

**Question 7** Écrire une fonction `degre_l(G:list, i:int) -> int`, d'argument un sommet i, renvoyant le nombre des voisins du sommet i, c'est-à-dire le nombre d'arêtes issues de i.

**Question 8** Écrire la fonction `ajout_sommet_l(G:list, L:list) -> None` permettant d'ajouter un sommet au graphe. L désigne la liste des sommets auxquels le nouveau sommet est relié. `ajout_sommet` agit avec effet de bord sur G.

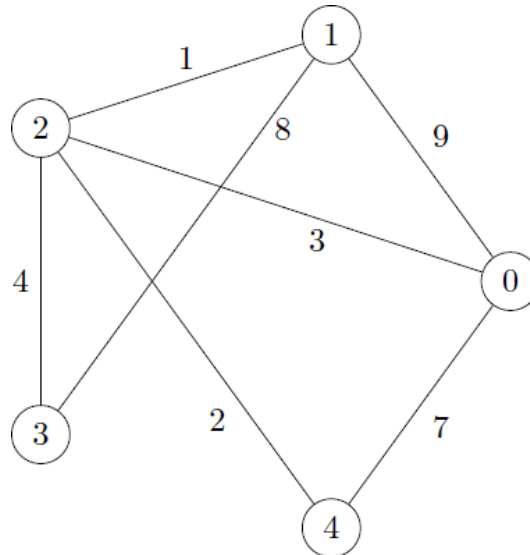
**Question 9** Écrire la fonction `supprime_sommet_l(G:list, i: int) -> None` permettant de supprimer le sommet i du graphe.

**Question 10** Écrire la fonction `from_list_to_matrix(G:list) -> list` permettant de convertir un graphe implémenté sous forme de liste d'adjacence en matrice d'adjacence.

**Question 11** Écrire la fonction `from_matrix_to_listmatrix(G:list) -> list` permettant de convertir un graphe implémenté sous forme de matrice d'adjacence en liste d'adjacence.

### Exercice 3 – Implémentation des graphes par une matrice d'adjacence

On considère le graphe G suivant, où le nombre situé sur l'arête joignant deux sommets est leur distance, supposée entière.



**Question 12** Construire la matrice  $(G_{ij})_{0 \leq i, j \leq 4}$ , matrice de distances du graphe G, définie par : « pour tous les indices  $i, j$ ,  $G_{ij}$  représente la distance entre les sommets  $i$  et  $j$ , ou encore la longueur de l'arête reliant les sommets  $i$  et  $j$  ». Cette matrice sera implémentée sous forme d'une liste de listes. (Chaque « sous-liste » représentant une ligne de la matrice d'adjacence. On convient que, lorsque les sommets ne sont pas reliés, cette distance vaut  $-1$ . La distance du sommet  $i$  à lui-même est égale à 0.

**Question 13** Écrire une fonction `voisins(G:list, i:int) -> list`, d'argument la matrice d'adjacence G et un sommet  $i$ , renvoyant la liste des voisins du sommet  $i$ .

**Question 14** Écrire une fonction `aretes(G:list) -> list`, renvoyant la liste des arêtes. Les arêtes seront constitués de couples de sommets (l'arête entre les sommets 0 et 1 sera donnée par (0,1)).

Les instructions suivantes permettent de tracer un graphe.

```

import networkx as nx
import matplotlib.pyplot as plt

def plot_graphe(G):
    Gx = nx.Graph()
    edges = aretes(G)
    Gx.add_edges_from(edges)
    nx.draw(Gx, with_labels = True)
    plt.show()
plot_graphe(M)
  
```

**Question 15** Écrire et tester la fonction `plot_graphe(G)`.

**Question 16** Écrire une fonction `degre(G:list, i:int) -> int`, d'argument un sommet  $i$ , renvoyant le nombre des voisins du sommet  $i$ , c'est-à-dire le nombre d'arêtes issues de  $i$ .

**Question 17** Écrire une fonction `longueur(G:list, L:list) -> int`, d'argument une liste L de sommets de G, renvoyant la longueur du trajet d'écrit par cette liste L, c'est-à-dire la somme des longueurs des arêtes empruntées.

Si le trajet n'est pas possible, la fonction renverra  $-1$ .

**Question 18** Écrire la fonction `ajout_sommet(G:list, L:list, poids : list) -> None` permettant d'ajouter un sommet au graphe. `L` désigne la liste des sommets (triés dans l'ordre croissant) auxquels le nouveau sommet est relié, `poids` la liste des poids respectifs. `ajout_sommet` agit avec effet de bord sur `G`.

**Question 19** Écrire la fonction `supprime_sommet(G:list, i: int) -> None` permettant de supprimer le sommet `i` du graphe.