TP n°2 sur les graphes : parcours de graphes non pondérés

Dans tout ce TP, nous ne considérerons que des graphes non pondérés. Ils peuvent être orientés ou pas.

Préliminaires

Le principe du parcours

Nous distinguerons trois types de sommets :

- les sommets blancs : ceux qui n'ont pas encore été explorés
- les sommets gris : stockés dans une «salle d'attente» juste avant d'être explorés.
- les sommets noirs : ceux qui ont déjà été explorés et donc qui sont sortis de la salle d'attente.

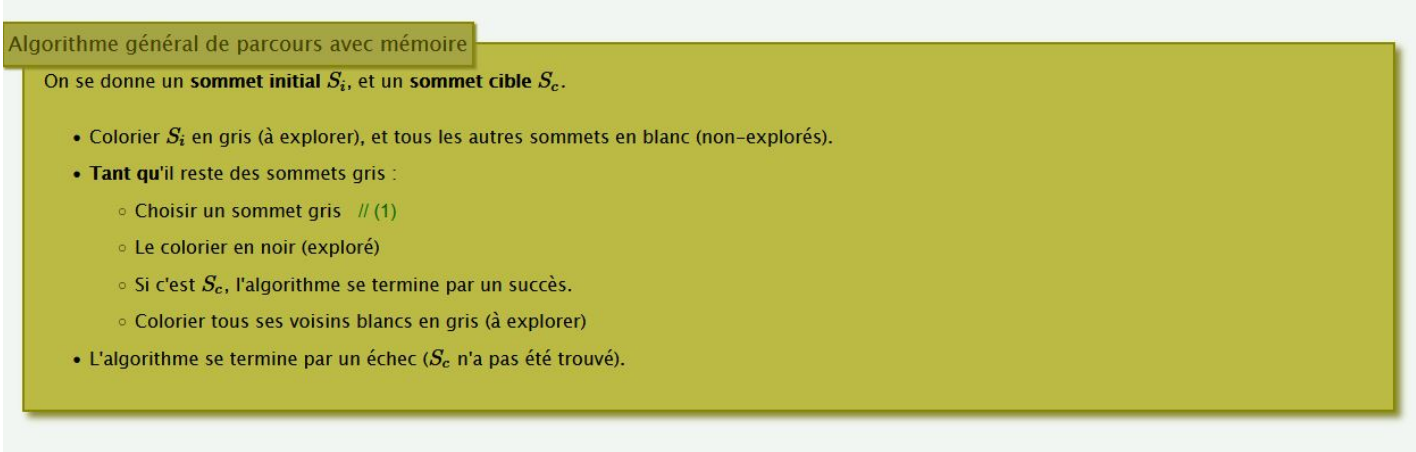

FIGURE 1 – Parcours générique

C'est la façon de choisir le prochain sommet gris dans la salle d'attente qui va déterminer le type de parcours et donc le choix de la structure de données pour implémenter efficacement la salle d'attente :

- dans un parcours en largeur, c'est le **premier** sommet gris arrivé dans la salle d'attente que l'on va explorer. C'est le principe du FIFO, First in, First out. Pour cela nous utiliserons la structure de **file** pour implémenter la salle d'attente.
- dans un parcours en profondeur, c'est le **dernier** sommet gris arrivé dans la salle d'attente que l'on va explorer. C'est le principe du LIFO, Last in, First out. Pour cela nous utiliserons la structure de **pile** pour implémenter la salle d'attente.

1 Découverte de deux nouvelles structures de données : la pile et la file

La `pile` (stack en anglais) est implémentée en Python par la notion de liste. Voici les opérations possibles sur une pile p, chacune d'entre elles a une complexité constante (en $O(1)$) :

- créer une pile vide p avec `p = []`
- accéder à l'élément «au-dessus» de la pile `p[-1]`
- empiler un élément x «au dessus» de p : `p.append(x)`
- dépiler (l'élément «du dessus») : `p.pop()`
- obtenir la longueur de la pile p : `len(p)`

La **file** (queue en anglais) est implémentée en Python par la structure de `deque` du module `collections` (on fera donc l'import suivant `from collections import deque`). Voici les opérations possibles sur une file q, chacune d'entre elles a une complexité constante (en $O(1)$) :

- créer une file vide q avec `q = deque()`
- enfiler un élément x dans q : `q.append(x)`
- défiler (l'élément le «plus à gauche») : `q.popleft()`
- obtenir la longueur de la file q : `len(q)`

Faisons un exercice où l'on utilise avec profit la notion de pile.

Exercice 1 (Expression bien parenthésée) Une expression bien parenthésée est une expression qui comporte une succession cohérente de parenthèses ouvrantes et fermantes. Par exemple, parmi les deux expressions suivantes

$$(4+7)\times((4-5)\times 3) \quad \text{et} \quad (4+(7\times((4-5)\times 3))$$

seule la première est bien parenthésée.

Écrire une fonction booléenne `bien_parenthesee(chaine)` qui teste si `chaine` est bien parenthésée. On utilisera pour cela le principe décrit ci-dessous :

on crée une pile vide p et on parcourt la chaîne de caractère :

- si l'on rencontre le caractère `'('`, on l'empile dans p
- si l'on rencontre le caractère `')'` :
  - si la pile est vide, on renvoie `False` (la parenthèse fermante ne peut correspondre à aucune parenthèse ouvrante)
  - sinon, on dépile un élément.

à la fin du parcours, si la pile est vide, on renvoie `True`, sinon `False` (on n'a pas fermé toutes les parenthèses ouvrantes).

2 Parcours en largeur (Breath First Search = BFS)

2.1 Principe et implémentations

Le parcours en largeur est obtenu en gérant la salle d'attente comme une file d'attente (FIFO).

Remarque : dans le cas où le graphe est un arbre (graphe non orienté connexe et sans cycle), ce parcours consiste à partir d'un sommet et à explorer tous ses fils, puis on explore tous ses petits-fils, puis tous ses arrières petits-fils...

Exercice 2 Dans le graphe 0 ci-dessous, décrire le parcours en largeur pour aller du sommet 0 au sommet 4. On décrira à chaque étape les sommets qui sont dans la salle d'attente, et lorsqu'un sommet a été exploré (extrait de la salle d'attente), on indiquera par une flèche son père. On indiquera ensuite le plus court chemin entre ces deux sommets, ainsi que sa longueur.

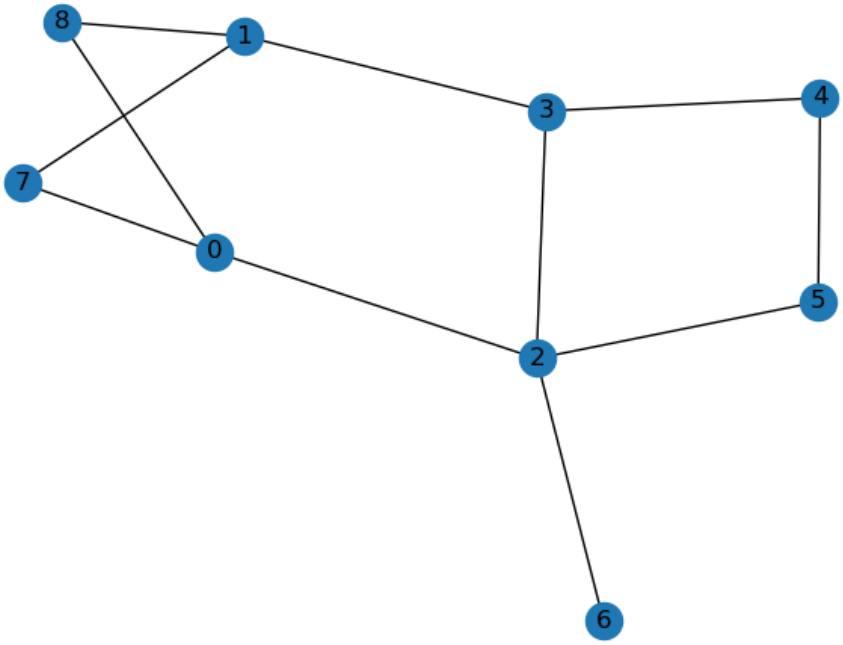

FIGURE 2 – Graphe 0

Exercice 3 (Parcours BFS, implémentation 1) On se donne G un graphe (non pondéré) défini par dictionnaire d'adjacence, puis `source` et `cible` deux sommets de ce graphe. On veut écrire en Python une fonction booléenne `est_accessible_BFS1(G, source, cible)` qui teste s'il existe un chemin dans G reliant les sommets `source` et `cible`.

Pour cela on utilisera :

- une file q (une `deque`) dans laquelle on va stocker les éléments de la salle d'attente. Elle sera initialisée à `source`.
- un dictionnaire `etat` dans lequel on va stocker les états (0 pour blanc, 1 pour gris et 2 pour noir) de chaque sommet.

Exercice 4 Tester votre fonction sur le graphes suivants dont les implémentations sont fournies en annexe.

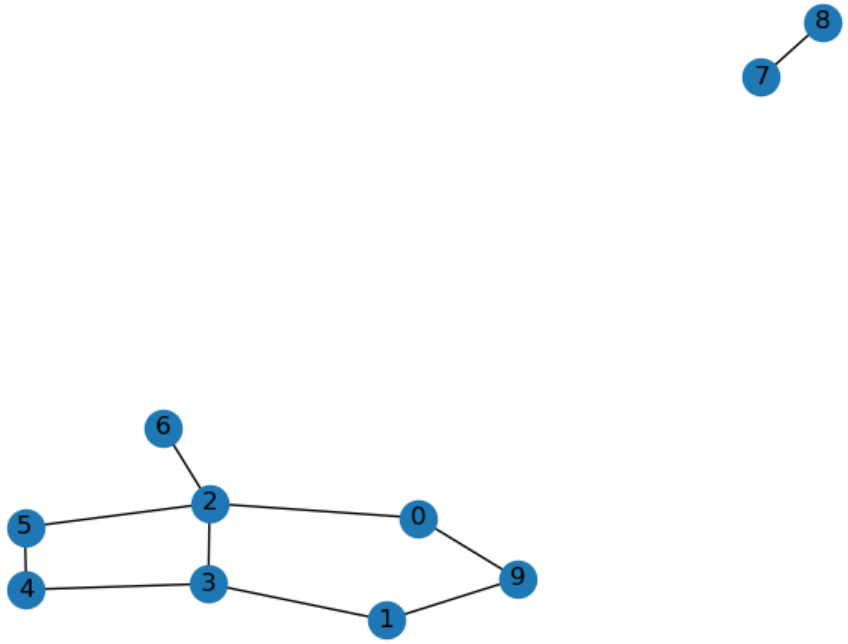

FIGURE 3 – Graphe non connexe

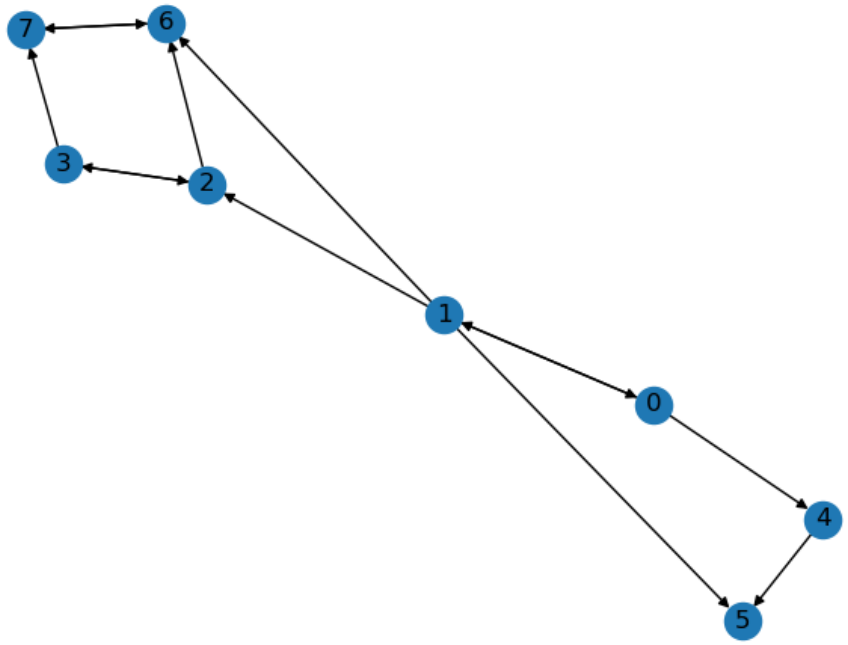

FIGURE 4 – Graphe orienté

Exercice 5 (Parcours BFS, implémentation 2) On se donne G un graphe (non pondéré) défini par listes d'adjacence, puis `source` et `cible` deux sommets de ce graphe. On veut écrire

en Python une fonction booléenne `est_accessible_BFS2(G, source, cible)` qui teste s'il existe un chemin dans G reliant les sommets `source` et `cible`.

Pour cela on utilisera :

- une file q (une `deque`) dans laquelle on va stocker les éléments de la salle d'attente. Elle sera initialisée à `source`.
- un dictionnaire `dejavu` dans lequel on va stocker les sommets qui sont rentrés dans la salle d'attente (on mettra `None` comme valeur).

  La condition «pour chaque voisin blanc» deviendra «pour chaque voisin qui n'est pas dans `dejavu`»

Exercice 6 Faire tout de suite l'exercice 11.

Exercice 7 (Complexité) Préciser la complexité des parcours BFS et DFS en fonction de $|S|$ et $|A|$ le nombre de sommets et le nombre d'arêtes.

2.2 Reconstruction du chemin et détermination d'un plus court chemin

Lorsque l'on explore le graphe, on dit qu'un sommet s est le père d'un sommet v si le sommet v est entré dans la salle d'attente en tant que voisin de s (v est donc dans la liste d'adjacence de s). On peut stocker cette relation de parenté dans un dictionnaire `parents` en écrivant `parents[v] = s`. On convient que le père du sommet `source` est `None`.

Par exemple, si l'on parcourt le graphe 1 pour accéder du sommet 0 au sommet 3, on obtient le dictionnaire `parents = {0: None, 9: 0, 2: 0, 1: 9, 3: 2, 5: 2, 6: 2}`. Le chemin entre 0 et 3 est donné par la liste $[0, 2, 3]$.

Exercice 8 (Des parents au chemin) Écrire une fonction `chemin(parents, source, cible)` qui renvoie le chemin parcouru pour aller de `source` à `cible`.

Si l'on veut reconstruire le chemin de parcours entre `source` et `cible`, il suffit de garder en mémoire pour chaque sommet exploré son père et donc de construire le dictionnaire `parents`.

Exercice 9 (Reconstruction du chemin) En déduire une fonction `chemin_BFS(G, source, cible)` qui renvoie un chemin dans G entre `source` et `cible` si celui-ci existe.

Par exemple, si $g1$ est le premier graphe, `chemin_BFS(g1, 1 ,6)` renverra `[1, 3, 2, 6]`.

Exercice 10 (Détermination d'un plus court chemin) Le parcours en largeur a une spécificité importante : on peut démontrer que le chemin obtenu entre la source et la cible est un **plus court chemin** (ceci est faux pour un parcours en profondeur). Pour comprendre cela, on mémorise la distance à la source de chaque sommet exploré (qui rentre dans la salle d'attente) à l'aide d'un dictionnaire `distance` initialisé avec `distance[source] = 0`. Si v est un sommet voisin du sommet s, sa distance à la source sera égale à la distance de s à la source augmentée de 1. Modifier ainsi la fonction `chemin_BFS(G, source, cible)` pour qu'elle renvoie aussi la longueur d'un plus court chemin.

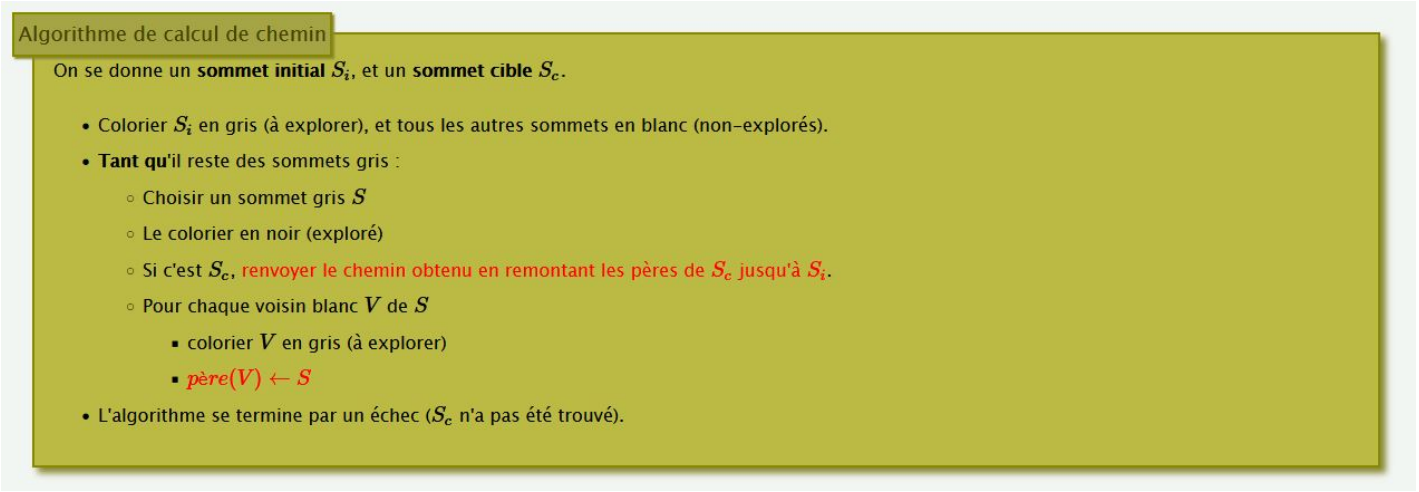

FIGURE 5 – Reconstruction du chemin

3 Parcours en profondeur (Deep First Search = DFS)

Le parcours en profondeur est obtenu en gérant la salle d'attente comme une pile (LIFO).

Remarque : dans le cas où le graphe est un arbre (graphe non orienté connexe et sans cycle), ce parcours consiste à partir d'un sommet et à explorer un des ses fils, puis un fils de son fils (les enfants sont visités avant les frères et soeurs)...

Exercice 11 (Parcours DFS, implémentation 1) On se donne G un graphe (non pondéré) défini par listes d'adjacence, puis `source` et `cible` deux sommets de ce graphe. On veut écrire en Python une fonction booléenne `est_accessible_DFS1(G, source, cible)` qui teste s'il existe un chemin dans G reliant les sommets `source` et `cible`.

Pour cela on utilisera :

- une pile p dans laquelle on va stocker les éléments de la salle d'attente. Elle sera initialisée à `source`.
- un dictionnaire `etats` dans lequel on va stocker les états (0 pour blanc, 1 pour gris et 2 pour noir) de chaque sommet.