Le parcours en largeur d'un graphe

Algorithmie

1 Ecriture de l'algorithme du parcours en largeur

1.1 Rappels sur le parcours en largeur

— Initialement, tous les sommets sont coloriés à blanc (traduisant le fait qu'ils n'ont pas encore été "découverts").

— Lorsqu'un sommet est "découvert" (autrement dit, quand on arrive pour la première fois sur ce sommet), il est colorié en gris. Le sommet reste gris tant qu'il reste des successeurs de ce sommet qui sont blancs (autrement dit, qui n'ont pas encore été découverts).

— Un sommet est colorié en noir lorsque tous ses successeurs sont gris ou noirs (autrement dit, lorsqu'ils ont tous été découverts).

De façon pratique, on va utiliser 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 colorié en gris. Un sommet gris dans la liste d'attente peut faire rentrer dans la liste ses successeurs qui sont encore blancs (en les coloriant en gris). 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.

1.2 Structures de données utilisées

— On utilise une file $\boxed{file}$, qui est en fait une liste, pour laquelle on sait faire les opérations suivantes :

◇ initialiser la liste $file$ à vide,

◇ ajouter un sommet s à la fin de la liste $file$,

◇ savoir si la liste est vide,

◇ enlever le sommet s au début de la liste $file$ (file.pop(0))

— un dictionnaire $\boxed{pere}$ (déjà évoqué) qui associe à chaque sommet le sommet qui la fait entrer dans la file,

— un dictionnaire $\boxed{couleur}$ qui associe à chaque sommet sa couleur (blanc, gris ou noir).

— un dictionnaire $\boxed{distance}$ qui associe à chaque sommet son niveau de profondeur par rapport au sommet de départ s_0 (autrement dit, $d[s_i]$ est la longueur du chemin dans l'arborescence $Pere$ de la racine s_0 jusque s_i). Ce tableau sera utilisé plus tard.

1.3 Algorithme de la fonction bfs(G, S)

Cette fonction doit retourner trois dictionnaires : pere, distance, et couleur, tels qu'ils ont été définis précédemment.

Déclaration et initialisation des variables

déclarer couleur, distance, et pere comme des dictionnaires

pour chaque clé x de G

couleur(x) initialisée à 'blanc'

distance(x) initialisée à 'infini'

pere(x) initialisée à **None**

Caractérisation du sommet

couleur(S) passe à 'gris'

distance(S) passe à 0

pere(S) reste à **None** (on ne fait donc rien)

on met (S) dans la file

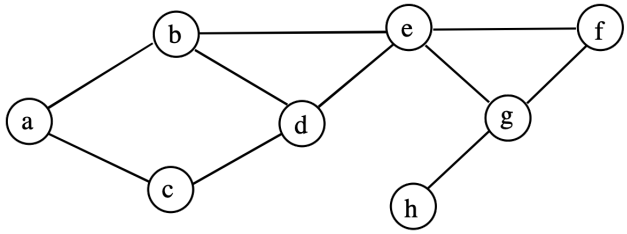
Processus de remplissage des dictionnaires pere, distance, et couleur

| tant que la file n'est pas vide
  | on regarde le premier élément de la file
  | pour chaque voisin de ce premier élément
    | si sa couleur est 'blanc'
      | le père de ce voisin est le premier élément de la file
      | la couleur de ce voisin passe à 'gris'
      | la distance de ce voisin augmente d'une unité
      | on place ce voisin dans la liste (file)
    | sinon
      | on ne fait rien
  | on colorie le premier élément en noir
  | on retire le premier élément de la liste, car on a épuisé tous ses successeurs
| on renvoie les dictionnaires pere, distance, couleur

1.4 Ecriture de l'algorithme du parcours en largeur

Activité 1 **Implémenter un graphe**

Écrire le dictionnaire G qui permet d'implémenter le graphe ci-dessous.



Activité 2 **Ecriture d'une fonction qui implémente le parcours de graphe en largeur**

nom : `bfs`

arguments : G,S (dict,str)

effet : renvoie trois dictionnaires : *pere*, *distance*, *couleur*. Ces trois dictionnaires renferment les renseignements décrits dans l'algorithme de parcours en largeur d'un graphe.

Tester ensuite votre algorithme sur le graphe G décrit ci-dessus, pour le sommet 'd'. Pour les trois dictionnaires, vous devriez obtenir :

pere

```
{'a': 'b',
 'b': 'd',
 'c': 'd',
 'd': None,
 'e': 'd',
 'f': 'e',
 'g': 'e',
 'h': 'g'}
```

distance

```
{'a': 2,
 'b': 1,
 'c': 1,
 'd': 0,
 'e': 1,
 'f': 2,
 'g': 2,
 'h': 3}
```

couleur

```
{'a': 'noir',
 'b': 'noir',
 'c': 'noir',
 'd': 'noir',
 'e': 'noir',
 'f': 'noir',
 'g': 'noir',
 'h': 'noir'}
```

2 Utilisations du parcours en largeur

2.1 Recherche du plus court chemin entre deux sommets

Le parcours en largeur peut être utilisé pour chercher le plus court chemin (en nombre d'arcs ou arêtes) entre la racine s_0 et chacun des autres sommets du graphe accessibles depuis s_0.

Pour cela, il suffit de **remonter** dans l'arborescence *pere* du sommet concerné jusqu'à la racine s_0. On se souviendra que le dictionnaire *pere* est construit de la manière suivante : la valeur d'une clé correspond au sommet par lequel on est passé pour atteindre la clé. On se souviendra également que si un sommet n'a pas de « père », la valeur de la clé associée est 'None'.

Activité 3 **Ecriture d'une fonction qui trouve un chemin entre un sommet et un autre**

nom : `chemin`

arguments : dico_pere,depart,arrivee (dict,str,str)

effet : renvoie une chaîne de caractère correspondant au chemin entre le départ et l'arrivée, s'il existe. Par exemple, si le chemin entre le départ 'e' et la fin 'g' est e -> a -> b -> c -> g, alors la chaîne renvoyée doit être 'eabcg'.
Renvoie un message disant que le chemin n'existe pas dans le cas contraire.

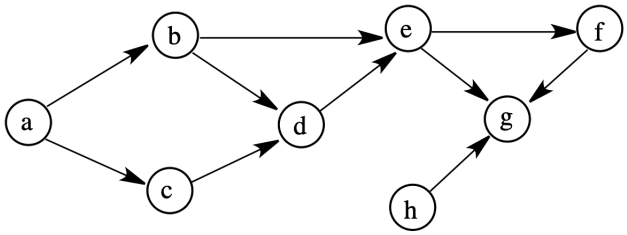
Cas du graphe non orienté

Testez cette fonction pour donner le chemin entre le départ 'd' et l'arrivée 'h'. Vous devez obtenir le chemin 'degh'.
Testez ensuite cette fonction sur d'autres chemins de votre choix ; vérifiez la cohérence du résultat annoncé.

Cas du graphe orienté

Testez ensuite cette fonction sur le graphe orienté ci-dessous, et pour les chemins suivants :

- a vers g
- h vers d
- f vers c
- f vers a



2.2 Graphe connexe ou pas ?

Le parcours en largeur peut être utilisé pour rechercher l'ensemble des sommets accessibles sommets noirs sont ceux accessibles depuis s_0 ; les sommets blancs sont ceux pour lesquels il n'existe pas de chemin/chaîne à partir de s_0.

Si le graphe est connexe, tous les sommets sont noirs à la fin de l'algorithme.

Activité 4 **Ecriture d'une fonction qui trouve si le graphe est connexe ou pas**

nom : `connexite`

arguments : couleur (dict)

effet : renvoie une chaîne de caractère 'Graphe connexe' si le graphe est connexe, 'Graphe non connexe' sinon.

Testez cette fonction pour voir si le graphe initial (non orienté) est connexe ou pas.