Objectif

☞ Problème dans des graphes.

Problème [Parcours dfs et applications]

La documentation générale pour ce travail se trouve ici, la documentation technique se trouve là, le répo git.

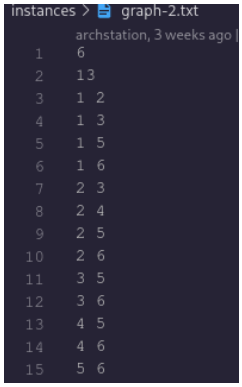
Partie A : Préliminaires

1. Implémenter une procédure permettant d'importer un graphe à partir d'un fichier.

La fonction `readGraph` dont la signature :

```
Graph *readGraph(const char *filename, bool digraph);
```

prend en argument le nom d'un fichier à lire (qui est supposé d'être dans le format spécifié dans l'énoncé du TP) et une `bool` indiquant si le graphe est orienté (vrai) ou pas (faux). En lisant le nombre de sommets, une structure `Graph` est alloué et un tableau qui contient des listes d'adjacènes est créé. Le pointeur qui est renvoyé pointe sur un bloc de mémoire nouvellement alloué.



(a) graph-2.txt

⇨

```
================= Read Graph ================
N edges: 26, N vertices: 6
1 -> 2 3 5 6
2 -> 1 3 4 5 6
3 -> 1 2 5 6
4 -> 2 5 6
5 -> 1 2 3 4 6
6 -> 1 2 3 4 5
```

(b) `printGraph(readGraph("graph-2.txt", false))`

FIGURE 1 – La transformation d'un fichier à un `Graph` non-orienté.

Un choix d'organisation de la structure se revèle. Pour les graphes non-oriéntés, j'ai décidé que chaque fois que l'on relie des sommets, je rajoute deux nouvelles arrêts qui vont dans le deux sens. C'est comme si implicitement je rajoutais deux connections orienté. La justification pour cela c'est quand il s'agit de parcourir un graphe par DFS. Concentrons sur les sommets 1 et 9. Sur ligne 5 de `graph-2.txt`, on relie les deux sommets 1 et 9. On voit cet arrêt stocké dans le graphe

2. Implémenter une procédure permettant d'exporter au format `.dot` un graphe donné en argument.

La fonction `createDot` dont la signature :

```
int createDot(const Graph *g, const char *filename);
```

prend en argument un pointer sur un `Graph` et qui exporte au format `.dot` au fichier `filename` un graphe qui a les mêmes connections que `g`. Cette fonction comporte différemment si le graphe est orienté ou pas. Si le graphe est orienté, le stockage des connexions est efficace et il n'y a pas d'arrêt "rédondant", comme c'est le cas pour les graphes

non-orientés. Comme expliqué au-dessus, les graphes non-orientés enregistre des arrêts allant dans les deux sens pour une seule connexion. Cela nous permet de traverser un `Graph` peu importe le côté duquel on commencé d'une connexion. Alors, pour créer un fichier dot qui exporte un graphe non-orienté, on devrait tout d'abord enlever ces arrêtes en rab. On réalise cela avec la fonction `removeMirrorConnection`.

Après un appel à `createDot(g, "graph2.dot")`, on contraste l'exportation en dot avec sa représentation interne en Figure 1.

(a) graph-2.txt

(b) graph2.dot

Figure 2 – L'exportation en `dot` d'un `Graph` non-orienté. L'exportation pour un graphe orienté a tous les arrêts indiquées avec −− remplacés par − >

Enfin, on visualise le graphe avec le logiciel `dot`.

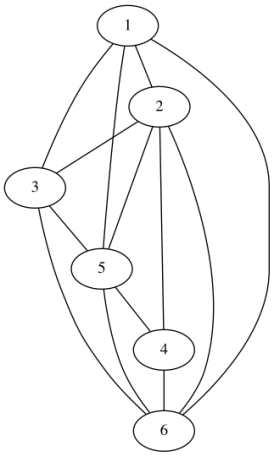


Figure 3 – `graph2.dot` en png.

On constate que (contrairement à une version précedente du code qui n'implémentait pas `removeMirrorConnection`) il n'y a pas d'arrêts doubles et que le graphe en Figure 3 correspond bien à celui défini par `graph-2.txt` en Figure 1

3. Implémenter une procédure permettant de visualiser l'arbre dfs d'un graphe donné en entrée.

J'ai interpreté les consignes libéralement et j'ai décidé que "visualiser l'arbre dfs" veut dire "afficher à l'écran les sommets que l'on visite lorsqu'on effecte un parcours dfs". Une autre version du code aurait peut-être exporté un fichier `.dot` pour chaque pas du parcours. La version du code au moment de rediger ce rapport ne le fait pas. On peut alors visualiser un parcours dfs d'un arbre à partir d'un sommet et un graphe donnés en argument à la procédure `dfsVisualize`.

Si on débarque à partir du sommet 5 du graphe2, on visite les noeuds dans l'ordre suivant :

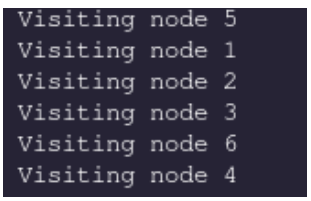


Figure 4 – dfs viz

A partir du sommet 5, on pourrait se déplacer vers les sommets adjacentes : 1, 2, 3, 4, ou 6. Comme le lien entre 5 et 1 a été établit avant les autre connexions, l'algorithme choisit de visiter le sommet 1. Après avoir visité le sommet 1, on a deux stratégies à considérer. Le premier, le DFS, consiste à continuer à déplacer vers le enfants du sommet 1, pour continuer en profondeur. L'autre stratégie, le BFS, consiste à visiter les enfants du premier sommet, 5, avant

de procéder aux autres descendants. Etant donné que cela est une implémentation de la stratégie DFS, on visite les enfants de 1 dans un premier temps, voilà donc l'explication pour laquelle on se déplace vers 2, le premier élément relié avec 1. La stratégie se répète récursivement.

4. Implémenter une procédure testant si un graphe est connexe.

L'implémentation d'une procédure pour voir si un graphe est connexe ou pas est assez simple à concrétiser. On commence un DFS à n'importe quel sommet du graphe et on compte le nombre de sommets que l'on visite pour la première fois. A la fin du parcours, si le nombre de sommets visités est égal au nombre de sommets qui existe dans le graphe, alors on a visité tous les noeuds et le graphe est connexe.

Ce comportement est Implémenté par la fonction `isConnected` dont la déclaration est :

```
bool isConnected(const Graph *g);
```

5. Implémenter une procédure inversant un graphe orienté donné en argument.

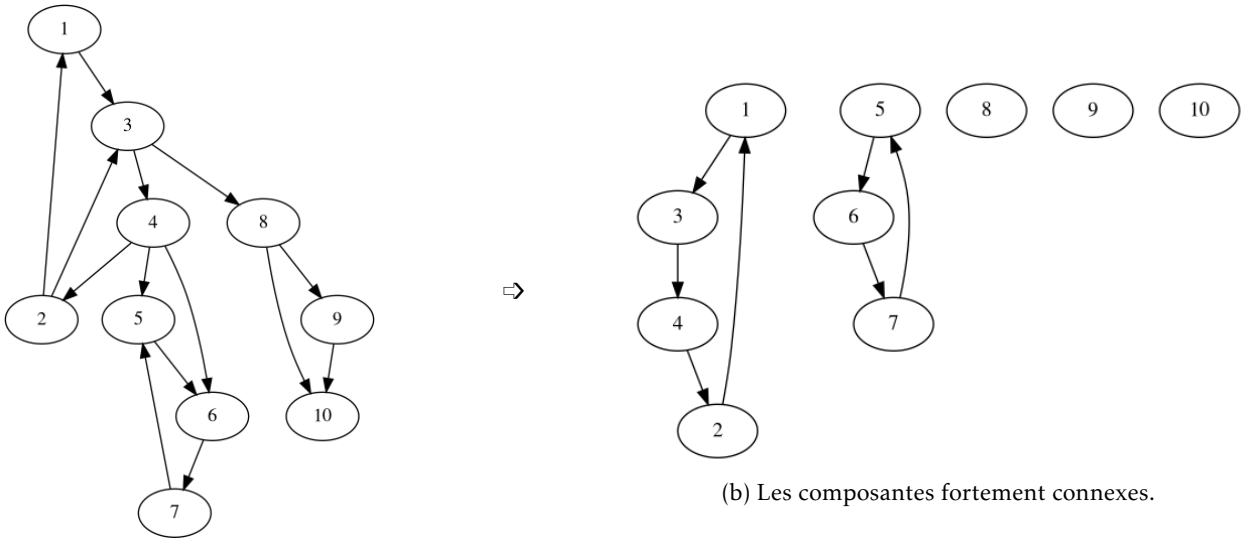
Afin d'implémenter une procédure inversant un graphe orienté, il faudrait tout simplement créer un autre graphe avec le même nombre de sommets. En parcourant le graphe originale, on relie les connexions déjà existentes dans l'autre sens. Pour écrire cela en code, on doit considérer la stratégie de parcourir le graphe. Pour ce faire, on laisse tomber les stratégies singulaire DFS ou BFS et on considère une manière de visiter toutes les connexions du graphe. On pose alors la stratégie de parcourir le *tableau* des listes d'adjacènces, en inversant toutes les connexions qui existe. On peut imaginer qu'on traverse le graphe verticalement par rapport à la representation du graphe2 en partie (a) dans Figure 2. La fonction pertinente est `reverseGraph`.

Pour plus des détails, veuilleuz consulter le code source en `cod/src/graph.c` ou regarder la documentation.

Partie B : Composantes fortement connexes

1. Implémenter un algorithme déterminant les composantes fortement connexes d'un graphe orienté.

Pour déterminer les composantes fortement connexes, on va se servir de deux parcours DFS. La raison pour laquelle on a implémenté la fonction `reverseGraph` c'est parce que le deuxième parcours va s'effecter sur le transpose du graphe dont on veut trouver les composantes fortement connexes. Donc, muni avec la structure d'une pile déclaré en `cod/include/stack.h`, on parcours le graphe dans le bon sens, on rajoute à la pile les sommets qui mènent à un sommet déjà visité ou bien inexistants. Après, on inverse le graphe et en partant du sommet au-dessus de la pile, on effectue encore un DFS pour tomber sur les composantes fortement connexes. Dans son état actuel, la procédure `stronglyConnected` crée un nouveau graphe dont les connexions sont des cycles des sommets existants dans la même composantes fortement connexes. Une visualisation est en ordre.



(a) Graphe définit en digraph-1.txt

(b) Les composantes fortement connexes.

FIGURE 5 – Algorithme qui souligne les composantes fortements connexes. On interprète le graphe à droite pour dire qu'il y a 2 composantes qui sont (pas trivialement) fortements connexes. Une version meilleure du code aurait implémenté une procédure qui crée une visualisation qui préserve les connexions originales du graphes, en soulignant les composantes fortement connexes avec des couleurs distinctes. Néanmoins, cette routine trouve correctement les composantes fortement connexes : {1, 2, 3, 4}, {5, 6, 7}, {8}, {9}, {10}.

2. Analyser la complexité de votre algorithme.

3. Implémenter une procédure permettant d'exporter au format `dot` un graphe donné en argument et indiquant ses composantes fortement connexes.

Comme cette fonctionalité a déjà été implémenté implicitement pour la première question, je propose tout simplement la combinaisons des fonctions : `createDot(stronglyConnected(g), "graph_scc.dot");` Cette combinaison marche parce que la fonction `stronglyConnected` renvoie un graphe dont les connexions sont qu'entre des composantes fortement connexes. On visualise donc ce graphe avec un appel à la fonction `createDot` que l'on a vu auparavant. Justement, cette suite de fonctions nous permet de créer l'image trouvée en Figure 5.

Partie C : Orientation forte d'un graphe

1. Implémenter une procédure testant si un graphe est sans pont. Quelle est sa complexité ?

Avant dimplémenter une fonction qui détermine si un graphe est sans pont, on devrait écrire une fonction qui détermine déjà si un arrêt individu est un pont ou pas. Si on enlève l'arrêt concerné d'un graphe et le graphe résultant n'est pas connexe, alors l'arrêt est par définition un pont. Un graphe entier est donc sans pont si les arrêts sont tous pas de ponts. Le code s'écrit aisement, on parcours tous les arrêts et si il y a un seul entre eux qui soit un pont, notre fonction `hasBridge` renvoie vrai.

2. Démontrer qu'un graphe G est fortement orientable si et seulement si G ne possède pas de pont.
3. Implémenter une procédure déterminant une orientation forte d'un graphe non orienté sans pont. Analyser sa complexité.
4. Exécecuter l'algorithme