

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, et voici le répertoire 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.

```
Instances > graph-2.txt
archstation, 3 weeks ago
1 6
2 13
3 1 2
4 1 3
5 1 5
6 1 6
7 2 3
8 2 4
9 2 5
10 2 6
11 3 5
12 3 6
13 4 5
14 4 6
15 5 6
```

(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 révèle. Pour les graphes non-orientés, j'ai décidé que chaque fois que l'on relie des sommets, je rajoute deux nouvelles arrêts qui vont dans les deux sens. C'est comme si implicitement je rajoutais deux connections orientés. La justification pour cela c'est quand il s'agit de parcourir un graphe par DFS. Concentrons sur les sommets 1 et 6. Sur ligne 6 de `graph-2.txt`, on relie les deux sommets 1 et 6. On voit cet arrêt stocké dans le graphe dans les deux sens; un sens $1 \rightarrow 6$ et l'autre sens $6 \rightarrow 1$. Imagine si on n'avait pas cette connexion allant dans le deuxième sens. Si on commence un DFS à partir du sommet 6, on va pas reconnaître que 1 est adjacent à 6. Cela ne semble pas grave pour cet exemple, mais il devient clair quand on considère le cas où 6 est adjacent au sommet 1 et qu'il n'a plus de voisin. Dans ce cas-là un DFS qui démarre de 6 s'arrête immédiatement comme il semble qu'il n'y a plus de chemins. Le cas où le graphe est orienté, on crée un seul arrêt entre les deux sommets dans le sens propre.

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 connexions 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 "rédundant", 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 arrêts 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.



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.

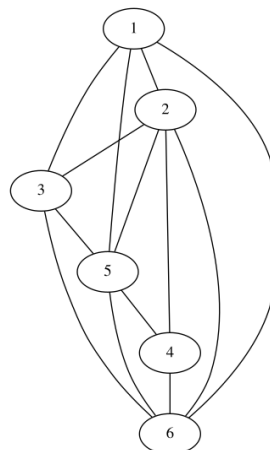


FIGURE 3 – graph2.dot en png.

On constate que (contrairement à une version précédente 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 interprété 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 effectue 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 rédiger 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 :

```
Visiting node 5
Visiting node 1
Visiting node 2
Visiting node 3
Visiting node 6
Visiting node 4
```

FIGURE 4 – Visualisation d'un parcours DFS.

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 établi avant les autres 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 les 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. Étant 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. À 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 original, on relie les connexions déjà existantes 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 singulières 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'adjacences, en inversant toutes les connexions qui existent. On peut imaginer qu'on traverse le graphe verticalement par rapport à la représentation du graphe2 en partie (a) dans Figure 2. La fonction pertinente est `reverseGraph`.

Pour plus des détails, veuillez 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'effectuer sur le transpose du graphe dont on veut trouver les composantes fortement connexes. Donc, muni avec la structure d'une pile déclarée en `cod/include/stack.h`, on parcourt 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 composante fortement connexe. Une visualisation est en ordre.

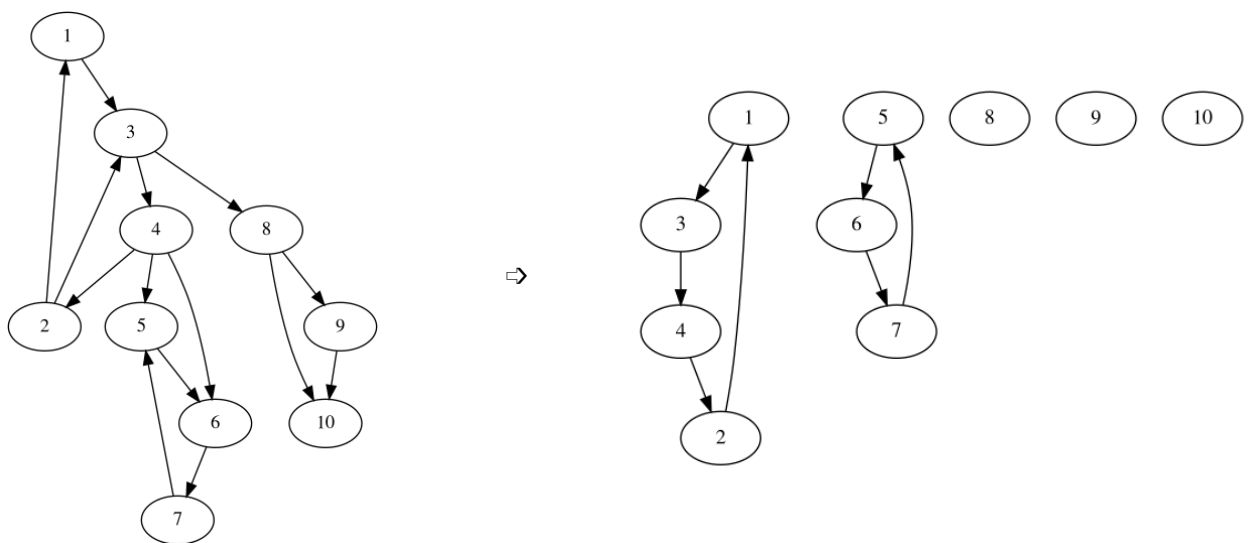


FIGURE 5 – Algorithme qui souligne les composantes fortement connexes. On interprète le graphe à droite pour dire qu'il y a 2 composantes qui sont (pas trivialement) fortement connexes. Une version meilleure du code aurait implémenté une procédure qui crée une visualisation qui préserve les connexions 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.

Cet algorithme consiste en deux appels à un parcours DFS, ce qui a une complexité de $O(V + E)$. Entretemps l'algorithme utilise une pile dont les actions associées ont une complexité de $O(V)$. Alors la complexité de cet algorithme est $2 * O(V + E) + O(V) \in O(V + E)$

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

Comme cette fonctionnalité a déjà été implémentée implicitement pour la première question, je propose tout simplement la combinaison 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.

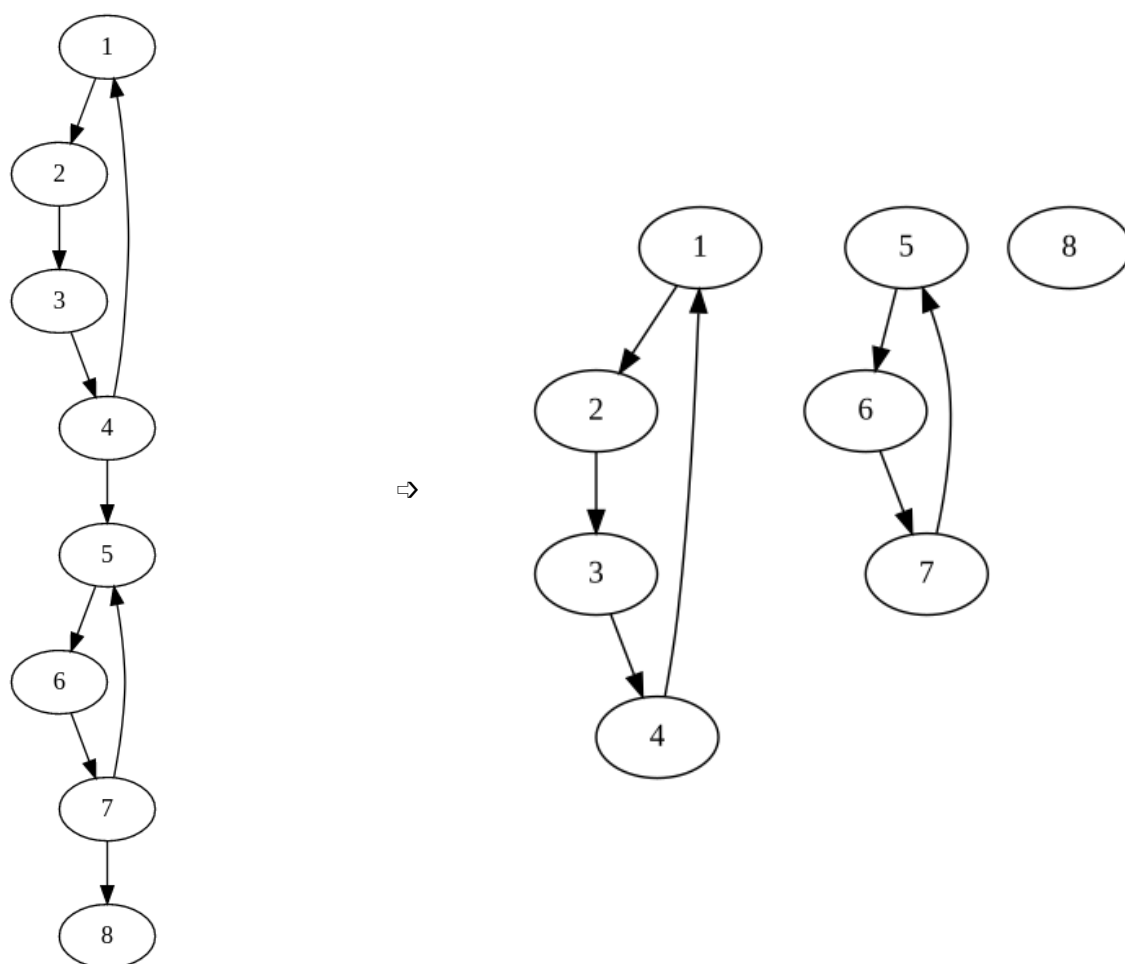


FIGURE 6 – Application de `stronglyConnected` avec un autre graphe qui est simple. L'image à droite souligne les composantes fortement connexes du graphe à gauche.

Une version future du code pourra utiliser des couleurs différentes dans la génération d'un fichier `.dot` pour distinguer les composantes fortement connexes.

Partie C : Orientation forte d'un graphe

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

Avant d'implémenter 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 aisément, on parcourt tous les arrêts et si il y a un seul entre eux qui soit un pont, notre fonction `hasBridge` renvoie vrai.

Pour déterminer sa complexité on considère le fait qu'on devrait vérifier si *chaque* arrêt est un pont. Donc, on devrait appeler la fonction `isBridge` E fois. La fonction `isBridge` teste si un arrêt est un pont en quittant la connexion et vérifiant si le graphe qui en résulte est connexe ou pas. Finalement, la procédure pour déterminer si un graphe est connexe ou pas consiste d'un seul parcours DFS. On conclut alors que `isBridge` consiste d'un parcours DFS et donc est de classe de complexité $O(V + E)$, et que la procédure `hasBridge` consiste de E appels d'une procédure de complexité $O(V + E)$. Alors, la complexité de la procédure testant si un graphe est sans pont est $O(E(V + E))$.

2. Démontrer qu'un graphe G est fortement orientable si et seulement si G ne possède pas de pont.

Pour démontrer cela, on suppose que G est fortement orientable. Par définition, cela veut dire qu'il existe une orientation fortement connexe de G . De plus, cela implique que pour deux sommets quelconque u et v , il existe un chemin de u vers v et un chemin de v vers u . Si on imagine le scénario où on enlève n'importe quel arc du graphe G , on peut - au pire - casser l'un des chemins entre u et v . Vu qu'un arc a une et seulement une orientation, il est impossible que le chemin en avance $u \rightsquigarrow v$ et le chemin $v \rightsquigarrow u$ en revers traverse l'arrêt qu'on va enlever. Autrement dit, comme un arc a un seul sens, si on enlève un arc quelconque on aura **toujours** un graphe qui reste quand même *connexe*, mais pas forcément *fortement connexe*. Par conséquence, G ne contient pas de pont et on a démontré l'égalité dans la direction en avant.

Pour attaquer la deuxième direction, on suppose que G ne possède pas de pont. Alors, on peut toujours supprimer un arrêt du G sans le priver d'être connexe. Il existe alors une orientation telle que pour deux sommets quelconque u et v , il existe un chemin de u vers v ou un chemin de v vers u . Cela est une conséquence immédiate de la connectivité du graphe. Pour vérifier une orientation forte, on doit garantir qu'il existe un chemin dans les deux sens! Comme il n'y a aucun pont, il y a toujours un arc "en extra" pour aller dans l'autre sens et compléter le cycle d'un graphe.

On a donc vérifié l'égalité dans les deux sens.

3. Implémenter une procédure déterminant une orientation forte d'un graphe non orienté sans pont. Analyser sa complexité.

Une version future du code qui sera mise à jour sur github implémentera cette routine.

4. Exécuter l'algorithme

Une version future du code qui sera mise à jour sur github testera cette procédure.

Remarques : Repo github

Une version minimale du répertoire sera inclu dans le zip à rendre sur moodle. Pour voir le répertoire complet et mis à jour, consultez le github. Les procédures basiques qui sont utilisées dans ce TP sont testées au moment de la construction avec le logiciel CTest et CMake. Si jamais cela vous intéresse, vous pouvez construire le projet en suivant les instructions sur le README .md du projet. De la documentation technique a été générée par Doxygen et de la documentation du projet qui est plus générale et qui contient des exemples pour montrer l'API a été construite avec Mkdocs et hébergé par readthedocs.