# Rapport du travail effectué sur les TP1 et 2 d’algorithmique générale

J’ai choisi pour ce projet de créer une forge pour pouvoir travailler chez moi et à l’école. De plus vu la taille du programme, j’ai choisis de programmer de façon modulaire afin que les temps de compilation soient réduits. J’ai donc écris un MakeFile que vous trouverez en annexe.

## Partie A : Préliminaire

J’ai choisi comme structure d’implémenter un graphe sous ses trois formes : une matrice d’adjacence, une liste pi et alpha et une liste de successeurs.

typedef struct Graphe{  
 int nb\_aretes;  
 int nb\_sommets;  
 int oriente;  
 int \*pi;  
 int \*alpha;  
 int \*\*liste\_adjacence;  
 int \*\*liste\_successeurs;  
}Graphe;

La première partie du TP était assez simple à implémenter, jusqu’au dfs, qui lui m’a pris beaucoup de temps. De plus la question sur la connexité a du être retravaillée pour répondre aux attentes du sujet. La compléxité de chacune des fonctions sera détaillée dans la dernière partie.

### Question 1 :

La fonction Graphe\* create\_from\_file(char nom\_de\_fichier[]) permet d’implémenter ma structure graphique à partir d’un graphe décrit en fichier texte. Cette fonction implémente tout d’abord la matrice d’adjacence, qui est plus facile à manipuler. Ensuite la fonctiuon void initialize\_all(Graphe \*graph) initialise les valeurs de pi, alpha et de la matrice des successeurs. Lorsque l’on execute le main, on peut entrer à l’aide d’un scanf, le nom du graphe que l’on souhaite étudier par exemple digraph-1.txt. Une fois ses représentations initialisées, on peut travailler avec le graphe.

### Question 2 :

Afin de répondre à cette question, j’ai utilisé la matrice d’adjacence, qui me paraissait plus simple à utiliser. A partir de celle-ci, je crée un fichier .dot en utilisant la syntaxe universelle dot. Je pourrais pour améliorer l’efficacité de cette fonction en utilisant la liste d’adjacence, mais je n’ai pas eu le temps de revenir dessus.

### Question 3 :

J’ai écrit 5 fonctions de dfs qui ont chacune une utilité particulière. Cependant la forme des ces 5 fonctions est toujours la même : la fonction void dfs(Graphe \*graphe,int sommet\_depart ,liste\_ordre \*\*ordre\_par\_sommet, int \*\*sommet\_marque\_dugraphe) que j’ai baptisé “le gros dfs” utilise la fonction void dfss(Graphe \*graphe,int sommet, int \*sommet\_marque, liste\_ordre \*ordre) que j’ai surnommée “le petit dfs”. Le petit dfs prend en entrée un sommet, dont il va parcourir le premier successeur en profondeur pour ensuite parcourir ses autres successeurs aussi en profondeur. Pour chaque sommet parcouru, le sommet est marqué et le sommet est aussi ajouté à l’ordre de parcours du dfs qui est stocké sous la forme d’une liste chainée. De plus, l’un des dfs permet de créer l’arbre dfs, ou l’arbre de parcour suprimant les arretes non utilisées par le parcour. Sur les 5 dfs codé, 4 utilisent la matrice pi et alpha, qui permettent d’avoir la meilleure implémentation possible. La forme des tableaux d’ordre renvoyés par les dfs s’affiche sous la forme :

[1->2->3->4] [5->6->7]

L’un des petits dfs utilise la matrice d’adjacence qui est nécessaire pour trouver l’orientation forte d’un graphe non orienté.

### Question 4 :

Afin de teste si un graphe est connexe, j’ai juste à utiliser la fonction de dfs que j’ai implémentée précédement. Si le graphe donné en entrée est orienté, je dois le “dés-orienter” le rendre non orienté car l’ordre n’est pas important pour la connexité. Ensuite il ne me reste plus qu’à lancer un dfs et si le tableau des ordre de parcour renvoyé par mon dfs contient deux ordre de parcours, cela veut dire que à partir d’un des sommets, le grand dfs ne parvient pas à marquer un autres des sommets et doit donc relancer le petit dfs sur celui ci. Cela implique qu’un sommet est donc isolé et que la graphe n’est donc pas connexe. La condition de connexité est donc très simple à vérifier.

### Question 5 :

L’inversion du graphe orienté se fait en utilisant la matrice d’adjacence - c’est la fonction la plus couteuse en terme de mémoire - et la fonction void initialize\_all(Graphe \*graph) pour ensuite mettre à jour le graph obtenu.

## Partie B : Composantes fortement connexes

Cette partie du TD à été la plus longue et laborieuse : c’est nottement pour cette partie que j’ai du écrire plusieurs dfs.

### Question 1 & 3:

Pour déterminer les composantes fortement connexes j’ai utilisé l’algorithme de kosaraju, qui utilise un premier dfs (pu l’on ajoute le sommet parcouru qu’une fois qu’on a parcouru tous ses successeurs) puis un deuxième dfs appliqué à l’inverse du graphe en utilisant l’ordre de parcour inverse obtenue lors du premier dfs. Le tableau d’ordre retourné par ce deuxième dfs contient les composantes fortement connexes de ce graphe. A partir de ce tableau, je crée un deuxième graphe, contenant en sous graphe les composantes fortement connexes. J’ai aussi utilisé l’hypothèse qu’une composante toute seule est fortement connexe. Voici le graphe rendu par mon algorythme, appliqué sur le digraph-1.



Image

### Question 2 :

La complexité de cette algorithme est : O(N²) à cause de l’invertion. Sans cette étape d’invertion, la complexité est linéaire.

### Question 3 :

’cf question 1 pour la premi’ai du écrire une nouvelle fonction appelée void create\_dot2(Graphe \*graphe, char nom\_de\_fichier[],liste\_ordre \*\*composantes, int nbc); qui prend donc en entrées les composantes fortement connexe et re-écrit le graph, en utilisant une petite astuce: toutes les sommets des composantes fortement connexes sont simplement placées dans un sous graphe, sans aucune liaisons. Ci après le fichier.dot et .ps rendu par l’algorithme après exécution sur le digraph-1.

digraph premier\_graph {  
 subgraph cluster\_0 {  
 3;  
 1;  
 2;  
 4;  
   
 }  
 subgraph cluster\_1 {  
 8;  
 } subgraph cluster\_2 {  
 10;  
 } subgraph cluster\_3 {  
 9;  
 } subgraph cluster\_5 {  
 5;  
 7;  
 6;  
   
 }  
1 -> 3 ;  
2 -> 1 ;  
2 -> 3 ;  
3 -> 4 ;  
3 -> 8 ;  
4 -> 2 ;  
4 -> 5 ;  
4 -> 6 ;  
5 -> 6 ;  
6 -> 7 ;  
7 -> 5 ;  
8 -> 9 ;  
8 -> 10 ;  
9 -> 10 ;  
}

![Image](data:application/pdf;base64,)

Image

## Partie C : Orientation forte d’un graphe

Partie assez facile seulement, la dernière question n’est pas implémentée de la bonne manière.

### Question 1 :

La procédure utilise la matrice d’adjacence pour supprimer chacunes des arretes. Ensuite on vérifie si le graphe donné en entreé est encore connexe ou pas. La complexité de cette fonction est polynomiale : O(N²). Je n’ai pas trouvé de méthode linéaire permettant la suppression d’une arrete.

### Question 2 :

A faire

### Question 3 :

On doit tout d’abord vérifier que le graphe ne possède pas de ponts, ensuite on effectue une procédure que j’ai trouvé sur Wikipédia, qui conciste à effectuer un dfs à partir d’un sommet, pour ensuite orienter toutes les arretes des sommets parcouruts dans le sens du dfs, et toutes les arretes restantes vers la racine, donc dans l’autre sens.

### Question 4:

Vous trouverez ci joint une orientation du graphique K proposé dans l’énoncé ainsi qu’une autre orientation du graphique H.