Sujet du mini-projet fil rouge

Algorithmique avancée et Programmation

Table des matières

Organisation

Chaque groupe rendra au plus tard au début de la demi-journée de soutenance qui le concerne (8:00 ou 13:30) une seule archive sur Moodle, contenant :

- Les fichiers sources du programme produit, accompagnés d'un fichier indiquant les commandes de compilation (texte ou Makefile) ;
- Le diaporama qui sera présenté lors de la soutenance la semaine de la rentrée, de préférence au format PDF ;
- Un compte-rendu rédigé (avec introduction, développement, conclusion, perspectives et bibliographie) de 15 pages maximum présentant :
  - l'organisation des programmes rendus,
  - les structures de données manipulées,
  - le fonctionnement de chaque programme,
  - les choix effectués par rapport au sujet,
  - les performances des programmes,
  - l'organisation du groupe (qui a fait quoi, avec quel planning, etc.),
  - les difficultés rencontrées et les aspects de l'énoncé qui n'ont pas pu être traités.

V2

Des soutenances seront organisées à la rentrée des congés de fin d'année. Elles auront lieu en groupe TD, avec un jury composé de deux enseignants. Chaque soutenance durera **25** minutes : **15** minutes de présentation, 10 minutes de questions. Ces soutenances reprendront les éléments de votre compte-rendu et vous permettront de présenter votre travail au reste du groupe. Il est encouragé de faire une démonstration de vos programmes pendant la soutenance, mais prenez garde à bien préparer cette démonstration pour ne pas perdre de temps. L'évaluation du fil rouge portera sur le code source (portion de l'énoncé traité, qualité de la livraison), le compte-rendu et la soutenance.

Types de données pour les graphes

Cette section donne un certain nombre d'éléments d'implémentation. En fonction de votre aisance sur le sujet, vous êtes tout à fait libres de modifier ces éléments à votre guise. Pensez à le mentionner dans le rapport.

Le but de ce projet est d'implémenter les graphes sous forme de matrice d'adjacence avec allocation dynamique, et sous forme de listes d'adjacence. Pour chacun des programmes ci-dessous, vous devrez donc fournir deux versions du programme.

Il n'est pas demandé d'implémenter les graphes sous forme de matrice d'adjacence avec allocation statique (tableau de taille fixe).

Code fourni : listes chaînées, piles, etc.

Un certain nombre d'éléments de code sont fournis dans un fichier.

En premier lieu, on y définit les types représentant un sommet de graphe et un booléen :

```
typedef int t_bool;     // Booléen
typedef int t_vertex;   // Sommet de graphe
```

Suivent les définitions de deux types déjà étudiés en cours : les listes chaînées et les piles (définies grâces aux listes chaînées) qui seront utiles dans la suite. Les valeurs portées par ces deux types sont les sommets de graphe (**`t_vertex`**).

```
// Maillon de liste chaînée
typedef struct node {
  t_vertex val;           // Valeur (sommet)
  struct node * p_next;   // Pointeur vers le maillon suivant
} t_node;
// Liste chaînée (de sommets)
typedef t_node * t_list;
// Pile (de sommets)
typedef t_list t_stack;
```

V2

Parcours de liste chaînée

Toutes les fonctions nécessaires à l'utilisation des listes chaînées et des piles sont aussi fournies. Leurs noms on pu changer par rapport aux TD et TP pour plus de clarté. Vous avez évidemment la liberté d'en ajouter de nouvelles si vous le jugez nécessaire.

De nouvelles fonctions sur les listes chaînées ont aussi été ajoutées pour rendre leur parcours plus simple, et conserver l'aspect type abstrait de données. Pour cela, nous utilisons la notion de *curseur de liste*, qui représente en fait un pointeur vers un maillon. Pour parcourir une liste, le curseur est initialement créé sur le premier maillon, et à chaque itération il est déplacé sur le maillon suivant.

Type	Nom	Description
Parcours	`list_cursor_new(L) → LC`	Crée un *curseur de liste* **`LC`** permettant de parcourir la liste chaînée **`L`**
Parcours	`list_cursor_at_end(LC) → bool`	Retourne 1 si le curseur **`LC`** a atteint la fin de la liste, 0 sinon
Parcours	`list_cursor_next(LC) → LC'`	Passe le curseur **`LC`** au maillon suivant.

Les prototypes et définitions de ces fonctions sont donnés dans le programme. Typiquement, un parcours de liste chaînée **`L`** doit ressembler à ceci :

```
t_note * lc;
lc = list_cursor_new(L);
while (!list_cursor_at_end(lc)) {

  // Ici : traitement du maillon courant...

  lc = list_cursor_next(lc);
}
```

Ce type de parcours de liste chaînée sera particulièrement utile pour les graphes représentés par listes d'adjacence.

Matrice d'adjacence avec allocation dynamique

Pour définir un graphe comme matrice d'adjacence avec allocation dynamique, le type de données est le suivant :

```
typedef struct {
  int size;     // Taille
  t_bool ** m; // Contenu
} t_graph;
```

Dans ce type, **`size`** représente le nombre de sommets du graphe, et **`m`** est la matrice d'adjacence : « **`t_bool ** m`** » permet de stocker un tableau de tableaux de booléens, ce qui revient à un tableau à deux dimensions.

Il est nécessaire d'allouer l'espace mémoire pour le graphe ainsi que pour sa matrice. Cela peut se faire de la façon suivante :

```c
int i;
t_graph * p_g;
// Allocation de la structure t_graph
p_g = malloc(sizeof(*p_g));
assert(p_g != NULL);
p_g->size = nb_nodes;  // Où nb_nodes est le nombre de sommets
// Allocation de la première dimension de la matrice d'adjacence
// (tableau de tableaux de booléens)
p_g->m = malloc(p_g->size * sizeof(*(p_g->m)));
assert(p_g->m != NULL);
// Allocation de la deuxième dimension de la matrice d'adjacence
// (chaque élément est un tableau de booléens)
for (i = 0; i < p_g->size; i++) {
  p_g->m[i] = malloc(p_g->size * sizeof(*(p_g->m[i])));
  assert(p_g->m[i] != NULL);
}
```

Une fois cela fait, « **`p_g->m[i][j]`** » permet d'accéder à l'indice (*i*, *j*) de la matrice. Il est à noter qu'après le code ci-dessus, la structure est bien allouée en mémoire, mais que la matrice d'adjacence n'est pas initialisée. Une seconde étape consisterait donc à l'initialiser à 0.

Listes d'adjacence

Pour définir un graphe comme listes d'adjacences, il est nécessaire d'utiliser le type des listes chaînées défini plus haut. Nous définissons ensuite un graphe de la façon suivante :

V4

```c
typedef struct {
  int size;   // Taille
  t_list * l; // Contenu
} t_graph;
```

Dans ce type, « **`t_list *`** » permettra de stocker un tableau de listes chaînées : chaque élément du tableau sera une tête de liste, c'est-à-dire un pointeur vers le premier maillon de la liste chaînée (ou le pointeur **`NULL`** pour une liste vide). À nouveau, il est nécessaire d'allouer l'espace mémoire avant de l'utiliser :

```c
int i;
// Allocation de la structure t_graph
t_graph * pg = malloc(sizeof(*pg));
assert(pg != NULL);
pg->size = nb_nodes;  // Où nb_nodes est le nombre de sommets
// Allocation du tableau de têtes de listes d'adjacence
pg->l = malloc(pg->size * sizeof(*(pg->l)));
assert(pg->l != NULL);
```

À nouveau, ce code permet d'effectuer l'allocation en mémoire mais pas son initialisation : il faut maintenant parcourir le tableau `p_g->l` pour initialiser chaque cellule comme étant une nouvelle liste vide.

Type de données abstrait

Il est recommandé de développer les types représentant les graphes dans des fichiers séparés, et de les traiter comme des type de données abstrait, c'est-à-dire en développant les fonctions nécessaires à toutes les opérations que vous pourriez vouloir faire dessus : création d'un nouveau graphe, lecture depuis un fichier, ajout d'un arc, vérification de la présence d'un arc, affichage du graphe dans le terminal, écriture au format DOT dans un fichier, recherche des CFC, etc.

Formats des fichiers de graphes

La définition de graphes « à la main » ou en demandant à l'utilisateur de saisir des valeurs permet de faire des tests. Cependant, pour des graphes de plus de quelques sommets, cela n'est plus envisageable. Il est donc attendu que vos programmes puissent lire un graphe stocké dans un fichier selon un format précis.

Deux types de formats de graphes sont proposés, et les exemples de graphes fournis avec l'énoncé respectent un de ces deux formats. Le premier format encode les sommets du graphe sous forme de numéros (entiers commençant à zéro) tandis que le deuxième format leur attribue des noms (chaînes de caractères).

Dans les deux formats, une ligne vide dans le fichier doit être ignorée, où qu'elle se trouve.

Vous devez choisir si vous souhaitez traiter le format 1, le format 2, ou les deux.

Format 1 : Par numéros

Le format 1 est le suivant :

- une ligne contenant le nombre N de sommets du graphe,
- puis une liste d'arcs de taille non déterminée : chaque arc est codé avec le numéro du sommet prédécesseur et le numéro du sommet successeur, séparés par un caractère d'espacement ou de tabulation (`'\t'`).

Un numéro est donc ici un entier en 0 et $N - 1$.

Voici un exemple de fichier dans ce format :

```
4

0 1
0 2
2 1
2 3
3 2
```

Format 2 : Par noms

Le format 2 est le suivant :

- une ligne contenant le nombre *N* de sommets du graphe, puis un caractère d'espace et le caractère **`n`** qu'il n'est pas nécessaire de lire mais qui permet de distinguer les fichiers par noms des fichiers par numéros si nécessaire,
- la liste des noms des sommets sur *N* lignes,
- une liste d'arcs de taille non déterminée : chaque arc est codé avec le nom du sommet prédécesseur et le nom du sommet successeur, séparés par un caractère d'espacement ou de tabulation (**`'\t'`**).

Voici un exemple représentant le même graphe que ci-dessus au format 2 :

```
4 n

start
stop
working
paused

start stop
start working
working stop
working paused
paused working
```

Pour ce format, il sera nécessaire d'appliquer quelques modifications au traitement du graphe :

- Il faudra ajouter un champ permettant de stocker la liste des noms dans la structure du graphe,
- À la lecture des arcs du graphe depuis un fichier, il faudra effectuer des recherches dans le tableau de noms pour retrouver les numéros de sommets correspondants.

Deux types de recherche sont possibles, au choix :

  - de façon naïve avec un parcours de tableau classique, ce qui est assez lent,
  - ou avec une recherche dichotomique[1] pour accélérer le processus, mais cela nécessite d'avoir trié le tableau (cf. TP2).

1 Voir par exemple : https://fr.wikipedia.org/wiki/Recherche_dichotomique

- Il faudra aussi prendre soin d'afficher les noms des sommets plutôt que les numéros à chaque fois que ça sera nécessaire (conversion du graphe en DOT, énumération des CFC…).

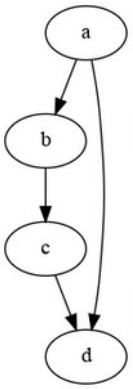
1. Conversion d'un graphe au format DOT

Le premier programme doit permettre de changer un graphe pour ensuite l'écrire dans un fichier au format DOT. Voici un exemple de fichier DOT :

```
digraph nom_du_graphe {
  a -> b;
  b -> c;
  a -> d;
  c -> d;
}
```

Avec le format 1 (par numéros) il est seulement demandé d'afficher les numéros de sommets dans le fichier DOT ; pour le format 2, il faut plutôt utiliser le noms.

Il est ensuite possible de traduire ce fichier en image représentant le graphe à l'aide d'un logiciel comme Graphviz. Ce logiciel peut être utilisé en ligne[2] ou installé sur votre ordinateur sous Ubuntu en tapant dans le terminal :




```
sudo apt install graphviz
```

Cette commande ajoute le programme **`dot`** sur votre machine et on peut ensuite générer une image à partir d'un fichier DOT en tapant la commande suivante dans le terminal :

```
dot fichier_entree.dot -T png -o fichier_sortie.png
```

Un exemple de résultat est donné à droite.

Arguments acceptés par le programme

Votre programme devra accepter les options suivantes, dans n'importe quel ordre :

- **`-i`** suivi du nom du fichier de graphe à lire ; si omis, le programme lit sur l'entrée standard (**`stdin`**),
- **`-o`** suivi du nom du fichier DOT à générer ; si omis, le programme écrit sur l'entrée standard (**`stdout`**).

Un exemple d'appel à votre programme serait donc :

```
./fil_rouge_1 -o ex1.dot -i ex1.txt
```

Le résultat de cette exécution serait la lecture du fichier **`ex1.txt`** (selon un des deux formats décrits ci-dessus) et la création d'un fichier **`ex1.dot`** contenant le même graphe au format DOT.

2 Par exemple ici : https://dreampuf.github.io/GraphvizOnline

Si le programme est appelé avec des arguments qui ne sont pas reconnus, alors le programme doit afficher un message détaillant le problème et les arguments possibles, puis s'arrêter. Par exemple, si un utilisateur donne un argument invalide « **`-a`** », le message pourrait être :

```
Argument non reconnu : -a
Utilisation :
  ./fil_rouge_1 [-i <fichier_graphe>] [-o <fichier-dot>]
Arguments :
  -i <fichier_graphe> : le fichier graphe à lire
  -o <fichier_dot> : le fichier dot à produire
```

V2

2. Recherche d'un chemin

Le deuxième programme doit permettre la recherche d'un chemin entre deux sommets donnés dans un graphe (un sommet de départ et un sommet cible) en affichant l'ordre de passage des sommets sous la forme :

```
sommet_départ -> sommet_1 -> sommet_2 -> sommet_cible
```

On ne s'intéressera pas à l'énumération de tous les chemins possibles ni à la recherche du plus court chemin : on cherchera seulement à trouver un exemple de chemin, s'il existe, et à afficher les sommets traversés dans l'ordre. Pour cela, il est nécessaire de mettre en œuvre un *parcours de graphe en profondeur*.

Parcours de graphe en profondeur

Un parcours de graphe en profondeur consiste à parcourir l'ensemble du graphe en suivant les arcs entre les sommets. Pour cela, pour chaque sommet visité, on extrait la liste de ses successeurs et on les visite à nouveau un à un, donc on visite encore le successeurs, etc. avant de revenir au sommet initial. On marque chaque sommet visité pour éviter de le visiter à nouveau en cas de cycle dans le graphe.

Lorsqu'on a atteint le sommet recherché, on peut alors arrêter le traitement et remonter les arcs jusqu'à sommet initial. Cependant, si on affiche les sommets à ce moment-là, ils seront affichés dans l'ordre inverse : depuis le sommet cible jusqu'au sommet de départ. Une solution consiste à utiliser une pile pour empiler les sommets depuis le sommet cible jusqu'au sommet de départ, puis à les dépiler pour les afficher dans l'ordre inverse.

Si un chemin est trouvé, on souhaite afficher les sommets dans l'ordre, du sommet de départ jusqu'au sommet cible.

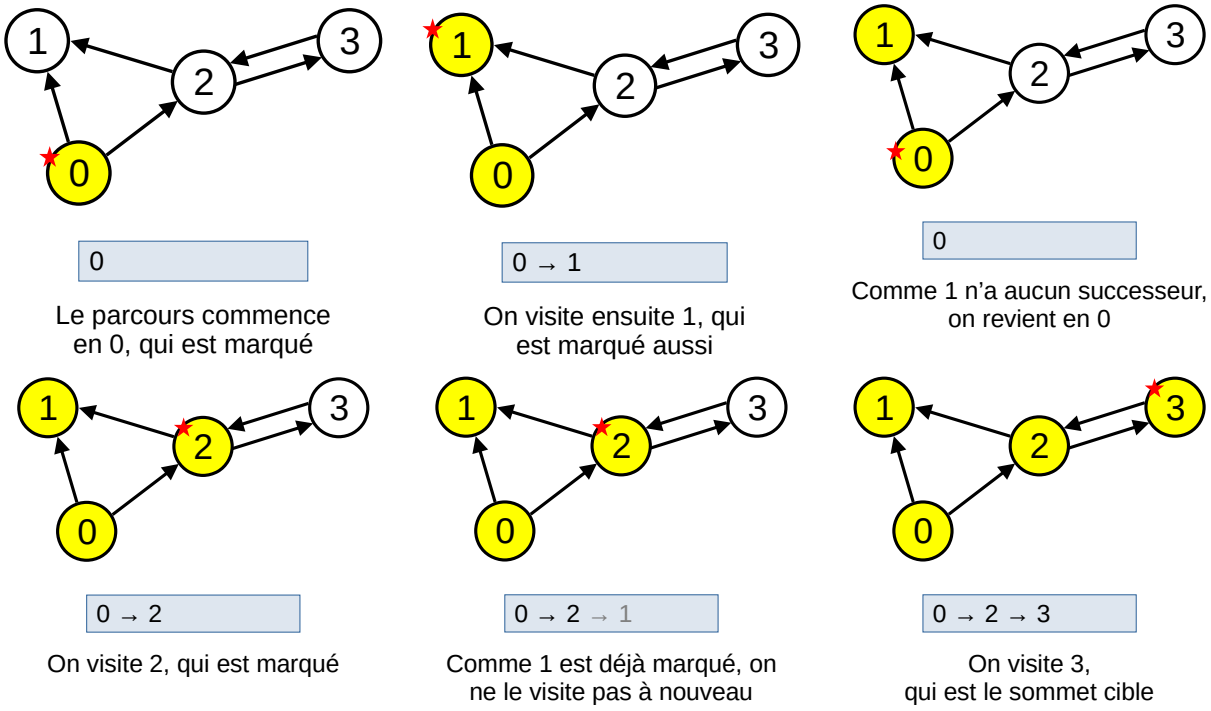
Si vous le souhaitez, vous pouvez aussi implémenter l'affichage de l'ordre inverse des sommets, du sommet cible jusqu'au sommet de départ. Dans ce cas, vous devrez réfléchir à une façon de faire en adaptant les algorithmes donnés plus bas.

Parcours itératif et parcours récursif

Il existe deux algorithmes de parcours de graphe :

- L'algorithme récursif utilise une fonction qui, appelée sur un sommet, se rappelle elle-même sur chacun des sommets successeurs. Cette approche est plus simple à écrire car très bien adaptée aux graphes, mais un peu plus complexe à comprendre.
- L'algorithme itératif utilise une simple boucle pour parcourir les sommets, ainsi que des piles pour retenir les sommets à visiter et le chemin découvert. Cette approche est plus longue à mettre en place mais évite les appels récursifs.

Dans les deux cas, les sommets peuvent être parcourus dans le même ordre. Voici un exemple de parcours pour le graphe d'exemple du cours, où le sommet de départ est 0 et le sommet cible est 3 ; le sommet qui est actuellement visité est indiqué par une étoile et les sommets marqués ont un fond jaune :



Vous pouvez choisir de n'implémenter qu'un des deux parcours : récursif ou itératif. Si vous décidez de faire les deux, alors ajoutez la possibilité de choisir le type de parcours à l'aide d'un argument en ligne de commande.

Parcours récursif

Une fonction récursive est une fonction qui s'appelle elle-même. Cela est bien adapté au parcours de graphe car la fonction de parcours peut se rappeler sur tous les sommets successeurs, ce qui entraînera encore des appels sur les successeurs…

Une forme typique d'un parcours récursif est :

Parcours récursif générique
Fonction : **Parcours**(**g**, **x**, **marking**)
Arguments :
    g : un graphe de **n** sommets
    x : le sommet de départ
    marking[**n**] : un tableau de booléens permettant de marquer les sommets déjà visités
Variables :
    w : un sommet de **g**
Algorithme :
    Si la fin du parcours est atteinte :
        // Effectuer le traitement de fin de parcours
    Sinon, si **marking**[**x**] = **FAUX**
        marking[**x**] ← **VRAI**
        // Effectuer le traitement préfixe (avant de visiter les successeurs)
        Pour chaque successeur **w** de **x** :
            Parcours(**g**, **w**, **marking**)
        Fin pour
        // Effectuer le traitement suffixe (après avoir visité les successeurs)
    Fin si

Il est aussi possible d'ajouter des arguments ou des variables à la fonction en fonction des traitement à effectuer.

Par exemple, l'algorithme récursif de recherche de chemin est composé de deux fonctions. La première initialise les variables et appelle la seconde fonction. La seconde effectue le parcours en profondeur à partir d'un sommet donné, si le marquage indique qu'il n'a pas encore été traversé, et se rappelle sur chaque successeur du sommet. Elle s'arrête quand elle arrive sur le sommet cible (succès), ou quand elle a visité tous les sommets atteignables (échec).

Recherche de chemin (initialisation de la fonction récursive)
Fonction : **Recherche_recur**(**g**, **x**, **y**)
Arguments :
    g : un graphe de **n** sommets
    x : le sommet de départ
    y : le sommet cible
Variables :
    marking[**n**] : un tableau de booléens permettant de marquer les sommets
    stack : une pile de sommets
Algorithme :
    Initialiser toutes les cellules de **marking** à **FAUX**
    Initialiser **stack** en tant que nouvelle pile vide
    Recherche_recur_f(**g**, **x**, **y**, **marking**, **stack**)

Recherche de chemin (fonction récursive)
Fonction : **Recherche_recur_f**(**g**, **x**, **y**, **marking**, **stack**)
Arguments :
    g : un graphe de **n** sommets
    x : le sommet de départ
    y : le sommet cible
    marking[**n**] : un tableau de booléens permettant de marquer les sommets
    stack : une pile de sommets
Valeur de retour :
    Un booléen indiquant si le sommet cible a été trouvé
Variables :
    w : un sommet de **g**
    trouvé : un booléen
Algorithme :

```
Si x = y :
    Empiler x sur stack
    Retourner VRAI
Sinon :
    Si marking[x] = FAUX
        marking[x] ← VRAI
        Pour chaque successeur w de x :
            trouvé ← Recherche_recur_f(g, w, y, marking, stack)
            Si trouvé est vrai :
                Empiler x sur stack
                Retourner VRAI
            Fin si
        Fin pour
    Fin si
    Retourner FAUX
Fin si
```

Remarque : il est possible de faire des modifications à l'algorithme, par exemple intégrer **marking** et **stack** dans le graphe pour ne pas devoir le passer à chaque appel de fonction. Il est possible aussi d'en faire des variables globales, mais cela est généralement considéré comme une mauvaise pratique de programmation.

Parcours itératif

L'algorithme itératif du parcours en profondeur utilise le même principe que l'algorithme récursif, mais deux piles permettent de remplacer les appels récursifs : **stack_traversal** permet de stoker la liste des sommets suivants à visiter, à la place de l'algorithme récursif, tandis que **stack_path** stocke la progression du chemin découvert depuis le sommet de départ. Enfin, **stack_path_final** permet simplement d'inverser **stack_path** pour afficher le chemin trouvé dans le bon sens.

Recherche de chemin (fonction itérative)
Fonction : **Recherche_iter**(**g**, **x**, **y**)
Arguments :
    g : un graphe de **n** sommets
    x : le sommet de départ
    y : le sommet cible
Variables :
    w : un sommet de **g**
    trouvé : un booléen
    marking[**n**] : un tableau de booléens permettant de marquer les sommets
    stack_traversal : une pile de sommets pour le parcours du graphe
    stack_path : une pile de sommets stockant le chemin en ordre inverse
    stack_path_final : une pile de sommets stockant le chemin dans le bon ordre
Algorithme :
    Initialiser toutes les cellules de **marking** à **FAUX**
    Initialiser **stack_traversal**, **stack_path** et **stack_path_final** en tant que piles vides
    Empiler **x** sur **stack_traversal**
    Tant que **stack_traversal** n'est pas vide :
        x ← Dépiler **stack_traversal**
        Si **x** = **y** :
            Empiler **x** sur **stack_path**
            Quitter la boucle Tant Que
        Sinon
            Si **marking**[**x**] = **FAUX** :
                marking[**x**] = **VRAI**
                Empiler **x** sur **stack_path**
                Empiler –1 sur **stack_traversal**
                Pour chaque successeur **w** de **x** :
                    Empiler **w** sur **stack_traversal**
                Fin pour
            Fin Si
        Fin Si
    Fin Tant Que
    Si **stack_path** n'est pas vide : // On a trouvé un chemin !
        Tant que **stack_path** n'est pas vide :
            Dépiler **stack_path** et empiler la valeur sur **stack_path_final**
        Fin Tant que
        Tant que **stack_path_final** n'est pas vide :
            Dépiler **stack_path_final** et afficher la valeur
        Fin Tant que
    Fin Si

Arguments acceptés par le programme

Votre programme devra accepter les options suivantes, dans n'importe quel ordre :

- **`-i`** suivi du nom du fichier de graphe à lire ; si omis, le programme lit sur l'entrée standard (**`stdin`**),

- **`-o`** suivi du nom du fichier dans lequel écrire le chemin sommet par sommet ; si omis, le programme écrit sur l'entrée standard (**`stdout`**),
- **`-start`** suivi du numéro (ou du nom) du sommet de départ ; si omis, le sommet est lu sur l'entrée standard,
- **`-goal`** suivi du numéro (ou du nom) du sommet d'arrivée ; si omis, le sommet est lu sur l'entrée standard.

Un exemple d'appel du programme serait donc :

```
./fil_rouge_2 -start 0 -goal 3 -i ex1.txt
```

À nouveau, si un argument n'est pas reconnu, alors le programme doit afficher un message détaillant le problème et les arguments possibles, puis s'arrêter.

V2

3. Recherche des CFC

L'objectif de ce programme est d'énumérer toutes les composantes fortement connexes et d'en afficher la liste des sommets sur chaque ligne, ainsi que le nombre total trouvé. Par exemple, pour le graphe d'exemple du cours :

```
0
2 3
1
3 composantes fortement connexes trouvées
```

Ou, dans la version avec noms :

```
start
working paused
stop
3 composantes fortement connexes trouvées
```

Algorithme de Kosaraju

Il existe plusieurs algorithmes de recherche et nous utiliserons le plus connu : l'algorithme de Kosaraju[3]. Avec cet algorithme, rechercher les composantes fortement connexes nécessite notamment d'effectuer deux parcours en profondeur.

Il est uniquement demandé d'implémenter la version récursive de l'algorithme de Kosaraju, détaillée ci-dessous.

Voici l'algorithme général de cette recherche :

3 Voir : https://fr.wikipedia.org/wiki/Algorithme_de_Kosaraju

Énumération des composantes fortement connexes
Fonction : **enum_cfc_kosaraju**(**g**)
Arguments :
    g : un graphe de **n** sommets
Variables :
    marking[**n**] : un tableau de booléens permettant de marquer les sommets déjà visités
    order[**n**] : un tableau d'entiers permettant de stocker l'ordre de passage
Algorithme :
    1) Effectuer un parcours en profondeur sur **g** dans un ordre arbitraire et conserver l'*ordre suffixe de parcours* dans le tableau **order**
    2) **g'** ← Calculer le graphe inverse de **g**
    3) Effectuer un parcours en profondeur sur **g'** dans l'ordre inverse de celui donné par **order** et afficher les sommets des sous-graphes parcourus depuis chaque sommet initial

L'étape 1) est un parcours en profondeur, comme pour la partie 2. Elle nécessite de considérer un ordre arbitraire des sommets ; on utilise typiquement les numéros de sommets. Pour chaque sommet, on consiste calcule l'ordre suffixe de parcours, c'est-à-dire l'ordre dans lequel on quitte les sommets après les avoir parcourus. Si on reprend l'exemple de parcours de la partie 2 :

- L'ordre de parcours *préfixe* est celui où on arrive dans les sommets : 0, 1, 2, 3.
- L'ordre suffixe est celui dans lequel on quitte les nœuds en revenant au prédécesseur ; si on continue le parcours à la fin de l'exemple, cet ordre serait : 1, 3, 2, 0.

L'algorithme de cette étape est le suivant, avec à nouveau une partie d'initialisation et une fonction récursive :

Premier parcours récursif de l'algorithme de Kosaraju (fonction globale)
Fonction : **Kosaraju_1**(**g**)
Arguments :
    g : un graphe de **n** sommets
Variables :
    marking[**n**] : un tableau de booléens permettant de marquer les sommets déjà visités
    order[**n**] : un tableau d'entiers permettant de stocker l'ordre de passage suffixe
    x : un sommet de **g**
    step : un entier permettant de compter l'ordre suffixe
Algorithme :
    Initialiser **marking** à **FAUX**
    step ← 0
    Pour tout sommet **x** de **g** :
        Kosaraju_1_recur(**g**, **x**, **marking**, **order**, **step**)
    Fin Pour

Premier parcours récursif de l'algorithme de Kosaraju (appels récursifs)
Fonction : **Kosaraju_1_recur**(**g**, **x**, **marking**, **order**, **step**)
Arguments :
    g : un graphe de **n** sommets
    x : le sommet de départ
    marking[**n**] : un tableau de booléens permettant de marquer les sommets déjà visités
    order[**n**] : un tableau d'entiers permettant de stocker l'ordre de passage suffixe
    step : un entier permettant de compter l'ordre suffixe
Valeur de retour :
    L'étape finale du parcours actuel
Variables :
    y : un sommet de **g**
Algorithme :
    Si **marking**[**x**] = **FAUX** :
        marking[**x**] = **VRAI**
        Pour chaque successeur **y** de **x** :
            step ← **Kosaraju_1_recur**(**g**, **y**, **marking**, **order**, **step**)
        Fin Pour
        order[**x**] ← **step**
        step ← **step** + 1
    Fin Si
    Retourner **step**

V3

L'étape 2) consiste à calculer le graphe dont les sommets sont identiques à ceux de **g** mais dont les arcs sont tous inversés. Pour la représentation d'un graphe en matrice d'adjacence, cela consiste à calculer la transposée de la matrice. Pour la représentation en listes d'adjacence c'est un peu différent mais cela se calcule assez simplement aussi.

L'étape 3) consiste à reprendre l'ordre suffixe de l'étape 1) et à parcourir **g'** dans l'ordre inverse de cet ordre suffixe (dans le cas de l'exemple du cours : 0, 2, 3, 1). Pour chacun de ces sommets, on lance un nouveau parcours récursif dont les ensembles de sommets non marqués forment les composantes fortement connexes :

Deuxième parcours récursif de l'algorithme de Kosaraju (fonction globale)
Fonction : **Kosaraju_2**(**g**, **order**)
Arguments :
    g : un graphe de **n** sommets
    order[**n**] : un tableau d'entiers contenant l'ordre suffixe calculé à l'étape 1)

Variables :
    marking[**n**] : un tableau de booléens permettant de marquer les sommets déjà visités
    inv_order[**n**] : un tableau d'entiers pour stocker l'ordre suffixe inverse
    x : un sommet de **g**
    nb_scc : un entier comptant le nombre de composantes fortement connexes trouvées
Algorithme :
    Initialiser **marking** à **FAUX**
    Pour **x** allant de 0 à **n** :
        inv_order[(**n** – 1) – **order**[**x**]] ← **x** // On calcule l'ordre inverse de **order**
    Fin Pour
    Pour **x** allant de 0 à **n** :
        Si **Kosaraju_2_recur**(**g**, **order**[**x**], **marking**) retourne **VRAI** :
            nb_scc ← **nb_scc** + 1
            Afficher un retour à la ligne
        Fin Si
    Fin Pour
    Afficher : « **nb_scc** composantes fortement connexes trouvées »

V3

V3

Deuxième parcours récursif de l'algorithme de Kosaraju (appels récursifs)
Fonction : **Kosaraju_2_recur**(**g**, **x**, **marking**)
Arguments :
    g : un graphe de **n** sommets
    x : le sommet de départ
    marking[**n**] : un tableau de booléens permettant de marquer les sommets déjà visités
Valeur de retour :
    Un booléen indiquant si une nouvelle composante fortement connexe a été trouvée
Variables :
    y : un sommet de **g**
Algorithme :
    Si **marking**[**x**] = **VRAI** :
        Retourner **FAUX**
    Sinon :
        marking[**x**] = **VRAI**
        Afficher **x**
        Pour chaque successeur **y** de **x** :
            Kosaraju_2_recur(**g**, **y**, **marking**)
        Fin Pour
        Retourner **VRAI**
    Fin Si

Si vous le souhaitez, vous pouvez tenter de trouver une version itérative de l'algorithme, ou tester un autre algorithme de recherche de composantes fortement connexes[4]. Pensez à détailler votre démarche dans le rapport et la présentation.

L'algorithme ci-dessus affiche toutes les composantes fortement connexes trouvées. Cependant, si on souhaite les analyser, par exemple pour n'afficher que celles qui contiennent au moins un certain

4 Voir : https://fr.wikipedia.org/wiki/Composante_fortement_connexe#Algorithmes

nombre de sommets, ou pour déterminer celles qui sont terminales, il faut pour cela les stocker en mémoire.

Si vous le souhaitez, implémentez un filtre permettant de retourner seulement les composantes fortement connexes contenant plus d'un certain nombre de sommets, ou seulement les composantes fortement connexes terminales. Pensez à détailler votre démarche dans le rapport et la présentation.

V3

Arguments en ligne de commande

Concevez un ensemble d'arguments pertinent pour votre programme et détaillez-le dans votre compte-rendu.

Conclusion

Pour chaque programme, effectuez une analyse temporelle, par exemple avec le programme **`time`**, comme cela avait été vu au TP2, ou avec la fonction **`clock`**[5] intégrée au langage C. Comparez l'implémentation des graphes en tant que matrice d'adjacence et en tant que listes d'adjacence. Vous pouvez aussi comparer les temps d'exécution en fonction des graphes, mais gardez à l'esprit que des structures de graphes différentes peuvent avoir des effets importants sur l'exécution.

V3

Attention, le chargement du graphe peut prendre du temps et fausser les résultats, de même qu'un très grand nombre d'affichages sur le terminal. Pensez donc à retirer des instructions d'affichage, ainsi qu'à faire une analyse temporelle du chargement du graphe pour soustraite ce temps aux autres mesures.

Dans votre rapport et dans votre présentation, pensez à présenter :

- les structures de données utilisées en vous appuyant sur des schémas,
- la notion de parcours de graphe pour montrer que vous l'avez maîtrisée,
- les algorithmes que vous avez écrits et qui ne sont pas détaillés dans cet énoncé,
- tout autre élément montrant que ce travail vous a permis de maîtriser le sujet.

En revanche, dans le rapport comme dans la soutenance, évitez de présenter du code source ou des jeux d'essai non commentés. Préférez des explications écrites, des diagrammes, des courbes, de (courts) algorithmes… De courtes portions de code source sont acceptables si cela permet de mieux comprendre votre travail. Pour le reste, nous pourrons consulter votre code source si besoin est.

5 Voir : https://fr.wikiversity.org/wiki/Fonctions_de_base_en_langage_C/time.h#Fonction_clock()