TRAVAUX PRATIQUES I

Contrôle technique

Le but de ce TP est de réviser les algorithmes usuels sur les graphes. Il ne contient donc rien de nouveau !
Insistons tout de suite : **Il faut savoir écrire et utiliser des parcours de graphe !**

On rappelle le fonctionnement de la compilation séparée en C :

— Chaque fichier source (`.c`) doit d'abord être compilé en fichier objet (`.o`). Pour cela, utiliser l'option `-c` à la compilation ; elle signifie « compiler uniquement, ne pas lier ». La ligne de commande ressemble donc à :

```
gcc -o code_source.o -c code_source.c
```

Un fichier objet est un intermédiaire de compilation. C'est en quelque sorte *presque* un fichier compilé en code machine, à quelques points près dont l'édition de liens. Dans un `.o` , les appels à une fonction d'un autre fichier ne sont pas encore traduits : il est simplement écrit « ici il faut appeler la fonction f sur tels et tels arguments » [1]
L'édition de liens consiste justement à indiquer comment sauter à la fonction f (et comment revenir en quittant f) afin que les appels de fonctions fonctionnent.
Ainsi : pour compiler en objet, il n'est pas nécessaire d'avoir le code des fonctions externes ! C'est le linker, responsable de l'édition de liens, qui cherchera cette fameuse fonction f, et qui lèvera une erreur si il ne la trouve pas (ou si il en trouve plusieurs…).
Une fonction f particulière recherchée lors de l'édition de liens est la fonction `main` : l'exécution du programme C se résume à l'exécution du `main`, qu'il faut donc trouver. S'il n'y en a pas ou qu'il y en a plusieurs, une erreur sera levée par le linker.

— Notez que l'option `-o` permet de nommer la sortie de la compilation (l'**o**utput dans la langue d'Ada Lovelace). Elle n'a donc pas spécialement à voir avec les fichiers objets.

— Ensuite, il faut lier tous ces objets ensemble en un seul exécutable : on les prend donc tous en argument, et on les compile ensemble. La ligne de commande ressemble donc à :

```
gcc -o exec file0.o file1.o file2.o ...
```

On rappelle les options de compilation aidant au débuggage : `-Wall -Wextra -fsanitize=undefined` .
Il faut toujours mettre ces options de compilation, à la fin de la ligne de commande.

Rappelons aussi que débugguer au fur et à mesure est *bien* plus simple que tout débugguer à la fin.

Dans tout ce TP, S désigne l'ensemble des sommets d'un graphe et A celui de ses arcs/arêtes. Dans tout ce TP, sauf mention contraire les graphes sont non-pondérés.

A Représentation utilisée

Ce TP est à coder en C. Vous devrez rendre **unique le fichier `graphes.c` complété**. Les graphes sont représentés par listes d'adjacence. Chacune des listes d'adjacence est modélisée par un tableau dynamique, dont une interface et son implémentation sont fournies avec ce sujet. [2]. Le nombre de sommets des graphes est fixe.
On utilise le type `graphe` ci-dessous :

C : graphes.h

```c
struct graphe_s {
  dynarray* lA; //tableau des "listes" d'adjacence du graphe (sous forme de tableaux dynamiques)
  int n; //nombre de sommets du graphe
};
typedef struct graphe_s graphe;
```

N'oubliez pas d'ajouter une fonction `main` à votre fichier **graphes.c** !

1. La commande `objdump` permet d'aller lire des fichiers compilés, vous pourrez y jeter un oeil chez vous.
2. Vous n'avez pas à les relire ! Vos versions du passé de MP2I n'ont pas fait une belle interface pour rien !

1. Rappeler le prototype usuel d'un `main` . À quoi correspond le type de sortie ?
2. Rappeler le fonctionnement de la fonction `void* malloc(size_t size)` . Que ne faut-il **jamais** oublier de faire avec un `malloc` ?
   Le type `size_t` est un type d'entiers qui est garanti assez grand pour contenir la taille de tout objet mémoire allouable sur la machine. Il est peu présent au programme de MP(2)I ; et vous pouvez utiliser d'autres types d'entiers à la place.
3. Lire l'interface `dynarray.h` . Les fonctions `dyn_len` et `dyn_acces` seront très utiles dans tout ce TP !
4. Complétez les fonctions `void cree_graphe_exemple(void)` qui implémente le graphe-exemple ci-dessous, et `void graphe_free(graphe G)` qui libère un graphe.

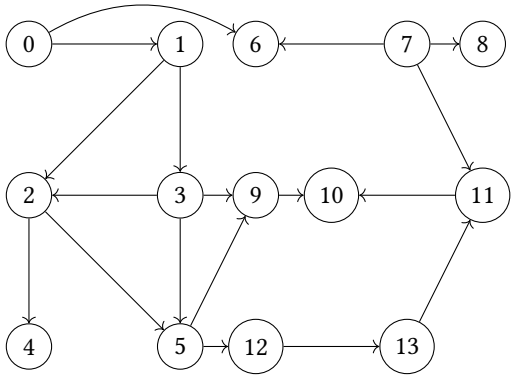

Figure I.1 – Le graphe-exemple.

5. **a.** Écrire une fonction `graphe graphe_transpose(graphe G)` qui prend en argument un graphe G et renvoie son graphe transposé ${}^tG$.
   Rappel : avec $G = (S, A)$, ${}^tG$ est défini par ${}^tG = (S, {}^tA)$ où ${}^tA = \{(b, a) | (a, b) \in A\}$.
   b. Testez-la. Vous pourrez utiliser la fonction `graphe_print` qui affiche les listes d'adjacence d'un graphe donné.

B Parcours en largeur

On rappelle que des fichiers de bibliothèques sont fournis avec ce TP, notamment les bibliothèques `pile` et `file` .

6. **a.** Écrire une fonction de parcours en largeur `void graphe_bfs_s0(graphe G, int s0, int ordre[])`
   La fonction prend en argument un graphe `G` et un tableau/zone-allouée `ordre` de longueur $|S|$ (initialement quelconque) ; et écrit dans `ordre` les sommets de G dans l'ordre d'exploration d'un parcours en largeur[3] depuis `s0`.
   ⚠ *Un tel ordre n'est pas unique.* Pour le rendre unique et passer les tests du testeur, vous devez renvoyer le parcours en largeur des sommets par ordre croissant d'étiquettes (en cas de choix). Par exemple, si 0 a pour voisins 1 et 6, on visite d'abord le voisin 1 (et on ajoute ses voisins à la file) puis le voisins 6. Donc 1 apparaîtra avant 6 dans le tableau `ordre`, et les voisins de 1 apparaîtront avant les voisins de 6, etc.
   On remplira les cases inutilisées par des -1[4].
   b. Testez-le, par exemple en affichant dans l'ordre de parcours les sommets du graphe-exemple depuis 0 et depuis 3.
7. On rappelle que pour parcourir un graphe en entier (et pas seulement les sommets accessibles depuis la source), on lance un parcours depuis un premier sommet, puis depuis un prochain sommet non-exploré, etc jusqu'à avoir exploré tous les sommets.
   a. Écrire une fonction `void graphe_bfs_complet(graphe G, int ordre[])` qui réalise un parcours en profondeur de `G` en entier et note l'ordre d'exploration dans `ordre`.
   Un tel ordre n'est pas unique, vous devez en renvoyer un valide.
   b. Testez-le.

3. « bfs » est l'acronyme de **B**readth-**F**irst-**S**earch, c'est à dire « recherche en largeur d'abord » dans la langue de Frances Allen.
4. On dit que -1 est une sentinelle de fin.

8. (bonus) Écrire une fonction `void graphe_dist_s0(graphe G, int s0, int dist[])` qui calcule les distances de `s0` à tout autre sommet de `G`.
   Votre code fonctionnerait-il pour un graphe pondéré ? Justifier.

C Parcours en profondeur

9. **a.** Écrire une fonction de parcours en profondeur :
   `void graphe_dfs_s0_pile(graphe G, int s0, int ordre[])` .
   La fonction prend en argument un graphe `G` et un tableau/zone-allouée `ordre` de longueur $|S|$ (initialement quelconque) ; et écrit dans `ordre` les sommets de G dans l'ordre d'exploration d'un parcours en profondeur [5] depuis `s0`.
   On n'utilisera pas de récursion.
   b. Testez-le, par exemple en affichant dans l'ordre de parcours les sommets du graphe-exemple.
10. **a.** Écrire une version récursive de la fonction précédente, nommée :
   `void dfs_s0_rec(graphe G, int s0, int ouverture[], int fermeture[])` .
   Cette fonction prend en argument un graphe `G` et deux tableaux/zones-allouées `ouverture` et `fermeture` tous deux de longueur $|S|$. Elle doit modifier :
   - `ouverture` de sorte à ce que pour tout sommet `s`, `ouverture[s]` contienne la date d'ouverture de `s` lors d'un parcours en profondeur récursif depuis `s0`, ou -1 si ce sommet n'a pas été ouvert (sommet vierge).
   - `fermeture` de sorte à ce que pour tout sommet `s`, `fermeture[s]` contienne la date de fermeture de `s` lors de ce même parcours, ou -1 si ce sommet n'a pas été fermé.

   Réalisez la avec une récursion [6].
   b. Testez-la, par exemple en affichant les dates d'ouverture et de fermeture pour un parcours du graphe exemple depuis un sommet de votre choix.
11. **a.** Écrire une fonction `void graphe_dfs_rec(graphe G, int ouverture[], int fermeture[])` effectue un parcours en profondeur du graphe `G` en entier, en utilisant la fonction récursive précédente.
   Comme précédemment, on lance des parcours successivement sur chaque sommet qui n'a pas déjà été vu. Cette partie n'a pas à être récursive.
   b. Testez-la.
12. (Bonus) Écrire une fonction `graphe_tri_topo` qui calcule un tri topologique d'un graphe.
   Le prototype précis (et la spécification associée) est volontairement laissée à votre décision.

D Si vous vous ennuyez

13. Écrire un nouveau type de graphes, permettant de pondérer les arcs.
14. Implémenter :
   a. Le calcul de la matrice des distances entre toute paire de sommet.
   b. Le calcul de la distance d'un sommet source donné à toute destination.

   Dans les deux cas, vous pourrez poser des hypothèses raisonnables et les spécifier ; ainsi que préciser ce qui peut arriver si elles ne sont pas respectées.
15. Écrire une fonction qui vérifie si un graphe (non-orienté non-pondéré) est biparti. *Ce n'est pas censé être compliqué ou technique.*
16. Écrire une fonction qui vérifie si un graphe (non-orienté non-pondéré) est 3-colorable, c'est à dire si on peut colorer ses sommets de sorte à ce que chaque sommet ait une couleur différente de ses voisins.
   Des indices sur un code polynomial pour ce problème vous attendent dans le prochain chapitre !

5. « dfs » est l'acronyme de **D**epth-**F**irst-**S**earch, c'est à dire « recherche en profondeur d'abord » dans la langue de Shafira Goldwasser.
6. La notion de date d'ouverture et de fermeture n'a pas de sens en dehors d'un parcours récursif.