La machine à parcours

Les objectifs de ce sujet de projet sont :

Révisions de C pointeurs génériques et pointeurs de fonctions ;

Algorithmique parcours génériques, parcours spécialisés (en profondeur et en largeur) et leurs applications : ordres topologiques et détection de circuits, calcul des composantes fortement connexes, calcul des distances depuis un sommet et diamètre d'un graphe.

Les fonctions à implémenter (ou qui le sont déjà) sont (partiellement) documentées (voir ENT).

Mise en route

1. Lire les consignes du projet sur l'ENT.
2. Si vous en avez besoin, consultez l'appendice A (à la fin de ce sujet) qui contient des rappels sur les pointeurs génériques et les pointeurs de fonctions.
3. Télécharger l'archive contenant les fichiers nécessaires et l'extraire dans un répertoire dédié à ce projet.
4. Les fichiers contenus sont les suivants :
   - `Makefile` pour compiler avec la commande `make` ;
   - `graphe-4.c` et `graphe-4.h` : la bibliothèque de graphes représentés par leurs listes de successeurs qui est, à quelques petits changements près, la même que celle décrite lors du TP précédent ;
   - `pile.c`, `pile.h`, `file.c`, `file.h`, `pile_ou_file.c`, `pile_ou_file.h` : structures de données pour l'écriture des parcours ;
   - `test-parcours.c`, `test-cfc.c`, `test-ordre_top.c` et `test-distances.c` : fonctions `main` de test pour les différentes parties du sujet ;
   - `conteneur_sommets.c` et `conteneur_sommets.h` : une petite bibliothèque pour abstraire la notion de conteneur de sommets et créer des conteneurs associés à des structures de données ;
   - `parcours.c` et `parcours.h` : une bibliothèque pour parcourir les graphes ;
   - `graphe_parcours.c` et `graphe_parcours.h` : fonctions de parcours de graphes particulières (profondeur, largeur, ...) et leurs applications.

   Les seuls fichiers à modifier sont `conteneur_sommets.c`**,** `parcours.c`**,** `graphe_parcours.c` et éventuellement les fichiers d'en-tête correspondant si vous trouvez utile d'ajouter des fonctions.

Petits changements dans `graphe-4`

Les fonctions

```
graphe *graphe_creer(int n, int est_or);
void graphe_liberer(graphe *g);
```

font leur apparition. La première alloue de la mémoire pour une variable de type `graphe` et pour ses champs et retourne l'adresse de la nouvelle variable. La seconde libère la mémoire obtenue.

De cette façon, la création, l'utilisation et la destruction de graphes se font de la façon plus classique suivante :

```
graphe *g = graphe_creer(32, 1);
graphe_ajouter_arc(g, 1, 3, 2.8);
/* (...) */
graphe_liberer(g);
```

Les fonctions `graphe_stable` et `graphe_detruire` restent dans la bibliothèque pour des raisons de compatibilité mais leur utilisation n'est pas recommandée.

La fonction

```
int graphe_ecrire_dot(graphe *g, char *nom_fichier, int ecrire_valeurs);
```

a un troisième argument pour savoir si l'on veut ou non représenter les valeurs des arcs.

Les fonctions

```
graphe *graphe_complet(int n, int est_or);
graphe *graphe_aleatoire(int n, double p, int est_or);
```

se comportent comme la fonction `graphe_creer` et retournent un pointeur vers le nouveau graphe.

1 Parcours générique

Le but de cette partie est de définir et/ou d'utiliser des structures de façon à pouvoir implémenter de façon efficace le parcours générique donné dans votre cours, qu'on rappelle ici :

```
Parcours générique :
Initialisation :
Le conteneur de sommets est vide.
Aucun sommet n'est visité.
Aucun sommet n'est exploré.

Tant que tous les sommets ne sont pas explorés,
    choisir une racine r parmi les sommets non visités
    ajouter r au conteneur
    marquer r comme visité
    Tant que le conteneur n'est pas vide
        choisir un sommet v dans le conteneur
        Si v a des successeurs non visités, alors
            choisir un tel successeur w
            marquer w comme visité
            ajouter w au conteneur
        Sinon
            marquer v comme exploré
            enlever v du conteneur
        FinSi
    FinTantQue
FinTantQue
```

On rappelle que l'ordre *préfixe* associé au parcours est l'ordre dans lequel les sommets sont visités et l'ordre *suffixe* l'ordre dans lequel ils sont explorés.

L'algorithme générique montre qu'on a besoin d'un conteneur dans lequel on peut : choisir un sommet, ajouter un sommet, supprimer un sommet (ou au moins supprimer le dernier sommet choisi) et savoir s'il est vide.

1.1 Le type `conteneur_sommet`

Le type `conteneur_sommet` est défini dans le fichier `conteneur_sommet.h`. Il permet l'abstraction du concept de conteneur où l'on peut ajouter un élément, choisir un élément, supprimer le dernier élément choisi et savoir si le conteneur est vide. Comme exemple concret de conteneur, penser à la pile et à la file.

C'est donc un type structuré contenant un pointeur générique vers la structure concrète et 5 pointeurs de fonctions.

Une variable de type `conteneur_sommet` est

— créée par un appel à une fonction spécifique à un type de conteneur, par exemple : `cs_creer_pile(int)` ;
— manipulée via les fonctions générales `cs_ajouter`, `cs_est_vide`, `cs_supprimer`, `cs_choisir` (qui utilisent elle-même les pointeurs de fonctions internes à la structure) ;
— détruite par un appel à la fonction générale `cs_detruire` (qui utilise elle-même un pointeur de fonction interne à la structure).

Les structures de données `pile`, `file` et `pile_ou_file` ont déjà été écrites pour vous : voir les fichiers correspondants. Vous connaissez bien-sûr les piles et les files, la structure `pile_ou_file` est plus originale : il s'agit d'une file d'attente où, lorsque l'on choisit un élément, plutôt que de choisir toujours le premier, on choisit aléatoirement soit le premier soit le dernier.

Question 1: Le constructeur `cs_creer_pile` est déjà écrit pour vous.

En s'en inspirant, écrire les définitions des constructeurs pour les deux autres structures.

--- * ---

Par la suite, on supposera (c'est le cas pour nos trois conteneurs concrets) que les 4 opérations (choisir, ajouter, supprimer et est_vide) sur nos conteneurs sont toutes en complexité $O(1)$.

1.2 Le type `struct parcours`

Le type `struct parcours` contient tous les éléments (nombreux) permettant de faire de façon efficace (en temps linéaire) le parcours d'un graphe en utilisant un conteneur de sommet. Il contient donc :

— un champ `conteneur`, pointeur vers un conteneur de sommets ;
— un champ `arbo`, pointeur vers un graphe voué à contenir l'arborescence ou la forêt associée au parcours ;
— un champ `prio` représentant la liste des sommets par ordre de priorité décroissante lors du choix d'une racine du parcours (ce peut être l'adresse d'un tableau de n entiers (où n est le nombre de sommets du graphe parcouru), ou l'adresse d'une file ou d'une pile contenant ces entiers) ;
— un champ pour contenir l'ordre préfixe associé au parcours ;
— un champ pour contenir l'ordre suffixe associé au parcours ;
— d'autres champs utiles pour faire des parcours efficaces !

Pour revenir sur le champ `prio` : imaginons qu'il pointe vers la file `{0, 3, 4, 2, 1}` : cela signifie que la première racine du parcours est `0`. Si les sommets accessibles depuis `0` sont `3` et `2`, la prochaine racine sera `4`.

La complexité linéaire est chose fragile : on fera particulièrement attention aux deux points suivants :

— si, à chaque choix de racine, vous devez reparcourir tout le tableau `prio`, votre parcours ne sera plus en temps linéaire dans le pire des cas ;
— si, à chaque fois que vous considérez un sommet, vous parcourez toute la liste de ses successeurs, votre parcours ne sera pas en temps linéaire dans le pire des cas.

Il va donc falloir choisir avec soin les données à garder dans la structure `parcours`,... (l'écriture des fonctions pourrait vous aider à bien choisir).

Question 2: Le but de cette question est de compléter entièrement le fichier source `parcours.c`. On rappelle que vous êtes libres dans l'ajout de champs dans la structure `parcours`. On peut procéder dans l'ordre suivant (que vous n'êtes évidemment pas obligés de respecter) :

1. Commencer par implémenter les fonctions d'une ligne :

```
int pc_conteneur_est_vide(struct parcours *p);
void pc_ajouter_dans_conteneur(struct parcours *p, int sommet);
void pc_supprimer_du_conteneur(struct parcours *p);
int pc_choisir_dans_conteneur(struct parcours *p);
```

2. Implémenter la fonction

```
int pc_est_visite(struct parcours *p, int sommet);
```

Elle doit avoir une complexité en $O(1)$ pour que l'algorithme soit efficace.

3. Implémenter les fonctions (complexité $O(1)$)

```
void pc_marquer_comme_visite(struct parcours *p, int sommet);
void pc_marquer_comme_explore(struct parcours *p, int sommet);
void pc_est_fini(struct parcours *p);
```

4. Implémenter la fonction

```
int pc_choisir_racine(struct parcours *p);
```

qui choisit la prochaine racine du parcours.

5. Implémenter la fonction (complexité $O(1)$)

```
msuc *pc_prochain_msuc(struct parcours *p, int sommet);
```

qui retourne un pointeur vers le prochain maillon de successeur du sommet `sommet` non encore considéré, ou bien `NULL` si la liste des successeurs de `sommet` est épuisée.

6. Écrire les fonctions (la seconde faisant appel à la première)

```
void pc_parcourir_depuis_sommet(struct parcours *p, int r);
void pc_parcourir(struct parcours *p);
```

en respectant l'algorithme donné au début de cette partie et en accédant le moins souvent possible directement aux champs de la structure. On suppose que la structure `parcours` est initialisée pour commencer le parcours.

7. Maintenant que l'on sait bien quels champs sont présents dans la structure, écrire la fonction

```
struct parcours *pc_init(graphe *g, conteneur_sommets *cs, int *prio);
```

qui retourne un pointeur vers un parcours initialisé, pour le graphe d'adresse `g`, le conteneur d'adresse `cs` et le tableau de priorité `prio`. Cette fonction se charge d'allouer de la mémoire pour tous les champs de la structure qui en ont besoin et de les initialiser de façon à pouvoir commencer le parcours.

Si le paramètre `prio` vaut `NULL`, cette fonction devra créer une liste de priorité par défaut qui est `{0, 1, 2, ..., n - 1}` (où n est l'ordre du graphe parcouru).

On testera les allocations mémoire. En cas d'échec d'une allocation, toute la mémoire allouée devra être libérée et la valeur de retour sera `NULL`.

8. Écrire la fonction

```
void pc_detruire(struct parcours *p);
```

qui libère toutes les ressources associées au parcours (ne fait rien si le pointeur `p` vaut `NULL`). On fera attention à ne pas libérer les zones mémoire réservées par l'appelant (i.e. les trois adresses passées en arguments à la fonction `pc_init`), même si elles ont été enregistrées dans la structure `*p`.

--- * ---

2 Parcours spécialisés

Question 3: Implémenter, dans le fichier `graphe_parcours.c` les fonctions suivantes en se servant des fonctions de `parcours.h` et `conteneur_sommets.h` :

```
int graphe_parcours_profondeur(graphe *g, graphe **arbo, int **suff, int *prio);
int graphe_parcours_largeur(graphe *g, graphe **arbo, int **suff, int *prio);
int graphe_parcours_larg_ou_prof(graphe *g, graphe **arbo, int **suff, int *prio);
```

Elles se chargent d'initialiser et de détruire le parcours et le conteneur.

--- * ---

Question 4: Tester tout le travail accompli à l'aide du programme de test `test-parcours`.

Le débuggage peut être difficile. Il pourrait être utile de lancer le programme avec `valgrind` en lançant la commande

```
valgrind ./test-parcours
```

Le programme `gdb` (GNU debugger) peut également être utile. On trouvera un petit tutoriel par exemple à l'adresse
http://perso.ens-lyon.fr/daniel.hirschkoff/C_Caml/docs/doc_gdb.pdf

Une fois que tout est réglé, c'est le bon moment pour prendre le temps de nettoyer un peu ses sources, de s'assurer que tout est bien présenté, indenté, nommé, satisfait les spécifications données dans les fichiers d'en-tête, que les points un peu subtils du code sont commentés, ...

--- * ---

3 Applications du parcours en profondeur

3.1 Test d'acyclicité et ordre topologique

On veut, toujours en temps linéaire, tester si un graphe est acyclique (c'est-à-dire n'a pas de circuit) et, si tel est le cas, retourner un ordre topologique.

On pourrait utiliser l'implémentation de l'algorithme de Bellman (avec les degrés entrants résiduels) donnée en cours, mais ici on va utiliser un parcours en profondeur.

Lors du parcours en profondeur, on détecte un arc arrière de la façon suivante : si, après avoir choisi un sommet `v` dans le conteneur, l'examen d'un de ses successeurs `w` (considéré pour la première fois) est déjà dans la pile, c'est que l'arc (v, w) est un arc arrière. Si cette situation ne se produit jamais, il n'y a pas d'arc arrière.

Question 5: En utilisant une version légèrement modifiée du parcours en profondeur (utiliser les fonctions de `parcours.h`), écrire dans le fichier `graphe_parcours.c` la définition de la fonction

```
int graphe_ordre_top(graphe *g, int **ordre_top);
```

Elle doit respecter la spécification donnée dans le fichier `graphe_parcours.h`.

Tester votre fonction à l'aide du programme de test `test-ordre_top`.

--- * ---

3.2 Calcul des composantes fortement connexes

Question 6: Utiliser l'algorithme de Kosaraju-Sharir pour calculer les composantes fortement connexes d'un graphe en temps linéaire en définissant dans le fichier `graphe_parcours.c` la fonction

```
int graphe_comp_fort_conn(graphe *g, int **reprs_cfc);
```

respectant les spécifications données dans le fichier `graphe_parcours.h`. La fonction devra choisir pour chaque composante fortement connexe un (et un seul) représentant `r` et, pour tout sommet `v` appartenant à cette composante, faire l'affectation `(*reprs_cfc)[v] = r`.

Tester votre fonction à l'aide du programme de test `test-cfc`.

Note : si nécessaire, vous pouvez ajouter des champs à `struct parcours` *et modifier l'algorithme générique de parcours pour qu'il marque chaque sommet visité avec la racine courante du parcours.*

--- * ---

4 Applications du parcours en largeur

On rappelle que la distance de graphe entre un sommet r et un sommet v du graphe est le nombre minimal d'arcs d'un chemin allant de r à v (et elle vaut $+\infty$ si un tel chemin n'existe pas).

On a défini dans `graphe_parcours.h` la constante symbolique `DIST_INF` (un `int` négatif) pour représenter une distance infinie.

Question 7: Définir la fonction

```
int *graphe_distances_depuis_sommet(graphe *g, int sommet);
```

respectant la spécification donnée dans `graphe_parcours.h`, en utilisant un parcours en largeur (légèrement modifié) depuis le sommet `sommet`.

Tester votre fonction à l'aide du programme de test `test-distances`.

--- * ---

Le diamètre d'un graphe est la plus grande distance entre deux sommets du graphe ($+\infty$ si le graphe n'est pas fortement connexe).

Question 8: Définir la fonction

```
int graphe_diamètre(graphe *g);
```

qui calcule le diamètre du graphe. Donner sa complexité dans le pire des cas (en fonction de `n` et `m`) en commentaire dans le fichier `graphe_parcours.c`.

--- * ---

A Appendice : Rappels de programmation en C

A.1 Pointeurs génériques

Un pointeur générique en C est une variable de type `void *`. Elle contient l'adresse de « quelque-chose » sans aucune information sur le type (et donc la taille) du « quelque-chose » en question.

Il est illicite (erreur à la compilation) de déréférencer un pointeur générique ou de faire de l'arithmétique sur de tels pointeurs.

Avant de faire ces opérations, il est donc nécessaire de *transtyper* (c'est-à-dire de faire un « cast » sur) ce pointeur générique.

Voyons tout de suite un exemple.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int n = 0x7B534644;
  int *k;
  void *p = &n;
  float *x;
  char *c;
```

```
  /* cast de p sous la forme d'un int */
  k = (int *) p; /* cast explicite */
  printf("valeur pointée sous forme int : %d\n", *k);
  /* affiche : 2069055044 */
  /* cast de p sous la forme d'un float */
  x = p;
  printf("valeur pointée sous forme float : %g\n", *x);
  /* affiche : 1.097e+36 */
  /* cast de p sous la forme d'un char */
  c = p;
  printf("valeur pointée sous forme char : %c%c%c\n", *c, c[1], c[2]);
  /* affiche : DFS */
  return EXIT_SUCCESS;
}
```

Quelques remarques :

- L'utilisation des pointeurs génériques est souvent **dangereuse** et parfois **peu portable**. Les types ont une raison d'être ! Il ne faut donc s'en servir qu'en cas de besoin et avec une grande précaution.
- En C, tous les transtypages entre types de pointeurs doivent être explicites, sauf :
  - d'un type pointeur non générique à un type générique et
  - d'un type pointeur générique à un type pointeur non générique.

D'ailleurs, vous connaissez déjà une fonction qui retourne un pointeur générique (et qui ne pourrait pas faire autrement !) : la fonction `malloc` et on peut donc indifféremment écrire

```
int *tab = malloc(n * sizeof(int));
```

ou

```
int *tab = (int *) malloc(n * sizeof(int));
```

(j'ai une certaine préférence pour l'écriture la plus courte, mais les débats à ce sujet sont sans fin...)

A.2 Pointeurs de fonctions

Il est possible en C d'utiliser des pointeurs vers des fonctions. Cela permet, entre autre, de passer une fonction en paramètre d'une autre fonction, et d'avoir donc des fonctions plus générales. Voyons d'abord un exemple inutile avant de voir, à la prochaine sous-section, un exemple plus utile tiré de la bibliothèque standard.

```
#include <stdlib.h>
#include <stdio.h>
int appliquer(int (*f)(int), int n);
int oppose(int n);
int triple(int n);
int main()
{
  int n = 5;
  int k = appliquer(&oppose, n);
  /* en fait le & est inutile sur les noms de fonction */
  int l = appliquer(triple, n);
  printf("opposé de %d : %d\n", n, k);
  /* affiche : opposé de 5 : -5 */
  printf("triple de %d : %d\n", n, l);
  /* affiche : triple de 5 : 15 */
  return EXIT_SUCCESS;
}
int appliquer(int (*f)(int), int n)
{
```

```c
  return f(n); /* ou (*f)(n), les deux syntaxes sont correctes */
}
int oppose(int n)
{
  return -n;
}
int triple(int n)
{
  return 3 * n;
}
```

Quelques remarques :

— un nom de fonction est automatiquement converti en pointeur de fonction si besoin ;

— si `pf` est un pointeur de fonction, on peut utiliser la syntaxe `pf(x)` pour appeler la fonction pointée avec le paramètre `x` sans qu'il y ait besoin de déréférencer `pf` (bref, `*pf(x)` et `pf(x)` sont équivalents).

A.3 Exemple tiré de la bibliothèque standard

Quand on combine ces deux notions, on peut écrire des fonctions très générales. Prenons l'exemple de `qsort` (pour « quick sort », tri rapide), déclaré dans `stdlib.h` :

```c
void qsort(void *base, size_t nmemb, size_t size,
int (*compar)(const void *, const void *));
```

Les arguments de `qsort` sont, dans cet ordre :

base un pointeur générique vers un tableau

nmemb le nombre d'éléments du tableau

size la taille du type des éléments du tableau

compar un pointeur vers une fonction qui compare, sans les modifier, deux éléments, étant donné leurs adresses et retourne -1, 0 ou 1 selon que la première valeur pointée est respectivement inférieure, égale ou supérieure à la seconde.

Dans l'exemple suivant, cette fonction est utilisée pour trier un tableau d'entiers et un tableau de chaînes de caractères.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int compar_int(const void *a, const void *b)
{
  const int *n = a; /* pointeur vers un int constant*/
  const int *m = b;
  return (*n > *m) - (*n < *m);
}

int compar_str(const void *s1, const void *s2)
{
  /* pointeur vers une chaine constante, c'est-à-dire
  * pointeur vers un pointeur constant vers des char constants */
  const char *const *ch1 = s1;
  const char *const *ch2 = s2;
  return strcmp(*ch1, *ch2);
}

int main()
```

```
{
  int tab_int[] = {3, 1, 9, 5, 0};
  char *tab_str[] = {"kosaraju", "dijkstra", "prim", "ford", "kruskal"};
  int i;

  qsort(tab_int, 5, sizeof(int), compar_int);
  for (i = 0; i < 5; ++i)
  printf("%d ", tab_int[i]);
  puts("");

  qsort(tab_str, 5, sizeof(char *), compar_str);
  for (i = 0; i < 5; ++i)
  printf("%s ", tab_str[i]);
  puts("");
  return EXIT_SUCCESS;
}
```