Plus courts chemins : Ford, Bellman et Ford-Bellman
26 novembre 2018

Introduction

Les objectifs de ce sujet (qui prendra deux séances) sont :

1. d'implémenter les algorithmes de Ford, de Bellman et de Ford-Bellman ;
2. de faire des révisions sur l'allocation dynamique de mémoire en C et sur les pointeurs ;
3. de voir comment on peut partir d'un algorithme simplifié, qui parfois ne termine pas, pour obtenir un algorithme plus robuste qui termine toujours ;
4. de bien s'amuser en programmant.

1 Mise en place

On commence par vérifier que tout fonctionne : prendre sur l'ENT les différents fichiers correspondant à ce sujet. L'un d'entre eux est un Makefile (peu sophistiqué). Il y a aussi le script shell suivant (lui aussi, peu sophistiqué) :

```
#!/bin/sh
PROG=./ford.exe
DOT_FILE=premier_graphe.dot
IMG_FILE=premier_graphe.png
make && $PROG && dot -Tpng $DOT_FILE -o $IMG_FILE \
    && eog $IMG_FILE &
```

Remarque : si vous n'avez pas le programme `eog` (eye of gnome) vous devrez la remplacer par un autre programme capable d'afficher une image au format `png`.

Question 0: Après avoir mis tous les fichiers dans le même dossier, vous commencerez par rendre ce script exécutable

`chmod u+x comp_dot_eog.sh`

et par l'exécuter

`./comp_dot_eog.sh`

Vous devriez voir apparaître le premier graphe sur lequel nous allons travailler (c'est celui que nous avons vu en TD lorsqu'on a fait la trace de l'algorithme de Ford).

--- * ---

Pour le moment, `main.c` ressemble à

```c
#include "graphe_mat.h"
#include "pcc_tab.h"
int main()
{
    graphe_mat *g;
    /* partie commentée */
    g = gm_stable(8, 1); /* construction */
    gm_ajouter_arc(g, 0, 1, 2.);
    /* etc */
    gm_ajouter_arc(g, 7, 5, -3);
    gm_ecrire_dot(g, "premier_graphe.dot");
    /* partie commentée */
    gm_detruire(&g); /* destruction */
#if 0
    /* parties commentées */
#endif
```

```
    return EXIT_SUCCESS;
}
```

Les fichiers `graphe_mat.h` et `pcc_tab.h` contiennent les *déclarations* des types et des fonctions utilisées (ou qui le seront par la suite) dans `main.c`.

2 La mini bibliothèque `graphe_mat`

Choix pédagogique : travailler avec les graphes représentés par *matrice d'adjacence* pour simplifier ce sujet mais changer l'allocation *statique* de cette matrice par une allocation *dynamique* pour faire des révisions (et aussi parce que c'est mieux ainsi). Le fichier d'en-tête `graphe_mat.h` ressemble à :

```
#ifndef GRAPHE_MAT_H
#define GRAPHE_MAT_H
#define GM_COUT_PAR_DEF 1 /* cout par défaut d'un arc/arete */

/* graphe représenté par sa "matrice d'adjacence" allouée ↩
   dynamiquement.
 * les couts des arcs/arêtes sont représentés dans cout.
 * Attention au lieu de adj[i][j], on utilise maintenant
 * *(adj + i * n + j) ou adj[i * n + j]
 * Limitation : tous les arcs ayant même départ et même arrivée ont mê↩
    me
 * cout, afin de simplifier les choses */
struct s_graphe_mat {
    int n; /* nombre de sommets => V = {0 ,..., n -1} */
    int m; /* nombre d'arcs/arêtes (entier naturel) */
    int *adj; /* matrice d'adjacence */
    float *cout; /* cout associé aux arêtes */
    int is_or; /* 1 si le graphe est orienté et 0 sinon */
};
typedef struct s_graphe_mat graphe_mat;
/* retourne un pointeur vers un graphe stable (sans arc/arête) à n ↩
   sommets
 * (NULL si une des allocations mémoire a échoué)
 * le cout des arcs/arêtes est de type float et vaut GM_COUT_PAR_DEF ↩
    */
graphe_mat *gm_stable(int n, int is_or);
/* libère la mémoire associée à *g et met *g à NULL */
void gm_detruire(graphe_mat **g);
void gm_ajouter_arc(graphe_mat *g, int dep, int arr, float cout);
void gm_supprimer_arc(graphe_mat *g, int dep, int arr);
int gm_get_n(graphe_mat *g);
int gm_est_oriente(graphe_mat *g);
int gm_get_m(graphe_mat *g);
/* retourne le nombre d'arcs/arêtes (dep, arr) */
int gm_est_successeur(graphe_mat *g, int dep, int arr);
int gm_get_degre_sortant(graphe_mat *g, int dep);
int gm_get_degre_entrant(graphe_mat *g, int arr);
float gm_get_cout(graphe_mat *g, int dep, int arr);
void gm_set_cout(graphe_mat *g, int dep, int arr, float cout);
int gm_ecrire_dot(graphe_mat *g, char *nom_fic);
#endif
```

Les champs de la structure `graphe_mat` sont :

`n` le nombre de sommets du graphe, il est fixé lors du premier appel au constructeur `gm_stable`.

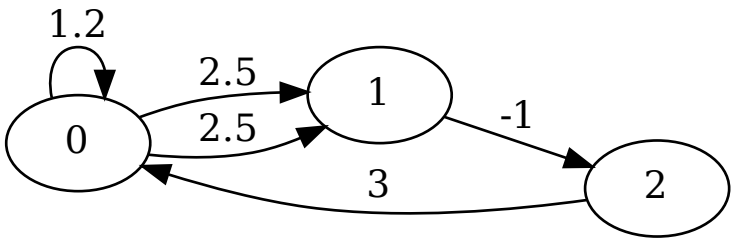
`m` le nombre d'arêtes du graphe, qui lui évolue. Il vaut 0 après l'appel du constructeur `gm_stable`.

`adj` pointeur vers une zone de la mémoire pouvant contenir n^2 éléments de type `int`. Cette zone est allouée dans `gm_stable` et est initialement remplie de 0. *Attention ! il ne s'agit donc plus d'un « vrai » tableau bi-dimensionnel !* Ainsi la variable égale au nombre d'arcs (ou arêtes) entre les sommets `dep` et `arr` est maintenant `g->adj[dep * g->n + arr]`, qu'on peut aussi écrire `*(g->adj + dep * g->n + arr)` car `g->adj` est l'adresse du premier élément de la zone allouée.

`cout` pointeur vers une zone de la mémoire pouvant contenir n^2 éléments de type `float`. Cette zone est allouée dans `graphe_stable` et est initialement remplie d'éléments égaux à la constante `GM_COUT_PAR_DEF`, qui est 1. Les mêmes remarques qu'au point précédent sont valables.

`is_or` égal à 1 si le graphe est orienté, 0 sinon. Fixé une fois pour toute lors de l'appel au constructeur `gm_stable`.

Question 1: Pour représenter avec cette structure, le graphe suivant



quelles doivent être les valeurs des différents champs de la structure (des zones mémoires pointées, s'il s'agit de pointeurs) ?

--- * ---

L'implémentation du constructeur `gm_stable` et du destructeur `gm_detruire` est la suivante, nous la discuterons juste après.

```
graphe_mat *gm_stable(int n, int is_or)
{
    graphe_mat *g;
    int i;
    if ( !(g = malloc(sizeof(*g))) ) {
        fprintf(stderr, "Erreur de malloc dans gm_stable\n");
        return NULL;
    }
    if ( !(g->adj = calloc( n*n, sizeof(int) )) ) { /* calloc met à 0 ↩
    */
        free(g);
        fprintf(stderr, "Erreur de malloc dans gm_stable\n");
        return NULL;
    }
    if ( !(g->cout = malloc( n*n*sizeof(float) )) ) {
        free(g->adj);
        free(g);
        fprintf(stderr, "Erreur de malloc dans gm_stable\n");
```

```
        return NULL;
    }
    for (i = 0; i < n*n; i++)
        g->cout[i] = GM_COUT_PAR_DEF;
    g->n = n;
    g->m = 0;
    g->is_or = is_or;
    return g;
}
void gm_detruire(graphe_mat **g)
{
    if ( ! *g )
        return;
    free((*g)->cout); /* parentheses nécessaires : -> plus prio que * ↩
       */
    free((*g)->adj);
    free(*g);
    *g = NULL;
}
```

Dans ces deux fonctions, les appels les plus importants sont ceux de `malloc`, `calloc` et `free`. Pour rappel (voir la deuxième édition française du K&R) :

`void *malloc(size_t size)` `malloc` retourne un pointeur sur un espace mémoire réservé à un objet de taille `size`, ou bien `NULL` si cette demande ne peut être satisfaite. La mémoire allouée n'est pas initialisée. [1]

`void *calloc(size_t nobj, size_t size)` `calloc` retourne un pointeur sur un espace mémoire réservé à un tableau de `nobj` objets, tous de taille `size`, ou bien `NULL` si cette demande ne peut pas être satisfaite. La mémoire allouée est initialisée par des zéros.

`void free(void *p)` `free` libère l'espace mémoire pointé par `p` ; elle ne fait rien si `p` vaut `NULL`. `p` doit être un pointeur sur un espace mémoire alloué par `calloc`, `malloc` ou `realloc`.

Dans la fonction `gm_stable`, dans l'expression `!(g = malloc ( sizeof (*g) ))` :

1. À la compilation, `*g` a la taille d'un élément de la structure `gm_graphe` (somme des tailles de 3 entiers et de 2 pointeurs, dépend de la machine, mais sur la mienne ça fait $3 \times 4 + 2 \times 8 = 28$ octets).
2. À l'exécution, les instructions du programme correspondant à `malloc(28)` (ou d'une autre valeur sur des machines différentes), réservent un bloc de 28 octets dans la mémoire de la machine et retourne un pointeur générique, de type `void *`, vers l'adresse du premier octet de ce bloc ou `NULL` en cas d'échec.
3. L'expression `g = malloc ( sizeof (*g) )` affecte l'adresse obtenue dans la variable `g` et fait un *cast* du type *pointeur générique* `void *` vers le type `graphe_mat *`. Ce cast est ici implicite et est licite car il s'agit d'un cast de `void *` (pointeur générique) vers un autre type de pointeur (seul cas où les *cast* implicites entre pointeurs sont autorisés).
4. La valeur de l'expression `g = malloc ( sizeof (*g) )` est celle de `g` *après l'affectation*. Ainsi, en cas d'erreur, `g` contiendra la valeur `NULL` et `!(g = malloc ( sizeof (*g) ))` sera évalué à `1`. On entrera alors dans le bloc `if`, comme il se doit.

Le reste de la fonction est du même genre (mais si vous ne la comprenez pas, posez des questions).
Question 2: Que se passe-t-il si dans `gm_stable` les 2 premières acquisitions de mémoire sont réussies mais que le `malloc` pour `g->cout` échoue ? Pourquoi est-ce que le paramètre de la

1. Le type `size_t` dépend en principe de la machine mais il s'agit toujours d'un type entier non signé, en général `long`. Sur ma machine, la directive `#include <stdlib.h>` conduit à l'inclusion de la ligne `typedef long unsigned int size_t;`

fonction `gm_detuire` est un double pointeur ? Dans les appels à `free` de cette fonction, lesquels sont interchangeables ?

--- * ---

Passons maintenant aux fonctions suivantes. Elles illustrent les règles de priorité du C. Les opérateurs de plus forte priorité en C sont

`()` appel de fonction ou parenthésage d'expression

`[]` accès à un élément de tableau ou à une valeur pointé.

`->` accès à un champ d'une structure via un pointeur vers une variable de type structure.

`.` accès à un champ d'une structure via une variable de type structure.

Ces 4 opérateurs ont *la même priorité* et sont *associatifs de gauche à droite*, ce qui signifie que, par exemple `a->b->c` est équivalent à `(a->b) -> c` et `a[b].champ->c` est équivalent à `((a[b]).champ)->c`.

```c
void gm_ajouter_arc(graphe_mat *g, int dep, int arr, float cout)
{
    g->m ++;
    g->adj[dep * g->n + arr] ++;
    g->cout[dep * g->n + arr] = cout;
    if (! g->is_or ) {
        g->adj[arr * g->n + dep] ++;
        g->cout[arr * g->n + dep] = cout;
    }
}
void gm_supprimer_arc(graphe_mat *g, int dep, int arr)
{
    /* on ne supprime pas une arête qui n'existe pas*/
    if (! g->adj[dep * g->n + arr] )
        return;
    g->m --;
    g->adj[dep * g->n + arr] --;
    if (! g->is_or )
    g->adj[arr * g->n + dep] --;
}
```

Question 3: Si les `int` sont représentés en machine par 4 octets, alors `g->adj + 1` (choisir la bonne réponse)

1. est l'adresse de l'octet qui suit celui d'adresse `g->adj`.
2. n'est pas une adresse : c'est un `int`.
3. est l'adresse correspondant à 4 octets après `g->adj`.
4. n'est pas une expression valide du C.

--- * ---

3 Mini bibliothèque de plus courts chemins représentés par tableaux

Pour donner la solution de nos problèmes de plus courts chemins, on travaille avec le type structuré `pcc_tab` introduit dans `pcc_tab.h`.

```c
#include <math.h> /* pour INFINITY , isfinite, isinf */
#include "graphe_mat.h"
#define INFINI INFINITY
#define NON_DEF -1 /* lorsque le prédecesseur est non défini */
```

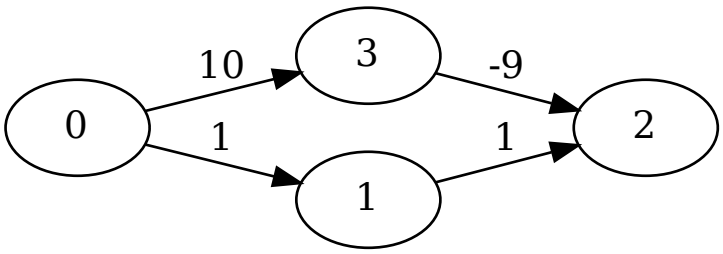
```c
typedef struct s_pcc_tab {
    graphe_mat *g; /* le graphe sur lequel on se pose le pb */
    int s; /* départ des pcc */
    int *pred; /* tableau des prédecesseurs du pcc */
    float *pot; /* tableau des potentiels */
} pcc_tab;

pcc_tab *pcct_allouer(graphe_mat *g);
void pcct_detruire(pcc_tab **p);
pcc_tab *pcct_ford(graphe_mat *g); /* 1ere version : toujours depuis 0 */
int pcct_ecrire_dot(pcc_tab *p, char *nom_fic);
```

On a besoin :

1. de la constantes symboliques `INFINI` dont on a fixé la valeur à `INFINITY` de la bibliothèque standard, défini dans `math.h`). Dans `math.h` sont aussi définies les macros `isinf` et `isfinite` qui testent si leur paramètre, un nombre à virgule flottante est infini (respectivement, est fini) [2] ;
2. de la constante symbolique `NON_DEF` qu'on a fixée à -1 (car ce n'est pas un sommet du graphe) pour dire qu'un prédecesseur d'un sommet n'est pas encore défini ;
3. d'un sommet de départ `s` ;
4. du graphe `g` sur lequel on cherche à trouver les plus courts chemins issus de s.
5. d'un tableau (dynamique) de n entiers pour stocker les prédecesseurs pendant la résolution du problème (`pred`) ;
6. d'un tableau (dynamique) de n flottants pour les potentiels (`pot`) ;

Question 4: Pour représenter avec cette structure, la solution du problème des plus courts chemins issus de 0 sur le graphe suivant



quelles doivent être les valeurs des différents champs de la structure (des zones mémoires pointées, s'il s'agit de pointeurs) ?

--- * ---

On a besoin d'un allocateur de mémoire pour cette structure et d'un destructeur. C'est le rôle joué par

2. Les calculs portant sur les infinis (par exemple `INFINI + 2` ou `INFINI < 2`) peuvent fonctionner (c'est le cas avec `gcc`) mais peuvent aussi donner des résultats inattendus car ne sont pas spécifiés dans le standard. On essaiera donc de les éviter en utilisant les tests `isinf` et `isfinite`.

```
pcc_tab *pcct_allouer(graphe_mat *g)
{
    pcc_tab *p;
    if ( ! *g ) {
        fprintf(stderr,
                "Erreur : allocation de pcc sur un graphe non instanci↩
                   é.\n");
        return NULL;
    }
    if ( !(p = malloc(sizeof(*p))) ) {
        fprintf(stderr, "Erreur de malloc dans pcct_allouer\n");
        return NULL;
    }
    p->g = g;
    if ( !(p->pot = malloc(sizeof(float) * g->n)) ) {
        fprintf(stderr, "Erreur de malloc dans pcct_allouer\n");
        free(p);
        return NULL;
    }
    if ( !(p->pred = malloc(sizeof(int) * g->n)) ) {
        fprintf(stderr, "Erreur de malloc dans pcct_allouer\n");
        free(p->pot);
        free(p);
        return NULL;
    }
    return p;
}
void pcct_detruire(pcc_tab **p)
{
    if (! *p )
        return;
    free( (*p) -> pot );
    free( (*p) -> pred );
    free(*p);
    *p = NULL;
}
```

L'implémentation est très similaire à celle du constructeur et du destructeur dans `graphe_mat.c` donc nous passons à la suite. Mais si vous avez des difficultés pour les comprendre, posez des questions à votre chargé de TP.

4 Algorithme de Ford

L'allocateur précédent n'est pas un constructeur car la plupart des champs ne sont pas initialisés. Notre premier constructeur sera une implémentation de l'algorithme de Ford. À vous de compléter l'implémentation de la fonction suivante :

```
pcc_tab *pcct_ford(graphe_mat *g)
{
    pcc_tab *p;
    /* déclarations supplémentaires */

    /* allocation */
    if ( ! g ) {
        fprintf(stderr,
          "Erreur : pcc_ford sur graphe non instancié\n");
        return NULL;
```

```
    }
    if (! (p = pcct_allouer(g)) )
        return NULL;
    p->s = 0; /* 1ere version */
    /* initialisation */

    /* boucle principale */

    return p;
}
```

On commence en imposant que s vaut 0 pour coller à l'algorithme vu en cours. D'ailleurs on le rappelle ci-dessous.

Entrée $G = (V, E)$, où $V = \{0; 1; ...; n-1\}$, graphe orienté, arc-valué par c, sans circuit absorbant.

Sortie Plus courts chemins issus de 0 représentés par les relations $P : V \to V$ (parent dans l'arborescence associée) et $\pi : V \to \mathbb{R} \cup \{+\infty\}$ (le potentiel des plus courts chemins).

`/* Initialisation de` P `et` π `*/`.

1. $P(0) \leftarrow 0$ et $\pi(0) \leftarrow 0.0$.
2. Pour v de 1 à $n-1$, faire $(P(v), \pi(v)) \leftarrow$ (`NON_DEF`, `INFINI`).

`/* Boucle principale */`.

3. $v \leftarrow 0$
4. Tant que $v \leq n-1$
   (a) `prochain` $\leftarrow v+1$
   (b) Pour tout successeur w de v :
      i. `val` $\leftarrow \pi(v) + c(v, w)$
      ii. Si `val` $< \pi(w)$
         A. $\pi(w) \leftarrow$ `val`
         B. $P(w) \leftarrow v$
         C. Si $w <$ `prochain`, alors `prochain` $\leftarrow w$.
   (c) $v \leftarrow$ `prochain`
5. Retourner P et π.

Question 5: Compléter la fonction `pcct_ford` en traduisant cet algorithme en langage C. Décommenter les deux premiers blocs dans `main` puis compiler et exécuter. Utiliser la commande `dot -Tpng ford1.dot -o ford1.png` (ou modifier le script fourni) pour visualiser le résultat.

--- * ---

Question 6: Modifier la fonction `pcct_ford` (et sa déclaration dans le fichier d'en-tête) de manière à ce qu'elle admette un second argument s donnant le point de départ des plus courts chemins. Indice : si une variable `i` parcourt les entiers de 0 à $n-1$, alors `(s+i) % n` parcourt dans cet ordre les entiers s, $s+1$, ..., $n-1$, 1, ..., $s-1$.

--- * ---

Question 7: On veut maintenant tester notre fonction sur un autre graphe. Commenter toute la première partie de `main` (du début au commentaire `/* Ford : 2ème essai */`) et décommenter cette partie (jusqu'à `/* Bellman */`). Compiler et exécuter. Que se passe-t-il ? Pourquoi ? (Vous pouvez, pour vous aider, visualiser le graphe de départ).

--- * ---

Question 8: On souhaite que notre fonction `pcct_ford` termine toujours. D'après le cours, il faut pour cela que chaque fois que l'algorithme doit mettre à jour le potentiel d'un sommet w par la considération d'un arc (v, w) on doit tester si w est sur le chemin entre s et v dans l'arborescence issue de s construite jusqu'à maintenant (si c'est le cas, le graphe possède un circuit absorbant et il faut donc sortir avec un message d'erreur).

Créer une fonction intermédiaire `int pcct_est_ancetre(pcc_tab *p, int v, int w)` qui retourne 1 si w est un ancêtre de v dans p, c'est-à-dire si w appartient à $\{v, P(v), P(P(v)), ..., s\}$ et 0 sinon.

Utiliser cette fonction pour détecter les circuits absorbant dans `pcct_ford`. Tester cette modification.

--- * ---

5 Algorithme de Bellman

Comme vu en cours, l'algorithme de Bellman est très efficace mais a la contrepartie de ne pouvoir s'appliquer qu'aux graphes sans circuit. Comme précédemment, on va procéder par étapes et d'abord supposer que le graphe vérifie cette condition. On implémentera ensuite un test pour détecter les circuits.

L'algorithme donné dans le cours est le suivant.

Entrée $G = (V, E)$, graphe orienté, simple, sans circuit, arc-valué par c et s dans V.

Sortie Plus courts chemins issus de 0 représentés par les relations $P : V \to V$ (parent dans l'arborescence associée) et $\pi : V \to \mathbb{R}$ (le potentiel des plus courts chemins), *ainsi qu'une suite S de sommets qui représente un ordre topologique.*

`/* Initialisations */`

1. S est vide
2. Z est vide
3. Pour v dans V,
   (a) $deg[v] \leftarrow$ le degré entrant de v dans G
   (b) Si $deg[v]$ est nul, alors mettre v dans Z.
   (c) Si v est le sommet s, alors $P(v) \leftarrow v$ et $\pi(v) \leftarrow 0$.
   (d) Sinon, $P(v) \leftarrow$ `NON_DEF` et $\pi(v) \leftarrow$ `INFINI`.

`/* Boucle principale */`

4. Répéter n fois
   (a) Extraire un sommet v de Z et l'insérer en fin de S.
   (b) Pour w successeur de v,
      i. Diminuer $deg[w]$ de 1.
      ii. Si $deg[w]$ est nul, alors insérer w dans Z.
      iii. `val` $\leftarrow \pi(v) + c(v, w)$.
      iv. Si $\pi(w) >$ `val`, alors
         A. $\pi(w) \leftarrow$ `v` ;
         B. $P(w) \leftarrow v$
5. Retourner S, P et π.

Pour mettre en œuvre cet algorithme, nous devons nous poser la question de la représentation de S, Z et de la fonction de degré entrant résiduel deg.

Pour deg, un tableau de n entiers fera certainement l'affaire. Sur S, on sait qu'on doit pouvoir insérer à la fin (et on ne supprime jamais), cela suggère d'utiliser une pile ou une file. Pour Z, on sait seulement qu'on doit pouvoir extraire et ajouter donc une pile ou une file seraient toutes les deux adaptées.

Une autre particularité de S et de Z est qu'ils ne peuvent contenir au maximum que n sommets.

Par *soucis de simplicité*, nous allons utiliser deux piles, l'une pour S et l'une pour Z, et, n'ayez crainte, nous allons directement utiliser des tableaux pour les implémenter avec simplement deux compteurs de taille `tailleZ` et `tailleS` pour connaître la hauteur des piles.

Ainsi, au départ, `tailleZ` et `tailleS` sont nuls. Insérer un élément w à la fin de S devient

```
S[tailleS] = w;
tailleS ++;
```

qu'on peut écrire de façon plus compacte

```
S[tailleS++] = w;
```

Rappel : la *valeur* de l'expression `tailleS++` est celle de `tailleS`. La valeur de `++tailleS` est celle de `tailleS + 1`.

Extraire un élément (en fin) de Z devient

```
v = Z[tailleZ - 1];
tailleZ --;
```

qu'on peut abréger en

```
v = Z[--tailleZ];
```

Pour vous faire gagner du temps, on a déjà codé les parties d'allocations et de ménage de la fonction (la prochaine fois, ce sera à vous de le faire).

`pcc_tab *pcct_bellman(graphe_mat *g, int s, int **ordre_top)`

```
pcc_tab *pcct_bellman(graphe_mat *g, int s, int **ordre_top)
{
    pcc_tab *p;
    int *Z, *S, *deg;
    unsigned tailleZ, tailleS;
    /* autres déclarations si nécessaire*/

    /* allocations mémoire */
    Z = S = deg = NULL;
    p = NULL;
    if ( ! (p = pcct_allouer(g)) ||
         ! (Z = malloc( g->n * sizeof(int) )) ||
         ! (S = malloc( g->n * sizeof(int) )) ||
         ! (deg = malloc( g->n * sizeof(int) ))
       ) {
        fprintf(stderr, "Erreur de malloc dans pcct_bellman\n");
        pcct_detruire(&p); free(Z); free(S);
        return NULL;
    }
    /* initialisations */
    tailleZ = tailleS = 0; /* au départ Z et S sont vides */
```

```
    /* à vous de jouer ! */
    /* boucle principale */

    /* à vous de jouer ! */

    /* ménage et retour */
    free(Z); free(deg);
    *ordre_top = S;
    return p;
}
```

Question 9: Finir l'implémentation de cette fonction. La tester avec le graphe fourni dans `main` sous le commentaire `/* Bellman */` d'abord en partant du sommet 0, puis en partant de 3. Chaque fois, visualiser le résultat avec le fichier `.dot` écrit par `pcct_ecrire_dot`.

--- * ---

Question 10: Bien sûr c'est beaucoup plus drôle sur un graphe plus gros et aléatoire. Tester la même fonction en commentant la partie `/* Bellman */` et en décommentant `/* Bellman 2 */`. Visualiser le résultat. Vous pouvez aussi essayer avec des graphes beaucoup plus gros, mais c'est la partie visualisation qui risque de poser problème.

--- * ---

Question 11: Passer à la partie `/* Bellman 3 */`. Que se passe-t-il et pourquoi ?

--- * ---

Question 12: D'après le cours, on peut détecter la présence d'un circuit lorsqu'il arrive (en-dehors de l'initialisation) que Z soit vide alors que S n'est pas V tout entier. Implémenter cette modification et la tester avec la partie `/* Bellman 3 */`.

--- * ---

Question 13: Quelle est la complexité de l'algorithme de Bellman avec notre représentation des graphes ? Comment l'améliorer ?

--- * ---

6 Algorithme de Ford-Bellman

L'algorithme de Ford-Bellman, vu en cours, est l'un des plus robuste et a une complexité acceptable. Maintenant, vous devriez avoir le niveau pour implémenter cet algorithme sans aide.

Bien sûr, on commence par supposer qu'il n'y a pas de circuit absorbant, on verra ensuite pour implémenter la détection de tels circuits.

L'algorithme vu en cours est le suivant. Pendant l'algorithme, on maintient (en plus de π et P) :

1. une fonction π_{it} qui décrit l'évolution du potentiel pendant l'itération courante ;
2. l'ensemble M des sommets dont le potentiel a été modifié à l'itération précédente et
3. l'ensemble M_{it} des sommets dont le potentiel a été modifié pendant l'itération courante.

Entrée $G = (V, E)$, graphe orienté, simple, sans circuit, arc-valué par c et s dans V.

Sortie Plus courts chemins issus de 0 représentés par les relations $P : V \to V$ (parent dans l'arborescence associée) et $\pi : V \to \mathbb{R}$ (le potentiel des plus courts chemins).

`/* Initialisation */`

1. M contient seulement s.

2. Pour v dans V,
   (a) Si v est le sommet s, alors $P(v) \leftarrow v$ et $\pi(v) \leftarrow 0$;
   (b) sinon, $P(v) \leftarrow$ `NON_DEF` et $\pi(v) \leftarrow$ `INFINI`.
3. $k \leftarrow 1$.

`/* Boucle principale */`

4. Tant que $k \leq n-1$ et M est non vide,
   (a) M_{it} devient vide,
   (b) $\pi_{\text{it}} \leftarrow \pi$,
   (c) Pour `dep` dans M,
      i. pour `arr` successeur de `dep`,
         A. `val` $\leftarrow \pi(v) + c(v, w)$.
         B. Si `val` $< \pi_{\text{it}}(w)$, alors
            $\pi_{\text{it}}($`arr`$) \leftarrow$ `val`,
            $P($`arr`$) \leftarrow$ `dep` et
            mettre `arr` dans M_{it}.
   (d) $M \leftarrow M_{\text{it}}$, $k \leftarrow k+1$.
   (e) Pour v dans M_{it}, $\pi(v) \leftarrow \pi_{\text{it}}(v)$.
5. Retourner π et P.

Question 14: Comment représenter les ensembles M et M_{it} en mémoire ?

--- * ---

Question 15: Implémenter cet algorithme dans une fonction
`pcc_tab *pcc_fb(graphe_mat *g, int s)`.
Tester votre fonction avec le graphe aléatoire de la partie `/* Ford-Bellman avec graphe aléatoire à poids positifs */` de `main`. Visualiser le résultat.

--- * ---

Question 16: Dans le cours, il est dit qu'il y a un circuit absorbant (accessible depuis s) dans G si et seulement si lors d'une n-ième itération, l'algorithme améliore le potentiel d'au moins un sommet. Implémenter la détection de circuits absorbants depuis s dans votre fonction et la tester, par exemple avec le graphe de test de Ford numéro 2.

--- * ---