Informatique en MP(2)I

Marc de Falco

Travaux Pratiques - Graphes

I Parcours de graphes en C

I.1 Représentation

On va considérer le type suivant pour les graphes qui suppose qu'on n'aura jamais plus que `MAXV` sommets. On utilise ici une constante avec l'alias `#define` : partout où on écrit `MAXV`, il sera remplacé par la valeur 100.

```c
#define MAXV 100 /* nombre maximum de sommets */

struct edgenode {
  int y; // le voisin
  struct edgenode *next; // la suite de la liste
};
typedef struct edgenode edgenode;

struct graph {
  edgenode *edges[MAXV]; // tableau de listes d'adjacence
  int degree[MAXV]; // le degré de chaque sommet
  int nvertices;
  int nedges;
  bool directed; // indique si le graphe est orienté
};
typedef struct graph graph;
```

C

Question I.1.1 Écrire une fonction

```c
void initialize_graph(graph *g, bool directed);
```

C

qui initialise le graphe `g` passé par pointeur comme étant le graphe vide, dirigé ou non selon la valeur de `directed`.

Attention vous êtes libres d'initialiser les listes d'adjancence à la liste vide ou de considérer que sera fait dans la fonction `read_graph` ci-dessous.

■ **Preuve**

```c
void initialize_graph(graph *g, bool directed)
{
  g->directed = directed;
  g->nvertices = 0;
  g->nedges = 0;
  for (int i = 0; i < MAXV; i++)
  {
    g->edges[i] = NULL;
    g->degree[i] = 0;
  }
}
```

■

Question I.1.2 Écrire une fonction

```c
void insert_edge(graph *g, int x, int y, bool directed)
```

qui insère une arête de x à y en considérant `g` comme orienté ou non selon **`directed` et donc en ignorant `g->directed`**.

Attention il faudra prendre garde au fait que `g` soit orienté ou non. Dans le second cas, on représenté les arêtes comme dans un graphe orienté symétrique, donc il faut en ajouter deux. C'est la raison pour laquelle on utilise le paramètre `directed` plutôt que de consulter `g->directed`.

■ **Preuve**

```c
void insert_edge(graph *g, int x, int y, bool directed)
{
  edgenode *edge = malloc(sizeof(edgenode));
  edge->y = y;
  edge->next = g->edges[x];
  g->edges[x] = edge;
  if (!directed)
    insert_edge(g, y, x, true);
}
```

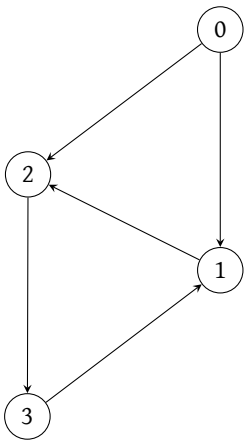
■

Pour travailler sur des graphes, on va écrire une fonction permettant de lire un fichier contenant le graphe sous le format suivant :

- première ligne contenant trois entiers, le nombre de sommets n, le nombre d'arêtes p et 0 ou 1 selon que le graphe soit non orienté ou orienté
- ensuite p lignes contenant deux entiers i et j et indiquant qu'il y a une arête de i vers j

Par exemple :

```
4 5 1
0 1
0 2
1 2
2 3
3 1
```



sera représenté par le graphe :

Remarque On va utiliser ici l'entrée standard, c'est-à-dire l'entrée de l'utilisateur depuis le terminal. Cependant, il est possible de rédiriger cette entrée depuis un fichier. En effet, si on appelle

```
./monprogramme < monfichier
```
C

Alors l'entrée standard sera le contenu de `monfichier`. Cela permet de ne pas avoir à se préoccuper d'ouvrir de fichier et d'utiliser directement `scanf`.

Rappel `scanf("%d %d", &x, &y);` va lire une ligne avec deux entiers et placer la valeur du premier dans `x` et la valeur du second dans `y`.

Question I.1.3 Écrire une fonction

```
void read_graph(graph *g)
```
C

qui lit un graphe depuis l'entrée standard et le place dans `g` **après l'avoir initialisé**.

■ **Preuve**

```c
void read_graph(graph *g)
{
  int directed;
  initialize_graph(g, false);
  scanf("%d %d %d", &g->nvertices, &g->nedges, &directed);
  g->directed = directed == 1;
  for (int i = 0; i < g->nedges; i++)
  {
    int x, y;
    scanf("%d %d", &x, &y);
    insert_edge(g, x, y, g->directed);
  }
}
```
C

■

Question I.1.4 Écrire une fonction

```
void free_edges(graph *g)
```
C

qui libère les listes d'adjacence.

■ **Preuve**

```c
void free_edges(graph *g)
{
  for (int i = 0; i < g->nvertices; i++)
```

```c
  {
    edgenode *n = g->edges[i];
    while(n != NULL)
    {
      edgenode *temp = n->next;
      free(n);
      n = temp;
    }
  }
}
```

■

I.2 Parcours en profondeur récursif

On va modifier la structure de graphe et rajouter trois nouveaux champs :

```c
bool discovered[MAXV]; // Quels sommets sont connus
bool processed[MAXV]; // Quels sommets sont traités
int parent[MAXV]; // parent[x] est le père de x dans le parcours
         // s'il n'y en a pas, c'est -1
```

Question I.2.1 Écrire une fonction

```c
void initialize_search(graph *g);
```

qui initialise ces tableaux pour commencer une nouvelle recherche.

■ **Preuve**

```c
void initialize_search(graph *g)
{
  for (int i = 0; i < g->nvertices; i++)
  {
    g->discovered[i] = false;
    g->processed[i] = false;
    g->parent[i] = -1;
  }
}
```

■

On va définir trois fonctions qui seront appelées lors d'un parcours et qu'on pourra redéfinir.

```c
void process_vertex_early(graph *g, int v)
{
  printf("processing vertex %d\n", v);
}

void process_vertex_late(graph *g, int v)
{
}

void process_edge(graph *g, int x, int y)
{
  printf("processed edge %d --> %d\n", x, y);
}
```

Question I.2.2 Écrire une fonction récursive

```c
void dfs(graph *g, int x)
```

qui effectue un parcours en profondeur en partant du sommet `x`.

Cette fonction va appeler `process_vertex_early` au début du traitement de `x`, `process_vertex_late` à la fin et `process_edge` pour chaque arête rencontrée.

Attention si le graphe est non orienté et qu'on a rencontré `x -> y` dans cet ordre, on ne fera pas de traitement dans le sens `y -> x`.

■ **Preuve**

```c
void dfs(graph *g, int x)
{
  // important uniquement pour le premier
  g->discovered[x] = true;
  process_vertex_early(g, x);

  edgenode *n = g->edges[x];
  while(n != NULL)
  {
    process_edge(g, x, n->y);
    if (!g->discovered[n->y])
    {
      g->discovered[n->y] = true;
      g->parent[n->y] = x;
      g->color[n->y] = !g->color[x];
      dfs(g, n->y);
    }
    n = n->next;
  }

  process_vertex_late(g, x);
  g->processed[x] = true;
}
```

■

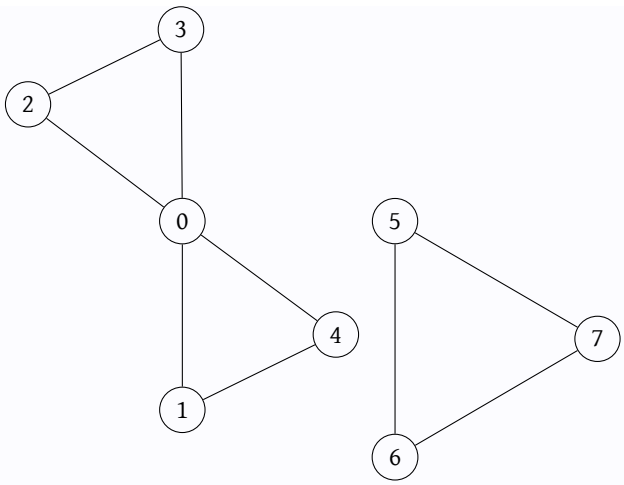
Question I.2.3 Comme on l'a démontré, le parcours dans un graphe non orienté permet de traiter exactement la composante connexe du sommet de départ.

Écrire une fonction

```c
void connected_components(graph *g)
```

qui affiche les composantes connexes sous la forme :

```
Component 1: 0 1 2 3 4
Component 2: 5 6 7
```



pour le graphe
Il sera nécessaire de modifier les fonctions `process_*`.

■ **Preuve**

On remplace les fonctions par :

```c
void process_vertex_early(graph *g, int v)
{
  printf(" %d", v);
}

void process_vertex_late(graph *g, int v)
{
}

void process_edge(graph *g, int x, int y)
{
}
```

et on définit :

```c
void connected_components(graph *g)
{
  initialize_search(g);
  int comp = 0;
  for(int i = 0; i < g->nvertices; i++)
  {
    if(!g->processed[i])
    {
      comp = comp+1;
      printf("Component %d:", comp);
      dfs(g, i);
      printf("\n");
    }
  }
}
```

On relance ainsi le parcours en repartant d'un sommet non traité.

■

Question I.2.4 Adaptez le parcours pour détecter des cycles. Quand vous détectez un cycle, affichez les sommets qui le compose. Dans le graphe précédent il faut afficher :

```
Cycle : 0 2 3
Cycle : 0 1 4
Cycle : 5 6 7
```

■ **Preuve**

Il suffit de changer `process_edge` dans le parcours de toutes les composantes pour détecter les arêtes arrières :

```c
void process_edge(graph *g, int x, int y)
{
  // Teste si on a une arête arrière
  if (g->discovered[y]
      && !g->processed[y]
      && (g->directed || g->parent[x] != y))
  {
    printf("Cycle : %d", y);
    int cur = x;
    while(cur != y & cur > -1)
    {
      printf(" %d", cur);
      cur = g->parent[cur];
    }
    printf("\n");
  }
}
```

■

Question I.2.5 A l'aide d'un parcours, déterminez si un graphe non orienté est biparti.

■ **Preuve**

On rajoute des champs

```c
bool colors[MAXV];
bool bipartite;
```

dans `graph` et on commence en donnant la couleur `true` au sommet de départ du DFS. Ensuite, il suffit de colorer avec la couleur différente de x quand on voit une arête $x \to y$ puis de vérifier à chaque arête qu'elle relie des sommets de couleur différente.

```c

```

■

I.3 Temps et classification des arêtes

On rajoute à la structure `graph` deux tableaux et un entier :

```c
int time; // l'horloge
int entry_time[MAXV];
int exit_time[MAXV];
```

Question I.3.1 Rajouter à `dfs` le maintien de l'horloge et des temps d'entrée et de sortie.

■ **Preuve**

Il suffit de modifier les traitements de sommets :

```c
void process_vertex_early(graph *g, int v)
{
  g->entry_time[v] = g->time;
  g->time = g->time + 1;
  printf("%d\n", v);
}
```

```c
void process_vertex_late(graph *g, int v)
{
  g->exit_time[v] = g->time;
  g->time = g->time + 1;
}
```

■

On définit des constantes :

```c
#define TREE 0
#define BACK 1
#define FORWARD 2
#define CROSS 3
```

Question I.3.2 Écrire une fonction :

```c
int edge_classification(graph *g, int x, int y)
```

qui pourra être appelée dans `process_edge` pour déterminer la classe d'une arête selon les constantes précédentes. On renverra -1 si l'arête ne peut être déterminée (est-ce possible ?).

■ **Preuve**

```c
int edge_classification(graph *g, int x, int y)
{
  if (!g->discovered[y])
    return TREE;
  if (!g->processed[y])
    return BACK;
  if (g->entry_time[x] < g->entry_time[y])
    return FORWARD;
  return CROSS;
}
```

■

I.4 Parcours avec une structure

On va réutiliser ici des implémentations de files et de piles dans un tableau de taille fixe.

Question I.4.1 Compléter les implémentations en se basant sur ce qui a déjà été fait dans les TP précédents.

```c
struct queue {
  int elts[MAXV];
  int front;
  int back;
};
typedef struct queue queue;

struct stack {
  int elts[MAXV];
  int back;
};
typedef struct stack stack;

void init_stack(stack *s)
{
  // initialise la pile
}

int pop(stack *s)
{
  // depile un élément
}

void push(stack *s, int x)
{
  // empile x sur la pile s
}

void init_queue(queue *s)
{
  // initialise la file
}

int dequeue(queue *s)
{
  // defile un élément
}

void enqueue(queue *s, int x)
{
  // enfile x sur la file s
}
```

c

■ **Preuve**

```c
void init_stack(stack *s)
{
  s->back = 0;
}

int pop(stack *s)
{
  assert(s->back > 0);
  s->back = s->back-1;
  return s->elts[s->back];
}

void push(stack *s, int x)
{
  assert(s->back < MAXV-1);
  s->elts[s->back] = x;
  s->back = s->back+1;
}

bool empty_stack(stack *s)
```

```c
{
  return s->back == 0;
}

void init_queue(queue *s)
{
  s->front = 0;
  s->back = 0;
}

int dequeue(queue *s)
{
  int x = s->elts[s->front];
  s->front = (s->front + 1) % MAXV;
  return x;
}

void enqueue(queue *s, int x)
{
  s->elts[s->back] = x;
  s->back = (s->back + 1) % MAXV;
}

bool empty_queue(queue *s)
{
  return s->front == s->back;
}
```

C

■

Question I.4.2 Écrire des fonctions

```c
void dfs(graph *g, int src);
void bfs(graph *g, int src);
```

C

effectuant un parcours en utilisant pour les sommets à visiter une pile ou une file. On ne pourra pas maintenir les temps de sortie ici car cela n'a plus vraiment de sens.

■ **Preuve**

```c
void dfs(graph *g, int src)
{
  stack s;
  init_stack(&s);
  push(&s, src);
  while(!empty_stack(&s))
  {
    int x = pop(&s);
    if (!g->processed[x])
    {
      g->processed[x] = true;
      process_vertex_early(g, x);

      edgenode *n = g->edges[x];
      while(n != NULL)
      {
        push(&s, n->y);
        n = n->next;
      }
    }
  }
}

void bfs(graph *g, int src)
```

```c
{
  queue s;
  init_queue(&s);
  enqueue(&s, src);
  while(!empty_queue(&s))
  {
    int x = dequeue(&s);
    if (!g->processed[x])
    {
      g->processed[x] = true;
      process_vertex_early(g, x);

      edgenode *n = g->edges[x];
      while(n != NULL)
      {
        enqueue(&s, n->y);
        n = n->next;
      }
    }
  }
}
```

c

■

Question I.4.3 A l'aide d'un parcours en largeur, le `bfs`, déterminez étant donné un sommet `x` et un sommet `y` de sa composante connexe, le plus court chemin de `x` vers `y`. On l'affichera sous la forme `0 -- 3 -- 1 -- 5`.

■ **Preuve**

On rajoute un champ comme vu dans le cours :

```c
int d[MAXV];
```

c

on marque `d[x] = 0` et on utilise le any-search modifié pour tenir compte de la parenté :

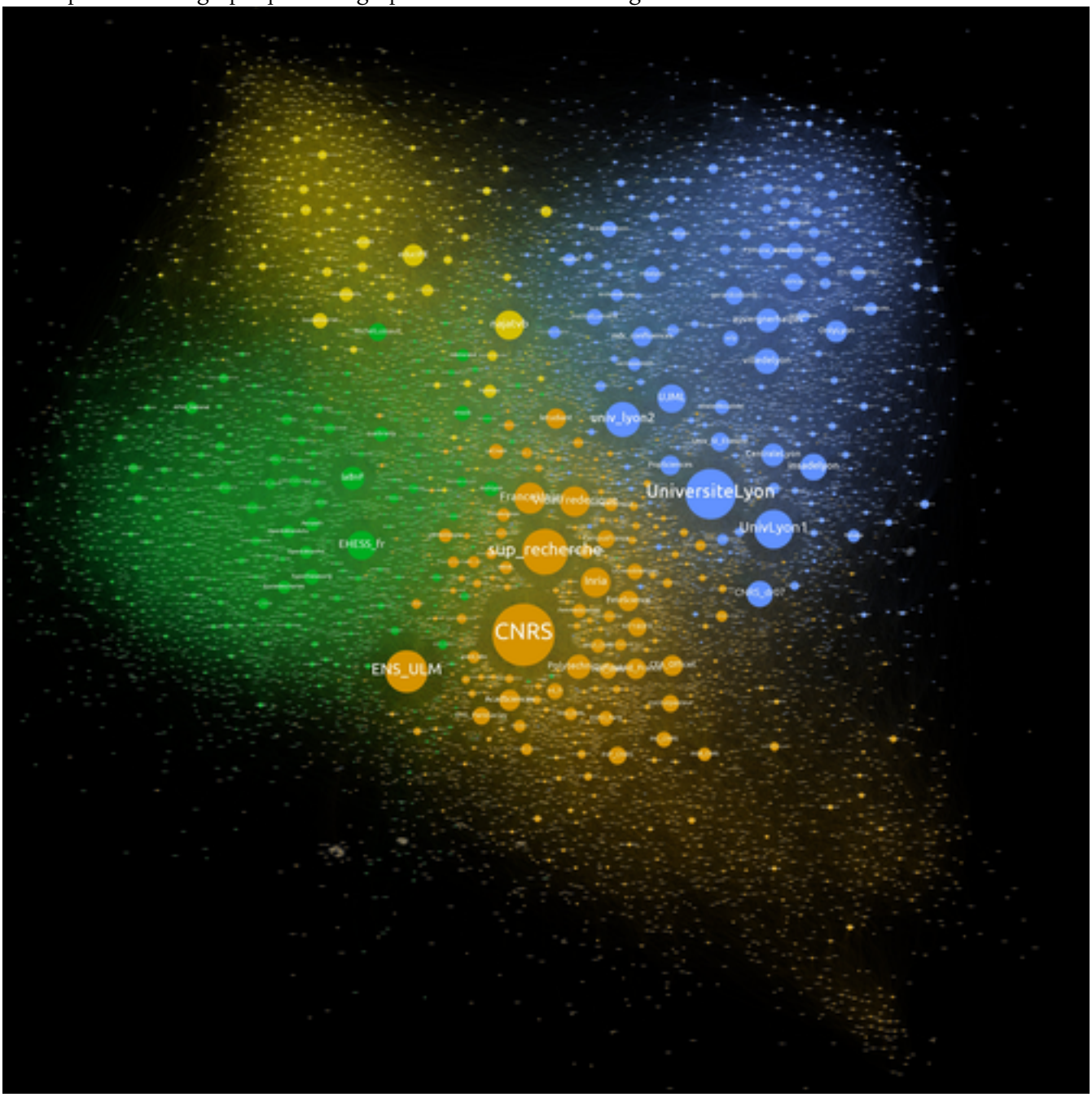
```c

```

c

■

II Étude d'un graphe issu d'un réseau social

Dans ce TP on va étudier le graphe **orienté** des followers du compte Twitter `@ENSdeLyon`.
Une représentation graphique de ce graphe est donné dans l'image suivante :



Les sommets sont les comptes donnés par leur identifiant (le `@identifiant` de Twitter) et une arête $x \to y$ indique que le compte x est abonné au compte y.

Ce graphe est assez conséquent : il comporte 8418 sommets et 305288 arêtes. Il nous permettra ainsi d'étudier en pratique la complexité des différents algorithmes étudiés. On va commencer par lire ce graphe depuis un fichier, ensuite, on en déduira différents graphes associés (sous-graphes, symétrisés par excès ou par défaut...) sur lesquels on pourra appliquer les algorithmes demandés. Une table de résultat est fourni en fin de TP pour vérifier vos résultats.

II.1 Définition et lecture du graphe

Le graphe est donné dans le fichier ENSdeLyon.graph. Il s'agit d'un fichier texte ayant la structure suivante :

- un entier `n_sommets` sur une ligne
- un entiers `n_aretes` sur une ligne

- `n_sommets` lignes contenant une chaîne de caractère représentant l'identifiant d'un sommet
- `n_aretes` couple de lignes comportant sur la première un entier `src` et sur la seconde un entier `tgt` indiquant une arête `src -> tgt`.

Question II.1.1 Expliquer pourquoi cela ne semble pas être une bonne idée de représenter ce graphe par une matrice d'adjacence.

On va utiliser le type suivant permettant de représenter le graphe par listes d'adjacence :

```
type graphe = {
  sommets : string array;
  aretes : int list array
}
```
OCaml

Pour lire le graphe depuis le fichier, le plus simple est de le rediriger sur l'entrée standard (*Rappel* `./monprogramme < monfichier`) et d'utiliser les deux fonctions suivantes :

- `read_int : unit -> int` lit une ligne composée d'un entier et renvoie sa valeur.
- `read_line : unit -> string` lit une ligne et la renvoie sans le caractère de saut de ligne, c'est-à-dire, sans le `'\n'`.

Alternativement, on peut lire le graphe depuis un fichier avec :

- `open_in : string -> in_channel` qui crée un descripteur de fichier en lecture pour le nom de fichier passé en paramètre
- `input_line : in_channel -> string` qui lit une ligne dans le descripteur et la renvoie sans le saut de ligne
- `int_of_string : string -> int` qui convertit une chaîne contenant un entier en entier.

Question II.1.2 Écrire une fonction `read_graphe : unit -> graphe` qui lit un graphe dans le format précédent sur l'entrée standard et tester que vous arrivez à lire le fichier.

Un graphe minimaliste de 3 sommets et 4 arêtes est donné dans le fichier test.graph afin de vous permettre de tester votre fonction.

■ **Preuve**

```
let read_graphe () =
  let nb_sommets = read_int () in
  let nb_aretes = read_int () in

  let sommets = Array.init nb_sommets
    (fun _ -> read_line ()) in

  let aretes = Array.make nb_sommets [] in

  for _ = 0 to nb_aretes - 1 do
    let src = read_int () in
    let tgt = read_int () in

    aretes.(src) <- tgt :: aretes.(src)
  done;
  { sommets=sommets; aretes=aretes }
```
OCaml

■

Si $G = (S, A)$ est un graphe dont les sommets sont énumérées $S = \{s_0, s_1, \ldots, s_{n-1}\}$, on note, pour $p \leq n$, G_p le sous-graphe induit par $\{s_0, \ldots, s_{p-1}\}$.

Question II.1.3 Écrire une fonction `restriction : graphe -> int -> graphe` tel que `restriction g p` où `g` est la représentation d'un graphe G renvoie la représentation du graphe G_p.

■ **Preuve**

On adopte une approche proche de la fonction précédente : on itère sur les arêtes du graphe initial et on ne sélectionne que celles qui sont compatibles avec la restriction.

```OCaml
let restriction g p =
  let sommets = Array.sub g.sommets 0 p in
  let aretes = Array.make p [] in
  for i = 0 to p - 1 do
    List.iter (fun j ->
      if j < p then aretes.(i) <- j :: aretes.(i))
      g.aretes.(i)
  done;
  { sommets = sommets; aretes = aretes }
```

Notons ici qu'on aurait pu avoir une approche plus fonctionnelle pour sélectionner les bonnes arêtes :

```OCaml
let restriction g p =
  let sommets = Array.sub g.sommets 0 p in
  let aretes = Array.map (List.filter ((>)p))
    (Array.sub g.aretes 0 p) in
  { sommets = sommets; aretes = aretes }
```

■

Si $G = (S, A)$ est un graphe orienté, on a vu au paragraphe Graphes non orientés les graphes non orientés par défaut et par excès, G^- et G^+ qui lui sont associés.

Question II.1.4 Écrire des fonctions `par_defaut : graphe -> graphe` et `par_exces : graphe -> graphe` qui, étant donné un graphe G, renvoie les graphes G^- et G^+ représentés en tant que graphes orientés symétriques.

■ **Preuve**

On adopte encore l'approche de création par remplissage.

```OCaml
let defaut g =
  let n = Array.length g.sommets in
  let sommets = Array.copy g.sommets in
  let aretes = Array.make n [] in
  for i = 0 to n - 1 do
    List.iter (fun j ->
      if List.mem i g.aretes.(j)
      then aretes.(i) <- j :: aretes.(i))
      g.aretes.(i)
  done;
  { sommets = sommets; aretes = aretes }

let exces g =
  let n = Array.length g.sommets in
  let sommets = Array.copy g.sommets in
  (* on recopie les arêtes existantes *)
  let aretes = Array.copy g.aretes in
  for i = 0 to n - 1 do
    List.iter (fun j ->
      (* on rajoute les retours absents *)
      if not (List.mem i g.aretes.(j))
      then aretes.(j) <- i :: aretes.(j))
      g.aretes.(i)
  done;
  { sommets = sommets; aretes = aretes }
```

■

Si $G = (S, A)$ est un graphe orienté, on note $rev(G) = (S, A')$ son miroir qui vérifie $(i, j) \in A \iff (j, i) \in A'$, c'est-à-dire qui renverse toutes les arêtes.

Question II.1.5 Écrire une fonction `miroir : graphe -> graphe` qui renvoie le miroir d'un graphe.

■ **Preuve**

Cette fonction c'est en fait le graphe par excès sans recopier les arêtes initiales sans retour :

```
let miroir g =
  let n = Array.length g.sommets in
  let sommets = Array.copy g.sommets in
  let aretes = Array.make n [] in
  for i = 0 to n - 1 do
    List.iter (fun j -> aretes.(j) <- i :: aretes.(j))
      g.aretes.(i)
  done;
  { sommets = sommets; aretes = aretes }
```

OCaml

■

Dans la suite du sujet on note $\mathcal{G}$ le graphe des followers contenu dans le fichier. On va considérer dans la suite les graphes :
$\mathcal{G}, rev(\mathcal{G}), \mathcal{G}^-, \mathcal{G}^+, \mathcal{G}_{500}, rev(\mathcal{G}_{500}), \mathcal{G}^-_{500}$ et $\mathcal{G}^+_{500}$.

II.2 Statistiques sur les degrés

Question II.2.1 Écrire une fonction `stat_degre : graphe -> int * float` qui renvoie un couple (d_{max}, d_{moy}) où d_max est le plus grand des degrés du graphe et d_{moy} est le degré moyen donné par un nombre flottant.

■ **Preuve**

On itère directement sur les sommets :

```
let stat_degre g =
  let n = Array.length g.sommets in
  let sum_d, max_d = ref 0, ref 0 in
  for i = 0 to n-1 do
    let d = List.length g.aretes.(i) in
    max_d := max !max_d d;
    sum_d := !sum_d + d
  done;
  !max_d, float_of_int !sum_d /. float_of_int n
```

OCaml

Notons qu'on peut, ici aussi, écrire une fonction utilisant la bibliothèque standard efficacement :

```
let stat_degre g =
  let a = Array.map List.length g.aretes in
  let fold = Array.fold_left in
  let foi = float_of_int in
  let n = Array.length g.aretes in
  fold max 0 a, foi (fold (+) 0 a) /. foi n
```

OCaml

■

II.3 Parcours en largeur

On va réaliser ici un parcours en largeur qui sera amené à être modifié et enrichi dans les questions suivantes. On vous laisse libre d'enrichir ce parcours en utilisant des fonctionnelles pour les traitements ou de modifier le code

du parcours directement.

Pour utiliser une file, on va utiliser le module Queue. Dans le parcours on va calculer la fonction de distance `d` et pour gérer les cas où $d(x) = \infty$, on va la représenter par un `int option array`. Si `d.(x) = None` c'est que `x` est inconnu, on peut donc se servir de ce tableau pour avoir l'état d'un sommet.

Question II.3.1 Écrire une fonction `bfs : graphe -> int -> int option array * int option array` telle que `bfs g x` renvoie un couple `(d,parent)` où `d.(y)` est la distance minimale de x à y et `parent.(y)` est l'indice du prédécesseur de `y` dans un tel chemin de x à y.

■ **Preuve**

On implémente directement le parcours en largeur vu dans le cours.

```OCaml
let bfs g x =
  let n = Array.length g.sommets in
  let p = Array.make n None in
  let d = Array.make n None in
  let a_traiter = Queue.create () in
  Queue.add x a_traiter;
  d.(x) <- Some 0;
  while not (Queue.is_empty a_traiter) do
    let x = Queue.take a_traiter in
    List.iter (fun y ->
        Queue.add y a_traiter;
        p.(y) <- Some x;
        d.(y) <- Some (Option.get d.(x) + 1))
      (List.filter (fun y -> d.(y) = None)
        g.aretes.(x))
  done;
  d, p
```

■

Question II.3.2 Écrire une fonction `chemin : graphe -> int option array -> int -> int list` tel que `chemin g parent y` renvoie les sommets présents dans un chemin de `x` à `y`.

■ **Preuve**

On utilise ici la récusivité terminale pour remettre le chemin dans l'ordre.

```OCaml
let chemin g p y =
  let rec aux y l =
    match p.(y) with
    | None -> y :: l
    | Some x -> aux x (y :: l)
  in aux y []
```

■

Question II.3.3 Écrire une fonction `affiche_chemin : graphe -> int list -> unit` qui prend un graphe et un chemin donné par la fonction précédente et l'affiche avec le format :

`compte1 -> compte1 -> ... -> compten`

■ **Preuve**

```OCaml
let affiche_chemin g p =
  Printf.printf "%s\n"
    (String.concat " -> "
```

```
      (List.map (fun i -> g.sommets.(i)) p))
```
OCaml

■

Si $x \in S$, on note $\underline{x} = \{\, y \in S \mid x \rightsquigarrow y \,\}$.

Question II.3.4 Écrire une fonction `accessibles : graphe -> int -> int` qui calcule le cardinal de $\underline{x}$ étant donné un graphe G et un sommet x donné par son indice.

■ **Preuve**

```
let accessibles g x =
  let d, p = bfs g x in
  Array.fold_left (+) 0
    (Array.map
      (fun v -> if v = None then 0 else 1) d)
```
OCaml

■

Question II.3.5 Écrire une fonction `max_accesibles : graphe -> int * int` qui renvoie un couple $(x, |\underline{x}|)$ où x est un sommet pour lequel $|\underline{x}|$ est maximal.

■ **Preuve**

```
let max_accessibles g =
  let m, v = ref 0, ref (accessibles g 0) in
  for i = 1 to Array.length g.sommets - 1 do
    let v' = accessibles g i in
    if v' > !v
    then ( v := v'; m := i )
  done;
  !m, !v
```
OCaml

■

II.4 Plus long chemin et diametre

Question II.4.1 Écrire une fonction `plus_loin : graphe -> int -> int list` telle que `plus_long_chemin g x` renvoie le chemin de `x` au sommet `y` qui lui est le plus éloigné.

■ **Preuve**

```
let plus_loin g x =
  let n = Array.length g.sommets in
  let d, p = bfs g x in
  let i = ref 0 in
  while d.(!i) = None do
    incr i
  done;
  let m = ref (!i) in
  for j = !i+1 to n - 1 do
    match d.(j) with
    | Some v -> if v > Option.get d.(!m) then m := j
    | None -> ()
  done;
  let v = Option.get d.(!m) in
  !m, v, chemin g p !m
```
OCaml

■

Question II.4.2 Écrire une fonction `diametre : graphe -> int list` qui renvoie un chemin réalisant le diamètre d'un graphe.

■ **Preuve**

```
let diametre g =
  let n = Array.length g.sommets in
  let v, p = ref 0, ref [] in
  for i = 0 to n-1 do
    let j, v', chemin = plus_loin g i in
    if !v < v'
    then begin
      v := v';
      p := chemin
    end
  done;
  !p
```

OCaml

■

II.5 Table de résultats

Attention : s'il faut peu de temps pour obtenir les résultats pour le sous-graphe de 500 sommets, c'est beaucoup plus long sur le graphe en entier.

- $\mathcal{G}_{500}$:

```
degré max 10
degré moyen 0.430000
max_accessibles Mishkalashnikov avec 16 sommets
Diamètre 7 réalisé par :
Isaac__K -> naxonlabs -> faezeh_db -> MooreInst ->
    fath_gabrielle -> hypothesesorg -> ScienceFactor -> savantures
```

- $rev(\mathcal{G}_{500})$:

```
degré max 31
degré moyen 0.430000
max_accessibles savantures avec 76 sommets
Diamètre 7 réalisé par :
savantures -> ScienceFactor -> hypothesesorg ->
    fath_gabrielle -> MooreInst -> faezeh_db -> naxonlabs -> Isaac__K
```

- $\mathcal{G}_{500}^{-}$:

```
degré max 4
degré moyen 0.176000
max_accessibles SeverineWozniak avec 8 sommets
Diamètre 4 réalisé par :
QLMB8mars -> giu_sapio -> louise_tbr -> GroupeImpec -> halfbloodqueenx
```

- $\mathcal{G}_{500}^{+}$:

```
degré max 32
degré moyen 0.684000
max_accessibles helloselyn avec 98 sommets
Diamètre 12 réalisé par :
TsamiyahL -> FES_AFNEUS -> FlorestanAFNEUS -> FedeAddiction ->
```

```
    LS46151053 -> hypothesesorg -> Osec2022 -> ardakaniz ->
    ValRobert974 -> DialloAIbrahim2 -> Defense137 -> KArthemis ->
    SGF_GEOSOC
```

- $\mathcal{G}$:

```
degré max 950
degré moyen 36.266096
max_accessibles Boris_Brana avec 6049 sommets
Diamètre 9 réalisé par :
MonaEmara10 -> SambitPhD -> MIT_CSAIL -> MehdiKaytoue -> gromuald ->
    ECHARDE_ENSL -> cerseilia_ -> dadoyeldado -> Deccefunjoogu -> stoicsalik
```

- $rev(\mathcal{G})$:

```
degré max 3655
degré moyen 36.266096
max_accessibles JustVonBraun avec 7532 sommets
Diamètre 9 réalisé par :
Bonusbasci -> TCebere -> Miruna_Rosca -> h2020prometheus ->
    barENdSonLyon -> INP_CNRS -> ThierryCoulhon -> Phil_Baty ->
    HigherEdFutures -> HEMobilities
```

- $\mathcal{G}^-$:

```
degré max 610
degré moyen 12.069850
max_accessibles augabcoh avec 5352 sommets
Diamètre 10 réalisé par :
LeaLescouzeres -> Gauthier_tls -> MorganeBoulch -> CSNB14 ->
    leo_chapuis -> CCILYONMETRO -> IsabelleHuault -> Phil_Baty ->
    UNIKEhighered -> HigherEdFutures -> HEMobilities
```

- $\mathcal{G}^+$:

```
degré max 3655
degré moyen 60.462343
max_accessibles augabcoh avec 7854 sommets
Diamètre 7 réalisé par :
GabrielMarseres -> caroched -> MarieMoroso -> L3vironaute -> najatvb ->
    LeankonCarotte -> JustVonBraun -> Sardine49160063
```

II.6 Aller plus loin

On propose ici plusieurs pistes de réflexions pour prolonger le TP :

- On a vu des algorithmes de dessin de graphes adaptés à des petits graphes. La présence de l'interaction sommet-sommet semble leur donner une complexité en $O(n^2)$ qui est rédhibitoire ici. Cependant, des sommets éloignés ont peu de chance d'interagir, comment pourrait-on modifier l'algorithme pour ignorer les interactions de répulsions entre sommets éloignés ? On remarque que la distance n'est pas un critère valide car les sommets peuvent être tous être superposés. Une manière de traiter cela efficacement est de découper le plan en région par des droites successives. Allez voir la page Binary Space Partionning et en déduire un algorithme effectif de dessin de graphe adapté.
- Pour estimer l'importance d'un compte, on ne peut pas se fier à son degré. En effet, celui-ci peut être augmenté artificiellement. Une manière fiable de mesurer l'importance est d'imaginer quelqu'un naviguant aléatoirement sur des comptes en suivant des liens d'abonnement et de mesurer la probabilité qu'il se retrouve sur un compte donné. C'est le principe qui est à la base de l'algorithme PageRank utilisé par Google. Implémenter cet algorithme et en déduire les comptes les plus importants dans cet exemple.

III Plus courts chemins en OCaml

Ce TP vous demande plus d'autonomie que les TP précédents. Il s'agit de mobiliser des connaissances plus anciennes et d'implémenter effectivement des algorithmes que l'on comprend bien en pseudo-code.

L'objectif est de pouvoir résoudre des problèmes comme :

- HIGHWAYS
- MICEMAZE
- TRAFFICN
- SAMER08A

Si vous êtes courageux, foncez sans lire la suite ;-)

III.1 Écriture naïve de Dijkstra

Écrire une implémentation de Dijkstra reposant sur une référence sur une liste de sommets pour effectuer la recherche du sommet de plus petite priorité. On pourra s'inspirer du programme Python ci-dessus.

On considérera, comme pour les autres parcours que le graphe est donné sous la forme de listes d'adjacence mais cette fois, les listes devront comporter les poids. En supposant que $S = \{0, 1, \ldots, n-1\}$ on utilisera directement le type suivant en supposant des poids entiers positifs :

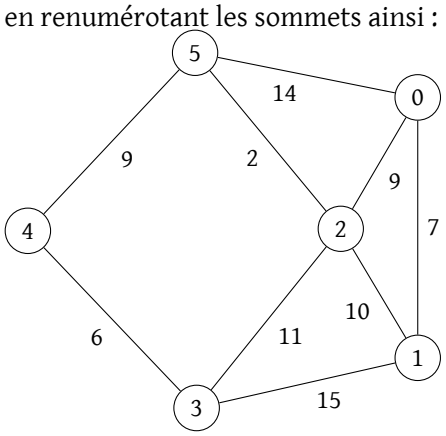
```OCaml
type graph = (int * int) list array
```

Pour un sommet x on a donc une liste de couples (y, w) où pour chaque arête $x \to y$ avec $\pi(x \to y) = w$.

Ainsi, le graphe du premier exemple est donné par la valeur :

```OCaml
let g = [|
  [ (1, 7); (2, 9); (5, 14) ];
  [ (0, 7); (2, 10); (3, 15) ];
  [ (0, 9); (1, 10); (3, 11); (5, 2) ];
  [ (1, 15); (2, 11); (4, 6) ];
  [ (3, 6); (5, 9) ];
  [ (0, 14); (2, 2); (4, 9) ]
|]
```

en renumérotant les sommets ainsi :



Pour le tableau des distances, on pourra définir un `type poids = I | N of int` ou considérer pour l'infini un entier plus grand que tous les poids de chemins, comme la somme des poids de toutes les arêtes plus 1.

■ **Preuve**

```OCaml
let rec extrait_plus_petit l d =
  match l with
  | [] -> failwith "impossible"
  | [x] -> (x, [])
  | x::q ->
    let (y, q') = extrait_plus_petit q d in
    if d.(y) < d.(x)
    then (y, x :: q')
    else (x, q)

let dijkstra g src =
  let n = Array.length g in
  let inf = max_int in
  let d = Array.make n inf in
  d.(src) <- 0;
  let marques = Array.make n false in
  let a_traiter = ref [ src ] in
  while !a_traiter <> [] do
    let x, q = extrait_plus_petit !a_traiter d in
```

```
    a_traiter := q;
    if not marques.(x)
    then begin
      marques.(x) <- true;
      List.iter (fun (y, poids) ->
        let n_d = d.(x) + poids in
        if n_d < d.(y)
        then begin
          d.(y) <- n_d;
          a_traiter := y :: !a_traiter
        end
      ) g.(x)
    end
  done;
  d
```

OCaml

■

III.2 Réalisation d'une file de min-priorité

Reprendre l'implémentation des files de priorité donnée dans le corrigé du TP18 (cliquez sur les liens dans les titres de parties) en faisant en sorte de gérer des couples $(sommet, distance)$ et en faisant attention au fait que *distance* peut être ∞.

Attention il s'agit ici de file de priorité minimale !

Remarque Ici, on peut aussi considérer directement la file de priorité maximale et ajouter des opposés.

III.3 Écriture de Dijkstra efficace

Utiliser la file de priorité pour en déduire une implémentation efficace.

■ **Preuve**

```
let rec extrait_plus_petit l d =
  match l with
  | [] -> failwith "impossible"
  | [x] -> (x, [])
  | x::q ->
    let (y, q') = extrait_plus_petit q d in
    if d.(y) < d.(x)
    then (y, x :: q')
    else (x, q)

let dijkstra g src =
  let n = Array.length g in
  let inf = max_int in
  let d = Array.make n inf in
  d.(src) <- 0;
  let marques = Array.make n false in
  let a_traiter = ref [ src ] in
  while !a_traiter <> [] do
    let x, q = extrait_plus_petit !a_traiter d in
    a_traiter := q;
    if not marques.(x)
    then begin
      marques.(x) <- true;
      List.iter (fun (y, poids) ->
        let n_d = d.(x) + poids in
        if n_d < d.(y)
        then begin
          d.(y) <- n_d;
          a_traiter := y :: !a_traiter
        end
```

```
      ) g.(x)
    end
  done;
  d
```

OCaml

■

III.4 Floyd-Warshall

Lire et implémenter l'algorithme de Floyd-Warshall au-dessus.

■ **Preuve**

```
let (+$) a b =
  (* une addition pour avoir max_int + * = max_int *)
  if a = max_int || b = max_int
  then max_int
  else a+b

let floyd_warshall poids =
  let n = Array.length poids in
  let m = Array.make_matrix n n max_int in
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      if i = j
      then m.(i).(i) <- 0
      else m.(i).(j) <- poids.(i).(j)
    done
  done;
  for k = 0 to n-1 do
    for i = 0 to n-1 do
      for j = 0 to n-1 do
        m.(i).(j) <-
          min m.(i).(j) (m.(i).(k) +$ m.(k).(j))
      done
    done
  done;
  m
```

OCaml

qu'on peut raffiner pour avoir les liens de parentés en :

```
let floyd_warshall poids =
  let n = Array.length poids in
  let m = Array.make_matrix n n max_int in
  let parent = Array.make_matrix n n None in
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      if i = j
      then m.(i).(i) <- 0
      else begin
        m.(i).(j) <- poids.(i).(j);
        parent.(i).(j) <- i
      end
    done
  done;
  for k = 0 to n-1 do
    for i = 0 to n-1 do
      for j = 0 to n-1 do
        let par_k = m.(i).(k) +$ m.(k).(j) in
        if par_k < m.(i).(j)
        then begin
          m.(i).(j) <- par_k;
          parent.(i).(j) <- parent.(k).(j)
        end
```

```
      done
    done
  done;
  m, parent
```

OCaml

■

III.5 Problèmes

Résoudre les problèmes donnés au dessus. On pourra utiliser la fonction suivante en `OCaml` pour lire des entiers séparés par des espaces sur une ligne :

```
let read_int_list () =
  List.map int_of_string (String.split_on_char ' ' (read_line ()))
```

OCaml