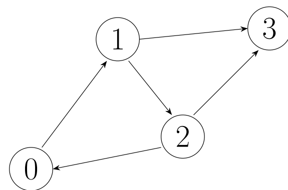
TP15: Graphes

Exercice 1: Listes d'adjacence

On considère des graphes orientés. Pour $G = (S, A)$ un graphe avec n sommets, on identifie S à l'ensemble $\{0, \dots, n-1\}$. On peut alors représenter un graphe par tableau de listes d'adjacences:

```
type graphe = int list array
```

Pour un graphe `g: graphe`, chaque case `g.(i)` est la liste des successeurs du sommet i. Par exemple, voici un graphe G_0 et sa représentation :




```
let g0 = [|
  [1];   (* successeurs de 0 *)
  [2;3]; (* successeurs de 1 *)
  [0;3]; (* successeurs de 2 *)
  []     (* successeurs de 3 *)
|]
```

Q1. Écrire une fonction `degre_moyen : graphe -> float` renvoyant le degré sortant moyen des sommets d'un graphe.

Q2. Écrire une fonction `est_arc: graphe -> int -> int -> bool` prenant en entrée un graphe G et deux sommets u, v, et renvoyant un booléen indiquant si (u, v) est un arc de G.

Q3. Écrire une fonction `ajouter_arc : graphe -> int -> int -> unit` prenant en entrée un graphe G et deux sommets u, v, et ajoutant à G l'arc (u, v). On supposera que l'arc n'existe pas déjà.

Q4. Écrire une fonction `renverser: graphe -> graphe` prenant en entrée un graphe G, et renvoyant le graphe $\bar{G}$ obtenu en **renversant** le sens de tous les arcs, sans modifier G. On pourra suivre l'algorithme suivant:

Algorithme 1 : renverser(G)

Entrée(s) : $G = (S, A)$ graphe orienté
Sortie(s) : $\bar{G}$ sous forme de tableau de listes d'adjacence.

1 $n \leftarrow |S|$;
2 $T \leftarrow$ tableau de n cases valant toutes [];
3 **pour** $u \in S$ **faire**
4   **pour** v ***successeur de*** u **faire**
        // L'arc (u, v) dans G devient (v, u) dans $\bar{G}$
5   Ajouter u à $T[v]$;
6 **retourner** T

Exercice 2: Premier parcours

On garde la même représentation des graphes que dans l'exercice précédent. OCaml dispose d'un module `Stack` permettant de manipuler des piles **mutables**. Le type d'une pile contenant des éléments `'a` est `'a Stack.t` Ses opérations sont:

```
(* Stack.create () renvoie une pile vide *)
Stack.create: unit -> 'a Stack.t

(* Stack.pop p dépile le sommet de p (non vide) et le renvoie *)
Stack.pop: 'a Stack.t -> 'a

(* Stack.push x p empile x dans p *)
Stack.push: 'a -> 'a Stack.t -> unit

(* Stack.is_empty p renvoie true ssi p est vide *)
Stack.is_empty: 'a Stack.t -> bool
```

De plus, on représente l'ensemble des sommets visités par un tableau `v: bool array`, de sorte que `v.(i) = true` lorsque le sommet i a déjà été visité. Une implémentation directe du parcours est:

```
(* affiche les sommets de g accessibles depuis s,
   dans l'ordre d'un parcours en profondeur. *)
let dfs (g: graphe) (s: int) : unit =
  let n = Array.length g in
  let vus = Array.make n false in
  let p = Stack.create () in
  vus.(s) <- true;
  Stack.push s p;
  while not (Stack.is_empty p) do
    let u = Stack.pop p in
    print_int u; print_newline ();
    let voisins = g.(u) in
    (* List.iter f l exécute f sur tous les éléments de l *)
    List.iter (fun v ->
      if not vus.(v) then begin
        vus.(v) <- true;
        Stack.push v p
      end)
      voisins
  done
```

Q1. Expliquer ce que fait l'instruction aux lignes 14-19.

Q2. Recopier la fonction précédente et la tester sur un exemple, puis modifier la fonction pour qu'elle renvoie le **nombre** de sommets accessibles depuis le sommet source s.

Q3. En suivant le même schéma que la fonction précédente, écrire une fonction qui renvoie la liste des sommets accessibles depuis un sommet s dans un graphe G.

Exercice 3: Recherche de chemin

On représente l'arborescence d'un parcours par son tableau des prédécesseurs. Dans ce tableau pr, pour chaque sommet u, $pr[u]$ est le sommet ayant permis d'empiler u. On posera que $pr[s] = s$ pour s la source du parcours, et que $pr[u] = -1$ si u n'a pas été visité lors du parcours.

Q1. Écrire une fonction `arborescence_dfs: graphe -> int -> int array` prenant en entrée un graphe G, un sommet s de G, appliquant un parcours en profondeur depuis s et renvoyant le tableau des prédécesseurs de l'arborescence du parcours.

Q2. Écrire une fonction `reconstruction : int array -> int -> int list` prenant en entrée un tableau pr obtenu via la fonction précédente, un sommet u, et renvoyant une liste de sommets formant un **chemin** de la source du parcours jusqu'à u s'il en existe un. *Indication: cette question se fait bien par récursivité.*

Exercice 4: Détection de cycle

On considère des graphes non-orientés. Les algorithmes de parcours permettent de trouver efficacement des cycles dans un graphe qui en contient. En effet, au cours d'un parcours, si l'exploration des voisins d'un sommet u trouve un sommet v ayant déjà été vu, alors c'est que le parcours a trouvé simultanément une chaîne C_1 de s à u, et une autre chaîne C_2 de s à v, ainsi qu'une arête (u, v). Alors, la chaîne C_1C_2 est un cycle valide.

Q1. Écrire une fonction `arete_cycle : graphe -> int array * (int * int)option`. Cette fonction prendra en entrée un graphe G, et renverra un couple (pr, a) avec:

- pr une arborescence de parcours en profondeur
- a une arête u, v tels que $pr[u] \neq v$ et $pr[v] \neq u$, ou None s'il n'existe pas de telle arête.

Q2. En déduire une fonction `cycle : graphe -> int list` prenant en entrée un graphe G, et renvoyant une liste de sommets formant un cycle, ou bien la liste vide s'il n'existe pas de cycle.

Q3. Améliorer la fonction précédente pour que le cycle renvoyé ne contienne **aucun** sommet en double, sauf le sommet d'arrivée / de départ.

Graphes en C

On propose d'utiliser les types suivants en C pour représenter les graphes par listes d'adjacence:

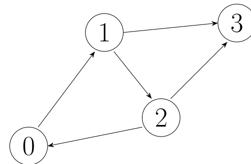
```
// liste chainée pour stocker les voisins / successeurs d'un sommet
struct adj{
  unsigned int voisin;
  struct adj* suiv;
};
typedef struct adj adj_t;

struct graphe {
  int n;
  adj_t** lv; // listes des voisins / successeurs
};
typedef struct graphe graphe_t;
```

Dans la suite, on appellera également "voisins" les successeurs d'un sommet d'un graphe orienté, par abus de langage.
L'archive du TP contient un couple de fichiers `graphe.h/.c` implémentant quelques opérations de base sur les graphes, permettant notamment de les lire depuis un fichier, selon le format suivant. Pour $G = (S, A)$ un graphe, avec $n = |S|, m = |A|$, on stocke G en écrivant dans un fichier texte:

- Sur la première ligne, les entiers n et m, ainsi qu'un entier 1 ou 0 indiquant si le graphe est orienté ou non.
- Sur les m lignes suivantes, m couples (u, v) représentant les arcs/arêtes du graphe.

L'archive du TP contient un fichier à ce format, représentant le graphe G_0 ci-dessous, ainsi qu'un fichier `test.c` lisant et affichant G_0.



Q1. Lire le code de la fonction `afficher_graphe` dans `graphe.h/.c` pour vous familiariser avec la structure.

Q2. Sur le même principe, écrire une fonction `est_arc` prenant en entrée un graphe et deux sommets u, v, et renvoyant un booléen indiquant si (u, v) est un arc.

Q3. Dessiner un graphe orienté avec une dizaine de sommets qui vous servira d'exemple pour tester les prochaines fonctions, et créer un fichier texte au format décrit plus haut pour stocker ce graphe.

On rappelle que le parcours en largeur (ou BFS pour Breadth-First Search) permet de calculer des plus courts chemins dans un graphe orienté.

Implémentation de la file Le parcours en largeur nécessite d'utiliser une file pour stocker les indices des différents sommets explorés.
Notons n le nombre de sommets du graphe. Puisque le parcours n'enfile chaque sommet qu'au plus une fois, on peut stocker la file dans un tableau à n cases. On utilisera donc trois variables pour manier la file:

- un tableau `int* file` de taille n;
- un indice `int tete` indiquant la case à défiler;
- un indice `int queue` indiquant la prochaine case à remplir lorsqu'un élément est enfilé.

De plus, on utilisera un tableau `int* pred` pour stocker les prédécesseurs dans l'arborescence du parcours. La fonction de parcours en largeur commencera donc de la manière suivante:

```c
/* Parcourt g en largeur depuis s, et renvoie l'arborescence
   du parcours sous forme de tableau des prédécesseurs */
int* bfs(graphe_t* g, int s){
  int n = g->n;
  int* pred = malloc(n * sizeof(int));
  pred[s] = s;
  int* file = malloc(n * sizeof(int));
  // initialiser la file avec un seul élément
  int tete = 0;
  int queue = 1;
  file[0] = s;
  while (tete != queue){
    ...
  }
  return pred;
}
```

Q4. Compléter la fonction `bfs`, et bien la tester.

Q5. Écrire une fonction `unsigned int* plus_court_chemin(graphe_t* g, unsigned int s, unsigned int t)` qui renvoie un plus court chemin entre s et t. Le résultat sera un tableau contenant les sommets du chemin, en commençant par s et en terminant par t.

Testez bien vos fonctions avant de passer à la suite.

On souhaite maintenant utiliser ces fonctions pour résoudre des labyrinthes. On considère des grilles 2D, où chaque case peut être soit libre, soit occupée par un mur. On peut marcher d'une case à une autre si les deux sont libres et adjacentes (i.e. avec un côté en commun), et l'objectif est d'aller de la case en haut à gauche jusqu'à la case en bas à droite.

L'archive du TP contient des fichiers stockant de tels labyrinthes, au format suivant:

- Sur la première ligne, deux entiers n et m, les dimensions du labyrinthe
- Sur chacune des n lignes suivantes, m caractères: `X` pour un mur, un espace pour une case vide.

Par exemple, le fichier `lab/21x23.txt` de l'archive représente le labyrinthe ci-contre (l'entrée et la sortie sont marquées E et S).

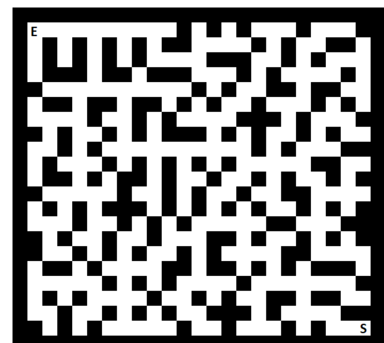


Figure 1: 21×23 cases

Un labyrinthe de $n \times m$ cases peut ensuite être modélisé par un graphe à nm sommets, un par case. Deux cases $u = (i_1, j_1)$ et $v = (i_2, j_2)$ sont reliées par une arête si et seulement si elles sont toutes les deux libres et adjacentes. Tous les sommets seront donc de degré au plus 4.

De plus, on encode les cases comme des entiers: pour un labyrinthe de n lignes et m colonnes, on encode la case (i, j) par l'entier $im + j$. En supposant que les labyrinthes ont moins de 2^{16} lignes et colonnes, on peut alors encoder les cases sur 32 bits, nous permettant donc de réutiliser directement notre structure de graphe.

0	1	$\dots$	$m-1$
m	$m+1$	$\dots$	$2m-1$
$\dots$	$\dots$	$\dots$	$\dots$
$(n-1)m$	$(n-1)m+1$	$\dots$	$nm-1$

Figure 2: Numérotation des cases de la grille

En particulier, l'entrée est le premier sommet, et la sortie est le dernier sommet.

Les fichiers `labyrinthe.h/.c` de l'archive contiennent deux fonctions `lire_labyrinthe` et `graphe_labyrinthe` permettant respectivement de lire un labyrinthe et de le stocker dans une structure `lab_t` décrite dans `labyrinthe.h`, et de créer un graphe correspondant à un labyrinthe.

Q6. Créer un nouveau fichier avec une fonction `main` lisant le fichier `lab/petit.txt` de l'archive et affichant le graphe correspondant.

Un script python `image_labyrinthe.py` est fourni dans l'archive du TP. Il prend en entrée un fichier texte contenant un labyrinthe, ainsi qu'un autre fichier texte contenant un chemin, et:

1. vérifie que le chemin est valide, et qu'il va de l'entrée à la sortie;
2. génère une image du labyrinthe;
3. génère une image du labyrinthe avec le chemin tracé en rouge.

Le fichier du chemin doit contenir les cases visitées, sur une ligne chacune, les coordonnées devant être séparées d'une espace:

```
0 0
0 1
0 2
1 2
...
```

On exécute le script Python en tapant la commande suivante dans le terminal (en remplaçant les noms des fichiers):

```
python3 image_labyrinthe fichier_labyrinthe.txt fichier_chemin.txt
```

Q7. Chercher à la main une solution du labyrinthe `petit.txt` et tester le script Python.

Q8. Dans le programme C, créer une fonction qui prend en entrée un nom de fichier contenant un labyrinthe, et générant un fichier contenant un chemin vers la sortie. Vérifiez avec les labyrinthes de l'archive.