TP16: Graphes

MP2I Lycée Pierre de Fermat

Graphes en OCaml

Dans cette section, nous allons manipuler des graphes représentés par listes d'adjacence. On propose d'utiliser le type suivant pour les graphes:

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

Un graphe sera donc un tableau, où la case i sera la liste des voisins du sommet numéro i. Par exemple:

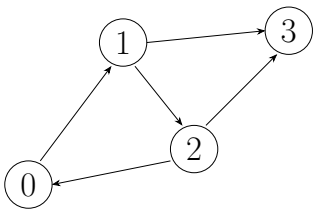


Figure 1: Graphe G_0

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

Figure 2: Représentation de G_0 par listes d'adjacence en OCaml

Q1. Écrire une fonction `nb_aretes: graphe -> int` renvoyant le nombre d'arêtes d'un graphe. On considèrera que le graphe est orienté.

Q2. Écrire une fonction `nb_aretes_no: graphe -> int` renvoyant le nombre d'arêtes d'un graphe non-orienté. La fonction ne vérifiera que le graphe est effectivement non-orienté.

Boucles while Afin d'implémenter des parcours de graphe comme vus en cours, il faut apprendre à faire des boucles while en OCaml. La syntaxe est la suivante:

```
while CONDITION do
  INSTRUCTION 1;
  INSTRUCTION 2;
  ...
  INSTRUCTION k
done
```

Par exemple, la fonction suivante calcule la racine entière d'un entier:

```
let racine (n: int) : int =
  let r = ref 0 in
  while !r * !r <= n do
    r := !r + 1;
  done;
  !r - 1
```

Q3. Implémenter l'exponentiation rapide avec une boucle while.

Pile mutable On implémentera les piles en OCaml avec le type suivant:

```
type 'a pile = 'a list ref
let pile_vide = ref []
```

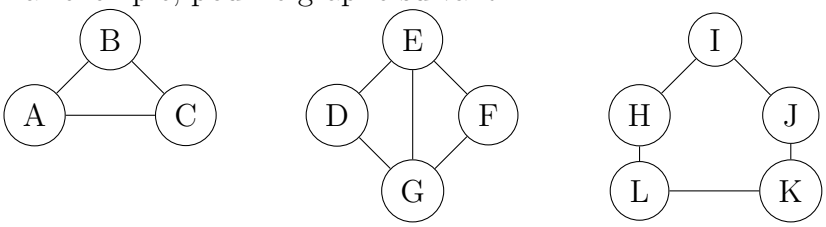
Une pile est donc une référence vers une liste d'éléments: le sommet de la pile est le premier élément de la liste.

Q4. Écrire une fonction `empiler: 'a pile -> 'a -> unit` permettant d'empiler un élément sur une pile.

Q5. Écrire une fonction `depiler: 'a pile -> 'a` permettant de dépiler le sommet d'une pile non vide et de le renvoyer.

Calcul des composantes connexes On s'intéresse tout d'abord au calcul des composantes connexes d'un graphe non-orienté. On rappelle le principe: effectuer un parcours, en profondeur ou en largeur, depuis chaque sommet u du graphe, afin d'explorer et de noter la composante connexe de u. On souhaite stocker le résultat de l'algorithme sous la forme d'un tableau C, où la case $C[i]$ contient un entier indiquant le numéro de la composante connexe du sommet i.

Par exemple, pour le graphe suivant:



Le tableau des composantes connexes contiendra:

i	0	1	2	3	4	5	6	7	8	9	10	11
Sommet n° i	A	B	C	D	E	F	G	H	I	J	K	L
Composante $C[i]$	1	1	1	2	2	2	2	3	3	3	3	3

Q6. Écrire une fonction `noter_composante_source: graphe -> int -> int -> int array -> unit` utilisant un parcours en profondeur et ayant la spécification suivante:

```
(* Entrées:
   - g graphe avec n sommets
   - 0 <= s < n indice d'un sommet de g
   - k entier positif
   - c tableau d'entiers, tel que pour tout sommet u de la composante connexe
     de s, c.(u) = -1
   Effet: modifie c pour écrire k dans toutes les cases correspondant aux sommets
   de la composante connexe de s dans g
     *)
let noter_composante_source (g: graphe) (s: int) (k: int) (c: int array) : unit =
  ...
```

On pourra utiliser directement le tableau `c` pour savoir quels sommets ont déjà été visités au cours du parcours.

Q7. Écrire une fonction `composantes_connexes: graphe -> int array` qui prend en entrée un graphe et calcule un tableau C associant à chaque sommet le numéro de sa composante connexe.

Q8. Faire des tests sur quelques graphes non-orientés.

Graphes en C

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

```
1  // liste chainée pour stocker les voisins / successeurs d'un sommet
2  struct adj{
3    unsigned int voisin;
4    struct adj* suiv;
5  };
6  typedef struct adj adj_t;
7
8  struct graphe {
9    int n;
10   adj_t** lv; // listes des voisins / successeurs
11 };
12 typedef struct graphe graphe_t;
```

Dans la suite, on appellera "voisins" les successeurs d'un sommet d'un graphe orienté.

Vous trouverez dans l'archive un fichier "graphe.c" avec les définitions de ces types ainsi que des instructions dans le `main` permettant de représenter le graphe G_0.

Q9. Modifier le code pour rajouter un arc $(3, 0)$ et pour renverser le sens de l'arc entre 2 et 1.

Q10. Écrire une fonction `void print_graphe(graphe_t* g)` permettant d'afficher chaque sommet du graphe suivi de la liste de ses voisins. Pour le graphe G_0 plus haut par exemple, la fonction affichera:

```
Graphe à 4 sommets:

0 -> 1
1 -> 2, 3
2 -> 0, 3
3
```

Q11. Écrire une fonction `free_graphe(graphe_t* g)` libérant toute la mémoire allouée pour un graphe. Testez-la sur le graphe G_0.

Q12. Écrire une fonction `graphe_t* graphe_vide(int n)` créant un graphe avec n sommets et aucune arête.

Q13. Écrire une fonction `void ajouter(graphe_t* g, unsigned int u, unsigned int v, bool oriente)` ajoutant l'arête (u, v) au graphe g. Si le booléen `oriente` est faux, alors la fonction ajoute aussi l'arête (v, u). Cette fonction devra éviter de créer des doublon dans les listes d'adjacence de u et v.

Une représentation basique des graphes est de donner la liste de ses arêtes. Cette représentation peut être facilement stockée dans un fichier, mais n'est pas pratique à manipuler, notamment du point de vue des parcours de graphe.
On se propose d'utiliser le format suivant pour stocker un graphe $G = (S, A)$ avec $n = |S|, m = |A|$:

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

On utilisera en C le type `unsigned int**` pour stocker des listes d'arêtes, on accèdera donc aux deux sommets de la i-ème arête d'une liste d'arêtes L avec $L[i][0]$ et $L[i][1]$.

Q14. Écrire une fonction `graphe_t* lire_graphe(char* filename)` qui lit dans un fichier les informations d'un graphe.

Q15. Stocker le graphe G_1 suivant dans un fichier, et l'utiliser pour tester votre fonction.

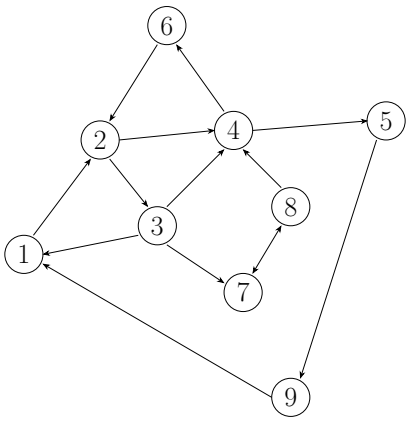


Figure 3: Graphe G_1

Parcours en largeur

On rappelle que le parcours en largeur, ou **BFS** (**B**readth-**F**irst **S**earch), permet de calculer des plus courts chemins dans un graphe orienté. Plus précisément, le parcours en largeur à partir d'un sommet s permet de construire une arborescence enracinée en s et contenant tous les sommets accessibles à partir de s. A partir d'un sommet u accessible donné, remonter l'arborescence jusqu'à s permet de calculer un plus court chemin.

Algorithme 1 : Parcours en largeur: calcul des distances

Entrée(s) : $G = (S, A)$ graphe non-orienté, $s \in S$ sommet de départ
Sortie(s) : Pred tableau des prédécesseurs de l'arborescence

```
1  d ← [−1, ..., −1] // tableau des distances, -1 indique un sommet
       inaccessible
2  Pred ← [−1, ..., −1] // -1 indique un sommet inaccessible
3  F ← file_vide() ;
4  enfiler(F, s) ;
5  d[s] ← 0 ;
6  Pred[s] ← s// convention:  la source est son propre prédécesseur
7  tant que F non vide faire
8      u ← defiler(F);
9      pour v voisin de u faire
10         si d[v] = −1 alors
11             enfiler(F, v);
12             d[v] ← d[u] + 1;
13             Pred[v] ← u;
14 retourner Pred
```

Implémentation de la file Notons n le nombre de sommets du graphe. Comme on sait que le parcours n'empile chaque sommet qu'au plus une fois, on peut stocker notre file dans un tableau à n cases. On utilisera donc trois variables pour manier la file:

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

Q16. Écrire une fonction `unsigned int* bfs(graphe_t* g, unsigned int s)` appliquant un parcours en largeur et renvoyant l'arborescence construite.

Q17. Écrire une fonction `unsigned int* plus_court_chemin(graphe_t* g, int unsigned s, unsigned int t)` qui renvoie un plus court chemin entre s et t. Le résultat sera un tableau contenant les sommets du chemin, s et t compris, ce qui permettra de le parcourir sans en connaître la taille: s et t marqueront le début et la fin du tableau.

Testez bien vos fonctions avant de passer à la suite.

Résolution de labyrinthe

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).

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, et `E` / `S` pour l'entrée et la sortie

Par exemple, le fichier `21x23.txt` de l'archive représente le labyrinthe ci-contre.

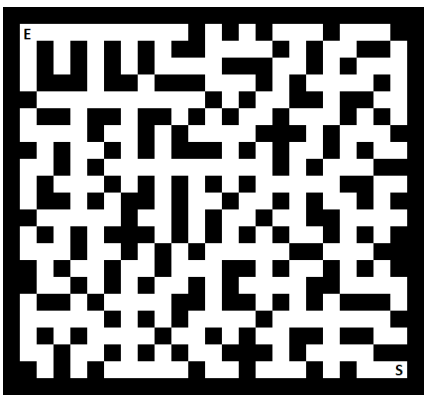


Figure 4: Labyrinthe de 21 × 23 cases

On utilisera la structure suivante pour stocker les informations d'un labyrinthe:

```
typedef struct{
  int n; // nombre de lignes
  int m; // nombre de colonnes
  bool** grille; //grille[i][j] = true s'il y a un mur, false sinon
  int ie, je ; //coordonnées de l'entrée
  int is, js ; //coordonnées de la sortie
} lab_t;
```

Q18. Écrire une fonction `lab_t* lire_lab(char* filename)` qui charge les informations d'un labyrinthe depuis un fichier.

On veut maintenant transformer un labyrinthe en un graphe. Pour un labyrinthe de $l \times c$ cases, on prendra un graphe à lc 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, pour pouvoir réutiliser les structures de graphe précédentes, on encode les cases comme des entiers. On suppose que nos labyrinthes ont moins de 2^{16} lignes et colonnes, et on peut alors encoder les cases sur 32 bits. Plus précisément, la case (i, j) sera associée à l'entier $m * i + j$, ce qui revient à numéroter les cases dans l'ordre naturel ligne par ligne:

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

Ceci nous permettra de réutiliser notre structure de graphe sans modification.

Q19. Écrire une fonction `graphe_t* graphe_labyrinthe(lab_t* lab)` permettant de construire un graphe à partir d'un labyrinthe comme décrit plus haut.

Q20. Écrire une fonction `char* resoudre_labyrinthe(lab_t* lab)` qui renvoie la suite des **directions** à prendre dans un labyrinthe pour aller de l'entrée à la sortie, en notant **U**, **R**, **D**, **L** les 4 directions possibles (up, right, down, left).
Vérifiez que sur le labyrinthe `500x500.txt` de l'archive, vous trouvez un chemin de longueur 1442.