TP15: Graphes

MP2I Lycée Pierre de Fermat

On étudie dans cette section des graphes non pondérés, représentés par liste d'adjacence. On manipulera des graphes dont les sommets sont des entiers positifs consécutifs, on pourra donc stocker les informations d'un graphe dans un tableau de listes. Si l'on voulait manipuler des sommets d'une autre nature, on pourrait utiliser une structure de dictionnaire.

OCaml impératif

Au cours des précédents TP, certaines sections de fin introduisaient quelques notions d'impérativité en OCaml. Le but de cette section est de présenter plus amplement l'aspect impératif d'OCaml. Le principe de l'impérativité est d'introduire des constructions proches du C: les boucles while / for, les variables mutables, les tableaux...

Expressions de type unit

On rappelle que le type `unit` a une unique valeur : `()`. Les expressions de type `unit` ne portent donc aucune information, ce n'est pas leur valeur qui importe, mais l'effet que produit leur évaluation, appelé **effet de bord**. On a déjà vu plusieurs expressions de type `unit`.

- `assert condition` qui n'a aucun effet si l'évaluation de l'expression booléenne *condition* donne la valeur `true`, mais qui a pour effet d'interrompre le programme en déclenchant une erreur si celle-ci conduit à la valeur `false`.
- `print_string s` qui a pour effet d'afficher la chaîne de caractères s.

Les expressions de type `unit` sont communément appelées des **instructions**, parce qu'elles ont un effet, mais elles sont bien des expressions, on peut donc écrire `let a = e_unit` où `e_unit` est une expression de type `unit`. Notons que l'effet de bord n'ayant lieu qu'à l'évaluation, appeler `a` après une telle définition n'a aucun intérêt : la valeur `a` est seulement `()`.

De plus, comme on ne peut pas juxtaposer des expressions en général, on ne peut pas juxtaposer les expressions de type `unit`, il faut les séparer par points-virgules, comme on le fait dans les jeux de tests. D'une manière générale, `e_1;e_2` est une expression de type et de valeur ceux de `e_2`. L'évaluation de cette expression consiste en l'évaluation de `e_1` puis l'évaluation de `e_2`, et seule la valeur de `e_2` est conservée, de sorte que si `e_1` n'a pas d'effet de bord, `e_1` ne sert à rien. Bien sûr, on peut généraliser cette **mise en séquence** à plus de 2 expressions : `e_1;e_2;e_3; ...`.

Références

Jusque là, nous avions appelé (à tort) "*variables*" des **valeurs nommées** dont la valeur ne changeait pas au cours du temps. Après la définition `let a = 2` par exemple, l'expression `a` vaut toujours 2, à moins d'une nouvelle définition comme `let a = 3`. Mais ce genre de re-définition ne peut être faite dans une fonction tout en gardant un effet une fois l'appel de fonction terminé. En Ocaml, lorsque l'on veut une "vraie" variable, c'est-à-dire une case mémoire dont on va pouvoir modifier la valeur, il faut le préciser grâce au mot-clé `ref`. On parle alors de **référence** (évitant le terme équivoque *variable*).

`ref e` où e est une expression de type `t` de valeur v
est une expression de type `t ref`, et de valeur `{contents = v}`, i.e. une case mémoire contenant v.
L'évaluation de cette expression consiste en :

1. l'évaluation de l'expression e en une valeur v ;
2. la réservation d'une case mémoire pouvant stocker une donnée de type t ;
3. l'initialisation de la donnée stockée dans cette case mémoire à la valeur v.

On peut donc voir une référence comme une case mémoire allouée avec malloc (mais il n'y a pas besoin de la libérer manuellement). On peut donc créer une référence comme suit: `let var = ref e` où *var* est un nom et e une expression.

Question 1. Après la définition `let var1 = ref '3'` que vaut `var1` et quel est son type ?
Après la définition `let var2 = ref 4` que vaut `var2` et quel est son type ?
Peut-on additionner `var1` avec 3 ? et `var2` avec 4 ? *Vérifiez en utilisant* utop.

On accède à la valeur de type t d'une référence de type `t ref` grâce à l'opérateur `!` (lu "bang"), et on la modifie grâce à l'opérateur `:=` (lu "reçoit"). Ces deux opérations se font en temps constant. Par exemple, après la définition `let a = ref 2`, `!a` vaut 2 (et est de type `int`), et après `a := 8`, `!a` vaut 8 (et est toujours de type `int`).

Question 2. Créer une référence nommée `var1` initialisée à la valeur 12. Interpréter `var1` puis `!var1`.

Question 3. Modifier `var1` en doublant son contenu, puis en ajoutant 10. Interpréter à nouveau `var1` puis `!var1`.

Question 4. Écrire une fonction `incrementer: int ref -> unit` telle que `incrementer a` ajoute 1 au contenu de `a`.

Question 5. Prédire ce que valent `a`, `b` et `c` à la fin du code suivant, puis vérifier votre réponse avec utop.

```
1  let a = ref 2
2  let b = ref (!a + 1)
3  let c = a
4
5  incrementer a;
6  b := !b + 3;
7  c = b
```

Il faut donc bien distinguer une référence (de type `t ref`) de son contenu (de type `t`).

Boucles while et boucles for

Dans cette section on présente les différentes boucles existant en Ocaml.

```
while condition do
  corps
done
```

où *condition* est une expression de type `bool`
et *corps* une expression de type `unit`

est une expression de type `unit` (et donc de valeur `()`).

Son évaluation consiste en :

1. l'évaluation de *condition* ;
2. si la valeur obtenue est `false`, l'évaluation est terminée
3. sinon, on évalue *corps*, et on recommence.

En pratique :

- on crée d'abord (au moins) une référence, disons `i` ;
- *condition* fait appel à sa valeur, et fait donc apparaître `!i` ;
- *corps* modifie la valeur de la référence `i` (via `i:=...`)

Par exemple, pour calculer la factorielle d'un entier:

```
let factorielle n =
  let x = ref 1 in
  let i = ref 1 in
  while !i <= n do
    x := !x * !i;
    i := !i + 1
  done;
  !x
```

Regardons maintenant la syntaxe des boucles for:

```
for i = init to final do
  corps
done
```

où i est un identificateur (**attention** i n'est pas une référence),
$init$ et $final$ sont deux expressions de type `int`, et
corps une expression de type `unit` pouvant faire apparaître i.

est une expression de type `unit` (et donc de valeur `()`).

Son évaluation consiste en :

1. l'évaluation de $init$ et de $final$ en $d \in \mathbb{Z}$ et $f \in \mathbb{Z}$;
2. pour chaque $k \in [\![d, f]\!]$, l'évaluation de *corps* où i est évaluée à k.

Question 6. En utilisant une boucle for puis une boucle while, écrire deux versions d'une fonction qui prend en entrée un entier n et affiche $1, 4, 9, \ldots, n^2$.

Tableaux

Les tableaux de type `'a array` permettent de stocker un nombre fini d'éléments modifiables de même type `'a`. On peut penser que les tableaux permettent de générer et manipuler une collection de références de même type, car chaque case du tableau est comme une référence : elle a un type, une valeur, peut être modifiée ou lue. De plus la taille les tableaux est fixe : on n'ajoute pas d'éléments à un tableau (contrairement aux listes en Python). On doit donc fixer à la création d'un tableau à la fois sa taille et ses valeurs initiales. Cela peut être fait de manière explicite ou bien en utilisant la fonction `make` du module `Array` pour une initialisation de toutes les cases à la même valeur.

`let t = [|e_1; e_2; ...; e_n|]` où les `e_i` pour $i \in [1..n]$ sont des expression du même type.

`let t = Array.make n e` où e est une expression de type quelconque et n de type `int`. Cela crée un tableau de n cases, valant toutes e.

Par exemple:

```
1 let t1 = [|1;3;5;7|]
2
3 let t2 = Array.make 15 "bla"
```

Une fois le tableau `t` initialisé, on accède aux valeurs de ses cases par leurs indices avec la syntaxe `t.(i)` (comme en C, les tableaux sont indexés à partir de 0), et on les modifie grâce à la syntaxe `t.(i) <- e`. Ces deux opérations se font en temps constant.

Par exemple, la fonction suivante prend en entrée un tableau et renvoie un nouveau tableau constitué des sommes partielles:

```
1 let sommes_partielles (t: int array) : int array =
2   let n = Array.length t in (* longueur du tableau *)
3   let res = Array.make n 0 in
4   let s = ref 0 in (* somme partielle actuelle *)
5   for i = 0 to n-1 do
6     s := !s + t.(i);
7     res.(i) <- !s
8   done;
9   res
```

Question 7. Recopier le code précédent, et le tester sur quelques tableaux.

Question 8. Écrire une fonction de tri (par insertion, par sélection ou par bulle) de signature `'a array -> 'a array`.

Le module `Array` contient de nombreuses fonctions pratiques sur les tableaux, dont certains équivalents des fonctions classiques sur les listes (map, fold, filter, etc...).

Graphes en OCaml

On propose d'utiliser le type suivant pour les graphes:

```
1 type graphe = int list array;;
```

Un graphe sera donc un tableau, où la case i sera la liste des voisins du sommet i.
Par exemple, pour le graphe G_0 suivant:

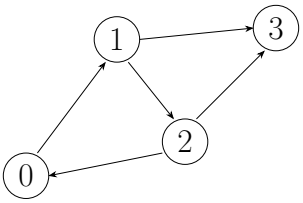

Figure 1: Graphe G_0

on pourra écrire:

```
1 let g0 =
2 [|
3   [1];    (* voisins de 0 *)
4   [2;3];  (* voisins de 1 *)
5   [0;3];  (* voisins de 2 *)
6   []      (* voisins de 3 *)
7 |]
```

Question 9. Écrire une fonction `nb_aretes: graphe -> int` renvoyant le nombre d'arêtes d'un graphe. On considèrera que le graphe est orienté.

Question 10. Écrire une fonction `nb_aretes_no: graphe -> int` renvoyant le nombre d'arêtes d'un graphe non-orienté. Cette fonction ne s'occupera pas de vérifier que le graphe est effectivement non-orienté.

On s'intéresse maintenant au calcul des composantes connexes d'un graphe. 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 utilisera un tableau de booléens pour se rappeler des sommets vus, afin de ne pas visiter la même composante connexe deux fois. On implémentera deux fonctions:

- `liste_composante_connexe: graphe -> int -> bool array -> int list` prend en entrée un graphe G, un sommet source s et un tableau de booléens V, et calcule la composante connexe de s dans G. Cette fonction mettra à jour le tableau V en mettant à vrai les cases correspondant à des sommets visités.
- `composantes_connexes: graphe -> int list list` prend en entrée un graphe et calcule la liste de ses composantes connexes.

Le fichier "composante_connexe.ml" contient du code à trou implémentant ces deux algorithmes, et le fichier "composante_connexe_rempli.ml" contient une correction de ce code. Ne regardez pas ce dernier tout de suite ! Passez au moins 20 minutes sur les questions 11 et 12 avant de regarder la correction.

On rappelle le pseudo-code:

Algorithme 1 : Liste_Composante_Connexe

Entrée(s) : $G = (S, A)$ graphe, $s \in S$ sommet de départ, V tableau de sommets visités

Sortie(s) : Liste des sommets dans la composante connexe de s. Met à jour V

```
1  P ← pile_vide();
2  P.empiler(s);
3  V[s] ← 1;
4  C ← [];
5  tant que P non vide faire
6      u ← P.depiler();
7      Traiter u;
8      pour v voisin de u faire
9          si V[v] = 0 alors
10             Ajouter v à C;
11             V[v] ← 1;
12             P.empiler(v);
13 retourner C
```

Algorithme 2 : Composantes connexes

Entrée(s) : $G = (S, A)$ graphe non-orienté

Sortie(s) : CC liste de listes représentant les composantes connexes de G

```
1  V ← [0, 0, ..., 0] // tableau de |S| booléens, indiquant les sommets vus
2  CC ← [] // liste des composantes connexes
3  pour u ∈ S faire
4      si V[u] alors
5          Passer au sommet suivant;
6      C ← Liste_Composante_Connexe(G, u, V);
7      Ajouter C à CC;
8  retourner CC
```

En OCaml, la pile P et la liste des voisins de u seront des listes. Donc, on remplacera les boucles par des fonctions récursives qui itèrent sur les listes.

Question 11. Compléter le code de la fonction `liste_composante_connexe`

Question 12. Compléter le code de la fonction `composantes_connexes`

Question 13. Faire des tests sur quelques graphes non-orientés.

On rappelle le principe pour détecter si un graphe G est biparti: on parcourt G en assignant à chaque sommet alternativement une couleur (1 ou 2). Si au cours du parcours on rencontre une contradiction, par exemple si l'on essaie de colorer un sommet avec la couleur 1 alors qu'il est déjà de couleur 2, alors le graphe n'est pas biparti. Sinon, on peut renvoyer une 2-coloration de G, qui est donc biparti.

Question 14. Écrire une fonction `bipartition: graphe -> bool array option` en adaptant les fonctions précédentes. Cette fonction renverra None si le graphe n'est pas biparti, et sinon un tableau indiquant la couleur de chaque sommet dans une bipartition valide.

Graphes en C

On propose d'utiliser les types suivants en C:

```c
// liste chainée pour les listes des voisins des sommets
typedef struct adj{
  unsigned int voisin;
  struct adj* suiv;
} ADJ;

typedef struct {
  int n;
  ADJ** lv; // listes des voisins
} GRAPHE;
```

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.

Question 15. Modifier le code pour rajouter un arc $(3, 0)$ et pour renverser le sens de l'arc entre 2 et 1.

Question 16. Écrire une fonction `void print_graphe(GRAPHE* g)` permettant d'afficher chaque sommet du graphe suivi de la liste de ses voisins. Pour le graphe G_0 ci dessus par exemple, la fonction affichera:

```
Graphe à 4 sommets:

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

En pratique, lorsqu'on construit un graphe, la manière la plus naturelle de le décrire est de donner la liste de ses arcs/arêtes. Cependant, cette représentation est bien moins maniable que les listes d'adjacences. On utilisera en C le type $int**$ 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 $L[i][1]$.

Question 17. Écrire une fonction `GRAPHE* construire_graphe(int n, int m, int** aretes)` qui prend en entrée un entier `n`, une liste `aretes` de `m` couples représentant les arêtes/arcs d'un graphe et un booléen `oriente`, et construit la liste d'adjacence correspondant au graphe sur `n` sommets dont les arêtes sont celles de la liste `aretes`, en considérant le graphe comme orienté ou non selon la valeur d'`oriente`. Votre fonction fera attention à ne pas créer de doublons dans la liste d'adjacence, même s'il y en a dans la liste des arcs.

Afin de pouvoir décrire des graphes plus simplement qu'en écrivant le code permettant de les construire, nous allons utiliser des fichiers. 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
- Sur les m lignes suivantes, m couples (u, v) représentant les arcs du graphe.

Question 18. Écrire une fonction `GRAPHE* lire_graphe(char* filename)` qui lit dans un fichier les informations d'un graphe.

Question 19. Stocker le graphe G_1 suivant dans un fichier, et l'utiliser pour tester votre fonction.

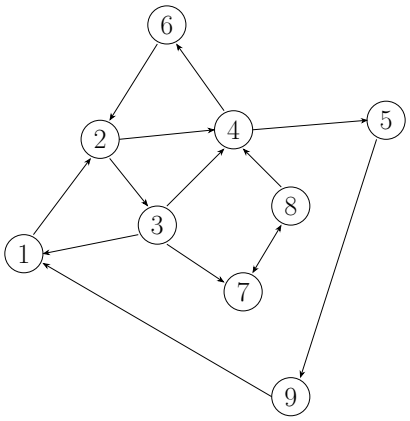

Figure 2: Graphe G_1

Parcours en largeur et labyrinthe

On rappelle l'algorithme du parcours en largeur, qui permet de calculer les plus courts chemins dans un graphe non-pondéré:

Algorithme 3 : Parcours en largeur: calcul des distances

Entrée(s) : $G = (S, A)$ graphe non-orienté, $s \in S$ sommet de départ
Sortie(s) : **d** tableau des distances de s à tous les sommets de G

```
1  d ← [−1, ..., −1] // tableau des distances
2  Pred ← [−1, ..., −1] // tableau des prédecesseurs
3  P ← file_vide() ;
4  F.enfiler(s) ;
5  d[s] ← 0 ;
6  tant que F non vide faire
7      u ← P.defiler();
8      pour v voisin de u faire
9          si d[v] = −1 alors
10             d[v] ← d[u] + 1;
11             Pred[v] ← u;
12             P.enfiler(v);
13 retourner d, Pred
```

d$[u]$ contient la distance s à u dans le graphe (-1 si aucun chemin n'existe entre s et u), et **Pred**$[u]$ contient le prédécesseur de u dans un plus court chemin depuis s.

On utilisera la structure suivante pour pouvoir renvoyer le couple **d**, **Pred**:

```c
typedef struct{
  int* d; // distances
  int* p; // prédécesseurs
} RES_PEL; // résultat parcours en largeur
```

Vous pouvez réutiliser les structures de file que vous avez implémenté au premier semestre, ou bien utiliser celle fournie dans l'archive de TP (fichiers "file.h" et "file.c").

Question 20. Écrire une fonction `RES_PEL* parcours_largeur(GRAPHE* g, int s)` appliquant un parcours en largeur dans le graphe `g` depuis `s`.

Question 21. Écrire une fonction `int* plus_court_chemin(GRAPHE* g, int s, 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.

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. On représentera les labyrinthes en utilisant le symbole "#" pour les cases occupées, et un espace pour les cases libres. De plus, on représentera l'entrée du labyrinthe par un "E" et la sortie par un "S". Vous trouverez dans l'archive du TP un fichier "maze_21_23.txt" qui correspond au labyrinthe suivant (l'entrée se situe dans le coin supérieur gauche, la sortie dans le coin inférieur droit):

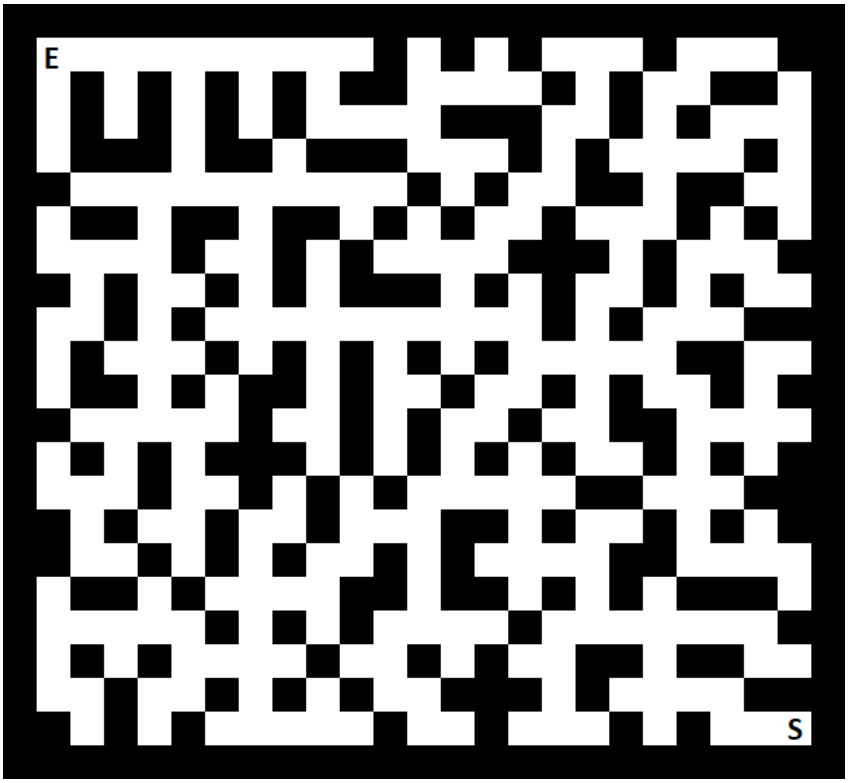

Figure 3: Labyrinthe de 21 × 23 cases

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

```
1 typedef struct{
2   int l; // nombre de lignes
3   int c; // nombre de colonnes
4   bool** grille; //grille[i][j] = true s'il y a un mur, false sinon
5   int ie, je ; //coordonnées de l'entrée
6   int is, js ; //coordonnées de la sortie
7 } LAB;
```

Question 22. Écrire une fonction `LAB* 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, et deux cases (i, j) sont reliées par une arête si et seulement si elles sont libres et adjacentes. Tous les sommets seront donc de degré au plus 4.

De plus, on suppose que nos labyrinthes ont moins de 2^{16} lignes et colonnes. On peut donc encoder les indices des cases sur 32 bits en tout, et donc se ramener à des entiers. Plus précisément, la case (i, j) sera associée à l'entier $i + 2^{16}j$. Ceci nous permettra de réutiliser notre structure de graphe sans modification.

Question 23. Écrire une fonction `GRAPHE* graphe_labyrinthe(LAB* lab)` permettant de construire un graphe à partir d'un labyrinthe comme décrit plus haut.

Question 24. Écrire une fonction `void resoudre_labyrinthe(LAB* lab)` qui affiche la suite des **directions** à prendre dans un labyrinthe pour aller de l'entrée à la sortie.

Graphes pondérés

Cette partie sera au début du TP16, elle est là si vous voulez déjà vous avancer.

On considère maintenant des graphes pondérés.Pour un graphe $G = (S, A, w)$ pondéré, on rappelle que les listes d'adjacence stockent dans ce cas des une liste de couples (voisin, poids de l'arête) pour chaque sommet $u \in S$, et que les matrices d'adjacences stockent les coefficients $w(u, v)$ pour $u, v \in S$. On considèrera des poids réels, pas forcément entiers ni positifs.

Question 25. Créer un nouveau fichier C, y recopier les types des listes d'adjacence de la section précédente, et faire les modifications nécessaires pour pouvoir gérer des graphes pondérés. On appelera le type `LADJ` et plus `GRAPHE`.

Question 26. Proposer un type pour les matrices d'adjacence, que l'on appelera `MADJ`.

Question 27. Écrire deux fonctions permettant de transformer une matrice d'adjacence en liste d'adjacence et inversement.

L'algorithme de Floyd-Warshall permet de calculer en $\mathcal{O}(n^3)$ la matrice des distances d'un graphe $G = (S, A, w)$, c'est à dire la donnée de la distance entre toute paire de sommet, pour un graphe ne contenant pas de cycle négatif. Faire une itération de l'algorithme de plus que nécessaire permet de détecter l'existence de cycles négatifs.
On rappelle le principe: pour un graphe $G = (S, A, w)$ avec $S = \{0, \dots, n-1\}$, on calcule pour $i, j \in [\![0, n-1]\!]$ et $k \in [\![0, n]\!]$ la quantité M_{ij}^k = le poids d'un plus court chemin entre i et j ne passant que par des sommets parmis 0 à $k-1$. On a la formule de récurrence suivante pour $k \geq 0$:

$$M_{ij}^{k+1} = \min(M_{ij}^k, M_{ik}^k + M_{kj}^k)$$

Cette formule exprime qu'un plus court chemin entre i et j peut soit passer par k, soit pas.

Question 28. Que vaut M_{ij}^0 pour $i, j \in [\![0, n-1]\!]$?

Question 29. Écrire une fonction `float** floyd_warshall(MADJ* g)` qui applique l'algorithme de Floyd-Warshall et renvoie la matrice des distances calculée.