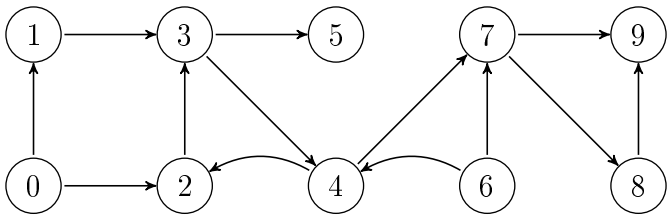
TD 5

Parcours d'un graphe

Correction

On considère le graphe suivant :



On donne les listes suivants :

```
let sommets_exemple = [0;1;2;3;4;5;6;7;8;9] ;;
let arcs_exemple = [(0,1);(1,3);(0,2);(2,3);(3,5);(3,4);(4,2)
;(4,7);(6,4);(6,7);(7,8);(7,9);(8,9)] ;;
```

1 Construction du graphe

On utilise les deux types utilisés au TD précédent :

```
type graphe_1 = bool array array ;;
type graphe_2 = int list array ;;
```

1. Écrire la fonction `construction_1` qui prend en arguments la liste des sommets et la listes des arcs et renvoies le graphe correspondant codé avec le premier type.

   ```
   construction_1 : 'a list -> (int * int) list -> graphe_1
   ```

   Correction

Listing 1 – La fonction `construction_1`

```
let construction_1 sommet arcs =
  let n = List.length sommet in
  let (graphe : graphe_1) = Array.make_matrix n n false in
  let rec aux liste = match liste with
    | [] -> ()
    | (a,b)::reste -> (graphe.(a).(b) <- true ; aux reste)
    in
  aux arcs ; graphe;;
```

2. Écrire la fonction `construction_2` qui prend en arguments la liste des sommets et la listes des arcs et renvoies le graphe correspondant codé avec le second type.

   ```
   construction_2 : 'a list -> (int * int) list -> graphe_2
   ```

Correction

Listing 2 – La fonction `construction_2`

```
let construction_2 sommet arcs =
  let n = List.length sommet in
  let (graphe : graphe_2) = Array.make n [] in
  let rec aux liste = match liste with
    | [] -> ()
    | (a,b)::reste ->( graphe.(a) <- b::graphe.(a) ; aux reste)
    in
  aux arcs ; graphe;;
```

2 Fonctions utiles

1. Écrire la fonction `appartient` qui prend en argument un élément et une liste, et renvoie le booléen correspondant à l'appartenance de l'élément à la liste.

   `appartient : 'a -> 'a list -> bool`

   (Remarque : cette fonction correspond à la fonction `List.mem`) Correction

Listing 3 – La fonction `appartient`

```
let rec appartient a liste = match liste with
  | [] -> false
  | x::reste when x = a -> true
  | x::reste -> appartient a reste;;
```

2. Écrire la fonction `retourne` qui prend en argument une liste, et renvoie la liste retournée.

   `retourne : 'a list -> 'a list`

   (Remarque : cette fonction correspond à la fonction `List.rev`) Correction

Listing 4 – La fonction `retourne`

```
let retourne l =
  let rec aux l_1 l_2 = match l_1 with
  | [] -> l_2
  | a::reste -> aux reste (a::l_2)
  in
  aux l [];;
```

3 Parcours en largeur

On utilisera le `type graphe_2` dans la suite du TD.
Le parcours en largeur (appelé BFS, pour Breadth First Search) consiste à traiter le sommet puis tous ses sommets voisins en évitant de traiter deux fois un sommet.

1. Donner la liste des sommets accessibles en partant du sommet 3 sur le graphe exemple.

   Correction

   [3 ; 4 ; 5 ; 7 ; 2 ; 9 ; 8]

2. Écrire la fonction `bfs` qui prend en argument un graphe et une liste de sommets déjà vus et une liste de sommets à voir, et renvoie la liste des sommets parcourus.

   `bfs : graphe_2 -> int list -> int list -> int list`

   Correction

   Listing 5 – La fonction `bfs`

```
let rec bfs (g: graphe_2) liste_vu liste_a_voir = match liste_a_voir with
| [] -> retourne liste_vu
| t :: q -> if appartient t liste_vu then bfs g liste_vu q
else bfs g (t :: liste_vu) (q @ g.(t));;
```

3. En déduire la fonction `parcours` qui prend en argument un graphe et un sommet, et renvoie la liste de sommets accessibles dans un parcours en largeur.

   `parcours : graphe_2 -> int -> int list`

   Correction

   Listing 6 – La fonction `parcours`

```
let parcours g sommet = bfs (g: graphe_2) [] [sommet];;
```

4. Le parcours en largeurs traite les sommets dans l'ordre de leur distance au sommet de départ. En notant (s_i^j) les sommets du parcours où j est la distance de s_i à d, on a :

$$\left[d, \underbrace{s_1^1, s_2^1, ..., s_n^1}_{distance=1}, \underbrace{s_{n+1}^2, ..., s_m^2}_{distance=2}, ..., \underbrace{s_{m+1}^k, ..., s_p^k}_{distance=k}\right]$$

   On cherche à déterminer le chemin le plus court entre deux sommets d et a du graphe. Si a est un élément du parcours, en notant k la distance entre d et a, on a :

$$\left[d, \underbrace{s_1^1, s_2^1, ..., s_n^1}_{distance=1}, \underbrace{s_{n+1}^2, ..., s_m^2}_{distance=2}, ..., \underbrace{s_{m+1}^k, ..., a, ..., s_p^k}_{distance=k}...\right]$$

Il suffit donc de reconstituer le parcours en partant du sommet d'arrivée.

(a) Écrire la fonction `liste_num` qui prend en argument une liste et un entier n et renvoie la liste des couples constituer des éléments de la liste, et de l'entier n.

```
liste_num : 'a list -> 'b -> ('a * 'b) list
```

Exemple :

```
liste_num ['a'; 'b' ; 'c'] 3;;
- : (char * int) list = [('a', 3); ('b', 3); ('c', 3)]
```

Correction

Listing 7 – La fonction `liste_num`

```
let rec liste_num l n =
  match l with
  | []-> []
  | a::reste -> (a,n)::(liste_num reste n);;
```

(b) Écrire la fonction `appartient_couple` qui prend un élément e et une liste de couple et renvoie le booléen correspondant au fait que e est le première élément d'un couple de la liste.

```
appartient_couple : 'a -> ('a * 'b) list -> bool
```

Exemple :

```
appartient_couple 'c' [('a',2) ; ('b',5);('c',2)];;
- : bool = true
```

Correction

Listing 8 – La fonction `appartient_couple`

```
let rec appartient_couple e l  =
  match l with
  | [] -> false
  | (a , b) :: reste when a = e -> true
  |  (a , b) :: reste -> appartient_couple e reste ;;
```

(c) Modifier la fonction `bfs`, en fonction `bfs_distance`, en y ajoutant la distance parcourue.

```
bfs_distance :
  graphe_2 -> (int * int) list -> (int * int) list -> (int * int)
list
```

Correction

Listing 9 – La fonction `bfs_distance`

```
let rec bfs_distance (g : graphe_2) liste_vu liste_a_voir =
  match liste_a_voir with
```

```
    | [] -> liste_vu
    | (s,d) :: reste ->
        if appartient_couple  s liste_vu then
          bfs_distance g liste_vu reste
        else

          bfs_distance g ((s,d)::liste_vu) (reste @ (liste_num g.(s)
(d+1)))
           ;;
```

(d) Écrire la fonction `parcours_distance` qui renvoie la liste des sommets accessibles, doublés de la distance.

```
parcours_distance : graphe_2 -> int -> (int * int) list
```

Correction

Listing 10 – La fonction `parcours_distance`

```
let parcours_distance (g:graphe_2) sommet  =
  List.rev (bfs_distance (g : graphe_2) [] [(sommet , 0 )]) ;;
```

(e) Écrire la fonction `chemin` qui renvoie la liste des sommets entre d et a, représentant le plus court chemin.

```
chemin : graphe_2 -> int -> int -> (int * int) list
```

Correction

Les sommets possibles pour le chemin sont des sommets accessibles depuis le sommets de départ. On récupère ces sommets avec la fonction `sommets_possibles`, qui renvoie une exception si l'arrivée n'est pas accessibles. Une fois la liste des possibles créée, on reconstitue le chemin en partant de l'arrivée.

Listing 11 – La fonction `chemin`

```
let chemin (graphe: graphe_2) sommet_init sommet_final =
  let accessibles = parcours_distance graphe sommet_init in
  let rec sommets_possibles element l_1 l_2= match l_1 with
    | [] -> failwith "␣pas␣de␣chemin"
    | (a,d)::reste when a = element -> (element,d)::l_2
    | (a,d)::reste -> sommets_possibles element reste ((a,d)::l_2)
  in
  let possibles = sommets_possibles sommet_final accessibles [] in
  let rec reconstitution l_1 l_2 = match l_1 with
    | [(a,d)] -> (a,d)::l_2
    | (a,d_a)::(b,d_b)::reste when (appartient a graphe.(b) && d_b =
d_a-1 )->
        reconstitution ((b,d_b)::reste) ((a,d_a)::l_2)
    | a::b::reste -> reconstitution (a::reste) l_2
  in reconstitution possibles [];;
```

4 Parcours en profondeur

Le parcours en profondeur (appelé DFS, pour Depth First Search) consiste à parcourir le graphe jusqu'au bout, puis de revenir aux sommets non traités.

1. Donner la liste des sommets accessibles en partant du sommet 0 sur le graphe exemple.

   Correction

   [0 ;1 ; 3 ; 4 ; ;2 ; 7 ; 8 ; 9 ; 5]

2. Écrire la fonction `dfs` qui prend en argument une graphe une liste de sommets déjà vus et une liste de sommets à voir, et renvoie la liste des sommets parcourus.

   `dfs : graphe_2 -> int list -> int list -> int list`

   Correction

   Listing 12 – La fonction `bfs`

```
let rec dfs (g: graphe_2) liste_vu liste_a_voir = match liste_a_voir with
| [] -> retourne liste_vu
| t :: q -> if appartient t liste_vu then dfs g liste_vu q
else dfs g (t :: liste_vu) (g.(t) @ q );;
```

3. En déduire la fonction `parcours_profond` qui prend en argument un graphe et un sommet, et renvoie la liste de sommets accessibles dans un parcours en long.

   `parcours_profond : graphe_2 -> int -> int list`

   Correction

   Listing 13 – La fonction `parcour_profonds`

```
let parcours_profond g sommet =  dfs (g: graphe_2) [] [sommet];;
```