TP : Graphes OCAML

1 Généralités

Exercice 1.

1. Montrer qu'un graphe simple non orienté a un nombre pair de sommets de degré impair.
2. Montrer que dans une assemblée de n personnes ($n > 2$), il y a toujours au moins deux personnes qui ont le même nombre d'amis présents (on considère que la relation d'amitié est symétrique mais pas réflexive).
3. Est-il possible de créer un réseau de 15 ordinateurs de sorte que chaque machine soit reliée avec exactement trois autres ?

Exercice 2. (Propriété de König) Montrer que dans un graphe $G = (V, E)$, s'il existe un chemin d'un sommet u vers v alors il existe un chemin élémentaire de u vers v

Exercice 3. On considère le type :

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

1. Ecrire la fonction `transpose (a:graphe) -> graphe` qui transpose un graphe.
2. On appelle *puits total* d'un graphe orienté, tout sommet p tel que
   — Il existe un arc de tout sommet vers p.
   — il n'y a qu'un arc d'origine p.

   Montrer que dans un puits p, le seul arc sortant est une boucle.
3. Écrire la fonction `puits (g:graphe) : bool` qui indique par un bouléen s'il existe un puits total dans le graphe représenté par `g`.

Exercice 4. On considère le type

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

Soit $G = (V, E)$ un graphe et $f : V \to [\![0, N]\!]$. la fonction f est un *coloriage* du graphe G si, pour tout (i, j) dans E avec $i \neq j$, $f(i) \neq f(j)$.

Écrire une fonction `coloriage (g:graphe) (f:int array) : bool` qui détermine si une fonction (donnée par un tableau de longueur $|V|$) est un coloriage du graphe G.

Exercice 5. Montrer que dans un arbre (non enraciné), pour tout couple u, v de sommets, il y a un unique chemin (sous-entendu *élémentaire*) de u à v.

Exercice 6. Si dans un graphe non orienté sans boucle, tout sommet est de degré supérieur à 2, alors G possède au moins un cycle.

Proposer deux preuves.

Exercice 7. On rappelle la définition suivante du cours

Définition 1.

— Un sommet r d'un graphe orienté $G = (S, E)$ est une *racine* de G si pour tout sommet x de G il existe un chemin de r à x.

— On dit qu'un graphe orienté $G = (V, E)$ est une *arborescence* s'il possède un unique élément x_0 de degré entrant nul, si tous les autres sont de degré entrant 1 et si il existe un chemin de x_0 à tous les autres sommets.

Soit $G = (V = \{x_0, x_1, \dots, x_n\}, E)$, une arborescence avec x_0 comme dans la définition et $n > 0$. Prouver les propriétés suivantes :

1. On considère un graphe avec un seul sommet de degré entrant 0 et tous les autres de degré entrant 1. Est-ce une arborescence ?
2. Montrer que G possède un élément de degré sortant nul et que celui-ci n'est pas x_0.
3. Soit x_0 comme dans la définition. C'est donc une racine. Montrer que c'est la seule.
4. Montrer que si x, y sont dans V et si il existe un chemin de x à y alors ce chemin est unique.
5. Montrer qu'une arborescence n'a pas de circuit.

Exercice 8. Dans cet exercice, les sommets d'un graphe orienté d'ordre n sont numérotés de 0 à $n-1$. Les voisins d'un sommet s sont listés par ordre croissant et cette liste est placée dans un tableau à la position s.

On donne les types et fonctions suivantes du cours :

```
1 || type graphe = int list array;;(*graphes orientés*)
```

On choisit donc une représentation des graphes par liste d'adjacence.

Si, pour former un graphe à n sommets, on part du graphe à n sommets sans arête et qu'on insère un à un les arcs dans l'ordre lexicographique, alors les listes d'adjacence sont triées par ordre croissant.

Nous supposons ce caractère (les listes d'ajacences sont triées) vérifié dans la suite et nous essayons de le conserver.

1. Question de cours. Écrire les fonctions suivantes :
   (a) **voisins : graphe** → **int** → **int list** qui donne les voisins d'un sommet.
   (b) **ajouter_o : graphe** → **int** → **int** → **unit** qui ajoute un arc.
   (c) **retirer_o : graphe** → **int** → **int** → **unit** qui retire un arc donné.

   On suppose que l'arc à ajouter ou à retirer est cohérent (il joint 2 sommets qui existent).
2. Comment sont codés les graphes non orientés ?
3. Écrire la fonction **ajouter_no : graphe** → **int** → **int** → **unit** qui ajoute une arête $\{i, j\}$ à un graphe non orienté.
4. Écrire la fonction **o2no : graphe** → **unit** qui transforme un graphe orienté en graphe non orienté.
5. Ecrire la fonction **check : graphe** → **bool** qui teste si un graphe est non orienté.
6. Une chaîne d'un graphe non orienté connexe est dite *eulérienne* si elle passe par toutes les arêtes une seule fois par chaque arête. Même définition pour les cycles.

   Théorème 1.1. — *Un graphe connexe sans boucle admet une chaîne eulérienne si et seulement s'il possède zéro ou deux sommet(s) de degré impair.*

   — *Un graphe connexe sans boucle admet un cycle eulérien si et seulement s'il ne possède que des sommets de degré pair.*

   Écrire la fonction **euler : graphe** → **int** qui dit si un graphe non orienté supposé connexe possède un cycle eulérien (réponse 0), une chaîne eulérienne mais pas de cycle eulérien (réponse 1), ni l'un ni l'autre (réponse 2) Notre graphe peut avoir des boucles.

Exercice 9. Combien existe-t-il de graphes orientés (resp. non orientés) à n sommets fixés ?

Exercice 10. Soit $G = (V, E)$ un graphe non orienté. On appelle *arbre couvrant* de G tout sous-graphe $G' = (V, E')$ tel que G' est un arbre (donc non orienté, acyclique, connexe, sans boucle).

1. Montrer que si G est connexe il existe un arbre couvrant de G.
2. Montrer que si G n'est pas connexe, il existe une forêt couvrante de G.

Exercice 11. On se place dans un graphe non orienté sans boucle.

Une chaîne est alors dite *eulérienne* si elle passe par toutes les arêtes exactement une fois. Un cycle est dit *eulérien* s'il vérifie les mêmes propriétés.

Soit $G = (V, A)$ un graphe non orienté connexe.

1. Montrer que G possède un cycle eulérien si et seulement si chaque sommet est de degré pair.
   Proposer un algorithme pour construire un chemin eulérien dans un graphe non orienté dont tous les sommets sont de degré pair.
2. Montrer que G possède une chaîne eulérienne si et seulement si chaque sommet est de degré pair sauf zéro ou deux d'entre eux.

Exercice 12. On considère un graphe non orienté $G = (V, E)$ sans boucle. Pour tout $v \in V$ on note $\mathcal{V}(v)$ l'ensemble de ses voisins. Pour un sommet v on note $G \setminus v$ le sous-graphe induit par $V \setminus \{v\}$.

On applique au graphe la fonction suivante :

```
1  fonction F(graphe : G = (V,E))
2      si E = ∅ alors renvoyer |V|
3      sinon
4          soit v ∈ V tel que deg v ≥ 1
5          p ← F(G \ v)
6          q ← F(G \ (v ∪ 𝒱(v)))
7          renvoyer max(p, q+1).
```

1. On suppose dans un premier temps que cette fonction est bien définie (et donc que quel que soit le choix des v successifs de degré ≥ 1, le résultat sera le même).
   Que retourne la fonction F lorsque :
   (a) G est un graphe sans arête.
   (b) G est un graphe complet.
   (c) G est un chemin non ramifié.
   (d) G est un cycle élémentaire.
   On demande des preuves et pas simplement des impressions.
2. On appelle *stable* un sous-ensemble de V dans lequel ne figure aucun couple de voisins. On rappelle que G est considéré non orienté sans boucle.
   Prouver avec soin que la fonction F calcule le cardinal maximal d'un stable de G (le *nombre de stabilité* de G).
3. Quel est le coût de cet algorithme ?

2 Parcours

Couleurs

Dans toute cette section on dispose du type somme suivant :

```
1  type couleur = R| V| B;;
```

Nous l'utilisons pour désigner les couleurs que prennent les sommets des graphes à explorer.

Parcours en largeur

Dans l'optique du parcours en largeur, voici les primitives à connaître du module **Queue** :

```
1  val create : unit -> 'a t
2  (*Return a new queue, initially empty.*)
3
4  val add : 'a -> 'a t -> unit
5  (*add x q adds the element x at the end of the queue q.*)
6
7  val push : 'a -> 'a t -> unit
8  (*push is a synonym for add.*)
9
10 val take : 'a t -> 'a
11 (*take q removes and returns the first element in queue q, or raises Queue.Empty if the queue
       is empty.*)
12
13 val is_empty : 'a t -> bool
14 (*Return true if the given queue is empty, false otherwise.*)
```

Insistons une fois encore sur la différence interface/implémentation des structures de données : nous connaissons les noms des primitives et ce qu'elles font (c'est l'interface) mais nous ne savons pas comment elles le font (c'est l'implémentation ; elle nous est cachée).

Exercice 13. Les graphes sont implantés par tableau de listes d'adjacence.

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

On utilise le module **Queue** de Ocaml qui implante la notion de file FIFO.

1. Écrire la fonction **distance g x** det ype **int list array -> int -> int array** qui retourne le tableau des distances du sommet x aux autres sommets de g (-1 si la distance est infinie) au moyen d'un parcours en largeur. Tous les arcs de g sont considérés de valuation 1.

```
# let g=[|[0;1;4];[2;5];[6;7];[2];[4;5];[];[7];[5]|] in
distance g 0;;
  - : int array = [|0; 1; 2; -1; 1; 2; 3; 3|]
```

2. Ecrire le fonction **chemin** de type **int list array -> int -> int -> int list** qui prend en paramètre un graphe et 2 sommets et retourne un PCC du premier au second.

```
# let g=[|[1;4];[2;5];[6;5];[2];[5];[];[7];[3]|] in
chemin g 0 3;;
  - : int list = [0; 1; 2; 6; 7; 3]
```

3. Le module **Queue** est pratique mais non indispensable car on maîtrise bien ici la taille maximale de la file.

   Proposer une façon simple de se passer de ce module au moyen d'un type enregistrement adapté à notre problème et comportant un tableau et deux indices . Implémenter alors la fonction de création **create n** de file vide à n éléments possibles puis **push,take,is_empty**.

Exercice 14. Répondre sans structure impérative dans le code.

Écrire la fonction **accessible g x** de type **int list array -> int -> int list** qui retourne la liste des accessibles du sommet **x** par un parcours en largeur dans le graphe **g**. On utilise ici une liste comme une file (ce qu'il ne faut en général pas faire pour des raisons de complexité).

```
# let g=[|[0;1;4];[2;5];[6;7];[2];[4;5];[];[7];[5]|] in
accessible_bfs g 0;;
  - : int list = [0; 1; 4; 2; 5; 6; 7]
```

Parcours en profondeur

Exercice 15. On donne :

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

Écrire la fonction `accessible_dfs (g:graphe) (x:int) : couleur array` qui renvoie le tableau RVB des couleurs des sommets après un DFS depuis x.

```
# let g = [|[1;4];[4;2];[2;5];[2;4];[8];[6];[7];[];[]|]
in accessible_dfs g 2;;
  - : couleur array = [|B; B; R; B; B; R; R; R; B|]
```

Exercice 16. Dans certains ouvrages, les auteurs présentent les parcours en utilisant un accumulateur(une pile (pour le DFS) ou une file (pour le BFS)) de sommets à visiter ainsi qu'une liste de sommets déjà vus. On ne retrouve pas exactement les 3 couleurs du cours.

L'algorithme pour les deux parcours (DFS et BFS) est alors :

Initialisation Au départ l'accumulateur contient seulement le point de départ ; la liste des sommets déjà rencontrés est vide.

Déroulement tant que l'accumulateur n'est pas vide :

1. retirer un sommet s de l'accumulateur et l'insérer dans la liste des sommets déjà vus.
2. Ajouter en une seule fois les voisins de s à l'accumulateur.

Cela offre l'avantage de la simplicité : le code est alors très court (5 lignes).

1. Implanter le DFS selon ce pincipe.
2. Idem pour le BFS.
3. Que dire de la complexité ?
4. Avec le DFS du cours, combien de fois un même sommet apparaît-il au dessus de la pile ? Combien de fois avec le DFS proposé ici ? Donner un exemple de recherche permise avec l' algorithme du cours et plus difficile ici.

Exercice 17. On admet qu'un graphe fortement connexe orienté admet un circuit eulérien si et seulement si chaque sommet a le même degré entrant que sortant.

Ecrire une fonction `eulerien : int list array -> int list` qui prend un tel graphe et retourne un circuit eulérien.

```
# let g = [|[1;2];[3];[4];[3;5];[0];[0;6];[7];[8];[5;9;10];[10];[8] |] in
eulerien g;;
  - : int list = [0; 1; 3; 3; 5; 6; 7; 8; 9; 10; 8; 10; 5; 0; 2; 4; 0]
```

Exercice 18. Ecrire la fonction **hascycle** de type **int list array -> int list** qui prend un graphe en paramètre et renvoie une liste de sommets représentant un cycle s'il en existe un et la liste vide sinon. La fonction effectue un parcours en profondeur RVB.

```
# let g = [|[1;4];[4;2];[2;5];[2;4];[8];[6];[7];[3];[]|]
    in hascycle g;;
  - : int list = [2; 5; 6; 7; 3; 2]
# let g = [|[1;4];[4;2];[2;5];[2;4];[8];[6];[7];[8];[]|]
    in hascycle g;;
  - : int list = []
```

Exercice 19. Les graphes sont représentés par matrices d'adjacence.

Ecrire la fonction **tritopo** de type **int array array -> int array** qui retourne un tableau contenant une rénumérotation des sommets dans un tri topologique.

Avec

```
  let g = let t = Array.make_matrix 10 10 0 in t.(0).(1)<-1;
  t.(0).(6)<-1; t.(1).(2)<-1; t.(1).(6)<-1; t.(2).(3)<-1;t.(2).(7)<-1;
  t.(3).(8)<-1; t.(4).(9)<-1;t.(4).(3)<-1; t.(6).(5)<-1;t.(6).(7)<-1;
  t.(7).(8)<-1; t.(9).(8)<-1; t;;
```

```
# tritopo g;;
- : int array = [|3; 4; 7; 9; 1; 6; 5; 8; 10; 2|]
```

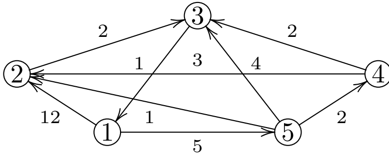
PCC

Exercice 20. Les graphes sont implémentés par un tableau de listes d'adjacence et les valuations des arcs sont données dans une matrice. Ainsi :

```
let g = [|[1;4];[2];[0];[1;2];[1;2;3]|];;

let w = let mat = Array.make_matrix 5 5 (-1) in
  for i = 0 to 4 do
    mat.(i).(i) <-0;
  done;
  mat.(0).(1)<-12;mat.(0).(4)<-5;
  mat.(1).(2)<-2;mat.(2).(0)<-1;
  mat.(3).(2)<-2;mat.(3).(1)<-3;
  mat.(4).(1)<-1;mat.(4).(2)<-4;
  mat.(4).(3)<-2; mat;;
```

représente



Comme les arcs sont positifs, l'infini est codé par -1. Il faudra en tenir compte dans les additions de distances et les comparaisons.

Notre objectif est une implantation de l'algorithme de Dijkstra utilisant une file de priorité (un tas-min ordonné par distances croissantes à la source).

Remarque. Évidemment, on peut toujours retrouver le tableau des listes de voisins à partir de la matrice des valuations mais cela engendre un coût. Nous supposons dans la suite que les deux informations sont connues.

On considère les types suivants :

```
type color = R|V|B;;

type ('a, 'b) data = {mutable distance: 'a;
                      vertice: 'b; pred :'b } ;;

type ('a, 'b) priority_file = {mutable n: int; tbl: ('a, 'b) data array} ;;
```

Une donnée contient le nom d'un sommet, une information sur sa distance à la source et son prédecesseur dans un PCC depuis la source.

On implémente une variante de l'algorithme de Dijksta dans laquelle la file de priorité contient des sommets verts ou rouges. On commence par insérer la source dans la file avec une distance de 0 et la source elle-même comme prédecesseur.

Tant que la file de priorité n'est pas vide :

- On récupère la donnée ayant la plus faible distance à la source.
- Si elle concerne un sommet x qui n'est pas rouge, on rend ce sommet rouge. On ajoute à la file toutes les données (nom, distance, prédecesseur) concernant les voisins de x (quelle que soit la couleur des voisins).

  Les voisins bleus deviennent verts.

Avec ce procédé, la file peut contenir des informations contradictoires concernant un sommet x donné. Mais ce n'est pas grave : seule la plus petite distance à la source renseignée pour x sera prise en compte. De plus, on n'a pas vraiment besoin de 3 couleurs : seule l'information Rouge/ Non rouge est pertinente.

1. L'infini est représenté par -1. Implanter les fonctions

```
val add : int -> int -> int (*addition d'entiers*)
val less : int -> int -> bool (*implémente "<"*)
```

2. Implanter la fonction

```
val nba : int list array -> int
```

   Elle renvoie le nombre d'arcs d'un graphe.

3. Implémenter les primitives (création, insertion, suppression et percolations) des files de priorité.

```
(*généré par
        ocamlc -i dijkstra_always_add.ml*)
val creer : int -> 'a -> int -> ('a, int) priority_file
val swap : int -> int -> 'a array -> unit
val is_empty : ('a, 'b) priority_file -> bool
val is_full : ('a, 'b) priority_file -> bool
exception Empty
exception Full
val push : (int, 'a) data -> (int, 'a) priority_file -> unit
val take : (int, 'a) priority_file -> (int, 'a) data
```

4. Écrire la fonction

```
val dijkstra_all :
  int list array -> int array array -> int -> int array * int array
```

Si g est un graphe représenté par tableau de listes de voisins, w une matrice des valuations et s un sommet, la fonction retourne le tableau des prédecesseur et celui des distances pour tous les PCC depuis s.

```
# dijkstra_all g w 0;;
- : int array * int array = ([|0; 4; 1; 4; 0|], [|0; 6; 8; 7; 5|])
```

Estimer complexité satiale et temporelle. Comparer avec l'algorithme du jour dans lequel au plus une information concernant un sommet donné figure dans la file.

Exercice 21. On considère le type des entiers-avec-infini. Écrivons ceci dans l'interpréteur et observons le résultat :

```
# module E = struct
  type elt = I of int | Infty

  let (++) x y = match x,y with
    | I(a), I(b) -> I (a+b)
    | _,_ -> Infty

  let (<<) x y = match x,y with
    | I(a), I(b) -> a < b
    | I(a), _ -> true
    | _, I(b) -> false
    | _,_ -> false

  let infini = Infty

  let zero = I 0

  let min x y = if x << y then x else y

end;;
module E :
  sig
    type elt = I of int | Infty
    val ( ++ ) : elt -> elt -> elt
    val ( << ) : elt -> elt -> bool
    val infini : elt
    val zero : elt
    val min : elt -> elt -> elt
  end
```

On a ici encapsulé tout ce dont on a besoin (type, addition, comparaison, infini, zéro et minimum) dans un *module* OCaml. Comme le nom de *module* le suggère, il s'agit avant-tout d'une manière d'introduire de la modularité dans un programme, i.e. de le découper en unités de taille raisonnable. Observer que l'interpréteur nous renseigne sur la *signature du module* (type interne, signature des opérateurs) mais pas sur le code.

Voici comment faire des additions et des comparaisons en utilisant une ouverture locale du module :

```
# let open E in
I(6) ++ I(3), I(6) ++ Infty,
I(6) << I(3),
I(6) << Infty;;
      - : E.elt * E.elt * bool * bool = (E.I 9, E.Infty, false, true)
```

L'aspect infixe des opérateurs est accessible du fait de l'ouverture (locale) du module. Sans cette ouverture, il aurait fallu écrire les opérations en préfixant par le nom du module. Et l'addition aurait été préfixée :

```
# E.(++) (E.I(6)) (E.Infty);;
- : E.elt = E.Infty
```

Q.1 Écrire la fonction `fw : E.elt array array -> E.elt array array * int array array` qui implémente l'algorithme de Floyd-Warshall dans lequel les graphes sont représentés par des matrices de `E.elt`.

La fonction retourne la matrice des distances et celle des prédecesseurs. On ne s'autorise qu'une ouverture locale du module `E`.

```
# let g = let g =Array.make_matrix 5 5 E.Infty in
  g.(0).(1)<-E.I(2);
  g.(1).(2)<-E.I(6);
```

```
  g.(2).(1)<-E.I(7);
  g.(3).(2)<-E.I(1);
  g.(3).(4)<-E.I(3);
  g.(4).(0)<-E.I(1);
  g.(4).(1)<-E.I(4);
  g;;
                  val g : E.elt array array =
  [|[|E.Infty; E.I 2; E.Infty; E.Infty; E.Infty|];
    [|E.Infty; E.Infty; E.I 6; E.Infty; E.Infty|];
    [|E.Infty; E.I 7; E.Infty; E.Infty; E.Infty|];
    [|E.Infty; E.Infty; E.I 1; E.Infty; E.I 3|];
    [|E.I 1; E.I 4; E.Infty; E.Infty; E.Infty|]|]
# fw g;;
- : E.elt array array * int array array =
([|[|E.Infty; E.I 2; E.I 8; E.Infty; E.Infty|];
   [|E.Infty; E.I 13; E.I 6; E.Infty; E.Infty|];
   [|E.Infty; E.I 7; E.I 13; E.Infty; E.Infty|];
   [|E.I 4; E.I 6; E.I 1; E.Infty; E.I 3|];
   [|E.I 1; E.I 3; E.I 9; E.Infty; E.Infty|]|],
 [|[|-1; -1; 1; -1; -1|]; [|-1; 2; -1; -1; -1|]; [|-1; -1; 1; -1; -1|];
   [|4; 4; -1; -1; -1|]; [|-1; 0; 1; -1; -1|]|])
```

Il serait dommage de devoir écrire un nouveau code pour chaque structure de données implémentant la notion d'infini, une addition et une comparaison.

On définit tout d'abord une signature de module contenant ces trois éléments :

```
module type S = sig
  type elt
  val (++) : elt -> elt -> elt
  val (<<) : elt -> elt -> bool
  val infini : elt
  val zero : elt
  val min : elt -> elt -> elt
end;;
```

Observons que cette signature est exactement celle du module des entiers-avec-infini `E`. C'est tout l'intérêt !

Il est temps de parler de la notion (totalement hors programme) de *foncteur*. Il s'agit d'un module paramétré par la signature d'un autre module (exactement comme il y a des types paramétrés par d'autres types). Par exemple, considérons le module qui contient une unique fonction, laquelle multiplie un élément par 2 :

```
module TWICE(X:S) = struct
  let twice x = let open X in x ++ x;;
end;;
```

On voit bien que ce foncteur ne contient qu'une fonction. Instancions le de deux façons différentes. D'abord en utilisant les entiers-avec-infini :

```
# module TWE = TWICE(E);;
module TWE : sig val twice : E.elt -> E.elt end
# TWE.twice (I(3));;
- : E.elt = E.I 6
```

Génial, on peut multiplier un entiers-avec-infini par 2 !

Construisons un type des flottants :

```
module F = struct
  type elt = float
  let (++) (x:elt) (y:elt) = x +. y
  let (<<) (x:elt) (y:elt) = x < y
  let infini = infinity
  let zero = 0.
  let min = min
end;;  (*la signature est la même que S !!*)
```

Instancions `TWICE` avec ce type `F` et réalisons une multiplication par 2 :

```
# module TWF = TWICE(F);;
module TWF : sig val twice : F.elt -> F.elt end
# TWF.twice 3.;;
- : F.elt = 6.
```

Et voilà : on peut multiplier un flottant par 2 !

Revenons maintenant à Floyd-Warshall :

Q.2 Écrire un foncteur `FW(X:S)` qui contient deux fonctions. La première initialise la matrice des prédecesseurs (elle est renseignée ensuite durant le déroulement de Floyd-Warshall). La seconde implémente Floyd-Warshall : la fonction `fw` lance un Floyd-Warshall spour un graphe représenté par une matrice d'éléments de type `X.elt`. Un tuple (matrice des valuations, matrice des prédecesseurs) est renvoyé.

```
functor (X : S) ->
    sig
      val init_pred : X.elt array array -> int array array
      val fw : X.elt array array -> X.elt array array * int array array
    end
```

Appplications.

— D'abord avec des entiers :

```
let g = let g =Array.make_matrix 5 5 E.Infty in
  g.(0).(1)<-E.I(2);
  g.(1).(2)<-E.I(6);
  g.(2).(1)<-E.I(7);
  g.(3).(2)<-E.I(1);
  g.(3).(4)<-E.I(3);
  g.(4).(0)<-E.I(1);
  g.(4).(1)<-E.I(4);
  g;;

module FWE =  FW(E);;

FWE.fw g;;
```

On obtient :

```
# FWE.fw g;;
- : E.elt array array * int array array =
([|[|E.Infty; E.I 2; E.I 8; E.Infty; E.Infty|];
   [|E.Infty; E.I 13; E.I 6; E.Infty; E.Infty|];
   [|E.Infty; E.I 7; E.I 13; E.Infty; E.Infty|];
   [|E.I 4; E.I 6; E.I 1; E.Infty; E.I 3|];
   [|E.I 1; E.I 3; E.I 9; E.Infty; E.Infty|]|],
 [|[|-1; -1; 1; -1; -1|]; [|-1; 2; -1; -1; -1|]; [|-1; -1; 1; -1; -1|];
   [|4; 4; -1; -1; -1|]; [|-1; 0; 1; -1; -1|]|])
```

On fait le choix de la détection de cycle : les coefficients diagonaux de la matrice des valuations ne sont pas initialisés à zéro.

— Ensuite avec des flottants :

```
module FWF =  FW(F);;

let g' = let g =Array.make_matrix 5 5 infinity in
  g.(0).(1)<-2.;
  g.(1).(2)<-6.;
  g.(2).(1)<-7.;
  g.(3).(2)<-1.;
  g.(3).(4)<-3.;
  g.(4).(0)<-1.;
  g.(4).(1)<-4.;
  g;;

FWF.fw g';;
```

On obtient :

```
# FWF.fw g';;
- : F.elt array array * int array array =
([|[|infinity; 2.; 8.; infinity; infinity|];
   [|infinity; 13.; 6.; infinity; infinity|];
```

```
    [|infinity; 7.; 13.; infinity; infinity|]; [|4.; 6.; 1.; infinity; 3.|];
    [|1.; 3.; 9.; infinity; infinity|]|],
 [|[|-1; -1; 1; -1; -1|]; [|-1; 2; -1; -1; -1|]; [|-1; -1; 1; -1; -1|];
    [|4; 4; -1; -1; -1|]; [|-1; 0; 1; -1; -1|]|])
```

Q.3 Sans aucune modification du code de `FW`, écrire un module `B` instanciant `S`. Les graphes sont alors représenté par une matrice d'adjacence bouléenne indiquant la présence ou non d'un arc.

La fonction `fw` du module `FW(B)` renvoie toujours un tuple de matrices. La première est une matrice de bouléens qui représente la *fermeture transitive* du graphe passé en argument. Le coefficient ligne i colonne j $(i \neq j)$ de cette matrice vaut `true` si et seulement si j est accessible depuis i.

```
module FWB =  FW(B);;

let gb = let g = Array.make_matrix 5 5 false in
g.(0).(1)<-true;g.(1).(2)<-true;g.(2).(3)<-true;g.(3).(4)<-true;
g.(3).(1)<-true; g;;

FWB.fw gb;;
```

Avec le code ci-dessus, on obtient :

```
# FWB.fw gb;;
- : B.elt array array * int array array =
([|[|false; true; true; true; true|]; [|false; true; true; true; true|];
   [|false; true; true; true; true|]; [|false; true; true; true; true|];
   [|false; false; false; false; false|]|],
 [|[|-1; 0; 1; 2; 3|]; [|-1; 3; 1; 2; 3|]; [|-1; 3; 1; 2; 3|];
   [|-1; 3; 1; 2; 3|]; [|-1; -1; -1; -1; -1|]|])
```

En particulier, on remarque que 0 accède à tous les autres sommets et 4 à aucun autre.