DS 2

Devoir sur table

(4 heures)

Le sujet a pour but l'étude de labyrinthes.
Dans tous le sujet, on considère des graphes non orientés, sans boucles. On note $v \longleftrightarrow w$ l'arête reliant les sommets v et w.
La représentation d'un graphe en OCaml est donné par le type suivant :

```
type graphe = { n : int ; (* les sommets sont 0,1,...,n-1 *)
adj : int list array (*  adj.(v) est la liste des voisins de v *)
};;
```

Les sommets sont les entiers $\{0, 1, ..., n-1\}$. Pour un sommet v du graphe `g`, la liste d'adjacence `g.adj.(v)` contient tous les voisins de v dans un ordre arbitraire et sans doublon.

I. Fonctions utiles sur les graphes

Q1| Écrire la fonction `ajout_sans_doublon` qui prend en argument une liste l (sans doublon) et un élément e, et renvoie la liste sans doublon constituée des éléments de la liste l et de l'élément e.

```
ajout_sans_doublon : 'a list -> 'a -> 'a list
```

Correction

Listing 1 – Fonction `ajout_sans_doublon`

```
let rec ajout_sans_doublon liste element =
 match liste with
 | [] -> [element]
 | a::reste when a = element -> liste
 | a::reste -> a::(ajout_sans_doublon reste element );;
```

Q2| Écrire la fonction `ajoute_arete` qui prend en argument un graphe g et deux entiers i et j (avec $0 \leq i \leq g.n-1$ et $0 \leq j \leq g.n-1$). La fonction ajoute l'arête $i \longleftrightarrow j$ si elle n'était pas déjà présente.

```
ajoute_arete : graphe -> int -> int -> unit
```

Correction

Listing 2 – Fonction `ajoute_arete`

```
let ajoute_arete g i j =
  g.adj.(i) <- ajout_sans_doublon g.adj.(i) j;
  g.adj.(j) <- ajout_sans_doublon g.adj.(j) i;;
```

Q3| Écrire la fonction `graphe_vide` qui prend en argument un entier n et renvoie un graphe à n sommets sans aucune arête.

```
graphe_vide : int -> graphe
```

Correction

Listing 3 – Fonction `graphe_vide`

```
let graphe_vide n =
{ n = n ; adj = Array.make n [] };;
```

Q4| Écrire la fonction `aretes` qui prend en argument un graphe, et renvoie la liste des couples (i, j), tel que $i \longleftrightarrow j$ soit une arête de g. (Le graphe étant non orienté, si (i, j) est dans la liste, alors (j, i) y est aussi.)

```
aretes : graphe -> (int * int) list
```

Correction

Listing 4 – Fonction `aretes`

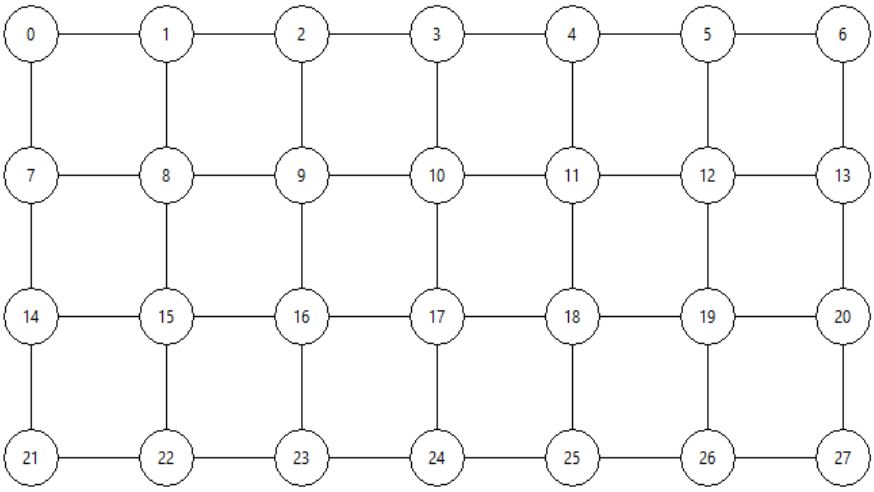
```
let aretes g =
  let rec aux indice liste =
      if indice = g.n then liste
      else
      let rec aux_2 lis liste_cumul = match lis with
        | [] -> aux (indice +1) liste_cumul
        | a::reste -> aux_2 reste ((indice,a)::liste_cumul)
    in aux_2 g.adj.(indice) liste
    in aux 0 [];;
```

II. Construction d'un labyrinthe

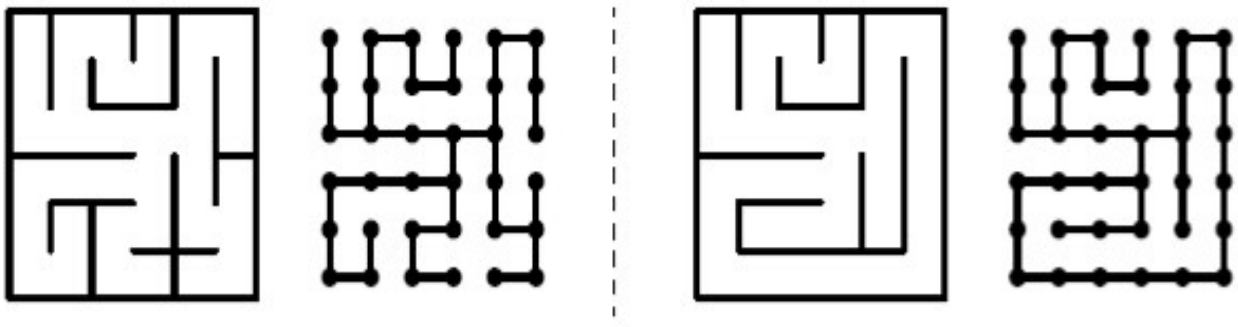
Pour deux entiers n et m, la grille de taille $n \times m$ est le graphe dont les sommets sont $\{0, 1, ..., n \times m - 1\}$ et dont les arêtes sont les paires de sommets qui sont :

- soit de la forme $v \longleftrightarrow (v + 1)$ pour $0 \leq v < nm$ tel que v modulo m est distinct de $m - 1$,
- soit de la forme $v \longleftrightarrow (v + m)$ pour $0 \leq v < (n - 1)m$.

Par exemple, la grille de taille 4×7 est le graphe suivant :



On se donne un graphe g connexe. On appelle **labyrinthe** sur g un sous-graphe connexe de g qui a le même ensemble de sommets que g.
Par exemple, on peut partir d'un graphe g dont les sommets sont les cases d'une grille rectangulaire et dont les arêtes correspondent aux cases adjacentes (nord, sud, est, ouest). Les arêtes du labyrinthe correspondent alors aux ouvertures permettant de passer d'une case à une autre. Les arêtes de g qui ne sont pas dans le labyrinthe correspondent aux murs.
Voici deux exemples :



On dit qu'un labyrinthe est parfait lorsqu'il existe un et un seul chemin entre toute paire de sommets. La labyrinthe de gauche dans l'exemple ci-dessus est parfait, alors que celui de droite ne l'est pas. Dans cette partie, on cherche à construire des labyrinthes parfaits.

Q5| Écrire la fonction `grille`, qui prend en argument deux entier n et m, et renvoie le graphe correspondant à la grille de taille $n \times m$.

```
grille : int -> int -> graphe
```

Correction

Listing 5 – Fonction `grille`

```
let grille ligne colonne =
  let g = graphe_vide (ligne*colonne) in
  for i = 0 to (ligne*colonne) -1 do
    if not (i mod colonne = colonne -1)
```

```
      then ajoute_arete g i (i+1)
  done ;
  for i = 0 to (ligne -1)*colonne -1 do
    ajoute_arete g i (i+colonne)
  done;
  g;;
```

Q6| Afin d'obtenir un mélange aléatoire d'une liste, on va écrire deux fonctions :
La première fonction, `i_eme_element`, prend en argument une liste l et un indice i (nécessairement compris entre 0 et la taille de la liste moins 1) et retourne le couple $(element, reste)$ qui représente l'élément en position i, et le reste de la liste.

```
i_eme_element : 'a list -> int -> 'a * 'a list
```

Écrire la fonction `melange_liste` qui reçoit en argument une liste et permute aléatoirement les éléments de la liste. (rappel : la fonction `Random.int` prend un argument n et renvoie un entier aléatoire entre 0 et $n-1$.)

```
melange_liste : 'a list -> 'a list
```

Correction

Listing 6 – Fonction `melange_liste`

```
let rec i_eme_element l i =
  if i=0 then (List.hd(l),List.tl(l))
  else
    let (e,l_reste) =i_eme_element (List.tl(l)) (i-1) in
    (e , List.hd(l)::l_reste);;

let melange_liste l =
  let n = List.length l in
  let rec aux n l l_result =
    if n = 1 then List.hd(l)::l_result
    else
      begin
        let i = Random.int n in
        let (element,l_reste) = i_eme_element l i in
        aux (n-1) (l_reste) (element:: l_result)
      end
    in
  aux n l [];;
```

Q7| Pour construire un labyrinthe aléatoire sur un graphe g, on peut utiliser l'algorithme suivant :
On effectue un parcours en profondeur (noté DFS par la suite) de graphe g. Lorsqu'un sommet v est traité, on permute aléatoirement la liste de ses voisins et on la parcourt. Pour chaque voisin w de v qui n'a pas encore été visité, on ajoute l'arête $v \longleftrightarrow w$ au labyrinthe et on traite w.
Écrire la fonction `labyrinthe1` qui prend en argument un graphe g connexe et renvoie un labyrinthe sur g construit avec un DFS aléatoire. On suggère d'écrire le parcours en profondeur comme une fonction récursive.

```
labyrinthe1 : graphe -> graphe
```

Correction

Listing 7 – Fonction `labyrinthe1`

```
let labyrinthe1 g =
  let laby = graphe_vide (g.n) in
  let rec parcours sommet  =
    let voisins = melange_liste g.adj.(sommet) in
      let rec aux liste =
        match liste with
          | [] ->()
          | s::reste ->
            begin
              if laby.adj.(s) = [] then
              begin
                ajoute_arete laby s sommet ;
                parcours s;
              end;
            aux reste;
            end
        in aux voisins
      in parcours 0;
    laby
    ;;
```

Q8| Montrer que le labyrinthe construit par la fonction `labyrinthe1` est parfait.
Indication : identifier un invariant du parcours en profondeur réalisé par `labyrinthe1`.

Correction

Pour la fonction `parcours`, l'invariant de boucle est le fait que le sous-graphe déjà parcouru a formé un sous-graphe de `laby` qui est parfait.
Lors d'un parcours d'un sommet, si les voisins ont déjà été parcourus, on n'ajoute pas de nouvelle arête (il n'y a donc pas de boucle). Le sous-graphe reste donc un labyrinthe parfait.

III. Classe disjointes

On s'intéresse maintenant à une strucure de données qui sera utilisée plus loin pour construire des labyrinthes par d'autre algorithmes. Cette structure de données représente une relation d'équivalence sur l'ensemble $\{0, 1, ..., n-1\}$. Pour cela on choisit un représentant arbitraire dans chaque classe d'équivalence et on se donne un tableau `lien` de taille n qui va permettre de retrouver ce représentant. Pour un représentant `r`, on a `lien.(r)` $=$ `r`. Pour tout autre élément `i` de la classe de `r`, on aboutit à `r` en suivant les valeurs données par le tableau `lieu` (c'est-à-dire que `lien.(i)` $=$ `r`, ou `lien.(lien.(i))` $=$ `r`, ou `lien.(lien.(lien.(i)))` $=$ `r`, etc). En particulier, `i` et `lien.(i)` sont toujours dans la même classe d'équivalence. Par exemple, le tableau

lien $=$

2	1	5	3	3	5	5

représente la relation d'équivalence dont les classes sont $\{0,2,5,6\}$, $\{1\}$ et $\{3,4\}$, pour lesquelles on a choisis les représentants 5, 1 et 3.

On se donne le type OCaml suivant pour cette structure de données :

```
type classes_disjointes = {
lien : int array ;
}
```

Q9| Écrire une fonction `cd_trouve` qui prend en arguments une relation d'équivalences sur $\{0,1,...,n-1\}$ et un entier i, avec $0 \leq i < n$, et renvoie le représentant de la classe de i.

```
cd_trouve : classes_disjointes -> int -> int
```

Correction

Listing 8 – Fonction `cd_trouve`

```
let cd_trouve classe i =
  let r = ref i in
  while classe.lien.(!r) <> !r do
    r := classe.lien.(!r)
  done;
  !r;;
```

On note que la complexité de `cd_trouve` dans le pire des cas est proportionnelle à la longueur du plus long chemin d'un élément à son représentant. Formellement, la longueur du chemin d'un élément `i` à son représentant `r` est définie comme le nombre d'éléments rencontrés en partant de `i` et en suivant les valeurs du tableau `lien` (en comptant `i` lui-même mais sans compter `r`).

On veut maintenant pouvoir modifier la relation d'équivalence, en offrant une opération `cd_union` permettant de fusionner la classes d'équivalence de deux éléments i et j donnés. L'idée consiste à cherche les représentants `ri` et `rj` des classes d'équivalence de i et j, avec la fonction `cd_trouve`, puis de faire pointer l'un vers l'autre en modifiant le tableau `lien`. Le choix du représentant `ri` ou `rj` pour la classe fusionnée n'est pas anodin, car il affecte le coût des opération `cd_trouve` ultérieures.

On introduit donc un second tableau `rang` de taille n. Pour chaque représentant r, la valeur de `rang.(r)` donne la longueur du plus long chemin d'un élément de la classe de `r` à `r`. Pour un élément i qui n'est pas un représentant, la valeur de `rang.(i)` n'est pas significative et ne sera jamais utilisée.Par exemple, les tableaux

lien = | 2 | 1 | 5 | 3 | 3 | 5 | 5 | et rang = | 0 | 0 | 1 | 1 | 0 | 2 | 0 |

représentent la relation d'équivalence dont les classes sont $\{0,2,5,6\}$, $\{1\}$ et $\{3,4\}$, mais les valeurs du tableau `rang` en position 0, 2, 4 , 6 ne sont pas significatives.

On modifie donc le type `classes_disjointes` de la façon suivante :

```
type classes_disjointes = {
lien : int array ;
rang : int array ;
}
```

Le code de la fonction `cd_trouve` reste inchangé.

Q10| Écrire une fonction `cd_union` qui prend en argument une relation d'équivalences sur $\{0, 1, ..., n-1\}$ et deux entiers i et j, avec $0 \leq i, j < n$, et fusionne les classes d'équivalence de i et j. Lorsque i et j sont déjà dans la même classe, cette fonction ne fait rien. Sinon, elle choisit pour nouveau représentant celui dont le rang est maximal.

```
cd_union : classes_disjointes -> int -> int -> unit
```

Correction

Listing 9 – Fonction `cd_union`

```
let cd_union cd i j =
    let ri = cd_trouve cd i in
    let rj = cd_trouve cd j in
    if ri <> rj
    then begin
         let rang_i = cd.rang.(ri)
         and rang_j = cd.rang.(rj) in
         if rang_i < rang_j
         then cd.lien.(ri) <- rj
         else if rang_i > rang_j
              then cd.lien.(rj) <- ri
              else (* egalite, on choisit de fusionner en i *)
                   (cd.lien.(rj) <- ri ;
                    cd.rang.(ri) <- cd.rang.(ri) + 1)
        end;;
```

Q11| On appelle rang d'une classe le rang de son représentant. Montrer que la fonction `cd_union` préserve les deux invariants suivants :

(a) toute classe de rang k possède au moins 2^k éléments ;

(b) dans une classe de rang k, la longueur du plus long chemin jusqu'au représentant est égale à k.

Correction

Pour la première propriété, si avant fusion, toutes les classes de rang k contiennent au moins 2^k éléments, alors lors de la fusion des classes (distinctes) de i et j, si elles ont des rangs différents, alors le rang de la classe obtenue est égal au rang d'une des classes de départ alors que son cardinal a augmenté. Ainsi, si la classe obtenue est de rang k, elle contient au moins 2^k éléments, ce qui était déjà le cas au départ.

Sinon, si les classes sont de même rang k, la classe obtenue est de rang $k+1$ et consiste en la fusion de 2 classes ayant chacune au moins 2^k éléments. La classe obtenue a donc bien au moins 2^{k+1} éléments.

Concernant la seconde propriété, à nouveau, si l'on fusionne des classes de rangs différents, le résultat est direct. Par contre si l'on fusionne deux classes distinctes de représentants i et j et de même rang k, en supposant que l'on lie j à i, alors par hypothèse d'induction :

- tous les chemins des classes initiales vers leurs représentant est de longueur au plus k ;
- il existe un sommet s de classe de j dont le chemin jusqu'à j est de longueur k.

Après fusion, tous les chemins des classes initiales vers i (le représentant de la classe obtenue) sont de longueur au plus $k+1$, et le chemin de s à i est bien de longueur $k+1$.

On se donne une fonction :

```
cd_init : int -> classes_disjointes
```

qui prend en argument un entier $n \geq 0$ et renvoie une relation d'équivalence sur $\{0, 1, ..., n-1\}$ dont toutes les classes sont des singletons. Pour tout i, on a donc `lien.(i)` $=$ `i` et `rang.(i)` $=$ `0`.

Q12| Déduire de la question 7 que, pour une relation d'équivalence sur $\{0, 1, ..., n-1\}$ construite à partir de `cd_init`, la complexité de `cd_trouve` est en $O(\log n)$.

IV Construction à l'aide des classes disjointes

On utilise maintenant la structure de classes disjointes pour construire un labyrinthe parfait `h` à partir d'un graphe `g` connexe à n sommets.
Le labyrinthe procède ainsi ;

-1- On construit une relation d'équivalence sur $\{0, 1, ..., n-1\}$ avec `cd_init` ;

-2- On construit la liste de toutes les arêtes du graphe g, puis on la mélange ;

-3- On parcourt la liste mélangée et, pour chaque arête $v \longleftrightarrow w$, si v et w ne sont pas dans la même classe d'équivalence, on ajoute l'arête $v \longleftrightarrow w$ au labyrinthe `h` et on fusionne les classes de v et w avec `cd_union` ;

Q13| Écrire une fonction `labyrinthe2` qui prend en argument un graphe `g` connexe et renvoie un labyrinthe sur `g` construit avec cet algorithme.

```
labyrinthe2 : graphe -> graphe
```

Correction

Listing 10 – Fonction `labyrinthe2`

```
let labyrinthe2 g =
  let laby = graphe_vide (g.n) in
  let cd = cd_init g.n in
  let liste_arete = aretes g in
  let liste_melangee = melange_liste liste_arete in
  let rec aux l = match l with
    | [] -> laby
    | (v,w)::reste ->
        if cd_trouve cd v <>  cd_trouve cd w then
          begin
```

```
            ajoute_arete laby v w ;
            cd_union cd v w
          end ; aux reste
  in aux liste_melangee;;
```

Q14| Montrer que l'étape 3 de l'algorithme maintient l'invariant suivant : pour toute classe d'équivalence X, le sous graphe `h` induit par X est un labyrinthe parfait du sous-graphe de `g` induit pas X. En déduire que le labyrinthe construit par la fonction `labyrinthe2` est parfait. Correction

Il faut uniquement vérifier que l'invariant est préservé lorsque l'on modifie effectivement le graphe et la relation d'équivalence, i.e. lorsque les deux sommets ne sont pas de la même classe d'équivalence de la relation X.

Or, dans ce cas, par hypothèse d'induction, les deux sommets initiaux s_1 et s_2 appartiennent à deux sous-graphes h_1 et h_2 qui sont des labyrinthes parfaits des sous-graphes g_1 et g_2 (ces sous-graphes étant respectivement ceux de h et g induits par les classes d'équivalence de s_1 et s_2). L'ajout d'une arête entre s_1 et s_2 conduit à un labyrinthe parfait h_+ du sous-graphe g_+ de g induit par la nouvelle relation d'équivalence. En effet,

- le sous-graphe h_+ obtenu est connexe, étant formé par deux graphes connexes que l'on a reliés par une arête,
- il existe un unique chemin reliant toute paire de sommets de h_+ (on rappelle que l'on ne considère que les chemins *simples*). En effet, deux sommets de h_1 et de h_2 sont reliés dans h_+ par un unique chemin, et pour tous sommets u de h_1 et v de h_2, il existe un unique chemin de h_+ allant de u à v, constitué de l'unique chemin de h_1 de u à s_1, puis de la nouvelle arête de s_1 à s_2, puis de l'unique chemin de h_2 de s_2 à v.

Ainsi, à la fin de l'exécution de l'algorithme, la relation d'équivalence X ne contient plus qu'une unique classe, et donc h est un labyrinthe parfait de g.

V Résoudre un labyrinthe

On se pose maintenant la question de résoudre un labyrinthe, c'est-à-dire de chercher un chemin d'un sommet source `src` donné à un sommet destination `dst` donné. On suppose toujours que le labyrinthe est connexe. En revanche, on ne suppose pas le labyrinthe parfait, c'est-à-dire qu'il peut exister plusieurs chemins de la source à la destination.

Files à deux bouts.

On commence par programmer une structure de données de file à deux bouts . Une telle file est représentée par le type suivant :

```
type file = {
  contenu : int array ;
  mutable debut : int ;
  mutable taille : int
  }
```

La file contient `taille` éléments, stockés dans le tableau `contenu` à partir de l'indice `debut` et dans le sens des indices croissants. Le tableau a un taille `cap`, qui est la capacité maximale

de la file. Si `debut`+`taille` $\leq$ `cap`, les éléments de la file sont stockés de façon consécutive dans le tableau `contenu`, entre `debut` et `debut`+`taille`-1. Sinon, le stockage des éléments se poursuit au début du tableau `contenu`, jusqu'à la position `debut` $+$ `taille` $-1-$ `cap`. Dans tous les cas, on a les inégalités suivantes : $0 \leq$ `debut` $<$ `cap` et $0 \leq$ `taille` $\leq$ `cap`.

Voici deux exemples de files de capacité 7 contenant 4 éléments (notés X) :

		X	X	X	X		`cap` $= 7$, `debut` $= 2$,`taille` $= 4$

X	X				X	X	`cap` $= 7$, `debut` $= 5$,`taille` $= 4$

La file est construite par une fonction

```
file_vide : int -> file
```

qui prend la capacité en argument, puis manipulée avec les quatre opérations suivantes :

```
ajoute_debut :file -> int -> unit
retire_file : file -> int
ajoute_fin : file -> int -> unit
retire_fin : file -> int
```

L'appel de `ajoute_debut f e` ajoute l'élément `e` au début de la file `f` (sans déplacer ou modifier les éléments déjà présents), en supposant que la file n'est pas pleine. L'appel de `retire_debut f` retire et renvoie l'élément du début de la file `f` (sans déplacer ou modifier les autres éléments), en supposant que la file n'est pas vide. Les deux autres opérations `ajoute_fin` et `retire_fin` opèrent de manière similaire à la fin de la file.

Q15| Donner le code de la fonction `ajoute_debut`. Correction

Listing 11 – Fonction `ajoute_debut`

```
let ajoute_debut f element =
  let cap = Array.length f.contenu in
    if f.taille = cap then failwith "␣file␣pleine" else
      begin
        let position =(( f.debut-1)+cap) mod cap in
        f.debut <- position ;
        f.contenu.(f.debut) <- element ;
        f.taille <- f.taille +1
      end;;
```

Q16| Donner le code de la fonction `retire_debut`. Correction

Listing 12 – Fonction `retire_debut`

```
let retire_debut f =
  if f.taille = 0 then failwith "␣file␣vide" else
    begin
      let cap = Array.length f.contenu in
      let element = f.contenu.(f.debut) in
      f.debut <- (f.debut +1) mod cap ;
      f.taille <- f.taille -1 ;
      element;
    end ;;
```

Parcours en largeur

Pour trouver la longueur d'un plus court chemin dans un labyrinthe à n sommets, on effectue un parcours en largeur, en partant du sommet source `src` et en s'arrêtant quand on trouve de sommet destination `dst`. On procède ainsi :

(a) créer un tableau `distance` de taille n, initialisé avec la valeur -1 dans toutes les cases ;

(b) créer une file à deux bouts `f` de capacité n ;

(c) ajouter le sommet source `src` dans `f` et initialiser `distance.(src)` à 0 ;

(d) tant que la file `f` n'est pas vide :

i. retirer l'élément v au début de `f` ;

ii. si $v =$ `dst` alors on a terminé et la réponde est `distance.(dst)` ;

iii. sinon, pour chaque voisin w de v pour lequel `distance.(w)` vaut -1 ; ajouter w à la fin de `f` et donner à `distance.(w)` la valeur `distance.(v)` $+ 1$.

Q17| Montrer que cet algorithme renvoie bien la longueur d'un plus court chemin de la source à la destination.

Correction

On note $d(s)$ la longueur minimale d'un chemin de `dst` vers s, sa distance.

On montre qu'à chaque moment

(a) la file d'attente contient des sommets s_1, s_2, …, s_p (dans l'ordre de début à la fin) tels que $d(s_1) \leq d(s_2) \leq \cdots \leq d(s_p) \leq d(s_1) + 1$,

(b) tous les sommets de distance $d(s_1)$ ou moins ont été insérés,

(c) toutes les valeurs de `distance.(i)` sont les distances des sommets correspondants.

Cette propriété est vrai lors de l'initialisation (étape 3) car la file ne contient que `src` qui est à distance 0 et c'est le seul sommet de distance 0.

On suppose que cette propriété est vraie avant un passage ; on retire le sommet s_1 et on ajoute les voisins de s_1 non encore marqués, t_1, …, t_q.

(a) On accède à ces sommets depuis s_1 donc leur distance est au plus $d(s_1)+1$, comme les sommets à distance $d(s_1)$ ont été déjà ajoutés, leur distance est exactement $d(s_1)+1$. On a donc $d(s_2) \leq \cdots d(s_p) \leq d(s_1) + 1 = d(t_1) \cdots = d(t_q) = d(s_1) + 1 \leq d(s_2) + 1$.

(b) Si on a $d(s_2) = d(s_1)$ alors tous les sommets de distance $d(s_2)$ au plus ont été insérés. Si on a $d(s_2) = d(s_1) + 1$ alors tous les sommets de distance $d(s_1)$ ont étés traités donc tous les sommets de distance $d(s_1) + 1 = d(s_2)$ ont été insérés.

(c) Les valeurs données dans le tableau `distance` sont `distance.(s1)` $+$ `1` qui est égale, selon l'hypothèse de récurrence à $d(s_1) + 1$, ce qui est bien la distance des t_j.

Ainsi la propriétés reste vraie.

Comme le graphe est connexe, le sommet `dst` est atteint donc la valeur de `distance.(dest)` est bien la valeur minimale des longueur des chemins de `src` à `dst`.

Q18| Écrire une fonction `resolution` qui prend en argument un graphe `g` et deux entiers `src` et `dst`, et renvoie la longueur d'un plus court chemin de la source à la destination.

```
resolution : graphe -> int -> int -> int
```

Correction

Listing 13 – Fonction `resolution`

```
let resolution g src dst =
  let distance = Array.make g.n (-1) in
  let f = file_vide g.n in
  ajoute_debut f src ;
  distance.(src) <- 0 ;
  try
  while f.taille > 0 do
    let v = retire_debut f in
    if v = dst then raise Trouve
      else
        begin
          let rec aux l = match l with
            | [] -> ()
            | w::reste ->
                begin
                  if distance.(w) = -1 then
                      ( ajoute_fin f w ;
                      distance.(w) <- distance.(v)+1 ;
                       aux reste)
                  else aux reste
                end
          in aux g.adj.(v)
        end;
    done;
    -100
  with  _-> distance.(dst);;
```