TD 2

Parcours

Hasley William

1 Parcours en Profondeur

1.1 Premiers Exemples

Ordonnancement des Sommets

Le choix du (des) premier(s) sommet(s) à considérer est en théorie laissé libre. Nous ordonnerons par convention les sommets, et choisirons en priorité les sommets d'étiquette minimale n'ayant pas été traité, lorsque le choix d'un sommet de base n'est pas précisé.

Pour les graphes proposés :

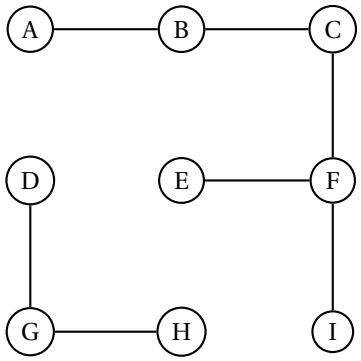
➢ (Ordre Lexicographique Usuel, a) Le parcours en profondeur donne (avec répétition des sommets déjà visités pour une meilleure compréhension) :

$$A \to B \to C \to F \to E \to F \to I \quad D \to G \to H$$

Ainsi, nous pouvons donner le tableau de Dates de Découverte et Fin de Traitement suivant :

	A	B	C	D	E	F	G	H	I
Date de Découverte	1	2	3	13	5	4	14	15	7
Date de Fin de Traitement	12	11	10	18	6	9	17	16	8

De plus, la forêt associée à ce parcours donne :



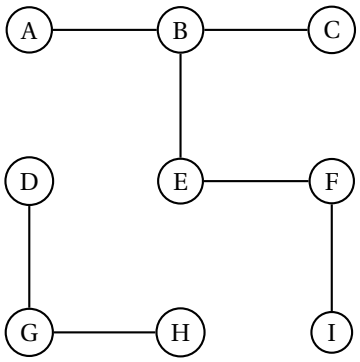
➢ (Ordre Lexicographique Inverse, a) Le parcours en profondeur donne :

$$I \to F \to E \to B \to C \to B \to A \quad H \to G \to D$$

Ainsi, nous pouvons donner le tableau de Dates de Découverte et Fin de Traitement suivant :

	A	B	C	D	E	F	G	H	I
Date de Découverte	7	4	5	15	3	2	14	13	1
Date de Fin de Traitement	8	9	6	16	10	11	17	18	12

De plus, la forêt associée à ce parcours donne :



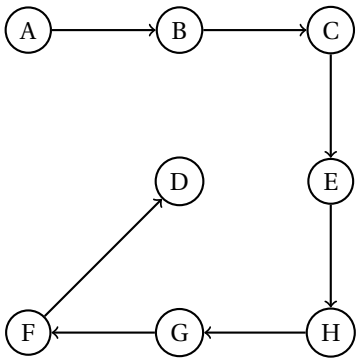
➢ (Ordre Lexicographique Usuel, b) Le parcours en profondeur donne :

$$A \to B \to C \to E \to H \to G \to F \to D$$

Ainsi, nous pouvons donner le tableau de Dates de Découverte et Fin de Traitement suivant :

	A	B	C	D	E	F	G	H
Date de Découverte	1	2	3	8	4	7	6	5
Date de Fin de Traitement	16	15	14	9	13	10	11	12

De plus, la forêt associée à ce parcours donne :



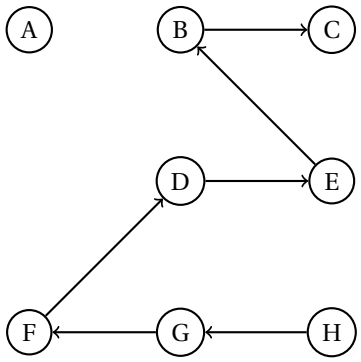
➢ (Ordre Lexicographique Inverse, b) Le parcours en profondeur donne :

$$H \to G \to F \to D \to E \to B \to C \quad A$$

Ainsi, nous pouvons donner le tableau de Dates de Découverte et Fin de Traitement suivant :

	A	B	C	D	E	F	G	H
Date de Découverte	15	6	7	4	5	3	2	1
Date de Fin de Traitement	16	9	8	11	10	12	13	14

De plus, la forêt associée à ce parcours donne :



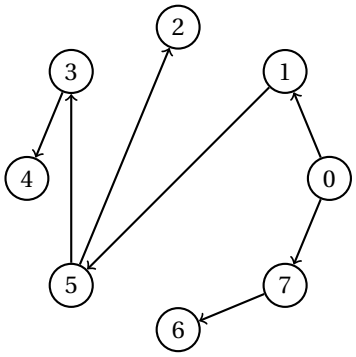
➢ (Ordre Lexicographique Usuel, c) Le parcours en profondeur donne :

$$0 \to 1 \to 5 \to 2 \to 5 \to 3 \to 4 \to 3 \to 5 \to 1 \to 0 \to 7 \to 6$$

Ainsi, nous pouvons donner le tableau de Dates de Découverte et Fin de Traitement suivant :

	0	1	2	3	4	5	6	7
Date de Découverte	1	2	4	6	7	3	13	12
Date de Fin de Traitement	16	11	5	9	8	10	14	15

De plus, la forêt associée à ce parcours donne :



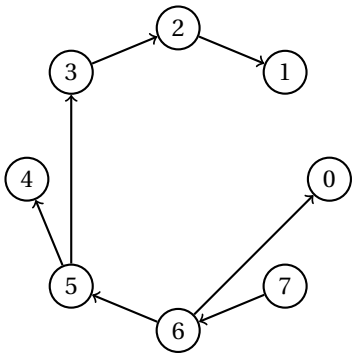
➢ (Ordre Lexicographique Inverse, c) Le parcours en profondeur donne :

$$7 \to 6 \to 5 \to 4 \to 5 \to 3 \to 2 \to 1 \to 2 \to 3 \to 5 \to 6 \to 0$$

Ainsi, nous pouvons donner le tableau de Dates de Découverte et Fin de Traitement suivant :

	0	1	2	3	4	5	6	7
Date de Découverte	13	7	6	6	4	3	2	1
Date de Fin de Traitement	14	10	11	9	12	10	15	16

De plus, la forêt associée à ce parcours donne :



Lemme Absence d'arête transverse

Soit $G = (S, A)$, un Graphe non orienté, dont on note $\mathscr{F} = (S, A')$ la forêt de l'un de ses parcours en profondeur.

Alors toute arête $a \in A$ telle que $a \notin A'$, a relie un sommet à son prédécesseur dans l'arborescence de $\mathscr{F}$.

DÉMONSTRATION.

Soit $a = \{u, v\} \in A \setminus A'$. En particulier, lors de la considération de u dans le parcours en profondeur, l'arête v n'est jamais empruntée. Or, ceci ne se produit que si v est déjà vu, ou bien est en train d'être traité.
Or, si v est déjà vu, $\{v, u\} \in A$, donc v a déjà appelé le parcours en profondeur sur u, ce qui est absurde car ce n'est pas le cas. Ainsi, v est en train d'être traité lors de l'appel sur u. Ainsi, $\text{dec}(v) < \text{dec}(u) < \text{dft}(u) < \text{dft}(v)$, et ceci caractérise le fait que u est un fils de v dans l'arborescence de $\mathscr{F}$. Il est donc impossible d'avoir une arête dite transverse dans une telle arborescence $\mathscr{F}$.

1.2 Questions

Question 1. Nous considérons dans cette question avoir un graphe G, donné sous forme de liste d'adjacence (c.f le code ci-dessous pour l'implémentation)

```
type sommet_t = int
type list_adj_t = sommet_t list
type graphe_t = list_adj_t array
type cc_t = int

let graphe_composante_connexe (g : graphe_t) : cc_t array =
  let n = Array.length g in
  let cc_arr = Array.make n (-1) in

  let rec parcours_rec sommet_act cc_act =
    if(cc_arr.(sommet_act) >= 0) then ()
    else begin
      cc_arr.(sommet_act) <- cc_act;
      iter_voisins g.(sommet_act) cc_act
    end

  and iter_voisins voisins_list cc_act = match voisins_list with
    |[] -> ()
    |h::t -> if(cc_arr.(h) = (-1)) then parcours_rec h cc_act; iter_voisins t cc_act

  in let rec build_cc_array cur_base_sommet cur_cc =
    if(cur_base_sommet >=n) then cc_arr
    else if (cc_arr.(cur_base_sommet) = (-1)) then begin
      parcours_rec cur_base_sommet cur_cc;
      build_cc_array (cur_base_sommet + 1) (cur_cc + 1)
     end
    else build_cc_array (cur_base_sommet + 1) cur_cc
  in build_cc_array 0 0
```

Question 2. Idem, nous remplaçons le tableau des composantes connexes par un tableau "vu".

```
let graphe_arborescence_profondeur (g : graphe_t) : sommet_t array =
  let n = Array.length g in
  let vu = Array.make n false in
  let peres = Array.make n (-1) in

  let rec parcours_rec sommet_act caller =
    if(vu.(sommet_act)) then ()
    else begin
      vu.(sommet_act) <- true;
      peres.(sommet_act) <- caller;
      iter_voisins g.(sommet_act) sommet_act
    end

  and iter_voisins voisins_list new_caller = match voisins_list with
    |[] -> ()
    |h::t -> if(not vu.(h)) then parcours_rec h new_caller; iter_voisins t new_caller

  in let rec build_peres_array cur_base_sommet =
    if(cur_base_sommet >=n) then peres
    else if (not vu.(cur_base_sommet)) then begin
      parcours_rec cur_base_sommet (-1);
      build_peres_array (cur_base_sommet + 1) end
    else build_peres_array (cur_base_sommet + 1)
  in build_peres_array 0
```

Question 3. L'algorithme de Tri Topologique est une variante du parcours en profondeur : Il est possible de donner un tri topologique en ordonnant les sommets par Date de fin de Traitement décroissante dans un parcours en profondeur, ce qui est l'algorithme ici implémenté :

```
let graphe_tri_topologique (g : graphe_t) : sommet_t list =
  let n = Array.length g in
  let vu = Array.make n false in

  let rec parcours_rec sommet_act ordre_topo_liste  =
      vu.(sommet_act) <- true;
      sommet_act::(iter_voisins g.(sommet_act) ordre_topo_liste)

  and iter_voisins voisins_list ordre_actuel = match voisins_list with
    |[] -> ordre_actuel
    |h::t -> if(vu.(h)) then iter_voisins t ordre_actuel
    else iter_voisins t (parcours_rec h ordre_actuel)

  in let rec build_ordre_topo cur_base_sommet buffer_list =
    if(cur_base_sommet >=n) then buffer_list
    else if (not vu.(cur_base_sommet)) then
      build_ordre_topo (cur_base_sommet + 1) (parcours_rec cur_base_sommet buffer_list)
    else build_ordre_topo (cur_base_sommet + 1) buffer_list
  in build_ordre_topo 0 []
```

2 Parcours Génériques

Question 1.

a) En définissant l'ordre du parcours comme l'ordre du marquage des nœuds, nous remarquons que l'ordre n'est pas modifié en supposant que l'opération "prendre un voisin" est déterministe et constante pour chaque sommet :

Montrons ceci par récurrence : Soit $G = (S, A)$, un graphe non orienté, ainsi que $s_0 \in S$ un sommet de base. Posons $\forall s \in S,\ \ d(s, s_0) = \min\{|c| \mid c \text{ chemin de } s \text{ à } s_0\}$, la profondeur du sommet s par rapport à s_0. Montrons donc que pour toute profondeur $n \in \mathbb{N}$, le parcours produit par WFS_1 est identique à celui de WFS_2 :

Remarquons premièrement que la structure de file impose que l'ordre donné par ces deux algorithmes est un ordre sur les sommets tel que la profondeur de ces derniers soit croissante. (Ceci est immédiat : s_0 enfile tout sommet de profondeur 1, puis ces sommets enfileront ceux de profondeur 2 etc...)

— Le résultat est immédiat pour une profondeur de zéro, le premier sommet enfilé et marqué est bien s_0 dans les deux cas.

— Soit donc $n \in \mathbb{N}$. Supposons que l'ordre donné est identique pour tout sommet de profondeur $\leq n$ (i.e l'ordre partiel est identique). Montrons donc que l'ordonnancement donné est inchangé pour les sommets à profondeur $n+1$ (dont on suppose l'existence) :

L'ordre partiel étant identique entre ces deux algorithmes, nous savons que la file de WFS_1 contient les sommets de profondeur $n+1$, entrelacés de sommets de profondeur $\leq n$. Or, l'ordre de ces sommets de profondeur $n+1$ est le même que celui de WFS_2, car par hypothèse de récurrence, les sommets de profondeur n sont marqués dans le même ordre, et là où WFS_1 enfile les sommets de profondeur $n+1$ pour les marquer dans le même ordre que l'enfilement, WFS_2 les marque dès l'enfilement. Ainsi, en marquant les sommets de profondeur n dans le même ordre, nous garantissons le marquage des sommets $n+1$ dans un même ordre, ceci grâce à la structure de file, qui nous permet d'ignorer les sommets déjà visités par WFS_1, qui pourraient modifier l'ordre si la structure utilisée était une structure de pile par exemple.

Il vient dès lors par principe de récurrence que l'ordre du parcours est identique entre ces deux algorithmes de parcours.

b) Nous supposerons que l'accès aux voisins de tout sommet de G se fait en $\mathcal{O}(1)$ (via une liste d'adjacence par exemple). De plus, les opérations sur la file sont également supposées en $\mathcal{O}(1)$ (push et pop).

Le premier algorithme marque chaque nœud individuellement, et enfile chaque voisin, même lorsque ces derniers sont déjà visités. Ainsi, un sommet s est enfilé exactement $\deg(s)$ fois : Une fois par marquage de ses voisins. Nous effectuons donc $\sum_{s\in S} \deg(s) = 2|A|$ tours de boucle pour vider le sac. Or, pour chaque sommet s, seul un défilage s'effectue avec $\deg(s)+1$ opérations. Le reste des défilages s'effectue en $\mathcal{O}(1)$.

D'où une complexité en $(2|A| - |S|) \times \mathcal{O}(1) + \sum_{s\in S}(\deg(s)+1) = \mathcal{O}(|A|+|S|)$.

Le deuxième algorithme n'enfile qu'une unique fois ses voisins (lors de leur marquage). Ainsi, nous effectuons exactement $|S|$ tours de boucle pour vider le sac. Or, lorsqu'un sommet est défilé, nous observons chaque voisin, et opérons en $\mathcal{O}(1)$ (vérifier si t est marqué, l'enfiler et le marquer sont des opérations en $\mathcal{O}(1)$). D'où une complexité en $\sum_{s\in S}(1+\deg(s)) = \mathcal{O}(|S|+|A|)$.

Dès lors, la complexité temporelle est inchangée entre ces deux algorithmes.

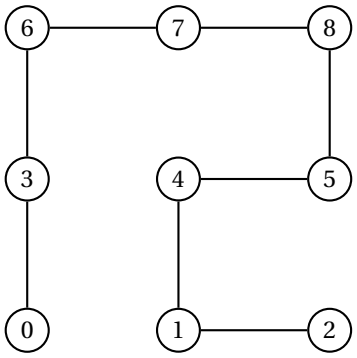
c) En revanche, nous avons remarqué que la complexité spatiale de ces deux algorithmes change drastiquement : WFS_1 est de complexité spatiale $\mathcal{O}(|S|^2)$ (via un graphe complet par exemple, chaque sommet nous fait enfiler $n-1$ sommets). Alors que WFS_2 est de complexité spatiale $\mathcal{O}(|S|)$, car chaque sommet n'est marqué et enfilé qu'une unique fois.

Question 2. Nous prendrons ici $s_0 = 0$.

➢ (Application de WFS_1) L'ordre obtenu est :

$$[0,3,6,7,8,5,4,1,2]$$

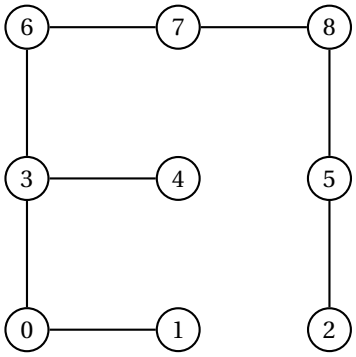
Avec pour arborescence (les arêtes "marquant" un sommet) :



➢ (Application de WFS_2) L'ordre obtenu est :

$$[0,1,3,4,6,7,8,5,2]$$

Avec pour arborescence :



3 Sommets Coupants

Question 1. Les sommets coupants de ce graphe sont les sommets $2,4,7,10$.

Question 2. Soit $G=(S,A)$ et $s \in S$. Montrons qu'un parcours en profondeur permet de déterminer si s est un sommet coupant : Supposons récupérer l'arborescence du parcours sous forme des tableaux dec (date de découverte) et dft (date de fin de traitement). Alors s est coupant si et seulement si le sommet $v \in S$, fils de s tel que $\text{dec}(v)=1$ (donc le premier fils de s rencontré) dans l'arborescence, est tel que $\text{dft}(v)+1 < \text{dft}(s)$ (autrement dit, après avoir traité entièrement v et sa descendance, s possède d'autres fils non découverts. Ceci revient à dire qu'il existe deux fils f et f' de s tels que $\text{dec}(f') > \text{dft}(f)$.

En effet, supposons que v existe (i.e G possède plus de deux sommets connectés). Alors si v possède cette propriété, il existe $f \in V$ tel que $\text{dft}(v) < \text{dec}(f) < \text{dft}(s)$, car s appelle le parcours en profondeur sur v, et si s n'a pas de tels descendants f, le parcours en profondeur se termine dès que v est entièrement traité, donc nous aurions $\text{dft}(v) = \text{dft}(s)-1$: Ceci est absurde par hypothèse. Or, puisque f n'est pas découvert avant d'avoir entièrement traité v, il n'existe aucun chemin $c : v \rightsquigarrow f$ sans passer par s, sans quoi f aurait été découvert par le parcours en profondeur sur v. Ainsi, $G \setminus \{s\}$ n'est pas connexe. La réciproque est de même.

Or, l'algorithme que nous décrivons (parcours en profondeur, puis vérification de la date de fin de traitement du sommet avec $\text{dec}(v)=1$) est bien linéaire en la taille du graphe. (de complexité $\mathcal{O}(|S|+|A|)$)

Question 3. Il vient alors un algorithme naïf : Nous appliquons l'algorithme précédent sur chaque sommet du graphe, ceci donne un algorithme en $\mathcal{O}(|S|^2+|S||A|)$

Question 4.

a) Cette formulation est équivalente à la propriété exhibée à la question deux : Si s est un sommet coupant de G, alors $G \setminus \{s\}$ n'est plus connexe. Ainsi, il existe deux sommets $a, b \in S$, tels que $\{s, a\} \in A$, $\{s, b\} \in A$ et tels qu'il n'existe pas de chemin $c : a \rightsquigarrow b$ ne passant pas par s (sinon $G \setminus \{s\}$ est connexe). Or, ceci correspond bien à deux fils de s dans l'arborescence du parcours : La relation de parenté entre deux nœuds u et v s'écrit : u est un ancêtre de v si et seulement si $\text{dec}(u) < \text{dec}(v) < \text{dft}(v) < \text{dft}(u)$. Ainsi, la spécification $\text{dft}(a) < \text{dec}(b)$ nous indique que a et b sont frères dans l'arborescence du parcours. Ainsi, s est bien de degré au moins 2 dans l'arborescence du parcours.

Réciproquement, si s est de degré au moins 2 dans l'arborescence du parcours : Du fait qu'il n'existe pas d'arête transverse (c.f Partie 1), $G \setminus \{s\}$ n'est plus connexe, car sinon il existerait une arête transverse, ce qui n'est pas le cas. Ainsi, s est bien coupant.

b) Soit $t \in S \setminus \{s\}$, un sommet coupant. Alors $G \setminus \{t\}$ n'est plus connexe. Alors, en particulier, t possède au moins deux voisins $a, b \in S$ tels qu'il n'existe aucun chemin $c : a \rightsquigarrow b$ qui n'emprunte pas t. Dès lors, dans l'arborescence du parcours en profondeur à partir de s, t possède bien au moins un fils. Prenons b comme étant ce fils sans perte de généralité, et a comme étant parent de t. Si b, ou l'un de ses descendants était voisin d'un ancêtre strict de t dans G, alors il existerait un chemin $c : a \rightsquigarrow b$ en empruntant l'arborescence, puis l'arête reliant un fils de b à un ancêtre strict de t. Ceci est absurde par choix de a et b, il vient alors que b n'est voisin d'aucun ancêtre strict dans l'arborescence du parcours.

Réciproquement, posons a ancêtre strict de t et b, tel qu'aucun de ses descendants n'est relié à aucun ancêtre strict de t dans G. Il vient alors que $G \setminus \{t\}$ n'est plus connexe, car sinon, nous pourrions relier a à b dans $G \setminus \{t\}$, donc il existerait dans $G \setminus \{t\}$ un chemin entre a et b. Or, ce chemin commence à un ancêtre strict de t et termine à un descendant de b (qui est b lui-même). Nous sommes donc capable d'exhiber une arête entre la descendance de b et les ancêtres strict de t. Ce qui est absurde par construction : t est bien un sommet coupant.

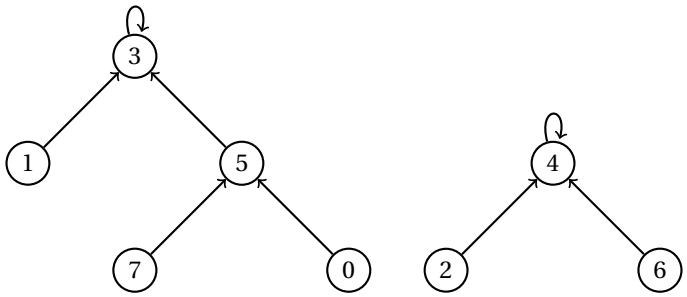
c) Proposons l'algorithme suivant :
Posons $\mathscr{T}$, l'arborescence du parcours en profondeur enraciné en s. (T s'obtient en temps linéaire en la taille du graphe G). On suppose donnée une structure d'arbre, dont les nœuds ont pour étiquette le sommet représenté, ainsi que les tableaux dec et dft. Procédons de manière récursive :

Notons d_{min}, l'application calculant la date de découverte minimale des voisins (dans G) de tout sommet u.

Une feuille n'est jamais coupante, nous renvoyons donc $([], d_{min}(f))$ si le sommet considéré est une feuille. Sinon, pour tout fils d'un sommet v dans l'arborescence, récupérons le couple (l, d) calculé précédemment. Si $d < \text{dec}(t)$, nous renvoyons $((v :: l), \min(d, d_{min}(v))$, sinon nous renvoyons $(l, \min(d, d_{min}(v))$ (nous continuons le processus sur les autres fils de v...). D'après ce qui précède, cet algorithme détermine récursivement tout sommet coupant de $\mathscr{T}$. De plus, cet algorithme évalue chaque nœud, et nous garantissons que chaque sommet fait tourner d_{min} avec une complexité $\deg(v)$, d'où une complexité en $\mathscr{O}(|S| + |A|)$

4 Union Find

Rappelons la figure de la première page du cours :



Question 1. Nous exhibons la suite d'opérations suivante :

1. union(3,1)
2. union(5,7)
3. union(5,1)
4. union(3,5)
5. union(4,2)
6. union(4,6)

Question 2. Nous proposons d'ajouter un champ "num_class", initialisé à n, et décrémenté lors d'une union concluante :

```
type uf = {n : int; num_class : int; repr : int array; height : int array}
let uf_make size = {n = size; num_class = size; repr = Array.init size (fun i -> i); height = Array.make
    size 0}

let uf_union uf a b =
  let ra = uf_find uf a in
  let rb = uf_find uf b in
  if(ra = rb) then false
  else if(uf.height.(a) < uf.height.(b)) then uf.repr.(a) <- b
    else if(uf.height.(b) < uf.height.(a)) then uf.repr.(b) <- a
    else begin uf.repr.(a) <- b; uf.height.(b) <- uf.height.(b) + 1 end
  uf.num_class <- uf.num_class - 1; true
```

Question 3. Il suffit, lors d'un parcours en profondeur, d'essayer de joindre les classes du sommet actuel et de chaque voisin distinct du caller. Si l'union rate, il vient que le fils est déjà dans la classe d'un des parents du sommet actuel, donc nous exhibons un cycle.

Question 4.

1.
```
type orientation_t = H | V
type coord_t = int * int
type porte_t = coord_t * orientation_t

let coord_identity (c : coord_t) : coord_t = c
let coord_y (c : coord_t) : int = match c with |(y, x) -> y
let coord_x (c : coord_t) : int = match c with |(y, x) -> x
let porte_identity (p : porte_t) : porte_t = p
let porte_coord (p : porte_t) : coord_t = match p with |(c, o) -> c
let porte_orientation (p : porte_t) : orientation_t = match p with |(c, o) -> o
```

```
let gen_laby_porte_table (n : int) : porte_t array =
  let tab = Array.make (2 * (n-1) * (n-1)) (0, 0, H) in

  let rec build_porte_rec cur_y cur_x cur_orient =
  if(cur_y >= n) then tab
  else if(cur_x >= n) then build_porte_rec (cur_y + 1) 0 cur_orient
  else begin match cur_orient with
    |H -> tab.(y * (n-1) + x) <- ((y, x), H); build_porte_rec cur_y cur_x V
    |V -> tab.(y * n + x + ((n-1) * (n-1))) <- ((y, x), V); build_porte_rec cur_y (cur_x + 1) H
    end
  in build_porte_rec 0 0 H
```

2. La bijection est assez usuelle : posons $\varphi : (x, y) \mapsto y * n + x$. Alors par unicité de la division euclidienne, nous déduisons le caractère bijectif de φ

3.
```
let build_labyrinth (n : int) : (bool array * bool array) =
  let porte_tab = gen_laby_porte_table n in
  let vert_arr = Array.make_matrix n n true in
  let hor_arr = Array.make_matrix n n true in
  let uf = uf_make (n * n) in

  let rec parcours_portes cur_coord =
    if(cur_coord >= 2 * (n - 1) * (n - 1)) then (vert_arr, hor_arr)
    else begin match porte_tab.(cur_coord) with
      |((y, x), or) -> begin match or with
        |H -> if(uf_union (phi y x) (phi y (x+1))) then hor_arr.(y).(x) <- false
        |V -> if(uf_union (phi y x) (phi (y+1) x)) then vert_arr.(y).(x) <- false
        end; parcours_portes (cur_coord + 1)
  in parcours_portes 0
```

4. Ce labyrinthe est bien parfait : S'il existe deux sommets a et b tels qu'il existe deux chemins distincts c_1 et c_2 (supposés élémentaires, et différant sur chaque sommet excepté aux bords a et b), alors lorsque ce cycle est formé dans le graphe, l'algorithme ouvre une porte joignant deux classes identiques (du fait qu'il existe déjà un chemin entre a et b, $a \sim b$). Or, ceci est absurde, donc le graphe obtenu est bien acyclique : Le labyrinthe est bien parfait.

5.
```
let melange_knuth (a : 'a array) : unit =
  let n = Array.length a in

  let rec recursive_swap cur_index =
    if(cur_index >= n) then ()
    else begin
      let rand_index = Random.int (cur_index + 1) in
      let temp = a.(rand_index) in
      a.(rand_index) <- a.(cur_index);
      a.(cur_index) <- temp
      end
```