Question 1 –

```
(* Solution 1 *)
let nombre_aretes (g: graphe): int =
  let res = Array.fold_left (fun m li -> m + List.length li) 0 g in
  res/2;; (* Chaque arête a été comptée deux fois *)
```

```
(* Solution 2 *)
let nombre_aretes (g: graphe): int =
  let res = ref 0 in
  for s = 0 to Array.length g - 1 do
    res := !res + List.length g.(s)
  done;
  !res/2;; (* Chaque arête a été comptée deux fois *)
```

Question 2 –

```
let adj_G32 = [|
    [|1;3|]; [|0;2;4|]; [|1;5|];
    [|0;4|]; [|1;3;5|]; [|2;4|]
  |];;
```

Question 3 –

```
(* Solution 1 *)
let adjacence (g: graphe): int array array =
  Array.map Array.of_list g;;
```

```
(* Solution 2 *)
let adjacence (g: graphe): int array array =
  let n = Array.length g in
  let adj = Array.make n [||] in
  for s = 0 to n-1 do
    adj.(s) <- Array.of_list g.(s)
  done;
  adj;;
```

Pour la complexité de la solution 2 : on remarque que la fonction parcourt tous les sommets du graphe. De plus, pour chaque sommet `s`, la liste d'adjacence de `s` notée `li` est convertie en tableau d'adjacence en temps $\mathcal{O}$(`len(li)`). Ainsi, le temps d'exécution de la boucle est de la forme :

$$T_1(G) = \sum_{s=0}^{n-1} T'(s) \text{ avec } T'(s) = \mathcal{O}(\deg(s)) + \mathcal{O}(1)$$

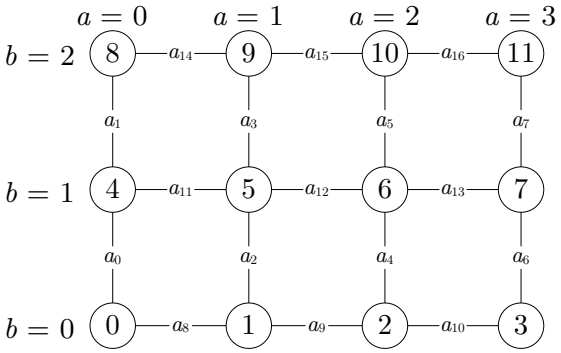
Donc :

$$T_1(G) = \mathcal{O}\left(\sum_{s=0}^{n-1} \deg(s)\right) + \mathcal{O}(n) = \mathcal{O}\left(2m\right) + \mathcal{O}(n) = \mathcal{O}\left(m\right) + \mathcal{O}(n)$$

Finalement, le temps d'exécution total est de la forme :

$$T(G) = T_1(G) + \mathcal{O}(n) = \mathcal{O}(m+n).$$

Question 4 – Dans le graphe $G_{p,q}$, on associe à chaque sommet des coordonnées (a, b) où $a \in [\![0; p-1]\!]$ est l'indice de la colonne ($a = 0$ est la colonne de gauche et $a = p-1$ est la colonne de droite) et $b \in [\![0; q-1]\!]$ est l'indice de la ligne ($b = 0$ est la ligne du bas et $b = q-1$ est la ligne du haut) :



On remarque que :

- ⋆ Le numéro du sommet de coordonnées (a, b) est $a + bp$.
- ⋆ Les coordonnées du sommet numéro s sont $(s \mod p, \lfloor s/p \rfloor)$

De plus, les arêtes verticales sont placées sur p colonnes et $q-1$ lignes, et les arêtes horizontales sur $p-1$ colonnes et q lignes. Ainsi, si comme dans l'énoncé, on suppose que s et t sont adjacents avec $s < t$ alors il y a deux cas :

- ⋆ s est le sommet directement en dessous de t. Dans ce cas, le rang de l'arête qui les relie est $b + a(q-1)$ avec (a, b) les coordonnées de s.
- ⋆ s est le sommet directement à gauche de t. Dans ce cas, le rang de l'arête qui les relie est $p(q-1) + a + b(p-1)$ avec (a, b) les coordonnées de s.

```
(* Attention: dans certaines implémentations, les cas p = 1 et q = 1
   doivent être traités à part.
   Ne pas oublier de tester votre fonction pour p = 1 et pour q = 1. *)
let rang ((p,q): int*int) ((s,t): int*int): int =
  let a = s mod p and
      b = s/p in
  if t = s+p then (* arête verticale *)
    b + a*(q-1)
  else            (* arête horizontale *)
    p*(q-1) + a + b*(p-1);;
```

Question 5 –

```
let sommets ((p,q): int*int) (i: int): int*int =
  if i < p*(q-1) then begin  (* arête verticale *)
      let a = i/(q-1) and
          b = i mod (q-1) in
      let s = a + b*p in
      s, s+p
    end
  else begin                 (* arête horizontale *)
      let j = i-p*(q-1) in
      let a = j mod (p-1) and
          b = j/(p-1) in
      let s = a + b*p in
      s, s+1
    end;;
```

Question 6 –

```
let quadrillage (p: int) (q: int): graphe =
  let n = p*q and
      m = p*(q-1) + q*(p-1) in
  let g = Array.make n [] in
  for i = 0 to m-1 do
    let s,t = sommets (p,q) i in
    g.(s) <- t :: g.(s);
    g.(t) <- s :: g.(t);
  done;
  g;;
```

Question 7 – Montrons les trois propriétés :

⋆ Pour tout $s \in S_n$, on a $s \in C_s$ car il existe un chemin de s à s (c'est un chemin de longueur nulle, voir l'énoncé). Ainsi :

$$\boxed{\forall s \in S_n, C_s \neq \varnothing}$$

⋆ D'après ce qui précède :

$$S_n = \bigcup_{s \in S_n} \{s\} \subset \bigcup_{s \in S_n} C_s$$

L'autre inclusion est évidente, ainsi :

$$\boxed{S_n = \bigcup_{s \in S_n} C_s}$$

⋆ Soient s et t deux sommets tels que $C_s \cap C_t \neq \varnothing$. Montrons que $C_s = C_t$ par double inclusion.

Comme $C_s \cap C_t \neq \varnothing$, il existe un sommet $u \in C_s \cap C_t$ et donc un chemin γ_1 (resp. γ_2) de s (resp. t) à u. Pour tout $v \in C_s$, il existe un chemin γ_3 de s à v. Si on note $\overline{\gamma_1}$ le chemin γ_1 parcouru à l'envers, alors la concaténation de γ_2, $\overline{\gamma_1}$ et γ_3 est un chemin de t à v et donc $v \in C_t$. Ainsi :

$$C_s \subset C_t$$

Le problème étant symétrique, on obtient $C_t \subset C_s$. De plus, étant donné que $C_s \neq \varnothing$, on ne peut pas avoir $C_s = C_t$ et $C_s \cap C_t = \varnothing$. Finalement :

$$\boxed{\text{Pour tous sommets } s \text{ et } t, \text{ soit } C_s = C_t, \text{ soit } C_s \cap C_t = \varnothing.}$$

Question 8 – On note $\omega(\gamma)$ la longueur d'un chemin γ et :

$$D = \Big\{\omega(\gamma) : \gamma \text{ est un chemin de } s \text{ à } t\Big\}$$

Comme $t \in C_s$, D est une partie non vide de $\mathbb{N}$ et admet donc un plus petit élément. En d'autres termes :

$$\boxed{\text{Il existe un plus court chemin de } s \text{ à } t.}$$

Soit $\gamma = (s = s_0, s_1, \ldots, s_k = t)$ un plus court chemin de s à t. En particulier, on a $k = \omega(\gamma) = \min(D)$. Supposons par l'absurde que γ passe deux fois par le même sommet, il existe $i < j$ tel que $s_i = s_j$. Soit :

$$\gamma' = (s = s_0, s_1, \ldots, s_{i-1}, s_i = s_j, s_{j+1}, s_k = t)$$

Alors γ' est bien un chemin de s à t et vérifie $\omega(\gamma') < \omega(\gamma)$ ce qui contredit la minimalité de $\omega(\gamma)$. On a une contradiction donc les sommets de γ sont deux à deux distincts.

Question 9 – Soit $k \in [\![0; m-1]\!]$, notons s et t les extrémités de a_k et supposons par l'absurde que s et t appartiennent à la même composante connexe de G_k. Il existe donc un plus court chemin γ de s à t dans G_k. Dans le graphe G_{k+1}, on note γ' la concaténation de γ avec l'arête $\{t, s\}$. Alors γ' est une cycle de G_k car :

- Il est de longueur $\geqslant 2$ puisque $s \neq t$.
- Ses arêtes sont deux à deux distinctes puisque les sommets de γ sont deux à deux distincts par la question précédente.

On a une contradiction car G est acyclique et donc G_{k+1} l'est aussi. Finalement :

$$\boxed{\text{Les extrémités de } a_k \text{ appartiennent à deux composantes connexes différentes de } G_k.}$$

Remarque. La relation ne fonctionne pas si $n = 0$. À partir de maintenant, on suppose que le graphe possède au moins un sommet (je ne crois pas que cette hypothèse apparaisse dans l'énoncé).

On remarque d'abord que G_0 est un graphe sans arête qui possède donc n composantes connexes et que $G_m = G$ est un graphe connexe qui possède donc 1 composante connexe.

Soit $k \in [\![0; m-1]\!]$ et s, t les extrémités de l'arête a_k. Le fait d'ajouter a_k dans G_k réunit les composantes connexes de u et v et laisse les autres composantes connexes inchangées. Formellement, si on note C_u (resp. C'_u) la composante connexe de u dans G_k (resp. G_{k+1}) :

$$\begin{cases} C'_u = C_s \sqcup C_t & \text{pour tout } u \in C_s \sqcup C_t \\ C'_u = C_u & \text{pour tout } u \notin C_s \sqcup C_t \end{cases}$$

Soit c_k le nombre de composantes connexes dans G_k, alors :

$$\begin{cases} c_0 = n \\ c_{k+1} = c_k - 1 \quad \text{pour tout } k \in [\![0; m-1]\!] \\ c_m = 1 \end{cases}$$

En conclusion :

$$\boxed{m = n - 1}$$

Question 10 – Les implications $(i) \Rightarrow (ii)$ et $(i) \Rightarrow (iii)$ sont des conséquences directes de la question précédente.

$\boxed{(ii) \Rightarrow (i).}$ Soit $\mathcal{H}$ l'ensemble de tous les graphes connexes obtenus en supprimant des arêtes de G :

$$\mathcal{H} = \Big\{ H : H = (S_n, B) \text{ avec } B \subset A \text{ et } H \text{ connexe} \Big\}.$$

On a $G \in \mathcal{H}$, donc $\mathcal{H} \neq \varnothing$ et on peut donc considérer $H = (S_n, B)$ un élément de $\mathcal{H}$ avec B de cardinal minimal.

Supposons par l'absurde que H contienne un cycle, alors en supprimant n'importe quelle arête de ce cycle, on obtient un nouveau graphe H' qui reste connexe. On a $H' \in \mathcal{H}$ ce qui contredit la minimalité de H. Ainsi, H est acyclique. Pour conclure, on remarque que H est un arbre. Par la question précédente, son nombre d'arêtes est donc $n - 1 = m$, c'est à dire que $H = G$:

$$\boxed{G \text{ est donc un arbre}}$$

$\boxed{(iii) \Rightarrow (i).}$ Même genre de raisonnement avec $\mathcal{H}$ l'ensemble de tous les graphes acycliques obtenus en ajoutant des arêtes à G :

$$\mathcal{H} = \Big\{ H : H = (S_n, B) \text{ avec } A \subset B \text{ et } H \text{ acyclique} \Big\}.$$

Question 11 –

```
let rec representant (part: int array) (s: int): int = match () with
  | _ when part.(s) < 0 -> s
  | _ -> representant part part.(s);;
```

Question 12 –

```
(* On suppose sans le vérifier que s et t sont bien deux représentants
   distincts *)
let union (part: int array) (s: int) (t: int): unit =
  let hs = - part.(s) - 1 in
  let ht = - part.(t) - 1 in
  if hs > ht then
    part.(t) <- s
  else begin
      let new_ht = max ht (1 + hs) in
      part.(s) <- t;
      part.(t) <- - new_ht - 1
    end;;
```

Question 13 – On montre cette propriété que l'on note $(\mathcal{H}_k)$ par récurrence sur $k \in \mathbb{N}$ le nombre de réunions utilisées pour construire $\mathcal{P}$.

$\star$ Si $k = 0$, alors $\mathcal{P} = \mathcal{P}_n^{(0)}$ et $X = \{s\}$ est un singleton avec $h(s) = 0$. On a bien :

$$|X| = 1 \geqslant 2^0 = 2^{h(s)}$$

D'où $\mathcal{H}_0$.

$\star$ Soit $k \in \mathbb{N}$. On suppose $\mathcal{H}_k$ et on montre $\mathcal{H}_{k+1}$.

Soit $\mathcal{P}$ une partition construite par $k+1$ réunions successives et $\mathcal{P}'$ la partition obtenue après les k premières réunions. Dans la suite on note h la fonction hauteur lorsqu'on s'intéresse à la partition $\mathcal{P}$ et on la note h' lorsqu'on s'intéresse à $\mathcal{P}'$. Soit s le représentant d'une partie $X \in \mathcal{P}$. On a alors plusieurs cas :

- Le dernière réunion ne concernait pas X, c'est à dire que $X \in \mathcal{P}'$ et s était déjà le représentant de X dans $\mathcal{P}'$. On a $h(s) = h'(s)$ et l'inégalité est vérifiée par hypothèse de récurrence.
- La dernière réunion concernait X, c'est à dire que $X = X' \cup Y'$ avec $X' \in \mathcal{P}'$ une partie de représentant $s' \in X'$ et $Y' \in \mathcal{P}'$ une partie de représentant $t' \in Y'$. Par hypothèse de récurrence :

  $$|X'| \geqslant 2^{h'(s')} \qquad\qquad |Y'| \geqslant 2^{h'(t')}$$

  Après la réunion, on obtient $s = s'$ ou $s = t'$, et dans tous les cas :

  $$\begin{cases} h(s) \leqslant h'(s') + 1 \\ h(s) \leqslant h'(t') + 1 \end{cases}$$

  Ainsi :

  $$|X| = |X'| + |Y'| \geqslant 2^{h'(s')} + 2^{h'(t')} \geqslant 2^{h(s)-1} + 2^{h(s)-1} = 2^{h(s)}$$

D'où $\mathcal{H}_{k+1}$.

Question 14 –

La fonction `union` s'exécute en temps $\mathcal{O}(1)$.

Pour la fonction `representant`, on remarque que le nombre d'appels récursifs est $\leqslant 1 + h(t)$ avec t le représentant du sommet s donné en entrée. Avec la question précédente :

$$|h(t)| \leqslant \log_2(n).$$

Ainsi :

La fonction `representant` s'exécute en temps $\mathcal{O}(\log_2(n))$.

Question 15 – Pour vérifier si un graphe est un arbre, on teste s'il est connexe et si $m = n - 1$ (voir question 10).

```
(* Renvoie le nombre de parties dans la partition *)
let compter_parties (part: int array): int =
  let n = Array.length part in
  let cpt = ref 0 in
  for s = 0 to n - 1 do
    if part.(s) < 0 then incr cpt;
  done;
  !cpt;;
```

```
(* Renvoie une partition contenant les composantes connexes du
   graphe *)
let make_cc (g: graphe): int array =
  let n = Array.length g in
  let part = Array.make n (-1) in
  for s = 0 to n-1 do
    List.iter
      (fun t ->
        let repr_s = representant part s in
        let repr_t = representant part t in
        if repr_s <> repr_t then union part repr_s repr_t)
      g.(s)
  done;
  part;;
```

```
let est_un_arbre (g: graphe): bool =
  let n = Array.length g in
  let m = nombre_aretes g in
  m = n-1 && compter_parties (make_cc g) = 1;;
```

Question 16 –

Il s'agit du chemin : $(1, 2, 5, 4)$

Question 17 – La terminaison de cet algorithme n'est pas garantie. En effet, il est possible que l'extrémité du chemin (notée s_k dans l'énoncé) ne soit jamais un sommet de $\mathcal{T}$. Par exemple, si $G = G_{3,2}$ et que $\mathcal{T}$ est réduit à la racine $r = 0$, alors en essayant de construire un chemin qui part du sommet initial $s = 1$, l'algorithme peut boucler indéfiniment en sélectionnant les sommets $2, 5, 4, 1, 2, 5, 4, 1, 2, 5, 4, \ldots$

Question 18 –

```
(* Remarque concernant la consigne "on supprime le cycle qui vient
   d'être formé et u devient le nouveau point d'arrivée".
   On remarque que le type chemin permet de gérer facilement cette
   consigne, il suffit de remplacer c.fin par u. Cela va
   automatiquement "couper le cycle qui vient d'être formé", sans
   avoir besoin de modifier c.suivant. *)
let marche_aleatoire (adj: int array array) (parent: int array) (s: int): chemin =
  let n = Array.length adj in
  let c = {debut = s; fin = s; suivant = Array.make n (-1)} in
  while parent.(c.fin) = -2 do
    let nb_voisins = Array.length adj.(c.fin) in
    let ind_nv_fin = Random.int nb_voisins in
    let nv_fin = adj.(c.fin).(ind_nv_fin) in
    c.suivant.(c.fin) <- nv_fin;
    c.fin <- nv_fin;
  done;
  c;;
```

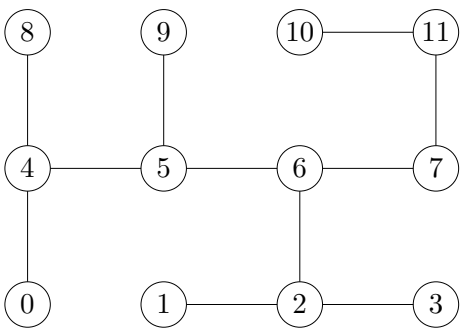
Question 19 –

```
let greffe (parent: int array) (c: chemin): unit =
  let s = ref c.debut in
  while !s <> c.fin do
    parent.(!s) <- c.suivant.(!s);
    s := c.suivant.(!s);
  done;;
```

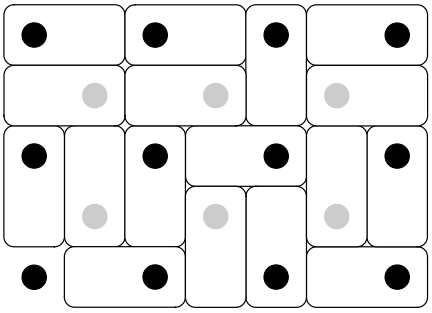
Question 20 –

```
let wilson (g: graphe) (r: int): int array =
  let n = Array.length g in
  let parent = Array.make n (-2) in
  let adj = adjacence g in
  parent.(r) <- -1;
  for s = 0 to n-1 do
    if parent.(s) = -2 then begin
      let c = marche_aleatoire adj parent s in
      greffe parent c;
      end;
  done;
  parent;;
```

Question 21 – On obtient :



Question 22 – On obtient :



Question 23 – Dans $E_{p,q}$, on repère les cases par le système de coordonnées (x, y) décrit au début de la partie IV. Dans $G_{p,q}$, on repère les sommets par le système de coordonnées (a, b) décrit à la question 4. L'énoncé indique que les cases noires de $E_{p,q}$ sont vues comme les sommets de $G_{p,q}$. Plus précisément, la case de coordonnées (x, y) de $E_{p,q}$ avec x et y pairs est en correspondance avec le sommet de coordonnées $(x/2, y/2)$ dans $G_{p,q}$. Voici la procédure pour obtenir le père d'un sommet $s \in \{1, \ldots, pq-1\}$:

- ⋆ On calcule les coordonnées (a, b) de s à l'aide des formules :

$$a = s \mod p \qquad\qquad b = \left\lfloor \frac{s}{p} \right\rfloor$$

- ⋆ On identifie les coordonnées $(a', b') \in \Big\{(a+1, b), (a, b+1), (a-1, b), (a, b-1)\Big\}$ du père de s en traitant quatre cas en fonction de la direction du domino de coordonnées $(2a, 2b)$.
- ⋆ On obtient le numéro s' du père de s à l'aide de la formule :

$$s' = pb' + a'.$$

Question 24 –

```
(* p et q sont les variables globales évoquées dans l'énoncé *)
let coord_noire (s: int): int*int =
  let a = s mod p in
  let b = s/p in
  (2*a, 2*b);;
```

Question 25 –

```
let sommet_direction (s: int) (d: direction): int =
  let a_s = s mod p in
  let b_s = s/p in
  let (a_t, b_t) = match d with
    | S -> (a_s, b_s-1)
    | W -> (a_s-1, b_s)
    | N -> (a_s, b_s+1)
    | E -> (a_s+1, b_s)
  in
  if 0 <= a_t && a_t < p && 0 <= b_t && b_t < q then
    p*b_t + a_t
  else
    -1;;
```

Question 26 –

```
let phi (pavage: direction array array): int array =
  let n = p*q in
  let parent = Array.make n (-1) in
  for s = 1 to n-1 do
    let x,y = coord_noire s in
    let pere_s = sommet_direction s pavage.(x).(y) in
    parent.(s) <- pere_s
  done;
  parent;;
```

Question 27 – Voici les fonctions `coord_grise` et `numero` évoquées dans l'énoncé :

```
(* Fonction supposée déjà définie *)
let coord_grise (s: int): int*int =
  let a = (s-1) mod (p-1) in
  let b = (s-1)/(p-1) in
  (2*a+1, 2*b+1);;
```

```
(* Fonction supposée déjà définie *)
let numero: int*int -> int = function
  | x,_ when x < 0 || x >= 2*p-1 -> 0
  | _,y when y < 0 || y >= 2*q-1 -> 0
  | x,y when x mod 2 = 0 && y mod 2 = 0 ->
     let a = x/2 in
     let b = y/2 in
     p*b + a
  | x,y when x mod 2 = 1 && y mod 2 = 1 ->
     let a = (x-1)/2 in
     let b = (y-1)/2 in
     (p-1)*b + a + 1
  | _ -> 0;;
```

```
let dual(): graphe =
  let n_etoile = (p-1)*(q-1) + 1 in
  let g_etoile = Array.make n_etoile [] in
  for s = 1 to n_etoile-1 do
    let (x,y) = coord_grise s in
    List.iter (fun coord ->
        let t = numero coord in
        g_etoile.(s) <- t :: g_etoile.(s);
        if t = 0 then g_etoile.(t) <- s :: g_etoile.(t);
      ) [(x-2, y); (x+2, y); (x, y-2); (x, y+2)];
  done;
  g_etoile;;
```

Question 28 – On remarque que $G^*_{p,q}$ est composé :

- ⋆ D'une copie du graphe $G_{p-1,q-1}$ dans laquelle on a changé la numérotation des sommets.
- ⋆ D'un sommet s_0 numéroté par 0 et relié à chaque sommet se trouvant sur la première (resp. dernière) ligne ou première (resp. dernière) colonne du quadrillage formé par $G_{p-1,q-1}$. Les arêtes issues de s_0 peuvent être multiples, par exemple pour $p \geqslant 2$ et $q \geqslant 2$, il y a exactement deux arêtes entre s_0 et le

sommet t se trouvant en bas à gauche du quadrillage ; la première arête vient du fait que t est sur la première ligne et la seconde du fait que t est sur la première colonne.

Par exemple avec $p = 4$ et $q = 5$, on obtient la figure 1 où le sommet s_0 n'est pas représenté. Sur cette figure, toutes les arêtes hachurées ont pour deuxième extrémité s_0 (notez qu'il serait possible de dessiner s_0 sans que ces arêtes ne se coupent).

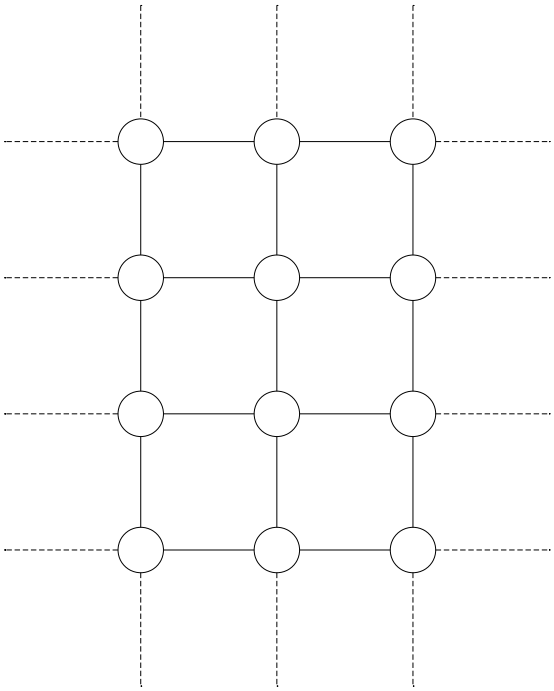


FIGURE 1

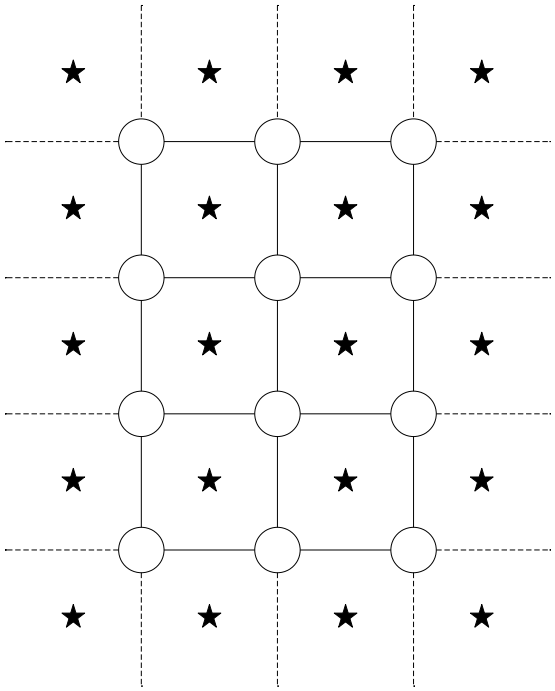


FIGURE 2

On s'intéresse maintenant au dual de $G^*_{p,q}$. Par définition, $G^{**}_{p,q}$ possède un sommet pour chaque face dans $G^*_{p,q}$. Graphiquement, ces sommets peuvent être placés sur un quadrillage de taille $p \times q$ comme l'indiquent les étoiles de la figure 2 :

- Les étoiles se trouvant sur les bords du quadrillage correspondent à des faces construites avec deux arêtes hachurées. L'une de ces étoiles correspond à la face non bornée de $G^*_{p,q}$ (l'étoile en question dépend de la façon dont on relie graphiquement les arêtes hachurées à s_0).
- Les autres étoiles (celle qui sont à l'intérieur du quadrillage) font parties du dual du sous-graphe $G_{p-1,q-1}$.

De plus, pour chaque arête a dans $G^*_{p,q}$, on a une arête dans $G^{**}_{p,q}$ qui relie les deux étoiles séparées par a.

En résumé, quitte à renuméroter les sommets, le dual de $G^*_{p,q}$ est $G_{p,q}$.

Question 29 – On suppose que le graphe (S_n, B) possède un cycle γ. Lorsqu'on trace graphiquement γ dans le plan $\mathbb{R}^2$, le plan est séparé en deux zones notées $X \subset \mathbb{R}^2$ (à l'intérieur du cycle) et $Y \subset \mathbb{R}^2$ (à l'extérieur du cycle). Par définition du graphe dual, une arête a^* coupe γ si et seulement si a fait partie de γ. Ainsi, la définition de B^* implique qu'aucune arête $a^* \in A^*$ coupant γ ne peut appartenir à B^*. Finalement, aucune arête de (S_{n^*}, B^*) ne relie les zones X et Y. Comme chaque zone contient au moins un sommet de S_{n^*}, le graphe (S_{n^*}, B^*) n'est pas connexe.

Question 30 – Soient $k = |B|$ le nombre d'arêtes dans $\mathcal{T} = (S_n, B)$ et $k^* = |B^*|$ le nombre d'arêtes dans $\mathcal{T}^* = (S_{n^*}, B^*)$. Par définition de B^* :

$$k^* = m - k$$

On remarque alors que :

$$\begin{aligned} k^* = n^* - 1 &\Leftrightarrow m - k = n^* - 1 \\ &\Leftrightarrow p(q-1) + q(p-1) - k = (p-1)(q-1) + 1 - 1 \\ &\Leftrightarrow k = pq - 1 \\ &\Leftrightarrow k = n - 1 \end{aligned}$$

Montrons l'équivalence par double implication :

($\Leftarrow$) Si $\mathcal{T}^*$ est un arbre couvrant de $G^*_{p,q}$, alors par la question 10, $\mathcal{T}^*$ est connexe et $k^* = n^* - 1$. Donc $k = n - 1$ et par la question précédente, $\mathcal{T}'$ est acyclique. Donc $\mathcal{T}$ est un arbre couvrant de $G_{p,q}$ (voir question 10).

($\Rightarrow$) La preuve est similaire. En effet, comme $G_{p,q}$ est le graphe dual de $G^*_{p,q}$ (question 28), on peut réutiliser les mêmes arguments que dans la question 29 pour montrer que :

Si (S_{n^*}, B^*) possède un cycle, alors (S_n, B) n'est pas connexe.

Question 31 –

```
(* Attention: pour utiliser la fonction rang, il faut que s < t *)
let vers_couple (parent: int array): int * bool array =
  let n = p*q in
  let m = p*(q-1) + q*(p-1) in
  let b = Array.make m false in
  let r = ref (-1) in
  for s = 0 to n-1 do
    let t = parent.(s) in
    if t = -1 then
      r := s
    else begin
        let i = rang (p,q) (min s t, max s t) in
        b.(i) <- true
      end;
  done;
  (!r, b);;
```

Question 32 –

⋆ La principale difficulté est que le tableau B seul ne permet pas de savoir dans quelle direction est orientée chaque arête de $\mathcal{T}$. Pour le déterminer, on lance un parcours (en largeur ou en profondeur) sur $\mathcal{T}$ à partir du sommet r (le tableau B indique si une arête de $G_{p,q}$ peut être empruntée). À chaque fois que le parcours utilise une arête d'un sommet u vers un sommet v, on peut en déduire que u est le père de v dans $\mathcal{T}$. À la fin du parcours, on connaît toutes les relations père/fils dans l'arbre, on peut alors construire le tableau `parent`.

⋆ Cet algorithme est effectivement applicable à un arbre couvrant de $G^*_{p,q}$. En réalité, on peut l'utiliser avec n'importe quel graphe G muni d'un arbre couvrant $\mathcal{T}$. En effet, un parcours sur $\mathcal{T}$ construit un arbre couvrant de $\mathcal{T}$. Or le seul arbre couvrant de $\mathcal{T}$ est $\mathcal{T}$ lui-même. De plus, le parcours emprunte les arêtes en s'éloignant de la racine.

Remarque. Pour pouvoir appliquer l'algorithme à $G^*_{p,q}$, il faut écrire une fonction `rang_etoile` similaire à la fonction `rang`. Pour cela, on numérote les arêtes de $G^*_{p,q}$ en utilisant la numérotation des arêtes de $G_{p,q}$: l'arête a^* a le même numéro que l'arête a. De plus, étant donné que le graphe $G^*_{p,q}$ contient des arêtes multiples, la fonction `rang_etoile` renvoie une liste de rangs.

Question 33 –

```
(* Indique si l'arête (s,t) est dans T *)
let dans_T s t b =
  let i = rang (p,q) (min s t, max s t) in
  b.(i);;
```

```
1  (* Dans la fonction parc_prof:
2   * - Le sommet s vient d'être visité
3   * - li est la liste d'adjacence de s dans T *)
4  let vers_parent ((r, b): int * bool array): int array =
5    let n = p*q in
6    (* let g = quadrillage p q in *) (* variable globale déjà définie *)
7    let parent = Array.make n (-2) in
8    let rec parc_prof s li = match li with
9      | [] -> ()
10     | t :: q when parent.(t) <> -2 -> parc_prof s q
11     | t :: q  -> parent.(t) <- s;
12                  parc_prof t (List.filter (fun u -> dans_T t u b) g.(t));
13                  parc_prof s q
14   in
15   parc_prof (-1) [r];
16   parent;;
```

Question 34 – Tout d'abord, on remarque que la fonction `rang` de la question 4 s'exécute en temps constant.

⋆ Dans la fonction `vers_couple`, la création du tableau à la ligne 5 s'exécute en temps $\mathcal{O}(m)$. De plus, la boucle `for` fait `n` tours et chaque tour s'exécute en temps constant. Finalement :

Le temps d'exécution de la fonction `vers_couple` est en $\mathcal{O}(n+m)$

⋆ Dans la fonction `vers_parent`, la création du tableau ligne 7 s'exécute en temps $\mathcal{O}(n)$. De plus, le parcours en largeur considère deux fois chaque arête $\{u, v\}$ du graphe $G_{p,q}$ (une fois de u vers v et une fois de v vers u). Finalement :

Le temps d'exécution de la fonction `vers_parent` est en $\mathcal{O}(n+m)$

Question 35 – Voici la fonction `vers_parent_etoile` évoquée dans l'énoncé :

```
(* Fonction supposée déjà définie *)
(* Suppose s < t et que s et t sont voisins *)
let rang_etoile ((p,q): int*int) ((s,t): int*int): int list = match s with
  | 0 ->
     let a = (t-1) mod (p-1) and
         b = (t-1)/(p-1) in
     let res = ref [] in
     if a = 0 then res := b :: !res;
     if a = p-2 then res := ((p-1)*(q-1)+b) :: !res;
     if b = 0 then res := (p*(q-1) + a) :: !res;
     if b = q-2 then res := (p*(q-1) + (p-1)*(q-1) + a) :: !res;
     !res
  | _ ->
     let a = (s-1) mod (p-1) and
         b = (s-1)/(p-1) in
     if t = s+(p-1) then (* arête verticale *)
       [p*(q-1) + (p-1) + a + b*(p-1)]
     else                 (* arête horizontale *)
       [(q-1) + b + a*(q-1)];;
```

```
(* Fonction supposée déjà définie *)
(* Attention: lorsqu'il y a plusieurs arêtes entre s et t dans
   G*_{p,q}, cette fonction indique si l'une de ces arêtes est dans T* *)
let dans_T_etoile s t b =
  let li = rang_etoile (p,q) (min s t, max s t) in
  List.exists (fun i -> b.(i)) li;;
```

```
(* Fonction supposée déjà définie *)
let vers_parent_etoile ((r, b): int * bool array): int array =
  let n_etoile = (p-1)*(q-1) + 1 in
  (* let g_etoile = dual() *) (* variable globale déjà définie *)
  let parent_etoile = Array.make n_etoile (-2) in
  let rec parc_prof s li = match li with
    | [] -> ()
    | t :: q when parent_etoile.(t) <> -2 -> parc_prof s q
    | t :: q  -> parent_etoile.(t) <- s;
                 parc_prof t
                   (List.filter (fun u -> dans_T_etoile t u b) g_etoile.(t));
                 parc_prof s q
  in
  parc_prof (-1) [r];
  parent_etoile;;
```

La fonction `arbre_dual` demandée dans la question :

```
let arbre_dual (parent: int array): int array =
  let (r,b) = vers_couple parent in
  let b_etoile = Array.map not b in
  vers_parent_etoile (0,b_etoile);;
```

Question 36 – Pour construire le pavage, $\mathcal{P}$, on s'intéresse d'abord aux cases noires de l'échiquier, puis aux grises. voici les étapes de l'algorithme :

(i) Chaque case noire (x, y) correspond à un sommet s dans $G_{p,q}$. On note « `d: direction` » la direction à emprunter pour aller de s à t dans $\mathcal{T}$. On place alors un domino dans la direction `d` sur la case (x, y).

(ii) On utilise le même procédé pour les cases grises (x, y) correspondant à un sommet s de $G^*_{p,q}$ dont le père n'est pas 0 dans $\mathcal{T}^*$.

(iii) Lorsque les deux étapes précédentes sont terminées, il reste seulement à placer les dominos sur les cases grises (x, y) correspondant à un sommet s de $G^*_{p,q}$ dont le père est 0 dans $\mathcal{T}^*$. Pour chacune de ces cases, on place un domino dans la seule direction possible (voir ci-dessous pour la justification qu'une seule direction est possible).

Avant cette étape, on construit donc un tableau « `occupee: bool array array` » de taille $(2p-1) \times (2q-1)$ indiquant pour chaque case de $E_{p,q}$ si elle est recouverte par un domino. On peut alors en déduire la direction du domino sur chaque case (x, y) considérée à cette étape.

Il s'agit maintenant de justifier qu'on obtient un pavage.

Pour commencer, on pose autant de dominos qu'il y a de cases noires et grises dans $E'_{p,q}$, c'est à dire $N = pq - 1 + (p-1)(q-1)$. Puisque $E'_{p,q}$ contient $(2p-1)(2q-1) - 1 = 2N$ cases, il suffit de montrer que les dominos ne se chevauchent pas pour montrer que c'est bien un pavage.

On remarque que chaque case blanche de l'échiquier correspond à une arête a dans $G_{p,q}$ et à l'arête a^* (associée à a) dans $G^*_{p,q}$. De plus, par définition de $\mathcal{T}^*$, soit a apparaît dans $\mathcal{T}$, soit a^* apparaît dans $\mathcal{T}^*$ (mais pas les deux). Enfin dans $\mathcal{T}$ et $\mathcal{T}^*$, si s est le père de t alors t n'est pas le père de s. Les arguments précédents permettent de conclure que les dominos posés aux étapes (i) et (ii) ne se chevauchent pas.

Il reste à justifier qu'à l'étape (iii), les dominos ne peuvent être posés que dans une seule direction. Si un domino peut être posé dans deux directions différentes sur une case grise, cela signifie que dans l'arbre $\mathcal{T}^*$, il existe un sommet s adjacent à deux arêtes n'ayant pas été considérées à l'étape (ii). L'une de ces arêtes, notée a_1, relie s à son père (qui est 0) et l'autre, notée a_1, relie s à un de ses fils t. C'est impossible puisque t (dont le père n'est pas 0) aurait dû être traité à l'étape (ii).

Question 37 –

```
(* Indique la direction à prendre pour aller de s à t dans G_{p,q} *)
let trouver_direction (s: int) (t: int) = match () with
  | _ when t = s + p -> N
  | _ when t = s - p -> S
  | _ when t = s-1 -> W
  | _ when t = s+1 -> E
  | _ -> failwith "Cas impossible"
```

```
(* Indique la direction à prendre pour aller de s à t dans G*_{p,q}
   Suppose s <> 0 et t <> 0 *)
let trouver_direction_etoile (s: int) (t: int)
      (pavage: direction array array) = match () with
  | _ when t = s + p-1 -> N
  | _ when t = s - (p-1) -> S
  | _ when t = s+1 -> E
  | _ when t = s-1 -> W
  | _ -> failwith "Cas impossible"
```

```
(* Renvoie les coordonnées de la case atteinte dans E_{p,q} lorsqu'on
   prend la direction d à partir de la case de coordonnées (x,y) *)
let prendre_direction (x: int) (y: int) (d: direction): int*int = match d with
  | S -> (x, y-1)
  | W -> (x-1, y)
  | N -> (x, y+1)
  | E -> (x+1, y);;
```

```
(* Renvoie la direction à prendre pour atteindre la seule case
   inoccupée parmi les voisins de (x,y) dans E_{p,q}.
   Déclenche une erreur si plusieurs voisins sont inoccupés *)
let trouver_direction_etoile2 (x: int) (y: int) (occupee: bool array array): direction =
  let li = List.filter (fun d -> let x2,y2 = prendre_direction x y d in
                                 not occupee.(x2).(y2))
             [S; W; N; E] in
  match li with
  | [d] -> d
  | _ -> failwith "Ne devrait pas arriver"
```

```
let pavage_aleatoire() =
  let parent = wilson g 0 in
  let parent_etoile = arbre_dual parent in
  let pavage = Array.make_matrix (2*p-1) (2*q-1) N in
  let occupee = Array.make_matrix (2*p-1) (2*q-1) false in
  for s = 1 to p*q-1 do
    let (x,y) = coord_noire s in
    let t = parent.(s) in
    pavage.(x).(y) <- trouver_direction s t;
    let x2,y2 = prendre_direction x y pavage.(x).(y) in
    occupee.(x).(y) <- true;
    occupee.(x2).(y2) <- true;
  done;
  for s = 1 to (p-1)*(q-1) do
    let (x,y) = coord_grise s in
    let t = parent_etoile.(s) in
    if t <> 0 then begin
        pavage.(x).(y) <- trouver_direction_etoile s t pavage;
        let x2,y2 = prendre_direction x y pavage.(x).(y) in
        occupee.(x).(y) <- true;
        occupee.(x2).(y2) <- true;
      end;
  done;
  for s = 1 to (p-1)*(q-1) do
    let (x,y) = coord_grise s in
    let t = parent_etoile.(s) in
    if t = 0 then
      pavage.(x).(y) <- trouver_direction_etoile2 x y occupee;
  done;
  pavage;;
```

Question 38 –

⋆ Le pavage d'un échiquier à $2p$ colonnes et $2q-1$ lignes correspond au pavage de $E'_{p+1,q+1}$ sur lequel la ligne du bas est pavée uniquement avec des dominos horizontaux. On reprend la méthode précédente, sauf

que l'algorithme de Wilson n'est pas lancé à partir de l'arbre réduit à la racine 0, mais à partir de l'arbre contenant tous les sommets et arêtes sur la ligne du bas de $G_{p+1,q+1}$ (chaque sommet $s \neq 0$ a pour père $s-1$).

$\star$ Pour un échiquier à $2p$ colonnes et $2q$ lignes, on applique la même logique en lançant l'algorithme de Wilson à partir de l'arbre contenant tous les sommets et arêtes sur la ligne du bas et la colonne de gauche.