Centrale Option 2021
Arbres couvrants et pavages

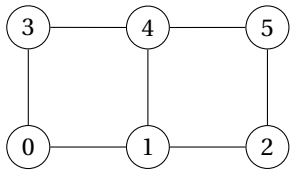
Correction proposée par Olivier Brunet.

I. Quelques fonctions auxilliaires

Question 1 Il ne faut pas oublier que le graphe est non orienté et donc que l'on va compter chaque arête deux fois.

```
let nombre_aretes g =
  let n = ref 0 in
  for i = 0 to Array.length g - 1 do
    n := !n + List.length g.(i)
  done;
  !n / 2
;;
```

Question 2 Le cycle $G_{3,2}$ se représente ainsi :



On en déduit directement son tableau de tableaux d'adjacence :

```
let g32 =
  [|
    [| 1; 3 |];
    [| 0; 2; 4 |];
    [| 1; 5 |];
    [| 0; 4 |];
    [| 1; 3; 5 |];
    [| 2; 4 |];
  |]
;;
```

Question 3 On utilise la fonction `Array.of_list`, comme suggéré par l'énoncé.

```
let adjacence g =
  let n = Array.length g in
  let adj = Array.make n [||] in
  for i = 0 to n - 1 do
    adj.(i) <- Array.of_list g.(i)
  done;
  adj
;;
```

Voici deux autres versions, de plus en plus courtes, de la même fonction :

```
let adjacence g =
  let n = Array.length g in
  Array.init n (fun i -> Array.of_list g.(i))
;;

let adjacence g = Array.map Array.of_list g ;;
```

Question 4 On a un total de $p(q-1)+q(p-1)$ arêtes réparties de la façon suivante :

- ⋆ les $p(q-1)$ premières sont regroupées en p lignes de $q-1$ arêtes verticales ;
- ⋆ les $q(p-1)$ suivantes sont regroupées en q lignes de $p-1$ arêtes horizontales.

En récupèrant l'abscisse et l'ordonnée du nœud de départ (qui sont respectivement le reste et le quotient de la division euclidienne de son numéro par p), on trouve facilement le rang de l'arête.

```
let rang (p, q) (s, t) =
  let x = s mod p
  and y = s / p in
  if t = s + 1 then (* arête horizontale *)
    (p * (q - 1)) + x + ((p - 1) * y)
  else (* arête verticale *)
    y + ((q - 1) * x)
;;
```

Question 5 De façon similaire à la question précédente, on distingue si une arête est horizontale ou verticale, et on récupère les coordonnées puis les numéros des sommets à l'aide de d'une division euclidienne.

```
let sommets (p, q) r =
  if r >= p * (q - 1) then (* Arête horizontale *)
    let x = (r - (p * (q - 1))) mod (p - 1)
    and y = (r - (p * (q - 1))) / (p - 1) in
    let s1 = x + (y * p) in
```

```
    let s2 = s1 + 1 in
    (s1, s2)
  else (* Arête verticale *)
    let y = r mod (q - 1)
    and x = r / (q - 1) in
    let s1 = x + (y * p) in
    let s2 = s1 + p in
    (s1, s2)
;;
```

Question 6 On construit le quadrillage en parcourant toutes les arêtes à l'aide de la fonction précédente, en ajoutant pour chaque extrémité de l'arête l'autre arête dans la bonne liste d'adjacence.

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

II. Caractérisation des arbres

1. Propriétés sur les arbres

Question 7 Cela découle directement du fait que l'existence d'un chemin entre deux arêtes forme une relation d'équivalence dont les classes sont précisément les composantes connexes.

Question 8 Si $t \in C_s$, alors t et s sont dans la même composante connexe, et donc l'ensemble des chemins reliant ces deux sommets est non vide. Parmi ces chemins, il en existe un de longueur minimale.

De plus, si un chemin passe deux fois par le même sommet, il contient une boucle. Si l'on supprime cette boucle, on obtient une chemin plus court de mêmes extrémités. Par contraposition, tous les sommets d'un chemin de longueur minimale sont distincts.

Question 9 Si les extrémités de a_k étaient dans la même composante connexe de G_k, alors l'ajout de l'arête a_k créérait une cycle que l'on retrouverait dans G, ce qui est impossible puisque G est un arbre, donc acyclique. Ainsi, les extrémités de a_k sont, dans G_k, dans deux composantes connexes distinctes.

Ainsi, puisque 1. G_0 comporte n composantes connexes (autant que de sommets), 2. l'ajout de chaque arête relie deux sommets dans des composantes connexes et donc fait baisser le nombre de composantes connexes de 1, et 3. après l'ajout de m arêtes, le nombre final de composantes connexes est égal à 1, on en déduit que :

$$n - m = 1 \qquad \text{soit, de façon équivalente,} \qquad m = n - 1.$$

Question 10 Si G est un arbre, alors il est connexe et acyclique par définition, et on vient de montrer que $m = n - 1$. Ainsi, on a prouvé les implications (i) $\implies$ (ii) et (i) $\implies$ (iii).

Pour prouver les implications réciproques, notons tout d'abord $d(s)$ le degré d'un sommet s. Dans un graphe à n sommets et m arêtes, on a :

$$\sum_{s \in S_n} d(s) = 2m$$

En particulier, si $d(s) \geq 2$ pour tout sommet $s \in S_n$, alors $m \geq n$. Réciproquement, si $m < n$, alors il existe au moins un sommet de degré au plus 1.

Prouvons maintenant par récurrence sur le nombre $n \geq 1$ de sommets que tout graphe G connexe tel que $m = n - 1$ est acyclique. Pour $n = 1$ (et donc $m = 0$), c'est évident. Supposons maintenant le résultat vérifié à l'ordre n, et soit G un graphe connexe comportant $n+1$ sommets et m arêtes. D'après la remarque précédente, il existe un sommet s de degré au plus 1. Par connexité de G, le degré est en fait égal à 1. En supprimant ce sommet et l'arête correspondante, on obtient un graphe connexe à n sommets et $n-1$ arêtes et, par hypothèse de récurrence, ce graphe est acyclique. L'ajout d'un sommet et d'une arête allant vers ce nouveau sommet implique l'acyclicité de G.

On a donc prouvé que (ii) $\implies$ (i).

On prouve de façon similaire que (iii) $\implies$ (i) puisqu'étant donné un graphe acyclique G comportant $n \geq 2$ sommets et $m = n - 1$ arêtes, il existe un sommet s de G d'ordre 1 (le graphe étant acyclique, il suffit de considérer l'extrémité d'un chemin de longueur maximale). En otant ce sommet et l'arête associée, on en déduit par récurrence que l'on obtient un graphe connexe. En rajoutant un sommet et une arête le reliant à un autre sommet, on en déduit que G est connexe lui aussi.

2. Manipulation de partitions

Question 11 On itére la fonction représentée par le tableau jusqu'à trouver une valeur négative.

```
let rec representant t s =
  let s' = t.(s) in
  if s' < 0 then s else representant t s'
;;
```

Question 12 Il ne faut pas oublier, si $h(s) = h(t)$, de d'augmenter $h(t)$ de 1 du fait du « branchement » de s vers t.

```
let union p s t =
  let hs = -p.(s) - 1
  and ht = -p.(t) - 1 in
  if hs > ht then
    p.(s) <- t
  else if hs < ht then
    p.(t) <- s
  else begin
    (* hs = ht *)
    p.(s) <- t;
    p.(t) <- p.(t) - 1
  end
;;
```

Question 13 Soient deux parties X_i et X_j de représentants s et t. On suppose que $|X_i| \geq 2^{h(s)}$ et $|X_j| \geq 2^{h(t)}$ et supposons que l'on fusionne ces deux parties. Si $h(s) \neq h(t)$, alors on ne modifie pas la hauteur associée à représentant de la partie fusionnée, mais son cardinal a augmenté. L'inégalité est donc vérifiée.

Si, par contre, on a $h(s) = h(t)$, alors la nouvelle partie $X_i \cup X_j$ a pour cardinal $|X_i| + |X_j| \geq 2^{h(s)} + 2^{h(t)} = 2^{h(t)+1} = 2^{h'(t)}$ où $h'(t)$ désigne la hauteur après fusion.

Comme cette inégalité est vérifiée par toutes les parties de $\mathscr{P}_n^{(0)}$ (les parties sont de cardinal $1 \geq 2^0$), on en déduit que cette inégalité demeurent vérifiée par toute partie obtenue par l'opération de fusion que l'on vient de décrire.

Question 14 Inversement, on a pour toute partie X de représentant s,

$$h(s) \leq \log_2 |X|$$

Les opérations de recherche de représentant et de fusion étant de complexité en $O\big(h(s)\big)$ (remontée d'une branche pour la première, remontée de branche et branchement en temps constant pour la seconde), ces fonctions sont donc dans le pire des cas de complexité logarithmique en fonction de n.

Question 15 Pour déterminer si un graphe G à n sommets est un arbre, on procède ainsi :

1. on part de la partition $\mathscr{P}_n^{(0)}$;
2. pour chaque arête du graphe, on s'assure que les sommets extrémités sont dans des parties différentes (sinon, le graphe contient un cycle) et on les fusionne,
3. on vérifie qu'à la fin la partition est de cardinal 1 (sinon, le graphe n'est pas connexe).

```
let est_un_arbre g =
  let n = Array.length g in
  let p = Array.make n (-1) in
  let est_acyclique = ref true in
  for i = 0 to n - 1 do
    List.iter
      (fun j ->
        if j > i then
          (* on ne traite chaque arête qu'une fois *)
          let s = representant p i
          and t = representant p j in
          if s = t then
            est_acyclique := false
          else
            union p s t)
      g.(i)
  done;
  !est_acyclique && est_connexe p
;;
```

où l'on a au préalable défini la fonction `est_connexe` :

```
let est_connexe p =
  (* p ne doit avoir qu'une unique valeur négative *)
  let n = Array.length p in
  let c = ref 0 in
  for i = 0 to n - 1 do
    if p.(i) < 0 then incr c
  done;
  !c = 1
;;
```

Remarque : On ne s'arrête pas lorsque l'on trouve un cycle, alors que cela implique que le graphe n'est pas un arbre. On pourrait faire cela en remplaçant la boucle **`for`** *parcourant le tableau g par une fonction récursive, ce qui alourdirait la syntaxe. Une méthode plus légère et idiomatique serait de lever une exception stoppant le parcours, mais les exceptions ne sont pas au programme.*

III. Algorithme de Wilson : arbre couvrant aléatoire

Question 16 Le chemin représenté par l'objet `c` de l'énoncé par de `c.debut` = 1. Ayant `c.suivant.(1)` = 2, le sommet suivant est `c.suivant.(1)` = 2, et ainsi de suite jusqu'à atteindre le sommet `c.fin` = 4. Le chemin est donc :

$$1 \longrightarrow 2 \longrightarrow 5 \longrightarrow 4$$

Question 17 La terminaison de l'algorithme n'est pas assurée, puisque l'on peut recréer indéfiniment la même boucle.

Question 18 Commençons par une petit fonction qui choisit aléatoirement le voisin d'un sommet.

```
let sommet_aleatoire adj s =
  adj.(s).(Random.int (Array.length adj.(s)))
;;
```

La construction d'un chemin aléatoire suit alors la procédure décrite dans l'énoncé.

```
let marche_aleatoire adj parent s =
  let chemin =
    {
      debut = s;
      fin = s;
      suivant = Array.make (Array.length adj) (-1);
    }
  in
  (* tant que l'on n'a pas rejoint l'arbre *)
  while parent.(chemin.fin) = -2 do
    (* on détermine un nouveau sommet voisin de la fin *)
    let nouveau_sommet = sommet_aleatoire adj chemin.fin in
    (* on vérifie si l'on a créé une boucle *)
    (if chemin.suivant.(nouveau_sommet) = -1 then
       chemin.suivant.(chemin.fin) <- nouveau_sommet
    else (* on a trouvé une boucle, on la supprime *)
      let courant = ref nouveau_sommet in
      while !courant <> chemin.fin do
        let suivant = chemin.suivant.(!courant) in
        chemin.suivant.(!courant) <- -1;
        courant := suivant
      done);
    (* c'est la nouvelle fin du chemin *)
    chemin.fin <- nouveau_sommet
  done;
  chemin
;;
```

Question 19 On parcourt le chemin en mettant à jour le tableau `parent` à chaque sommet.

```
let greffe parent c =
  let courant = ref c.debut in
  while parent.(!courant) = -2 do
    let suivant = c.suivant.(!courant) in
    parent.(!courant) <- suivant;
    courant := suivant
  done
;;
```

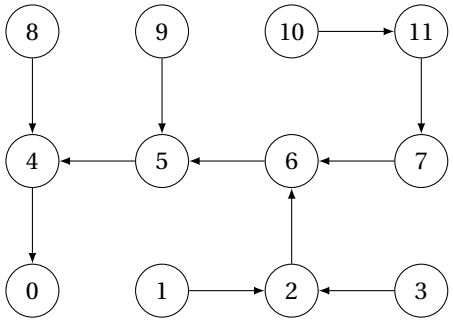
Question 20 Pour chaque sommet n'appartenant pas encore à l'arbre, on détermine un chemin qui l'y relie et on le greffe.

```
let wilson g r =
  let adj = adjacence g
  and n = Array.length g in
  let parent = Array.make n (-2) in
  parent.(r) <- -1;
  for i = 0 to n - 1 do
    if parent.(i) = -2 then
      let chemin = marche_aleatoire adj parent i in
      greffe parent chemin
  done;
  parent
;;
```

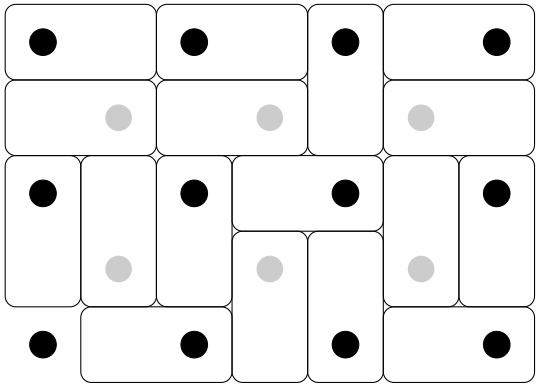
IV. Arbres couvrants et pavages par des dominos

1. Exemples

Question 21 On obtient l'arbre suivant :



Question 22 Le pavage correspondant est :



2. Calcul de l'arbre couvrant associé à un pavage

Question 23 Il s'agit du sommet « pointé » par le domino du sommet considéré.

Question 24 On réutilise les reste et quotient de la division euclidienne, comme expliqué plus haut.

```
let coord_noire c =
  (2 * (c mod p), 2 * (c / p)) ;;
```

Question 25 On utilise une fonction `case_noire` qui effectue l'opération inverse de `coord_noire`.

```
let case_noire x y = (x / 2) + (p * (y / 2)) ;;

let sommet_direction c d =
  let x, y = coord_noire c in
  let x', y' =
    match d with
    | S -> (x, y - 2)
    | W -> (x - 2, y)
    | N -> (x, y + 2)
    | E -> (x + 2, y)
  in
  case_noire x' y'
;;
```

Question 26 On remplit le tableau `parent` en indiquant, pour chaque case, la case désignée par la matrice de directions.

```
let phi pavage =
  let parent = Array.make (p * q) (-1) in
  for x = 1 to p - 1 do
    let c = case_noire x 0 in
    parent.(c) <- sommet_direction c pavage.(x).(0)
  done;
  for y = 1 to q - 1 do
    for x = 0 to p - 1 do
      let c = case_noire x y in
      parent.(c) <- sommet_direction c pavage.(x).(y)
    done
  done;
  parent
;;
```

V. Utilisation du dual pour la construction d'un pavage

1. Graphe dual de $G_{p,q}$

Question 27 Pour construire le graphe dual, on parcourt toutes les arêtes du graphe initial, on les « fait tourner d'un quart de tour » et obtient les arêtes du graphe dual.

```
let dual () =
  let nstar = 1 + ((p - 1) * (q - 1)) in
  let gstar = Array.make nstar [] in
  for i = 0 to (p * (q - 1)) + (q * (p - 1)) - 1 do
    (* pour chaque arête, *)
    let s1, s2 = sommets (p, q) i in
    (* on récupère les coordonnées des extrémités, *)
    let x1, y1 = coord_noire s1
    and x2, y2 = coord_noire s2 in
```

```
    (* on fait tourner l'arête d'un quart de tour *)
    let x3, x4 =
      if x1 = x2 then
        (x1 - 1, x1 + 1)
      else
        ((x1 + x2) / 2, (x1 + x2) / 2)
    and y3, y4 =
      if y1 = y2 then
        (y1 - 1, y1 + 1)
      else
        ((y1 + y2) / 2, (y1 + y2) / 2)
    in
    (* on enregistre la nouvelle arête *)
    let s3 = numero x3 y3
    and s4 = numero x4 y4 in
    gstar.(s3) <- s4 :: gstar.(s3);
    gstar.(s4) <- s3 :: gstar.(s4)
  done;
  gstar
;;
```

2. Dual d'un arbre couvrant

Question 28 Il est clair qu'il y a une correspondance bijective entre les faces du graphe dual et les sommets du graphe initial, ce que l'on vérifie en considérant les cases noires situées dans un coin, sur un bord ou à l'intérieur du pavage.

Question 29 Supposons que S_n, B admet un cycle $(s_0, s_1, \dots, s_r, s_0)$ et soit a_i l'arête du cycle reliant s_0 et s_1. On peut alors considérer l'arête duale $a_i^\star$ ainsi que les sommets duals $s^\star$ et $t^\star$ qu'elle relie.

Dans ce cas, il n'existe pas de chemin reliant $s^\star$ et $t^\star$ dans $(S_{n^\star}, B^\star)$ puisqu'un tel chemin contiendrait nécessairement une arête $a^\star$ croisant le cycle. Cela signifierait que l'arête duale a appartiendrait à B, ce qui est incompatible avec la définition de $B^\star$.

Question 30 Ayant vu que le dual de $G^\star_{p,q}$ est $G_{p,q}$, on en déduit que pour $B \subseteq A$, on a $B^{\star\star} = B$. La contraposée de l'implication de la question précédente peut s'écrire : si $(S_{n^\star}, B^\star)$ est connexe, alors (S_n, B) est acyclique. Mais en considérant leurs duals respectifs, on a: si $(S_{n^{\star\star}}, B^{\star\star}) = (S_n, B)$ est connexe, alors $(S_{n^\star}, B^\star)$ est acyclique.

Supposons donc que (S_n, B) est un arbre, il est connexe donc $(S_{n^\star}, B^\star)$ est acyclique. De plus, d'après la question 10, on a $|B| = n-1$. On en déduit que

$$\left|B^\star\right| = m - (n-1) = 2pq - p - q - pq - 1 = pq - p - q = n^\star - 1$$

Comme $B^\star$ est acyclique, on en déduit à nouveau d'après la question 10 que $(S_{n^\star}, B^\star)$ est un arbre, et même un arbre couvrant puisque l'ensemble de ses sommets est S_{n^star} en entier.

Question 31 En parcourant le tableau `parent`, soit on rencontre la valeur -1 ce qui indique que l'on est à la racine, ou bien on trouve l'autre extrémité d'un arête dont on trouve le numéro grâce à `rang`, et on mets la valeur correspondant à `true` dans `b`.

```
let vers_couple parent =
  let racine = ref (-1)
  and b =
    Array.make ((p * (q - 1)) + (q * (p - 1))) false
  in
  for i = 0 to (p * q) - 1 do
    if parent.(i) = -1 then
      racine := i
    else
      let arete =
        (* il faut faire attention à l'ordre
           des sommets en appelant rang *)
        if i < parent.(i) then
          rang (p, q) (i, parent.(i))
        else
          rang (p, q) (parent.(i), i)
      in
      b.(arete) <- true
  done;
  (!racine, b)
;;
```

Question 32 Pour reconstruire le tableau `parent` à partir de (r, B), on effectue un parcours de graphe (en profondeur, par exemple) à partir de r, les arêtes à visiter étant déterminés à partir des voisins du sommet courant et du tableau B.

Cette méthode n'est pas directement applicable à un arbre couvrant de $G^\star_{p,q}$, puisqu'alors il peut exister plusieurs arêtes reliant de même sommets.

Question 33 On applique le parcours que l'on vient de décrire. Un sommet `s` n'a pas encore été découvert si `parent.(s)` $= -2$.

```
let vers_parent (r, b) =
  let parent = Array.make (p * q) (-2) in
  parent.(r) <- -1;
  let rec visite s =
    List.iter
      (fun sommet ->
```

```
        let arete =
          if s < sommet then
            rang (p, q) (s, sommet)
          else
            rang (p, q) (sommet, s)
        in
        if parent.(sommet) = -2 && b.(arete) then begin
          parent.(sommet) <- s;
          visite sommet
        end)
      g.(s)
  in
  visite r;
  parent
;;
```

Question 34 La fonction `vers_couple` ne fait que parcourir le tableau `parent`, chaque valeur rencontrée étant en temps constant. Elle est donc en $O(n)$.

La fonction `vers_parent` repose elle sur un parcours en profondeur de graphe, sa complexité est donc en $O(n+m)$.

Question 35 D'après le lien entre B et $B^s tar$, il suffit, dans la représentation sous forme de couple, de prendre la négation de chaque booléen.

```
let arbre_dual parent =
  let r, b = vers_couple parent in
  for i = 0 to Array.length b - 1 do
    b.(i) <- not b.(i)
  done;
  vers_parent_etoile (0, b)
;;
```

3. Calcul du pavage associé à un arbre couvrant

Question 36 On l'a vu dans la partie IV, l'arbre $\mathcal{T}$ permet de placer les dominos couvrant une case noire. Reste à déterminer comment placer les dominos couvrant une case grise. C'est précisément le rôle de l'arbre dual $\mathcal{T}^\star$. En effet, chaque case blanche est recouverte par un domino recouvrant soit une case blanche, soit une case noire. Cela correspond exactement à une arête de $\mathcal{T}$ ou une arête de $\mathcal{T}^\star$ d'après la relation entre B et $B^\star$.

Il faut cependant au cas où un sommet dans un « coin » du graphe dual (dans l'exemple, il s'agit des sommets 1, 3, 4, et 6) est relié à 0 car il faut alors déterminer laquelle des deux arêtes utiliser.

Question 37 La fonction suivante est la traduction de l'algorithme précédent.

```
let pavage_aleatoire () =
  let t = wilson g 0 in
  let tstar = arbre_dual t in
  let pavage =
    Array.make_matrix ((2 * p) - 1) ((2 * q) - 1) N
  in
  (* on part de 1 car l'arbre est enraciné en 0 *)
  for i = 1 to (p * q) - 1 do
    let x1, y1 = coord_noire i
    and x2, y2 = coord_noire t.(i) in
    pavage.(x1).(y1) <- direction_de_sommet x1 y1 x2 y2
  done;
  (* les coins du graphe dual peuvent être faux *)
  for i = 1 to (p - 1) * (q - 1) do
    let x1, y1 = coord_grise i
    and x2, y2 = coord_grise tstar.(i) in
    pavage.(x1).(y1) <- direction_de_sommet x1 y1 x2 y2
  done;
  (* on traite les 4 coins séparément *)
  if tstar.(1) = 0 then
    pavage.(0).(0) <- oppose pavage.(1).(1);
  if tstar.(p - 1) = 0 then
    pavage.((2 * p) - 3).(1) <-
      oppose pavage.((2 * p) - 2).(0);
  if tstar.(((p - 1) * (q - 2)) + 1) = 0 then
    pavage.(1).((2 * q) - 3) <-
      oppose pavage.(0).((2 * q) - 2);
  if tstar.((p - 1) * (q - 1)) = 0 then
    pavage.((2 * p) - 3).((2 * q) - 3) <-
      oppose pavage.((2 * p) - 2).((2 * q) - 2);
  pavage
;;
```

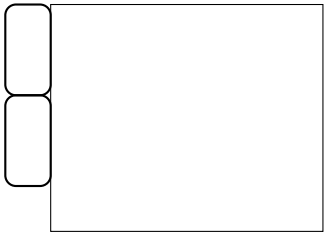
On a au préalable défini les fonctions utilitaires `direction_de_sommet` et `oppose` :

```
let direction_de_sommet x1 y1 x2 y2 =
  if x2 = x1 + 2 then
    E
  else if x2 = x1 - 2 then
    W
  else if y2 = y1 + 2 then
    N
```

```
  else
    S
;;

let oppose direction =
  match direction with
  | N -> S
  | S -> N
  | W -> E
  | E -> W
;;
```

Question 38 On peut adapter cette méthode pour paver un échiquier de taille $2p \times (2q-1)$ en ajoutant à gauche une colonne dont on otera le carré en bas. On se retrouve donc dans la configuration précédente. On peut alors chercher les pavages en pré-remplissant la colonne de gauche comme indiqué sur la figure ci-dessous :



Dans le cas d'un échiquier de taille $2p \times 2q$, on adapte l'idée précédente en ajoutant une colonne à gauchet et une ligne en bas.