I - Quelques fonctions auxiliaires

1. On commence par écrire une fonction qui donne la longueur d'une liste (on peut aussi utiliser `List.length`).

```
let rec longueur l = match l with
 | [] -> 0
 | t ::q -> 1 + longueur q;;
```

On peut alors écrire la fonction. Il suffit, à l'aide d'une boucle `for` de faire la somme des longueurs des listes.

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

2. On écrit une commande pour créer le graphe $G_{3,2}$:

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

3. On transforme chaque liste d'adjacence par un tableau d'adjacence grace à la fonction `Array.of_list`

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

4. Il faut déterminer le rang r d'une arêtes $\{s,t\}$.
On voit qu'il y a $p(q-1)$ arêtes verticales qui sont numérotées de 0 a $pq-p-1$, il y en a $q(p-1)$ qui sont horizontales, numérotées de $pq-p$ à $2pq-p-q-1$.
On remarque que si $t = s+1$ on a une arête horizontale que sinon l'arête est verticale.
Dans le premier cas, on récupère le numéro i de la ligne par $i = s/p$ puis l'emplacement dans la ligne par $j = s \bmod p$.
Le rang est alors

$$r = pq - p + i(p-1) + j$$

car la première arête horizontale est $pq-p$.
Dans le deuxième cas on récupère encore le numéro i de la ligne par $i = s/p$ puis l'emplacement dans la ligne par $j = s \bmod p$. Le rang est alors

$$r = j(q-1) + i$$

```
let rang (p, q) (s, t) = let i = s / p and j = s mod p in
 if t = s + 1 then p * q - p + i * (p - 1) + j else j * (q - 1) + i;;
```

5. Il suffit d'inverser les formules ci-dessus

```
let sommets (p, q) r = if r >= p * (q - 1) then
  (let u = r - p * (q - 1) in
   let i = u / (p - 1) and j = u mod (p - 1) in
   let s = p * i + j in (s, s + 1))
 else
  (let j = r / (q - 1) and i = r mod (q - 1) in
   let s = p * i + j in (s, s + p));;
```

6. On commence par créer le tableau qui contient des listes des vides. Ensuite, pour chaque k, on ajoute ses voisins dans la liste `g.(k)`. Il faut séparer le cas selon que le sommet est « en bas », « en haut », « à droite », « à gauche » ou au centre du graphe.

```
let quadrillage p q =
 let g = Array.make (p * q) [] in
 for k = 0 to p * q - 1 do
  if k mod p != 0 then g.(k) <- (k - 1) :: g.(k);
  if k mod p != (p - 1) then g.(k) <- (k + 1) :: g.(k);
  if (k / p) != 0 then g.(k) <- (k - p) :: g.(k);
  if (k / p) != (q - 1) then g.(k) <- (k + p) :: g.(k);
 done;
 g;;
```

II - Caractérisation des arbres

7. On considère sur l'ensemble S_n la relation binaire $\sim$ définie par

$$\forall (s,t) \in S_n^2, s \sim t \iff (\text{ il existe un chemin de } s \text{ vers } t)$$

Vérifions que la relation $\sim$ est une relation d'équivalence.
- Réflexivité : soit $s \in S_n$, par convention il existe un chemin de longueur nulle de s à s. On a $s \sim s$.
- Symétrie : soit $(s,t) \in S_n^2$. On suppose que $s \sim t$. Il existe donc un chemin $c = (s = s_0, s_1, \dots, s_k = t)$ de s à t. On voit alors que $\tilde{c} = (s_k = t, s_{k-1}, \dots, s_1, s_0 = s)$ est un chemin de t à s car, pour tout $i \in [\![1,k]\!]$, s_i et s_{i-1} sont voisins. On a bien $t \sim s$.
- Transitivité : soit $(s,t,u) \in S_n^3$. On suppose que $s \sim t$ et $t \sim u$. Il existe donc des chemins $c_1 = (s = s_0, s_1, \dots, s_k = t)$ et $c_2 = (t = t_0, t_1, \dots, t_r = u)$ dans le graphe. On voit que $\gamma = (s = s_0, s_1, \dots, s_k, t_1, \dots, t_r = u)$ est alors un chemin de s à u.

Pour tout sommet s, $C_s = \{t \in S_n \ , \ s \sim t\}$ est la classe d'équivalence de s. On sait alors (cours de maths de sup) que les classes d'équivalence d'une relation d'équivalence forment une partition de l'ensemble [1].

8. Soit $s \in S_n$ et $t \in C_s$. Par définition, il existe un chemin de s à t. Notons $A = \{\text{longueur de } c \ , \ c \text{ chemin de } s \text{ à } t\}$ l'ensemble des longueurs des chemins de s à t. L'ensemble A est une partie non vide de $\mathbb{N}$. Elle admet donc un plus petit élément. Il existe bien un chemin de s à t de longueur minimal.

Notons $c = (s = s_0, s_1, \dots, s_k = t)$ un chemin de longueur minimal. On montrer que les sommets de c sont deux à deux distincts. Supposons par l'absurde qu'il existe $0 \leq i < j \leq k$ avec $s_i = s_j$ alors $\tilde{c} = (s = s_0, \dots, s_i = s_j, s_{j+1}, \dots, s_k = t)$ est un chemin de s à t strictement plus court que c. C'est absurde.

Cela montre que les sommets d'un chemin de s à t de longueur minimale sont deux à deux distincts.

9. On suppose que G est un arbre. Supposons par l'absurde qu'il existe $k \in [\![0, m-1]\!]$ tel que les deux extrémités de a_k que nous noterons s et t appartiennent à la même composante connexe de G_k. En particulier, il existe un chemin qui relie s et t dans G_k. En choisissant un chemin de longueur minimale, les arêtes de ce chemin $c = (s = s_0, s_1, \dots, s_k = t)$ sont deux à deux distincts. On peut alors, dans G, considérer le cycle :

$$\tilde{c} = (s = s_0, s_1, \dots, s_k = t, s)$$

où la dernière arête qui va de s à t est l'arête a_k. Ce chemin est bien un cycle car ses arêtes sont deux à deux distinctes puisque les arêtes de c sont deux à deux distinctes et appartiennent à $\{a_0, a_1, \dots, a_{k-1}\}$ puisque c est un chemin dans le graphe G_k. C'est absurde car G est acyclique puisque c'est un arbre.

Finalement, pour tout $k \in [\![0, m-1]\!]$, les extrémités de a_k sont dans des composantes connexes différentes de G_k.

1. On peut refaire la démonstration.
- Pour tout $s \in S_n$, $C_s \neq \emptyset$ car $s \in C_s$.
- Pour tout $s_0 \in S_n$, $s_0 \in \bigcup_{s \in S_n} C_s$ car $s_0 \in C_{s_0}$.
- Soit s, t dans S_n. Si $s \sim t$ alors $C_s = C_t$ par la transitivité de la relation. Par contre si s n'est pas en relation avec t alors $C_s \cap C_t = \emptyset$ car s'il existait $u \in C_s \cap C_t$ alors $s \sim u$ et $t \sim u$ et donc $s \sim t$.

On remarque que $G_{-1} = (S_n, \emptyset)$ a n composantes connexes (une pour chaque sommet). Or, pour tout $k \in [\![-1, m-1]\!]$, G_{k+1} a une composante connexe de moins de G_k car ajouter l'arête a_k permet de fusionner les composantes connexes auxquelles appartenaient les extrémités de a_k. Cela montre que pour $k \in [\![0, m]\!]$, G_k a $n-k$ composantes connexes. Le graphe G a donc $n-m$ composantes connexes. Comme G est un arbre, $n-m=1$ c'est-à-dire, $m = n-1$.

10. — $\boxed{\text{(i)} \Rightarrow \text{(ii)} \text{ et } \text{(i)} \Rightarrow \text{(iii)}}$ Évident d'après la question précédente
    — $\boxed{\text{(iii)} \Rightarrow \text{(i)}}$ En reprenant la question précédente, on voit que dans la première partie de la question, on a juste utilisé que G était acyclique. En reprenant les notations de la question, on obtient que si G est acyclique alors G_k a $n-m$ composantes connexes. En particulier si $n = m-1$ on obtient que G est connexe. C'est donc un arbre.
    — $\boxed{\text{(ii)} \Rightarrow \text{(i)}}$ On suppose que G est connexe et que $m = n-1$. Supposons par l'absurde que ce n'est pas un graphe ce qui revient à supposer que G possède au moins un cycle. Soit $c = (s_0, s_1, \ldots, s_k = s_0)$ un cycle. Pour $i \in [\![1, k]\!]$ notons α_i l'arête $\{s_{i-1}, s_i\}$. Considérons alors le graphe $\tilde{G}$ obtenu en enlevant l'arête α_1. Il est encore connexe. En effet pour s, t deux sommets de G. Il existe un chemin γ dans G qui relie s à t car G est connexe. Si ce chemin n'emprunte pas l'arête α_1 alors γ est un chemin dans $\tilde{G}$. Si γ emprunte l'arête α_1, on peut quand même relier s à t dans le graphe $\tilde{G}$ en remplaçant l'arête α_1 par le chemin de s_1 à s_0 donné par

$$s_1 \overset{\alpha_2}{—} s_2 \overset{\alpha_3}{—} \cdots \overset{\alpha_k}{—} s_k = s_0$$

    En procédant ainsi, on peut enlever des arêtes jusqu'à obtenir un graphe acyclique qui sera encore connexe. Ce sera un arbre mais son nombre d'arêtes sera strictement inférieur à $n-1$ ce qui est absurde.
    Cela montre donc qu'un graphe connexe où $m = n-1$ est acyclique et donc est un graphe.

11. Soit `tab` un tableau représentant la partition et `s` un sommet. Si `tab.(s)` est strictement négatif alors `s` est le représentant cherché. Sinon, on renvoie le représentant de `tab.(s)` car `tab.(s)` est le père de `s`. Ils ont donc le même représentant.

```
let rec representant tab s =
if tab.(s) < 0 then s else representant tab tab.(s);;
```

12. Pour fusionner deux parties dont les représentants sont `s` et `t`, on compare $h(s)$ et $h(t)$ en utilisant `tab.(s)` et `tab.(t)`.
    — Si `-1-tab.(s)` > `-1-tab.(t)`, `s` devient le père de `t`. La hauteur $h(s)$ ne change pas.
    — Si `-1-tab.(t)` > `-1-tab.(s)`, `t` devient le père de `s`. La hauteur $h(t)$ ne change pas.
    — Si `-1-tab.(s)` = `-1-tab.(t)`, `t` devient le père de `s`. La hauteur $h(t)$ est incrémenté de 1 donc `tab.(s)` est diminué de 1.

```
let union tab s t =
if tab.(s) < tab.(t) then tab.(t) <- s
else if tab.(s) = tab.(t) then
  (tab.(s) <- t ;
  tab.(t) <- tab.(t)-1)
  else
  tab.(s)<- t ;;
```

13. Soit $k \in [\![1, n]\!]$. On note H_k le prédicat :

$$H_k : \text{« pour toute partie } X \text{ de cardinal } k,\ k \geq 2^{h(s)} \text{ où } s \text{ est le représentant de } X \text{ »}.$$

Montrons par récurrence forte que pour tout $k \in [\![1, n]\!]$, H_k est vrai.
— Initialisation : Pour $k = 1$. Les singletons sont représentés par des arbres de hauteur 0. On a bien $1 \geq 2^0$.
— Hérédité : Soit $d \in [\![2, n]\!]$, on suppose H_k vrai pour tout $k < d$. Montrons H_d. Soit X une partie de cardinal d. Elle est obtenue par fusion de deux parties de cardinal respectif k et k'. Notons h et h' les hauteurs des arbres représentant ce deux parties. On a, par hypothèse de récurrence, $k \geq 2^h$ et $k' \geq 2^{h'}$. Par symétrie, on peut supposer $h \geq h'$.
Si $h > h'$, la hauteur de l'arbre qui représente X est encore h et on a

$$d = k + k' \geq 2^h + 2^{h'} \geq 2^h$$

Si $h = h'$, la hauteur de l'arbre qui représente X est alors $h+1$ et on a

$$d = k + k' \geq 2^h + 2^{h'} = 2^h + 2^h \geq 2^{h+1}$$

— On a bien H_k vérifié pour k compris entre 1 et n.

14. Le calcul précédent permet de montrer que les hauteurs qui interviennent ne dépassent jamais $\log_2(n)$.
On voit alors que la complexité de la fonction `representant` est linéaire en la hauteur des arbres et donc en $O(\log_2(n))$. La fonction `union` s'exécute en temps constant. Cependant, on peut estimer que l'on ne connait pas nécessairement les représentants des parties que l'on veut fusionner. On peut donc d'abord avoir à chercher les représentants ce qui donnerait encore une complexité en $O(\log_2(n))$.

15. Pour savoir si un graphe `g` est un graphe on va, en utilisant la question Q10, vérifier si $n = m - 1$ et s'il est connexe. Pour étudier la connexité, on va utiliser un tableau `tab` qui va représenter les composantes connexes du graphe G_k (on va considérer les arêtes les unes après les autres). On calculera en parallèle le nombre de composantes connexes dans une variables `nb`.

```
let est_un_arbre g =
let n = Array.length g in
let m = nombre_aretes g in
if n <> m-1 then false
else (let tab = Array.make n (-1) in
  let nb = ref n in
  let rec parcours l i = match l with
    | [] -> ()
    | t :: q -> let rt = representant tab t and ri = representant tab i in
        if rt <> ri then
        (nb := !nb -1 ;
        union tab rt ri)
  in
for k = 0 to (n-1) do
  parcours g.(k) k
done;
!nb = 1);;
```

III - Algorithmes de Wilson : arbre couvrant aléatoire

16. Le chemin est $c = (1, 2, 5, 4)$.

17. Le graphe étant connexe, en partant d'un sommet s qui n'est pas dans $\mathcal{T}$, la marche aléatoire va presque surement atteindre un sommet de $\mathcal{T}$.

18. On applique l'algorithme proposé. La fonction récursive `parcours : chemin -> chemin` modifie le chemin. Initialement le chemin ne contient que le sommet s.
On récupère le nombre de voisin du dernier sommet du chemin (noté `nbv`) et on tire un sommet au hasard (noté `nvs`). On rajoute ce sommet au chemin (et on modifie la fin du chemin). Si ce sommet est dans $\mathcal{T}$, c'est-à-dire que `parent.(nvs)` ne vaut pas -2 on le renvoie le chemin, sinon on continue.

```
let marche_aleatoire adj parent s =
let n = Array.length adj in
let rec parcours c =
  let nbv = Array.length adj.(c.fin) in
  let nvs = adj.(c.fin).(Random.int nbv) in
  c.suivant.(c.fin) <- nvs;
  c.fin <- nvs;
  if parent.(nvs) <> (-2) then
    c
  else
    parcours c
in
  parcours {debut = s ; fin = s ; suivant = Array.make n (-1)};;
```

19. On parcourt le chemin grâce à la fonction `parcours : int -> unit`. Tant que l'on n'est pas arrivé à la fin du chemin, on modifie le tableau `parent` en récupérant le sommet suivant dans le chemin.

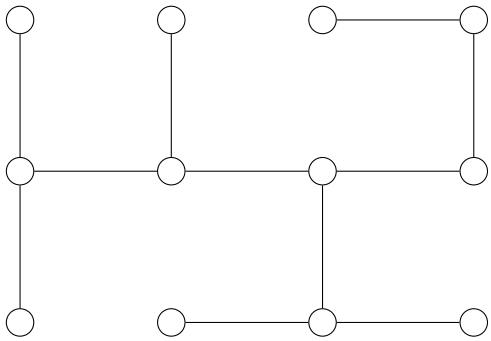
```
let greffe parent c =
  let rec parcours s =
    if s <> c.fin then
    (parent.(s) <- c.suivant.(s) ;
    parcours c.suivant.(s))
    in
  parcours c.debut;;
```

20. On applique l'algorithme.

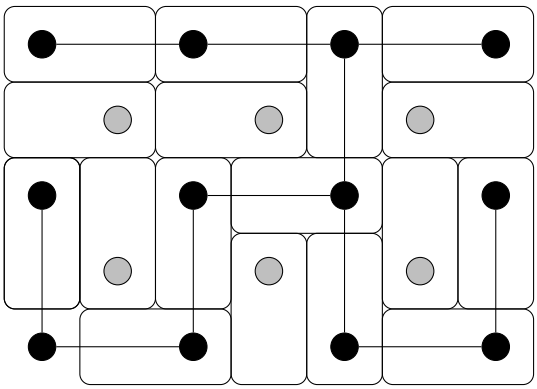
```
let wilson g r =
  let n = Array.length g in
  let parent = Array.make n (-2) in
  parent.(r) <- (-1);
  let adj = adjacence g in
  for s = 0 to (n-1) do
    if parent.(s) = -2 then
    (greffe parent (marche_aleatoire adj parent s))
  done;
  parent;;
```

IV - Arbres couvrants et pavage par des dominos

21. On obtient l'arbre couvrant suivant



22. On obtient le pavage suivant :



23. Soit $s \in \{1, \dots, pq-1\}$ un sommet du graphe $G_{p,q}$ autre que le sommet 0 (ce qui correspond à une case noire de l'échiquier $E'_{p,q}$). Pour trouver le père de s il suffit de se déplacer dans la direction du domino. Précisément en notant (i, j), les coordonnées de la case de l'échiquier qui correspond au sommet s,
    - Si `pavage.(i).(j) = W` le père de s est $s-1$
    - Si `pavage.(i).(j) = E` le père de s est $s+1$
    - Si `pavage.(i).(j) = N` le père de s est $s+p$
    - Si `pavage.(i).(j) = S` le père de s est $s-p$

24. On réalise des divisions euclidiennes et on multiplie le résultat par deux.

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

25. Si la direction est `N` ou `S` on change de ligne en ajoutant ou enlevant p. Sinon on ajoute ou enlève 1.

```
let sommet_direction s d =
let (i,j) = coord_noire s in
match d with
  | N -> if j = (2*q-2) then (-1) else s+p
  | S -> if j = 0 then  (-1) else s-p
  | E -> if i = (2*p-2) then (-1) else s+1
  | W -> if i = 0 then (-1) else s-1;;
```

26. On créé le tableau `parent`. Pour chaque sommet autre de 0, on détermine son père grâce à la fonction précédente.

```
let phi pavage =
let parent = Array.make (p*q) (-1) in
for s = 1 to (p*q)-1 do
  let (i,j) = coord_noire s in
  parent.(s) <- sommet_direction s pavage.(i).(j)
done;
parent;;
```

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

Voici les deux fonctions dont on ne demandait par l'écriture

```
let coord_grise s =  2*((s-1) mod (p-1))+1, 2*((s-1)/(p-1))+1;;
```

```
let numero (x,y) =
if (x mod 2)=0 && (y mod 2)=0 && (x < 2*p-1) && (0 <= x) && (y < 2*q-1) && (0 <= y)
then p*(y/2) + x/2
else if (x mod 2)=1 && (y mod 2)=1 && (x < 2*p-1) && (0 <= x) && (y < 2*q-1) && (0 <= y)
    then (p-1)*(y-1)/2 + (x-1)/2 +1
    else 0;;
```

27. On parcourt tous les couples (x, y) qui correspondent à des cases grises (les deux coordonnées sont impaires). À chaque couple on associe un sommet s grâce à la fonction numéro. Le sommet s a quatre voisins qui correspondent aux cases de coordonnées $(x-2, y)$, $(x, y-2)$, $(x+2, y)$ et $(x, y+2)$. La fonction numéro renvoyant 0 quand ces cases ne sont pas dans l'échiquier.
    On termine en ajoutant dans `g.(0)` tous les sommets ayant un arête vers le sommet 0.

```
let dual () =
let g = Array.make nstar [] in
for u = 0 to (p-2) do
  for v = 0 to (q-2) do
    let x = 2*u+1 and y = 2*v+1 in
    let s = numero (x,y) in
    g.(s) <- [numero ((x-2),y) ; numero (x,(y-2)) ; numero (x+2,y) ; numero (x,y+2)];
    if u = 0 || u= (p-2) then g.(0) <- s :: g.(0);
    if v = 0 || v= (q-2) then g.(0) <- s :: g.(0);
  done;
done;
g;;
```

28. Dans le graphe $G^*_{p,q}$ il y a $(p-1)(q-1)$ sommets qui sont issus des faces « internes » du graphe $G_{p,q}$. Cela va donner naissance à $(p-2)(q-2)$ faces internes au graphe $G^*_{p,q}$. Elles correspondent aux sommets de $G_{p,q}$ qui ne sont pas au bord du graphe. Maintenant, pour chaque sommet s du graphe $G_{p,q}$ qui n'est pas un coin (par exemple le sommet 4 de la figure 9), il y a deux sommets t et t' du dans $G^*_{p,q}$ qui sont issus d'une face ayant le sommet s comme angle.

Les sommets 0, t et t' du graphe $G^*_{p,q}$ vont donc créer une face que l'on fait correspondre au sommet s. On va créer ainsi $2(p-2+q-2)$ sommets

Pour finir si s est un angle du graphe $G_{p,q}$, il n'y a qu'un seul sommet t dans le graphe $G^*_{p,q}$ qui lui correspond mais ce sommet t a deux arêtes vers 0 ce qui permet donc de créer encore une face. On va créer ainsi 4 sommets.

Au total, le graphe dual de $G^*_{p,q}$ aura $pq-2p-2q+4+2p+2q-8+4 = pq$ sommets.

Finalement, chaque face du graphe $G^*_{p,q}$ correspond bien à un sommet et un seul de $G_{p,q}$. On peut vérifier aussi que les arêtes correspondent.

29. On suppose que (S_n, B) possède un cycle. On considère un cycle de longueur minimale dans (S_n, B). Il délimite les sommets de $G^*_{p,q}$ en deux parties : la partie intérieure au cycle que nous noterons U et la partie extérieure que nous noterons V. Ces deux parties ne sont pas vides. Comme B^* est constitué des arêtes a^* telles que $a \notin B$, on ne peut pas trouver un chemin qui relie un sommet de U à un sommet de V. Pour être plus précis, si on note s un sommet de U et t un sommet de V. En supposant qu'il existe une suite d'arêtes de B^* qui relie s à t, il existe nécessairement une arête $a^* \in B^*$ reliant un sommet s' de U à un sommet t' de V (par exemple t' est le premier sommet de V du chemin et s' est le sommet précédent). Cela n'est possible car cela implique que $a \in B$.

30. Supposons que (S_n, B) est un arbre couvrant. Il est sans cycle donc, d'après la question précédente, le graphe (S_{n^*}, B^*) est connexe. De plus, on sait d'après Q10 que $\#B = n-1 = pq-1$ donc

$$\#B^* = m - \#B = p(q-1) + q(p-1) - pq + 1 = n^* - 1$$

Cela implique, toujours d'après Q10 que (S_{n^*}, B^*) est un arbre couvrant.

Réciproquement, si on suppose que (S_{n^*}, B^*) est un arbre couvrant, en utilisant que $G_{p,q}$ est le dual de $G^*_{p,q}$, on obtient que $(S_n, (B^*)^*) = (S_n, B)$ est connexe (la connexité n'est remis en cause par la renumérotation des sommets). En utilisant le même argument de cardinalité on obtient que (S_n, B) est un arbre couvrant.

31. On parcourt le tableau `parent`. Si `parent.(s)` vaut -1 c'est que le sommet s est la racine de l'arbre. On le stocke dans la référence `r`. Dans le cas contraire, il y a une arête entre le sommet `s` et le sommet `parent.(s)`. On récupère son numéro grace à la fonction `rang` de la question Q4.

```
let vers_couple parent =
let b = Array.make m false in
let r =ref (-1) in
for s = 0 to (n-1) do
  if parent.(s) = -1 then r := s
  else b.(rang (p,q) (s,parent.(s))) <- true
done;
(!r,b);;
```

32. On fait un parcours en profondeur de l'arbre couvrant en partant de la racine. Plus précisément, notons p la pile des sommets à parcourir et V l'ensemble des arêtes qui ont été considérée. On initialise p à $\{r\}$ et V à vide.

Tant que la pile n'est pas vide, on considère le sommet s qui est au sommet de la pile. On regarde alors toutes les arêtes $a = \{s, t\}$ ayant s pour extrémité. Si cette arête est dans l'arbre couvrant (si `b.(a)` vaut `true`) et qu'elle n'est pas encore dans V, on ajoute a à V, on affecte s comme parent de t et on ajoute t à la pile.

33. Cela donne la fonction suivante où le tableau `visit` sert à savoir si une arête appartient à V ou non.

```
let vers_parent (r,b) =
let parent = Array.make n (-1) in
let visit = Array.make m false in
let adj = adjacence (quadrillage p q) in
let rec parcours s =
for k = 0 to (Array.length adj.(s)) - 1 do
let u = min s (adj.(s).(k)) and v = max s (adj.(s).(k)) in
  let i = rang (p,q) (u,v) in
  if b.(i) && (not visit.(i)) then
  (parent.(adj.(s).(k)) <- s;
  visit.(i) <- true;
  parcours adj.(s).(k)
  )
done;
in parcours r;
parent;;
```

34. Pour la fonction `vers_couple`, la complexité est en $O(n)$ du fait de la boucle `for`.
Pour la fonction `vers_parent` on réalise un parcours en profondeur du graphe. La complexité est en $O(n+m)$.

35. Par définition, de B^* :

```
let arbre_dual parent =
let (r,b) = vers_couple parent in
let b_etoile = Array.make m true in
for k = 0 to (m-1) do
  b_etoile.(k) <- not b.(k)
done;
vers_parent_etoile (r,b_etoile);;
```