

I Quelques fonctions auxiliaires

1. Attention à ne pas compter chaque arête deux fois.

```
let nombre_aretes g =
  let res = ref 0 in
  for i = 0 to Array.length g - 1 do
    res := !res + List.length g.(i)
  done;
  !res/2 (* on a compté chaque arête 2 fois *)
```

Autre possibilité :

```
let nombre_aretes g =
  (Array.map List.length g
   |> Array.fold_left (+) 0) / 2;;
```

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

3. Une possibilité : `let adjacence g = Array.map Array.of_list g`

Si n est le nombre de sommets et m le nombre d'arêtes de g , `Array.map Array.of_list g` crée un nouveau tableau de taille n (complexité $O(n)$) puis le remplit en appliquant `Array.of_list` sur chaque élément de g . Comme `Array.of_list g.(i)` demande autant d'opérations que le degré du sommet i , `Array.map Array.of_list g` demande au total $\sum_i \deg(i) = 2m =$

$O(m)$. Il y a donc bien une complexité $\boxed{O(n + m)}$ au total.

- 4.
- 5.
- 6.

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

II Caractérisation des arbres

7.
 - $s \in C_s$ donc $C_s \neq \emptyset$
 - $S_n \subseteq \bigcup C_s$ car si $s \in S_n$ alors $s \in C_s$. $C_s \subseteq S_n$ par définition donc $\bigcup C_s \subseteq S_n$. Donc $S_n = \bigcup C_s$.
 - Soient $s, t \in S_n$. Supposons $C_s \cap C_t \neq \emptyset$ et montrons $C_s = C_t$. Comme $C_s \cap C_t \neq \emptyset$, il existe un sommet $u \in C_s \cap C_t$. Comme $u \in C_s$, il existe un chemin C_1 de u à s . De même, il existe un chemin C_2 de u à t . En concaténant C_1 et C_2 , on obtient un chemin C de s à t . Alors, si $v \in C_s$, la concaténation d'un chemin de v à s et de C donne un chemin de v à t , ce qui montre $C_s \subseteq C_t$. De même, on montre $C_t \subseteq C_s$ et donc $\boxed{C_s = C_t}$.
8. Soit $\mathcal{C} = \{ \text{longueur de } C \mid C \text{ est un chemin de } s \text{ à } t \}$. Comme $t \in C_s$, $\mathcal{C} \neq \emptyset$. Comme \mathcal{C} est un sous-ensemble non vide de \mathbb{N} , il possède un minimum. Notons C un chemin réalisant ce minimum. Supposons que C passe plusieurs fois par le même sommet u . Alors on peut décomposer C en un chemin C_1 de s à u , puis un chemin C_2 partant de u et revenant en u , puis un chemin C_3 de u vers t . Alors, en supprimant C_2 , obtient un chemin de s vers t (composé de C_1 et C_3) de longueur strictement inférieure à C , ce qui est absurde. Donc les sommets de C sont distincts.
9. Supposons par l'absurde que les extrémités u, v de a_k soient dans la même composante connexe. Alors il existe un chemin C dans G_k de u à v . La concaténation de C et de a_k donne un cycle. Ce cycle existe aussi dans G , ce qui est absurde pour

un arbre.

Comme il y a initialement n composantes connexes, que chaque ajout d'arête diminue de 1 le nombre de composante connexe et qu'on obtient un arbre G avec 1 composante connexe (car G est un arbre donc connexe donc possède 1 seule composante connexe), $n - 1$ arêtes ont été ajoutées : $m = n - 1$.

10.

(i) \implies (ii) Si G un arbre alors G est connexe par définition et $m = n - 1$ par la question précédente.

(ii) \implies (iii) Si G est connexe et $m = n - 1$: supposons que G contienne un cycle C . Soit a une arête de C . Alors $G - a$ (le graphe obtenu en enlevant a dans G) est connexe. En effet : si u et v sont deux sommets de G alors ils sont reliés par un chemin C_{uv} dans G (car G est connexe) et, en remplaçant a par $C - a$ dans C_{uv} , on obtient un chemin de u à v dans $G - a$.

Ainsi $G - a$ est connexe et possède n sommets et $n - 2$ arêtes, ce qui est absurde d'après Q9.

(iii) \implies (i) Supposons G acyclique et $m = n - 1$. Soient C_1, \dots, C_k les composantes connexes de G et n_1, \dots, n_p leurs nombres de sommets. D'après Q9, chaque C_k possède $n_k - 1$ arêtes. Donc G possède $\sum_{k=1}^p (n_k - 1) = n - p$ arêtes. Comme $m = n - 1$ par hypothèse, $p = 1$. Donc G est connexe et (i) est démontré.

11.

```

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

```

12.

```

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

```

13. Montrons la proposition suivante par récurrence :

H_k : si \mathcal{P} est une partition de S_n construite à partir de $\mathcal{P}_n^{(0)}$ avec au plus k réunions et $X \in \mathcal{P}$ alors $|X| \geq 2^{h(s)}$.

- H_0 est vraie car on a alors $h(s) = 0$ et $|X| = 1$ (toutes les parties sont des singletons).
- Soit $k \in \mathbb{N}^*$. Supposons H_{k-1} et considérons \mathcal{P} une partition de S_n construite à partir de $\mathcal{P}_n^{(0)}$ avec k réunions. La dernière réunion a permis d'obtenir \mathcal{P} à partir d'une partition \mathcal{P}' en réunissant les parties X et Y associées à deux sommets s et t , pour obtenir une partie Z . On note $h(t)$ la hauteur de t dans \mathcal{P} et $h'(t)$ la hauteur de t dans \mathcal{P}' . Supposons que t ait été choisi comme représentant à l'issue de cette union (le cas où s l'a été est similaire).

D'après H_{k-1} , $|Y| \geq 2^{h'(t)}$. Il y a 2 cas : soit $h(t) = h'(t)$ soit $h(t) = h'(t) + 1$.

Si $h(t) = h'(t)$ alors $|Z| \geq |Y| \geq 2^{h'(t)} = 2^{h(t)}$.

Si $h(t) = h'(t) + 1$ alors $h'(s) = h'(t)$ (seule possibilité pour augmenter la hauteur) et :

$$|Z| = |Y| + |X| \underset{H_{k-1}}{\geq} 2^{h'(t)} + 2^{h'(s)} = 2^{h'(t)+1} = 2^{h(t)}$$

On a donc bien montré H_k .

D'après le principe de récurrence, H_k est donc vraie pour tout $k \in \mathbb{N}$.

14. **union** $p \ s \ t$ est clairement en $O(1)$. **representant** $p \ s$ est en complexité linéaire en la hauteur de l'arbre contenant s , qui est $h(s) \leq \log_2(n)$ (d'après Q13), c'est-à-dire $O(\log(n))$, où n est le nombre de sommets.

15.

```

let est_un_arbre g =
  let r = ref true in
  let n = Array.length g in
  let p = Array.init n (fun i -> -1) in
  for i = 0 to n - 1 do
    let ri = representant p i in
    List.iter (fun j ->
      let rj = representant p j in
      if ri = rj then r := false
      else union p ri rj) g.(i)
  done;
  !r && nombre_aretes g = n - 1

```

III Algorithme de Wilson

16. Le chemin {debut = 1; fin = 4; suivant = [| -5; 2; 5; 3; -1; 4 |]} part du sommet 1 pour aller en 2 puis 5 puis 4.

17. Cet algorithme peut ne pas terminer (si on tombe toujours sur un cycle avant de rencontrer \mathcal{T}) mais la probabilité que cela arrive est nulle.

18.

```

let marche_aleatoire adj parent s =
  let c = {
    debut = s;
    fin = s;
    suivant = Array.make (Array.length adj) (-1)
  } in
  while parent.(c.fin) = -2 do
    let i = Random.int (Array.length adj.(c.fin)) in
    let v = adj.(c.fin).(i) in
    c.suivant.(c.fin) <- v;
    c.fin <- v;
  done;
  c

```

19.

```

let rec greffe parent c =
  let u = ref c.debut in
  while !u <> c.fin do
    let v = c.suivant.(!u) in
    parent.(!u) <- v;
    u := v
  done

```

20.

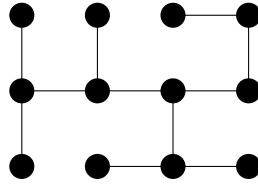
```

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

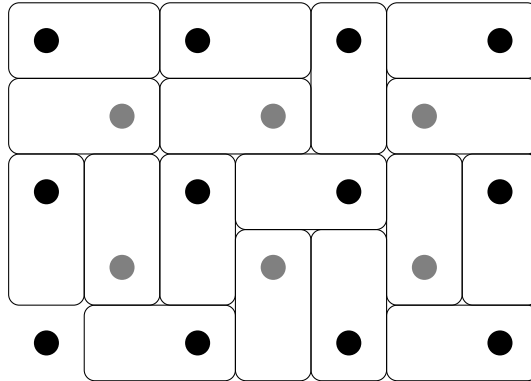
```

IV Arbres couvrants et pavages par des dominos

21.



22.



23. On peut récupérer les coordonnées (i, j) de s en utilisant la fonction `sommets` et utiliser la direction du domino en (i, j) .

24.

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

25.

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

26.

```
let phi pavage =
  let n = Array.length pavage in
  let parent = Array.make n (-1) in
  for i = 1 to n - 1 do (* on laisse -1 dans parent.(0) (racine) *)
    let k, l = coord_noire i in
    parent.(i) <- sommet_direction i pavage.(k).(l)
  done;
  parent
```
