

DM-Objective-Caml QuadTree for Image compression

Klepik Vladimir, TPB/TD2, 2186104

April 2020

1 Introduction

On présente une représentation d'images sous forme d'arbres. Cette représentation donne une méthode de compression plus ou moins efficace et facilite certaines opérations sur les images. Pour simplifier, on suppose les images carrées, de côté 2^n , et en noir et blanc. L'idée est la suivante : une image toute blanche ou toute noire se représente par sa couleur, tandis qu'une image composite se divise naturellement en quatre images carrées.

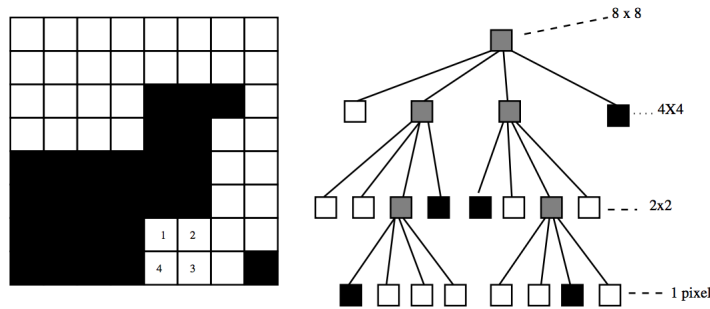


Figure 1: Quadtree image de côté $2^3 = 8$

Le carré de taille k lorsqu'il est de deux couleurs différentes se subdivise en quatre carrés chacun de taille $k/2$ considérés dans l'ordre donné par les chiffres dans la figure, i.e. dans l'ordre des aiguilles d'une montre en commençant par le carré en haut à gauche.

Conventions suivies:

- "B" , "b" , bit(0) : Blanc
- "." , "N" , "n" , bit(1) : Noir

(*Remarque : les outputs de matrices ont ete effectues a partir de mon terminal.*)

2 Questions

2.1 Question 1

Fonction récursive suivant la structure d'un quad-tree dont la racine est composée de 4 noeuds enfants et d'une profondeur de $k' = k + 1$ avec $k = \ln(n)/\ln(2)$ tq $n = 2^k$. Il y a donc deux manières de procéder récursivement, linéairement (en fonction de k tq pour $i + 1 : k = k - 1$) ou polynomialement (en fonction de n tq pour $i + 1 : n = n/2$). Pour cette question on ne fera pas en sorte d'optimiser l'arborescence du quad-tree en fonction d'une condition de dépendance (par ex: si toutes les feuilles induites par le noeud $N(i)$ sont d'une même couleur alors $N(i)$ devient une feuille de cette même couleur).

```
type couleur = Blanc |Noir;;
type quadTree = Feuille of couleur |Noeud of
    quadTree*quadTree*quadTree*quadTree;;
exception Invalid_coordinates of int*int;;

(*
    VERSION polynomiale avec n directement
    Remarque : utilisee dans le reste
*)
let rec quadTree_full n =
    if (mod) n 2=0 then
        match n with
        |2 -> Noeud(Feuille Noir,
                    Feuille Noir,
                    Feuille Noir,
                    Feuille Noir)
        |a -> Noeud(quadTree_full (n/2),
                    quadTree_full (n/2),
                    quadTree_full (n/2),
                    quadTree_full (n/2))
        else raise (Invalid_argument "taille invalide");;

quadTree_full 4;;
```

output :

```
val quadTree_full : int -> quadTree = <fun>
- : quadTree =
Noeud
  (Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
   Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
   Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
   Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir))
```

```

let rec quadTree_empty n =
  if (mod) n 2=0 then
    match n with
    |2 -> Noeud(Feuille Blanc,
                Feuille Blanc,
                Feuille Blanc,
                Feuille Blanc)
    |a -> Noeud(quadTree_empty (n/2),
                quadTree_empty (n/2),
                quadTree_empty (n/2),
                quadTree_empty (n/2))
  else raise (Invalid_argument "taille invalide");;

quadTree_empty 4;;

```

Dans la version qui suit on utilise k comme valeur récursive. Comme expliqué au dessus. Permet de vérifier directement si l'argument passé est correct ou non c-à-d si $n = 2^k$ (2ième version, non utilisée)

```

(*
  VERSION lineaire avec k directement
  Remarque : non utilisee dans le reste
  ATTENTION: requirements_libs : opam stdlib-shims => determiner le
             nbre de recurrences|profondeur
*)
let log n = (Stdlib.log n);;
let log_float n = ((/. ) (log n) (log 2.));;

let quadTree_full n =
  let n = float_of_int n in
  let rec aux k node_couleur =
    if k=0 then Feuille node_couleur
    else
      let k' = k-1
      in
        let px1 = aux k' node_couleur
        and px2 = aux k' node_couleur
        and px3 = aux k' node_couleur
        and px4 = aux k' node_couleur
        in
          match px1,px2,px3,px4 with
          |_,_,_,_ -> Noeud (px1,px2,px3,px4)
      in
        let x = ( *. ) ((log_float n)) 10.
        and y = float_of_int (( * ) (int_of_float (log_float n)) 10) in
        if x = y then
          let i = (int_of_float (log_float n)) in aux i Noir
        else raise (Invalid_argument "taille invalide");;

```

Possède le même output que la 1ère version. (pas de lien avec l'output qui suit)
output d'empty:

```
val quadTree_empty : int -> quadTree = <fun>
val empty : quadTree =
Noeud
  (Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Blanc),
   Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Blanc),
   Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Blanc),
   Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Blanc))
```

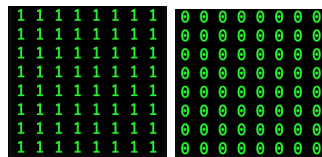


Figure 2: Quadtree image de côté $2^3 = 8$ full et empty

2.2 Question 2

Feuille Blanche devient Feuille Noire et vice-versa.

```
let rec inverse tree =
  match tree with
  |Feuille Noir -> Feuille Blanc
  |Feuille Blanc -> Feuille Noir
  |Noeud (n1,n2,n3,n4) -> Noeud (inverse n1,inverse n2,inverse
    n3,inverse n4);;

inverse (quadTree_full 2);;
```

Output :

```
val inverse : quadTree -> quadTree = <fun>
- : quadTree =
  Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Blanc)
```

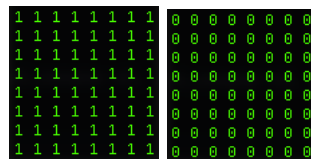


Figure 3: exemple A et ouput = inverse(A)

2.3 Question 3

On veut appliquer une rotation de $\pi/2$ anti-clockwise:

- zone 1 devient zone 4
- zone 2 devient zone 1
- zone 3 devient zone 2
- zone 4 devient zone 3

Il suffit donc d'interchanger les indices des neouds.

```
let rec rotate a = match a with
  | Feuille _ -> a
  | Noeud (n1,n2,n3,n4) -> Noeud (rotate n2,rotate n3, rotate n4,
    rotate n1);;
rotate (quadTree_empty 4);;
```

Output :

```
val rotate : quadTree -> quadTree = <fun>
- : quadTree =
Noeud (Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
  Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
  Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
  Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir))
```

Rotation appliquée à un quadtree modifié de condition $x < \text{taille}/2, y < \text{taille}/2$

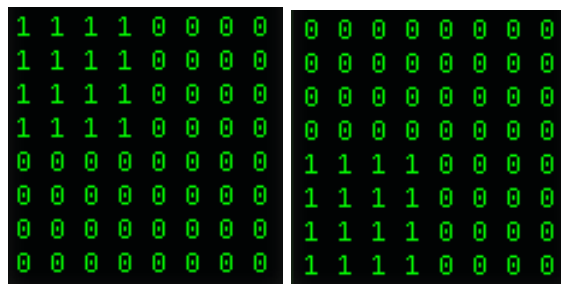


Figure 4: exemple A et ouput = rotate(A)

2.4 Question 4

On se donne les règles suivantes :

union (\cup)	intersection (\cap)
$\text{blanc} \cup x = x \cup \text{blanc} = x$	$\text{blanc} \cap x = x \cap \text{blanc} = \text{blanc}$
$\text{noir} \cup \text{noir} = \text{noir}$	$\text{noir} \cap \text{noir} = \text{noir}$

Figure 5: règles suivies

```

let rec union a b =
  match a,b with
  |Feuille Blanc,Feuille Noir -> Feuille Noir
  |Feuille Noir,Feuille Blanc -> Feuille Noir
  |Feuille Noir,Feuille Noir -> Feuille Noir
  |Feuille Blanc,Feuille Blanc -> Feuille Blanc
  |Noeud (a1,a2,a3,a4),Noeud (b1,b2,b3,b4)
    -> Noeud (union a1 b1, union a2 b2, union a3 b3, union a4 b4)

(* meme taille donc "inutile" tree pas opti *)
|Feuille Blanc,Noeud (_,_,_,_) -> failwith "tailles differentes"
|Noeud (_,_,_,_),Feuille Blanc -> failwith "tailles differentes"
|Noeud (_,_,_,_),Feuille Noir -> failwith "tailles differentes"
|Feuille Noir,Noeud (_,_,_,_) -> failwith "tailles differentes";;

union (quadTree_full 2) (quadTree_empty 2);

```

Output :

```

val union : quadTree -> quadTree -> quadTree = <fun>
- : quadTree = Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille
  Noir)

```

Union appliquée à un quadtree A modifié de condition $x < \text{taille}/2, y < \text{taille}/2$ et un quadtree B = rotationClockwise(inverse(A))

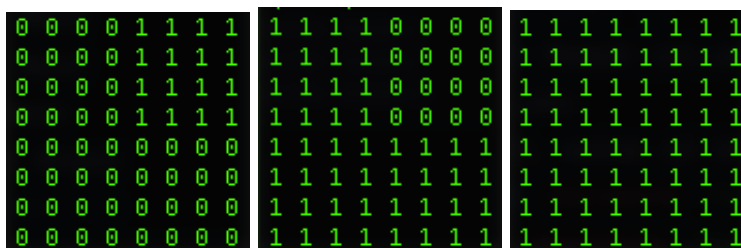


Figure 6: input A \cup B = output

2.5 Question 5

```

let rec intersection a b =
  match a,b with
  |Feuille Blanc,Feuille Noir -> Feuille Blanc
  |Feuille Noir,Feuille Blanc -> Feuille Blanc
  |Feuille Blanc,Feuille Blanc -> Feuille Blanc
  |Feuille Noir,Feuille Noir -> Feuille Noir
  |Noeud (a1,a2,a3,a4),Noeud (b1,b2,b3,b4)
    -> Noeud (intersection a1 b1, intersection a2 b2, intersection a3
      b3, intersection a4 b4)

(* meme taille donc "inutile" tree pas opti *)
|Feuille Blanc,Noeud (_,_,_,_) -> failwith "tailles differentes"
|Noeud (_,_,_,_),Feuille Blanc -> failwith "tailles differentes"
|Noeud (_,_,_,_),Feuille Noir -> failwith "tailles differentes"
|Feuille Noir,Noeud (_,_,_,_) -> failwith "tailles differentes";;

intersection (rotate((quadTree_full 4))) (quadTree_empty 4);;

```

Output :

```

val intersection : quadTree -> quadTree -> quadTree = <fun>
- : quadTree =
Noeud (Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille
  Blanc),
  Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Blanc),
  Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Blanc),
  Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Blanc))

```



Figure 7: input Full **INTER** Empty = Empty

2.6 Question 6

Information : coordonnée $(x,y) = (0,0) \Leftrightarrow$ haut-gauche ou nord-ouest.

On divise récursivement la taille de l'image et on applique les transformations suivantes à nos coordonnées avec $i = i + 1 < profondeurMax(Quadtree)$:

- si zone 1 (haut-gauche) alors $x = x, y = y$;
- si zone 2 (haut-droite) alors $x = x, y = y - taille/2$;
- si zone 3 (bas-droite) alors $x = x - taille/2, y = y - taille/2$;
- si zone 4 (bas-gauche) alors $x = x - taille/2, y = y$;

Il s'agit donc d'appliquer des transformations linéaires de re-scaling des coordonnées et du repère.

La récursivité est appliquée qu'à la zone qui nous intéresse (en fonction des coordonnées).

Les patterns du type tuple de bits permet de différencier les différentes zones. On divise les coordonnées par la **tailleActuelle**/2 (à noter que c'est une division entière donc tjrs 0 ou 1 car $(x < taille, y < taille)$) puis on match une combinaison composée de 0 ou/et de 1 pour déterminer dans quelle zone il faut itérer.

A noter que l'on applique (x-k), (y-k) à color si (1,1) ou (x-k), (y-0) si (1,0) pour appliquer un re-scaling des coordonnées selon la zone comme mentionné plus haut. (comme pour modify() Q7 sauf qu'on part de coords qui peuvent être différentes de 0 donc on soustrait, sinon on additionne).

```
let rec color (x,y) tree taille_i =  
  match tree with  
  |Feuille c -> c  
  |Noeud (n1, n2, n3, n4)  
    -> let k = (taille_i/2)  
        in  
          match (x/k, y/k) with  
          |(0, 0) -> color (x, y) n1 k  
          |(1, 0) -> color (x-k, y) n4 k  
          |(0, 1) -> color (x, y-k) n2 k  
          |(1, 1) -> color (x-k, y-k) n3 k  
          |_ -> raise (Invalid_coordinates (x,y));;  
  
let n = 4;;  
color (3,3) (quadTree_full (n)) n;;
```

Output :

```
val color : int * int -> quadTree -> int -> couleur = <fun>  
val n : int = 4  
- : couleur = Noir
```

2.7 Question 7

La fonction *modify* suit la même structure récursive que la fonction *color* sauf qu'elle s'applique à toutes les zones et non à une zone précise. On utilise une fonction auxiliaire qui permet d'opérer la récursivité en fonction des *taille * taille* feuilles du quadtree (puisque non optimisé),

à partir des coordonnées $x = 0, y = 0$ (d'où l'addition et non la soustraction).

On montre ensuite l'output de cette fonction en représentant l'arbre sous forme de matrice avec :

- "." : Feuille Noire;
- "B" : Feuille Blanche

Faire **ATTENTION** à la représentation du repère dans lequel on itère, une mauvaise interprétation peut tout biaiser et donc tout fausser... Comme savoir si les x représentent les rows ou columns etc.

La logique suivie est la suivante: coordonnée zone 1 (1,0) sera (5,4) dans la zone 3 par ex (pour la même itération). Le reste suit le même principe. L'orientation des coordonnées (rows et columns) a été pensée pour faciliter la lecture de la matrice.

```
(*
  Attention coordonnee (0,0) en haut a gauche avec
  row <=> y et column <=> x : regle en fonction des zones 1,2,3,4
*)
let modify tree taille_i p =
  let rec aux x y tree taille =
    match tree with
    |Feuille _ -> Feuille (p x y (color (x,y) tree taille_i))
    |Noeud (n1, n2, n3, n4)
      -> let k = taille/2 in
          Noeud(aux x y n1 k,
                aux (x) (y+k) n2 k,
                aux (x+k) (y+k) n3 k,
                aux (x+k) (y) n4 k)
  in aux 0 0 tree taille_i;;

let n = 8;;
let tree = quadTree_full (n);;
let new_tree = modify tree n
  (fun x y c -> if c=Noir && (x,y)=(0,5) || (x,y)=(n-1,n-1) then Blanc
    else Noir);;

(* Verification visuelle avec matrice *)
type matrice = string array array
let init f = Array.init n (fun i -> Array.init n (f i));;
let tree_matrix = (init (fun i j
  -> if (color (i,j) new_tree n)=Noir then "." else "B") : matrice);;
```

```

val modify :
  quadTree -> int -> (int -> int -> couleur -> couleur) -> quadTree =
  <fun>

val n : int = 8

val tree : quadTree =
  Noeud
    (Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille
      Noir),
      Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
      Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
      Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir)),
    Noeud (Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
      Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
      Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
      Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir)),
    Noeud (Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
      Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
      Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
      Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir)),
    Noeud (Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
      Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
      Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
      Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir)))

val new_tree : quadTree =
  Noeud
    (Noeud (Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille
      Noir),
      Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
      Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir)),
      Noeud (Noeud (Feuille Noir, Feuille Blanc, Feuille Noir, Feuille
        Noir),
        Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
        Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir)),
      Noeud (Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
        Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
        Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir)),
      Noeud (Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
        Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
        Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
        Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir)))

```

```
type matrice = string array array

val init : (int -> int -> 'a) -> 'a array array = <fun>

val tree_matrix : matrice =
[|["."; "."; "."; "."; "."; "."; "."; "."; "."]|;
 [|"."; "."; "."; "."; "."; "."; "."; "."; "."]|;
 [|"."; "."; "."; "."; "."; "."; "."; "."; "."]|;
 [|"."; "."; "."; "."; "."; "."; "."; "."; "."]|;
 [|"."; "."; "."; "."; "."; "."; "."; "."; "."]|;
 [|"B"; "."; "."; "."; "."; "."; "."; "."; "."]|;
 [|"."; "."; "."; "."; "."; "."; "."; "."; "."]|;
 [|"."; "."; "."; "."; "."; "."; "."; "."; "B"|]|]
```

2.8 Question 8

```
let rec optimise tree =
  match tree with
  |Feuille _ -> tree
  |Noeud (Feuille c1, Feuille c2, Feuille c3, Feuille c4)
    when c1=c2 && c2=c3 && c3=c4 -> Feuille c1
  |Noeud (n1,n2,n3,n4) -> Noeud (optimise n1,optimise n2,optimise
    n3,optimise n4);;

optimise (quadTree_full 2);;
```

```
val optimise : quadTree -> quadTree = <fun>
- : quadTree = Feuille Noir
```

2.9 Question 9

Structure utilisée :

```
code(Feuille Blanc) = 00
code(Feuille Noir) = 01
code(Noeud (a1,a2,a3,a4)) = 1 code(a1) code(a2) code(a3) code(a4)
```

Figure 8: list to quad-tree

QuadtreeToList permet de convertir un quadtree de taille n en liste de bits. La liste commencera obligatoirement par un bit de signe $B = 1$ puis on accède récursivement à toutes les Feuilles de couleur C et si:

- $C = \text{Blanc}$ alors on retourne $[Zero; Zero]$,
- $C = \text{Noir}$ alors on retourne $[Zero; Un]$,

en concaténant toutes les listes entre elles dans l'ordre des zones pour linéariser et donc retourner un vecteur (une liste, si vecteur alors forcément de dimension 1).

```
type bit = Zero | Un;;

let rec quadtree_to_list tree =
  match tree with
  | Feuille Blanc -> [Zero ; Zero]
  | Feuille Noir -> [Zero ; Un]
  | Noeud (n1,n2,n3,n4)
  -> Un::quadtree_to_list n1
      @ quadtree_to_list n2
      @ quadtree_to_list n3
      @ quadtree_to_list n4;;

quadtree_to_list (quadTree_empty 4);;
```

Output :

```
type bit = Zero | Un
val quadtree_to_list : quadTree -> bit list = <fun>
- : bit list =
[Un; Un; Zero; Zero; Zero; Zero; Zero; Zero; Zero; Zero; Zero; Un; Zero; Zero;
 Zero; Zero; Zero; Zero; Zero; Zero; Un; Zero; Zero; Zero; Zero;
 Zero; Zero; Zero; Zero; Un; Zero; Zero; Zero; Zero; Zero; Zero;
 Zero; Zero]
```

2.10 Question 10

Pour cette fonction on fait le process inverse à *quadTreeToList*, si $A \Rightarrow B$ alors ici on applique la contraposée : $B \Rightarrow A$. On utilise la curryfication (currying) pour se "débarasser" du reste suite à l'application de la fonction (qui retourne un tuple (*Feuille*, *reste*)) en ne conservant que ce qui nous intéresse. Cela permet d'alléger la fonction récursive.

A noter que l'on imbrique les Noeuds les uns dans les autres. Le reste de l'un sera le reste de l'autre (suivant), on dé-linéarise (puisque une paire de bits encode une Feuille c, que l'on doit décoder).

```
let rec list_to_quadtree l =  
  match l with  
  |Zero::Zero::r -> Feuille Blanc, r  
  |Zero::Un::r -> Feuille Noir, r  
  |Un::r ->  
    let a1,r = list_to_quadtree r in  
    let a2,r = list_to_quadtree r in  
    let a3,r = list_to_quadtree r in  
    let a4,r = list_to_quadtree r in  
    Noeud (a1,a2,a3,a4),r  
  (* pas oblige mais sinon matching pattern non exhaustif *)  
  |_ -> assert false;;  
  
let end_list_to_quadtree l = let n,_ = list_to_quadtree l in n;;  
end_list_to_quadtree (quadtree_to_list (quadTree_empty 4));;
```

Output :

```
val list_to_quadtree : bit list -> quadTree * bit list = <fun>  
  
val end_list_to_quadtree : bit list -> quadTree = <fun>  
  
- : quadTree =  
Noeud (Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille  
  Blanc), Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille  
  Blanc), Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille  
  Blanc), Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille  
  Blanc))
```

3 Questions BONUS

3.1 Question 1 VERSION 1

```
(* quadTree to matrix *)
type matrice = string array array

let n = 8;;
let tree = quadTree_empty (n);;
let init f = Array.init n (fun i -> Array.init n (f i));;
let tree_matrix = (init (fun i j
  -> if (color (i,j) tree n)=Noir then "n" else "b") : matrice));;

(* apres modification du tree *)
let tree = modify tree n (fun x y c -> if c=Blanc && (x,y)<(n-1,n-1)
  then Noir else Blanc);;
let tree_matrix = (init (fun i j
  -> if (color (i,j) tree n)=Noir then "n" else "b") : matrice));;
```

Output:

```
type matrice = string array array

val n : int = 8

val tree : quadTree =
  Noeud
    (Noeud
      (Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Blanc),
        Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Blanc),
        Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Blanc),
        Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Blanc)),
      Noeud (Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille
        Blanc),
        Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Blanc),
        Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Blanc)),
      Noeud (Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille
        Blanc),
        Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Blanc),
        Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Blanc)),
      Noeud (Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille
        Blanc),
        Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Blanc),
        Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Blanc)))

val init : (int -> int -> 'a) -> 'a array array = <fun>
```

[illegible]



Figure 9: Matrice conditionnée $x < \text{taille}/2$, $y < \text{taille}/2$ alors Feuille Blanche

3.2 Question 1 version 2 avec module Graphics

```

open Graphics
type image = couleur array array;;

let taille = 8

(* apres modification du tree *)
let tree = modify tree taille (fun x y c -> if c=Blanc &&
  (x,y)<(n-1,n-1) then Noir else Blanc);;
let tree_matrix = (init (fun i j
  -> if (color (i,j) tree taille)=Noir then "n" else "b") : matrice);;

let img = (init (fun i j
  -> if (color (i,j) tree taille)=Noir then Blanc else Noir) : image);;

let rec img_to_tree img i j k =
  if k <= 1 then Feuille img.(i).(j)
  else
    let k' = k/2 in
    let c1 = img_to_tree img i j k'
    and c2 = img_to_tree img i (j+k') k'
    and c3 = img_to_tree img (i+k') j k'
    and c4 = img_to_tree img (i+k') (j+k') k'
    in
    match c1,c2,c3,c4 with
    |Feuille n1, Feuille n2, Feuille n3, Feuille n4
      (* opti: if same children of c => node => feuille c*)
      when n1 = n2 && n2 = n3 && n3 = n4 -> c1
    |_,_,_,_ -> Noeud (c1,c2,c3,c4);;

let image_vers_arbre k img = img_to_tree img 0 0 k;;
let tree = image_vers_arbre (Array.length img) img;;

```

Suite code :

```
let rec output_img i j k tree =
  match tree with
  |Feuille Noir
    (* on veut output que les coords noires *)
    -> Graphics.fill_rect i j k k
  |Feuille Blanc
    -> ()
  |Noeud (c1,c2,c3,c4)
    ->
      let k' = k/2 in
      output_img i (j+k') k' c2 ;
      output_img (i+k') (j+k') k' c3 ;
      output_img i j k' c1 ;
      output_img (i+k') j k' c4;;
let dessine_quadtree taille tree = output_img 0 0 k tree;;
```

Output :

```
type image = couleur array array

val taille : int = 8

val img : image =
  [|Noir; Noir; Noir; Noir; Noir; Noir; Noir; Noir|];
  [|Noir; Noir; Noir; Noir; Noir; Noir; Noir; Noir|];
  [|Noir; Noir; Noir; Noir; Noir; Noir; Noir; Noir|];
  [|Noir; Noir; Noir; Noir; Noir; Noir; Noir; Noir|];
  [|Noir; Noir; Noir; Noir; Noir; Noir; Noir; Noir|];
  [|Noir; Noir; Noir; Noir; Noir; Noir; Noir; Noir|];
  [|Noir; Noir; Noir; Noir; Noir; Noir; Noir; Noir|];
  [|Noir; Noir; Noir; Noir; Noir; Noir; Noir; Noir|];

val img_vers_arbre : couleur array array -> int -> int -> int ->
  quadTree =
  <fun>

val image_vers_arbre : int -> couleur array array -> quadTree = <fun>

val a : quadTree = Feuille Noir
```

Si le module Graphics est installé alors se plot une matrice de coord $[[0,0],\dots,[taille-1,taille-1]]$ dont si $mat.(i).(j) = \text{Noir}$ alors case colorée en noir.

3.3 Main () unit

Voir le dossier dans le path ./DM/main contenant la structure adéquate à l'utilisation d'un main unit.

L'objectif de ce main () est de démontrer une application possible du code. Et non d'intégrer tout le code demandé (tâche non requise).

```
(* pour compiler localement si pas deja fait *)
```

```
#path_to_repo/ ocamlc -o --name main.ml
```

```
ex compilation : repo$ ocamlc -o file main.ml
```

```
ex utilisation : repo$ ./file info (* va output les info du main *)
```

Ici on utilise la **compilation locale** avec *ocamlc* qui est plus rapide que la version universelle bytecode, donc être conscient qu'elle (la version) peut ne pas run sur votre machine si pas compilée localement (os utilisé: mac osx).
Version universelle : *ocamlc* (pas utilisée pour l'exemple, sauf pour générer DMFile.mli)

```
(base) MBPdeVKSP0C2:DM SPOC$ ./file info

Structure de l'input : avec préfix ./name_compiled_file:

*) tree <type> <taille> <x> <y> <color> <rotate:false>
  - x et y : juste pour tester modify
  - rotate : false or true

*) ex d'usage : ./name_compiled_file tree full 16 15 15 Noir true

(base) MBPdeVKSP0C2:DM SPOC$ ./file tree full 16 8 8 Noir false
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Figure 10: Exemple commandes pour le main

```
(* FONCTIONS UTILES A L'OUTPUT DANS TERMINAL QUI ONT ETE ADD *)

let init f n = List.init n (fun i -> List.init n (f i));;

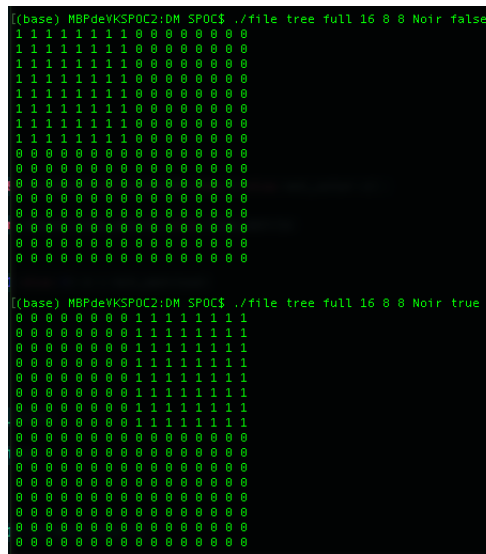
let rec print_bits_matrix l =
  match l with
  [] -> () ; print_string "\n"
  |e::l -> print_int e ; print_string " " ; print_bits_matrix l;;

let rec print_str_matrix l =
  match l with
  [] -> () ; print_string "\n"
  |e::l -> print_string e ; print_string " " ; print_str_matrix l;;

let outputBitsMat m = List.iter (fun x -> print_bits_matrix x) m;;
let outputStrMat m = List.iter (fun x -> print_str_matrix x) m;;

let to_color_type s = if s="Noir" then Noir else Blanc;;

let not_color c =
  match c with
  |Blanc -> Noir
  |Noir -> Blanc;;
```



19

Le main () :

```
let main () =

  if (Array.length Sys.argv) = 1 then begin
    print_string "Entrer la commande : ./file info";
    print_newline()

  end else begin
    try

      if Sys.argv.(1) = "tree" then

        let tree =
          if Sys.argv.(2) = "full" then
            quadTree_full (int_of_string Sys.argv.(3))

          else if Sys.argv.(2) = "empty" then
            quadTree_empty (int_of_string Sys.argv.(3))

          else
            (* sinon par default on assigne un tree *)
            quadTree_empty (int_of_string Sys.argv.(3))

        and n = int_of_string Sys.argv.(3)
        and coord1 = int_of_string Sys.argv.(4)
        and coord2 = int_of_string Sys.argv.(5)
        and tree_color = to_color_type Sys.argv.(6)
        in
          let tree_matrix =
            modify tree n (fun x y c -> if c=tree_color && x<coord1
              && y<coord2 then c else not_color c)
          in
            let last_tree = if (Sys.argv.(7) = "true") then rotate
              tree_matrix else tree_matrix
            in
              let bit_matrix = (init (fun i j
                -> if (color (i,j) last_tree n)=Noir then 1 else 0) n :
                bit_matrice)
              in
                outputBitsMat (bit_matrix)

          else if Sys.argv.(1) = "info" then

            print_string "
            Structure de l'input : avec prefix ./name_compiled_file:

            *) tree <type> <taille> <x> <y> <color> <rotate:false>
            - x et y : juste pour tester modify
            - rotate : false or true
```

```

*) ex d'usage : ./name_compiled_file tree full 16 15 15 Noir
    true ";
print_newline ();
print_newline ();

with
|Invalid_argument _ -> raise (Invalid_argument "entrer un arg
    valide.\n")
|Invalid_coordinates (a,b) -> raise (Invalid_coordinates (a,b))
|_ -> failwith "Fonction inexistante ou oubli params,\nEntrez la
    commande : ./file info\n"

end;;

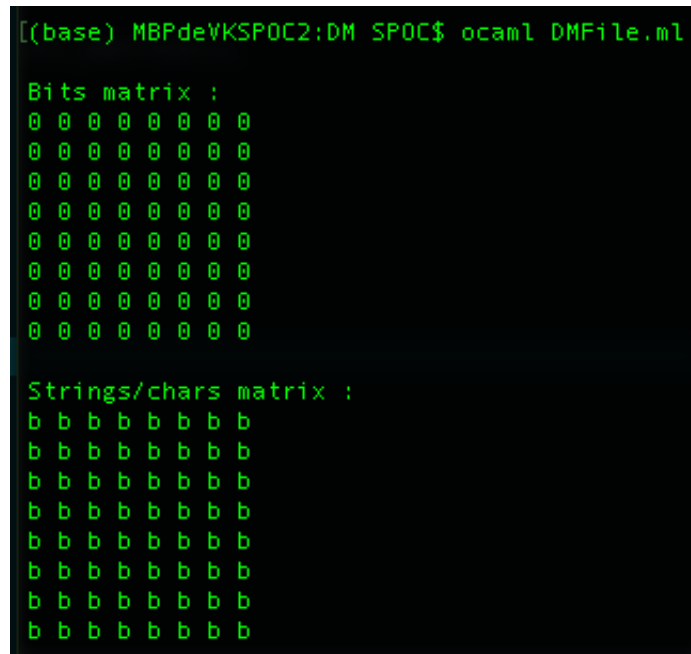
if !Sys.interactive then () else main ();;

```

Remarque : (pas de liens avec le main)

Vous pourrez tester les différentes fonctions directement avec *ocaml* DMFile.ml dans votre terminal.

Lisez les commentaires de ce fichier ainsi que le README.txt, merci 😊



```

[(base) MBPdeVKSP0C2:DM SPOC$ ocaml DMFile.ml

Bits matrix :
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

Strings/chars matrix :
b b b b b b b b
b b b b b b b b
b b b b b b b b
b b b b b b b b
b b b b b b b b
b b b b b b b b
b b b b b b b b
b b b b b b b b

```

Figure 12: Exemple de l'output avec la structure actuelle du fichier DMFile.ml