# Quelques programmes en ocaml illustrant le cours d'Algorithmique et complexité

D. Michelucci

2 octobre 2006

La lecture de ce polycopié ne dispense pas d'assister aux cours, aux TD, aux TP...

Elle ne dispense pas de lire : "Introduction à l'algorithmique".

Ce polycopié réunit quelques programmes en ocaml sur :

- quelques petits problèmes combinatoires
- Les tris
- les tables de hachage ; comment les utiliser pour optimiser ou simplifier le calcul de fonctions telles que fibonacci, ou la programmation dynamique.
- la recherche arborescente (*backtrack*)
- le calcul des racines réelles de polynômes, en utilisant les bases de Bernstein.

## 1 Quelques petits problèmes combinatoires

### 1.1 Sous ensembles de cardinalité $k$ d'un ensemble

```
let rec  k_subsets ensemble k =
        if k=0 then [  [] ]
        else let n=List.length ensemble in
             if n<k then []
             else if k=n then [ ensemble ]
             else match ensemble with
                 | t::q ->  (k_subsets q k)
                     @ (List.map (function e -> t::e) (k_subsets q (k-1)))
                 | [] -> failwith "impossible" ;;
# k_subsets [1;2;3;4] 2;;
- : int list list = [[3; 4]; [2; 4]; [2; 3]; [1; 4]; [1; 3]; [1; 2]]
```

### 1.2 Sous ensembles d'un ensemble

```
let rec subsets ensemble = match ensemble with
| [] -> [   []   ]
| t::q -> let parts_of_q = subsets q in
          let tete_union_parts_of_q =
               List.map (function ens-> t::ens) parts_of_q in
          parts_of_q @ tete_union_parts_of_q;;
# subsets [1;2];;
- : int list list = [[]; [2]; [1]; [1; 2]]
```

### 1.3 Permutations

```
(* insère e [a; b; c]: rend la liste
  [ [e;a;b;c]; [a;e;b;c]; [a;b;e;c]; [a;b;c;e]] *)
let rec insert e liste = match liste with
| [] -> [[ e]]
| t::q -> (e::liste)::(List.map (function y -> t::y)
                                (insert e q));;

let rec permutations l = match l with [] -> [[]]
| t::q -> List.flatten
            (List.map (function permu_sans_t->
              insert t permu_sans_t) (permutations q));;
# permutations [1;2;3];;
- : int list list =
[[1; 2; 3]; [2; 1; 3]; [2; 3; 1]; [1; 3; 2]; [3; 1; 2]; [3; 2; 1]]
```

# 2 Les méthodes de tris

## 2.1 Le tri par fusion (*mergesort*)

```
module L=List;;
let rec lalea n = if n=0 then []
      else (Random.int 10000) :: (lalea (n-1));;

let rec coupeen2 liste= match liste with
| [] -> [],[]
| [x]-> [x],[]
| x::y::q -> let q1,q2=coupeen2 q in
                           x::q1, y::q2;;

let rec fusion l1 l2 inf= match l1,l2 with
|[],_->l2
| _,[]->l1
| a1::q1, a2::q2 -> if (inf a1 a2)
             then a1::(fusion q1 l2 inf)
             else a2::(fusion l1 q2 inf);;

let rec mergesort l inf= match l with
| []  -> []
| [x] -> [x]
|  _  -> let l1,l2=coupeen2 l in
      fusion (mergesort l1 inf) (mergesort l2 inf) inf;;

let l= lalea 1000;;
let l'=mergesort l ( < );;
```

## 2.2 Le tri rapide (*quicksort*)

```
module L=List;;

let rec lalea n = if n=0 then []
      else (Random.int 10000) :: (lalea (n-1));;
```

```
let rec qksort inf l =
   let rec partition l pivot petits egaux grands=
    match l with
    | [] -> L.concat [qksort inf petits; egaux;
                      qksort inf grands]
    | t::q -> if inf t pivot
            then partition q pivot (t::petits)  egaux grands
            else if inf pivot t
            then partition q pivot petits egaux (t::grands)
            else partition q pivot petits (t::egaux) grands
in match l with
      | [] | [_] -> l
      | pivot::queue -> partition l pivot [][][];;

let l= lalea 1000;;
let l'= qksort ( < ) l;;
```

Nota bene : le tri rapide est en $O(n^2)$ si on n'a pas de chance. Il est possible de se prémunir contre cela, en mélangeant d'abord la liste. Le plus simple est de convertir la liste en tableau, et d'échanger chaque élément (de l'indice 0 à l'indice max) du tableau avec un autre choisi (pseudo) aléatoirement. Puis le tableau mélangé est de nouveau converti en liste.

Se reporter à "Introduction à l'algorithmique" pour une version sur les tableaux.

## 2.3 Le tri par tas (*heapsort*)

1. Écrire une fonction : `liste_aleatoire n` qui rende une liste de n nombres entiers aléatoires. Utiliser la fonction : `random__int k` qui rend un entier pseudo aléatoire entre 0 et $k-1$.

2. Écrire une fonction : `trinaif L` qui trie une liste d'entiers par ordre croissant : le principe est de trier la queue de la liste, puis d'insérer le premier élément en bonne place grâce à une fonction `insert_ds_liste_triee x liste`.

   Le but du jeu est d'écrire un tri efficace en utilisant la structure de données de tas (*heap* pour les saxophones). Un tas est un arbre binaire, dont les noeuds portent des éléments (ici des entiers) d'un ensemble complètement ordonné (ici $\mathbb{N}$). La valeur portée par tout noeud est inférieure (ou égale, mais pas supérieure) aux valeurs portées par le fils gauche et le fils droit. La valeur en un noeud est donc inférieure (ou égale) à toutes les valeurs portées par les descendants de ce noeud. Un tas est décrit par le type : `type Tas= Rien | Noeud of int * Tas * Tas;;`. On aura besoin de deux fonctions, l'une pour ajouter un élément (un entier) dans un tas, l'autre pour amputer un tas de sa racine.

3. Pour amputer, écrire une fonction : `enlever untas` qui utilise le principe suivant : soit `(x,fg,fd)` le tas. La racine du nouveau tas portera le petit élément entre `fg` et `fd`; supposons que ce soit l'élément de `fg`; alors le nouveau fils gauche est `enlever fg`, et le nouveau fils droit est `fd`.

4. Pour ajouter un élément `y`, écrire une fonction : `ajouter y untas`. C'est trivial si le tas est vide. Sinon soit `x` la valeur de la racine. La racine du nouveau tas est le minimum de `x` et de `y`. Le fils gauche du nouveau tas est le fils

droit de l'ancien, et le fils droit du nouveau tas est le résultat de l'ajout de `max x y` au fils gauche de l'ancien tas. Ainsi on insère "toujours à droite", mais on permute fils gauche et droit à chaque descente. Ces permutations évitent d'avoir un arbre plus lourd à droite qu'à gauche. On pourrait bien sûr prendre la conversion symétrique (insérer systèmatiquement à gauche, en permutant gauche et droite lors de chaque descente). On pourrait aussi ne pas permuter gauche et droite, mais choisir aléatoirement à chaque insertion entre la gauche et la droite. Toutes ces techniques (dites de randomisation) font que le tas obtenu reste équilibré, même après de nombreuses insertions et suppressions.

5. Écrire une fonction : `najouter liste untas` qui ajoute tous les éléments d'une liste d'entiers dans le tas.
6. Écrire une fonction : `vider untas` qui rend la liste triée des éléments contenus dans le tas. Bien sûr, cette fonction appelle la fonction `enlever` définie précédemment.
7. Écrire une fonction : `tripartas liste` qui trie une liste d'entiers, en les insérant dans un tas, puis en vidant le tas (*heapsort* en saxophone). Comparer les temps d'exécution de cette fonction avec `trinaif` sur une longue liste (dix mille éléments, disons).

Voici un programme caml, qui est une solution possible :

```
(* rend une liste d'entiers aleatoires *)
let rec liste_aleatoire = function
      0 -> []
|     n -> (random__int 10000)::(liste_aleatoire (n-1));;

type Tas= Rien | Noeud of int * Tas * Tas;;

exception Tas_Vide;;

(* rend le premier elt d'un tas; erreur si le tas est vide *)
let premier = function
      Rien -> raise Tas_Vide
|     Noeud (x,_,_) -> x;;

(* enleve le premier elt d'un tas *)
let rec enlever = function
      Rien -> Rien
|     Noeud (x,Rien,fd) -> fd
|     Noeud (x,fg,Rien) -> fg
|     Noeud (x, (Noeud (fg_x,fg_g,fg_d) as fg),
                (Noeud (fd_x,fd_g,fd_d) as fd))
      -> if fg_x < fd_x
        then Noeud (fg_x, enlever fg, fd)
        else Noeud (fd_x, fg, enlever fd);;

(* ajoute un element a un tas *)
(* nota bene: on insere systematiquement `a droite, mais on
   permute fils gauche et droit a chaque descente pour "melanger"
   et avoir un tas equilibre *)
let rec ajouter y = function
      Rien -> Noeud (y,Rien,Rien)
```

```
|     Noeud (x,fg,fd) -> Noeud( min x y, fd, ajouter (max x y) fg);;

(* ajoute une liste d'elements a un tas *)
let rec najouter liste tas = match liste with
      [] -> tas
|     x::q -> ajouter x (najouter q tas);;

(* cree un tas contenant une liste d'elements donn'es : *)
let remplir liste= najouter liste Rien;;

(* vide un tas et rend la liste (ordonnee) des elements *)
let rec vider = function
      Rien -> []
|     Noeud (x,_,_) as noeud -> x::(vider (enlever noeud));;

(* tri par tripartas *)
let tripartas l= vider (remplir l);;

(* variante de vider : on fusionne la liste sur le fg et celle
   sur le fils droit *)
let rec vider2 tas= match tas with
      Rien -> []
|     Noeud (x,fg,fd) -> x::(fusion (vider2 fg)(vider2 fd))
and (* fusion de 2 listes triees *)
fusion l1 l2= match (l1, l2) with
      [],_ -> l2
|     _,[] -> l1
|     x1::q1, x2::q2 -> if x1<x2 then x1::(fusion q1 l2)
                  else x2::(fusion l1 q2);;
(* variante correspondante de tripartas *)
let tripartas2 l= vider2 (remplir l);;

(* Calculer les longueurs min et max des chemins dans un tas
   permet de verifier qu'il est bien equilibr'e
*)
(* rend la longueur du plus court chemin dans un tas *)
let rec lmin= function
      Rien -> 0
|     Noeud (x,fg,fd) -> 1+ (min (lmin fg) (lmin fd));;
(* rend la longueur du plus long chemin dans un tas *)
let rec lmax= function
      Rien -> 0
|     Noeud (x,fg,fd) -> 1+ (max (lmax fg) (lmax fd));;

(* definition du tri naif par insertion *)
(* juste pour comparer avec tripartas ou tripartas2
   sur une liste aleatoire de 10 mille elts
   et verifier empiriquement que le tri naif est plus lent, en effet...
*)
let rec trinaif = function
      [] -> []
|     x::q -> insert_ds_liste_triee x (trinaif q)
and insert_ds_liste_triee x=function
      [] -> [x]
```

```
|     (y::q) as l-> if x<y then x::l else y::(insert_ds_liste_triee x q);;

(* fonction verifiant si une liste est triee *)
let rec triee l = match l with
      [] -> true
|     [x] -> true
|     x::(y::q as yq) -> x <= y && triee yq ;;
```

Remarque : cette gestion des tas diffère de celle, non fonctionnelle, utilisée avec des tableaux, qui a été vue en cours.

## 2.4 Le tri par base (*radix sort*)

Note : le tri par base utilise le tri par partition (*bucket sort*).

```
module A=Array;;
module L=List;;

let rec lalea n = if n=0 then [] else (Random.int 10000)::(lalea (n-1));;

(* rend le i ème chiffre du nombre dans la base specifiée
   (le dernier chiffre est le 0 ième) *)
let rec digit nombre base i =
        if i=0 then nombre mod base
        else digit (nombre/base) base (i-1);;

(* trie la liste en utilisant base tiroirs,
sur le k ième chiffre en base: base *)
let bucket_sort liste base k=
    let buckets = A.create base [] in
    L.iter (function nombre ->
         let index= digit nombre base k in
           buckets.( index ) <- nombre::buckets.(index)) liste;
    A.map L.rev buckets;;

let radix_sort liste base n = (* n est le nb de chiffres dans la base *)
        let etape i l = L.concat (A.to_list (bucket_sort l base i)) in
        (* retourne: ...(ETAP 3 (ETAP 2 (ETAP 1 (ETAP 0 liste))))..  *)
        let rec aux i liste_precedente =
          if i>n then liste_precedente
          else aux (i+1) (etape i liste_precedente)
        in aux 0 liste;;
let l=lalea 1000;;
let l'=radix_sort l 10 6;;
```

# 3 Les tables de hachage

## 3.1 Une implantation (naïve) des tables de hachage

```
module A=Array;;
module L=List;;

type ('clef, 'elt) matabledehachage =
```

```
  { mutable tab: ('clef * 'elt) list array;
    (* mutable: on pourrait changer la taille*)
    hashfonc : 'clef -> int;
    egal : 'clef -> 'clef -> bool };;

let create hashfonc fonction_egal =
{ tab = A.make 11 []; hashfonc = hashfonc; egal = fonction_egal };;

let find ht clef =
   snd (L.find (function (c,e) -> ht.egal c clef)
              ht.tab.( ht.hashfonc clef mod A.length ht.tab)) ;;

let add ht clef elt =
   let index = ht.hashfonc clef mod A.length ht.tab in
   ht.tab.(index) <- (clef, elt) :: ht.tab.(index);;

let remove ht clef = (* on supprime la 1ere occurence de la clef *)
   let index = ht.hashfonc clef mod A.length ht.tab in
   let rec balaie liste = match liste with
   [] -> raise Not_found
   | (key,elt)::q -> if ht.egal key clef
                         then q else (key,elt)::balaie q in
   ht.tab.(index) <- balaie ht.tab.(index);;

let replace ht clef elt =
   (try remove ht clef with Not_found-> ());
   add ht clef elt;;

let fold fonction ht valeur_initiale =
   let rec scanlist valeur l= match l with []-> valeur
      | (key,elt)::q -> scanlist (fonction key elt valeur) q
   in let rec aux i valeur =
      if i=A.length ht.tab then valeur
      else aux (i+1) (scanlist valeur ht.tab.(i))
   in aux 0 valeur_initiale;;

let hstring str =
   let rec accumule i total =
         if i=String.length str then total
      else accumule (i+1) (total + (int_of_char str.[i]))
   in accumule 0 0;;

let ht = create hstring ( = );;
add ht "adam" 127;;
add ht "eve" 140;;
find ht "adam";;

let contenu ht =
   fold (fun key elt liste -> (key,elt)::liste) ht [];;
```

Note : on utilise un champ mutable, pour pouvoir éventuellement modifier le nombre de tiroirs de la table. Mais on ne le fait pas. Ceci est un exercice possible.

## 3.2 Gestion d'ensembles par des tables de hachage

Ce programme fournit aussi le calcul de la fermeture transitive d'un graphe.

```
module H=hashtbl;;
module L=List;;

(* x est il dans la htable ? *)
let inht ht x = try H.find ht x; true with Not_found -> false;;

(* rend une htable d'un ensemble represente par une liste *)
let ht_of_set e = let ht= H.create 17 in
        L.iter (function x-> H.replace ht x x) e; ht;;

(* rend les elts dans une htable: sans repetition *)
let set_of_ht ht = H.fold (fun x y result -> x::result) ht [];;

(* pour enlever les elts multiples dans une liste en O(n) *)
let uniq e = set_of_ht (ht_of_set e) ;;

let intersection e f = uniq (L.filter (inht (ht_of_set e)) f);;

let filternot f l = L.filter (function x -> not (f x)) l;;

let difference e f = (* e - f *)
        let htf= ht_of_set f in uniq (filternot (inht htf) e);;

let reunion e f =
        let ht= ht_of_set e in
        L.iter (function x-> H.replace ht x x) f;
        set_of_ht ht;;

let diffsym e f = (difference e f) @ (difference f e);;

let meme e f = [] = (diffsym e f);;

let rec fixpt egal f x0 =
      let x1= f x0 in if egal x0 x1 then x1 else fixpt egal f x1;;

(* def de la fermeture transitive, un peu brutale *)
let sature voisins subset =
        reunion subset (L.flatten (L.map voisins subset)) ;;
let ft1 voisins subset = fixpt meme (sature voisins) subset;;

(* 2eme definition; on ne calcule les voisins que des
   nouveaux elts rencontres *)
let ft2 voisins subset =
        let ht= ht_of_set subset in
        let sature frontiere = match frontiere with
        | [] -> []
        | e::q -> let news= filternot (inht ht) (voisins e) in
                L.iter (function x-> H.add ht x x) news;
                news @ q
        in ignore(fixpt ( = ) sature subset); set_of_ht ht;;
```

```
let graphe1 x = if x <5 then [2*x; 2*x+1; x/2] else [x/2; x/3];;
ft1 graphe1 [1];;
ft2 graphe1 [1];;
```

### 3.3 Autres emplois des tables de hachage

Cf Fibonacci, et programmation dynamique.

## 4 Fibonacci

Ce programme montre comment les tables de hachage permettent d'accélérer le calcul naïf de la suite de Fibonacci. Le même principe peut être utilisé pour le calcul du nombre de combinaisons ou d'arrangements. La puissance rapide de matrices permet de calculer en $O(\log n)$ le $n$ ième terme de la suite.

```
(* voici une autre façon de definir fibonacci, c'est f x 0: *)
let rec f a b =
match a,b with
| 0, y -> y
| 1, y -> y+1
| x, y -> f (x-1) (f (x-2) y);;

(* la definition récursive naive *)
(* évidemment, c'est lent pour fib 40 *)
let rec fib x = match x with
| 0 -> 0
| 1 | 2 -> 1
| _ -> fib (x-1) + fib (x-2);;

(* Pour accélérer, on peut mémoriser les valeurs dans une table
   de hachage.
   Cette technique est très générale, et utilisable qq soit le
   langage ...
   Cette fois, le calcul de fibonacci 40 est instantané
  -- Bien notez que seules sont calculées les valeurs nécessaires --
*)
module H=Hashtbl;;

let fibonacci x =
let ht = H.create 7 in
let rec f x =
        try H.find ht x
        with Not_found ->
                let fx= match x with | 0 -> 0
                                     | 1 | 2 -> 1
                                     | _ -> f (x-1) + f (x-2)
                in H.add ht x fx; fx
in f x;;

(* si la table est extérieure :*)
let ht_fibo = H.create 17;;
let rec fibo x =
        try H.find ht_fibo x
```

```
        with Not_found ->
                ( let fx= match x with | 0 -> 0 | 1 | 2 -> 1
                                       | _ -> fibo (x-1) + fibo (x-2)
                  in H.add ht_fibo x fx; fx );;

(* On peut aussi optimiser le calcul, si la fonction n'est pas trop
méchante.
Si (f x) est une fonction lineaire , alors c'est tjs possible.
(f(x) , f(x-1)) = (f(x-1), f(x-2)) * [| 1; 0 |][| 1; 0|]|]
Donc il suffit de calculer mat^n rapidement *)

type mat = { a: int; b:int; c:int; d:int};;

let mult {a=a; b=b; c=c; d=d} {a=a'; b=b'; c=c'; d=d'}=
{ a= a*a'+b*c'; b=a*b'+b*d'; c=c*a'+d*c'; d=c*b'+d*d'};;

let square x = mult x x;;

let rec puiss m n = if n=0 then {a=1;b=0;c=0;d=1} else
        if n mod 2 = 0 then square (puiss m (n/2)) else
        mult m (puiss m (n-1));;

let fibm x = let m={a=1; b=1; c=1; d=0} in (puiss m (x-1)).a  ;;

(* les etudiants extrairont de la matrice le résultat souhaité ... *)

(* calcul du nombre de combinaisons
  (ou d'arrangements ?, je confonds)*)
let rec  choices k n =
   if k=0 then 1 else
   if k>n then 0
   else choices k (n-1) + choices (k-1)(n-1);;

let ht_choices = H.create 17;;
   let rec choix k n =
   try H.find ht_choices (k,n)
   with Not_found -> (
        let result= if k=0 then 1 else
                    if k>n then 0 else
                           choix k (n-1) + choix (k-1)(n-1) in
        H.add ht_choices (k,n) result;
        result
   );;

let ht_content ht =
H.fold (fun key attribut liste -> (key,attribut)::liste) ht [];;
```

# 5 Programmation dynamique

Soit à multiplier une suite de matrices $M_1 M_2 \ldots M_n$. Vu l'associativité du produit matriciel, il existe de nombreux arbres binaires de calculs. Leurs coûts peuvent être très différents. Voir "Introduction à l'algorithmique". L'arbre bi-

naire optimal qui a pour feuilles $M_1 \dots M_n$ a pour fils gauche et droits deux sous arbres binaires, l'un pour $M_1 \dots M_k$, l'autre pour $M_{k+1} \dots M_n$, et le coût de cet arbre est la somme des coûts des deux sous arbres, plus $l_1 \times c_n \times l_{k+1} = l_1 \times c_n \times c_k$ (car $c_i = l_{i+1}$). Il faut et il suffit d'essayer tous les $k$ possibles, et de calculer les sous arbres optimaux pour toutes les séquences $1 \dots k$, et $k \dots n$. Le principe de l'attribut-fonction simplifie grandement la programmation : on applique la formule récursive, sans avoir à savoir dans quel ordre faire les calculs.

## 5.1 Première implantation

Voici une version assez proche de celle de "Introduction à l'algorithmique" :

```
(* on multiplie n matrices a0 a1 ... an-1
   a.(i) a lig.(i) lignes et col.(i) colonnes.
   Il faut que col.(i)==lig.(i+1) bien sur
   on cherche le nb min de multiplications scalaires
   à effectuer, en jouant sur le parenthésage : le produit de
   matrices est associatif
*)

(*
la méthode est expliquée dans la bible, ie dans
"introduction à l'algorithmique"
de Cormen, Leiserson, Rivest, Stein.
Ici, j'utilise le principe de l'attribut-fonction (en pratique une
table de hachage) pour que la récursion fasse tout le travail; en
clair: je m'évite d'avoir à calculer
dans quel ordre il faut calculer (optimal i j) *)

module H=Hashtbl;;
module A=Array;;

let ht=H.create 17;;

let rec optimal ht lig col i j = try H.find ht (i,j)
with Not_found ->
    (let cout_k i j k = (optimal ht lig col i k)
                      + (optimal ht lig col (k+1) j)
                      + lig.(i) * col.(j) * col.(k) in
     let rec balaie i j k minimal_courant =
            if k=j then minimal_courant
            else balaie i j (k+1)
                 (min minimal_courant (cout_k i j k)) in
     let resultat =  if i=j then 0
            else balaie i j (i+1) (cout_k i j i) in
     H.add ht (i,j) resultat ; resultat
  );;

let cout_optimal lig col=
   let ht= H.create 17 in
   let cout = optimal ht lig col 0 (Array.length lig - 1) in
   H.fold (fun (i,j) cij liste -> ((i,j),cij)::liste) ht [];;
```

```
let rec iaj i j = if i=j then [i] else i::(iaj (i+1) j);;

let separation lig col couts i j =
   let cost i j = L.assoc (i,j) couts in
   let cij = cost i j in
   let cout_k i j k = (cost i k) + (cost (k+1) j)
                      + lig.(i) * col.(j) * col.(k) in
   (* chercher le k tel que cout_k i j k = cost i j *)
   L.hd (L.filter (function k-> cout_k i j k = cij) (iaj i (j-1)));;

let tailles = [| 30, 35; 35, 15; 15, 5; 5, 10; 10, 20; 20, 25 |];;
let lig = A.map fst tailles;;
let col= A.map snd tailles;;
let couts= cout_optimal lig col;;
separation lig col couts 0 5;;
(* rend 2 : l'optimal est de faire le produit (A0 A1 A2) * (A3 A4 A5) *)
(*  (A0 (A1 A2))  ((A3 A4) A5)   *)

(* question possible: construire l'arbre binaire associé *)
type arbre = Feuille of int | Noeud of arbre*arbre;;
let rec tree lig col couts i j =
   if i=j then Feuille i
   else let k=separation lig col couts i j in
   Noeud (tree lig col couts i k, tree lig col couts (k+1) j);;

tree lig col couts 0 5;;

(* variante: on peut ecrire une fonction optimal_tree qui rende
   l'arbre optimal et qui fabrique et compare des arbres; la table de
   hachage contient des arbres *)

let tailles = [| 30, 35; 35, 15; 15, 5; 5, 10; 10, 20;
                 20, 25; 25, 30; 30, 10 |];;
let lig = A.map fst tailles;;
let col= A.map snd tailles;;
let couts= cout_optimal lig col;;
tree lig col (cout_optimal lig col) 0 (Array.length tailles - 1);;
```

### 5.2 Seconde implantation

Et voici une version où on manipule purement et simplement les arbres binaires :

```
(* ceci est une deuxieme implanatation, + dans l'esprit fonctionnel:
   on génère, on compare, on stocke des arbres *)
module H=Hashtbl;;
module A=Array;;
type definition= Feuille of int | Noeud of arbre * arbre
and arbre = { cout: int; nblig:int; nbcol:int; def : definition };;

let rec iaj i j =
   if i>j then [] else if i=j then [i] else i::(iaj (i+1) j);;

let rec optimal_tree ht tablo i j=
```

```
(* tablo: tableau des feuilles; on cherche l'arbre optimal
   sur ces feuilles *)
        try H.find ht (i,j) (* s'il est dans la table de hachage *)
        with Not_found -> (* sinon on le calcule *)
                ( let resultat= compute_optimal_tree ht tablo i j in
                  H.add ht (i,j) resultat; resultat
                )
(* cut i j k rend le noeud avec
  fils gauche= optimal [i, k] et
  fils droit=optimal [k+1, j] *)
and cut ht tablo i j k =
     let ak=optimal_tree ht tablo i k in
     let ak' = optimal_tree ht tablo (k+1) j in
     let cost = ak.cout + ak'.cout + ak.nblig * ak'.nbcol * ak.nbcol in
     { cout=cost; nblig=ak.nblig; nbcol=ak'.nbcol; def=Noeud (ak, ak')}
and best_tree tree1 tree2 =
    if tree1.cout <= tree2.cout then tree1 else tree2
and compute_optimal_tree ht tablo i j =
    if i=j then tablo.(i)
    else List.fold_left best_tree (cut ht tablo i j i)
                     (List.map (cut ht tablo i j) (iaj (i+1) (j-1)));;

let optimal tablo =
        let ht=H.create 17 in
        let ot = optimal_tree ht tablo 0 (Array.length tablo - 1) in
        H.fold (fun key valeur liste -> (key,valeur)::liste) ht [] ;;

let tailles = [| 30, 35; 35, 15; 15, 5; 5, 10; 10, 20; 20, 25|];;
let feuilles = A.mapi (fun i (l,c) ->
                       { nblig=l; nbcol=c; def=Feuille i; cout=0 })
                     tailles;;
let trees = optimal feuilles;;
L.assoc (0,5) trees;;
```

# 6 La recherche arborescente (*backtrack*), le parcours en profondeur

Les programmes en ocaml réunis ici effectuent la recherche par parcours en profondeur (ou en largeur) de l'espace des possibles (*backtrack*). Les applications sont :
- la résolution du problème des reines
- la résolution du jeu : le compte est bon
- la résolution du casse tête des transvasements
- la résolution du casse tête des moutons (ou des chèvres)
- le coloriage (des sommets) de graphes
- le sudoku.

## 6.1 Un programme générique

Ce programme (fichier : depthsearch.ml) effectue une recherche en profondeur.

```
module L=List;;
module A=Array;;
type 'vertex pbm =
   { is_solution : 'vertex -> bool;
   neighbors : 'vertex -> 'vertex list;
   start_vertex : 'vertex };;

(* etape: effectue une etape de recherche.
l'entree est un couple: solutions connues, pile
si la pile est vide, rend l'entree : on ne peut rien faire
d'autre sinon, le premier chemin est depile, le premier
sommet du ce chemin est le sommet courant (le sommet initial
est le dernier element du chemin), si le sommet courant est
solution, le chemin est placé dans les solutions, sinon les
voisins du sommet courant sont calculés, ceux déjà présents dans
le chemin sont éliminés, et les chemins v::chemin sont empilés *)

let etape pbm ((known_solutions,stack) as e) =
match stack with [] -> e
| path::qstack -> let current_vertex=L.hd path in
  if pbm.is_solution current_vertex
  then (path::known_solutions, qstack)
  else let voisins = pbm.neighbors current_vertex in
    let voisins'= L.filter
        (function s-> not (L.mem s path)) voisins in
    let tmp= L.map (function s -> s::path) voisins' in
    (known_solutions, (tmp @ qstack)) ;;   (* Pour le parcours
en largeur, juste remplacer: (tmp @ qstack) par (qstack @ tmp) *)

let init_search pbm = ([], [[pbm.start_vertex]]);;

(* rend f(f(f...x0)) jusqu'a ce que condition soit vrai *)
let rec until f x0 condition =
   if condition x0 then x0 else until f (f x0) condition;;

let df_search pbm = until (etape pbm) (init_search pbm)
                          (function (sols,stack)-> []=stack);;

let rec fixpt egal f x0 =
    let x1= f x0 in if egal x0 x1 then x1 else fixpt egal f x1;;

(* autre definition possible de df_search *)
let df_search' pbm = fixpt ( == ) (etape pbm) (init_search pbm) ;;

(* prend en entrée un état de la recherche, et continue jusqu'à
trouver 1 nouvelle solution, ou jusqu'à ce que la pile soit vide *)
let rec next_solution pbm state_of_search =
   let next_state = etape pbm state_of_search in
   if fst next_state == fst state_of_search
      && snd state_of_search <>[]
   then next_solution pbm next_state
   else next_state;;

let solve_1 pbm = (* s'arrête à la premiére solution *)
```

```
  next_solution pbm (init_search pbm);;

(* exemple utilisation:
let path_pbm graphe a z =
      { is_solution = (function s -> s=z);
        neighbors = graphe; start_vertex = a };;
let mon_graphe x=
     if x<=5 then [ x/2; x*2; x*2 + 1 ] else [ x/2; x/3];;
let pb1= path_pbm mon_graphe 1 5;;
df_search pb1;;
solve_1 pb1;;
next_solution pb1 (solve_1 pb1);; *)
```

## 6.2 Application : le problème des reines

Ce programme (fichier : reine.ml) résout le problème des reines.

```
include Depthsearch;;
type echiquier = { reine_a_poser: int; pos: int array};;
(* pos.( une_colonne )= une_ligne *)

(* est ce que la dernière reine posée est correcte ? *)
exception Non;;

(* Dans la dernière colonne, peut-on poser la reine à
la hauteur ?
On pourrait le faire de façon récursive
(cf fonction essayer + bas), mais on utilise
une boucle pour montrer que c'est aussi possible *)

let possible pos derniere hauteur =
try  for i=0 to derniere-1 do
        if pos.(i) = hauteur then raise Non;
        if abs(pos.(i)-hauteur)=abs(i-derniere)
        then raise Non;
     done;
     true;
with Non -> false;;

(*voici la version récursive :
let possible pos derniere hauteur =
  let rec pas_de_conflit_avec i =
     i=derniere || pos.(i) <> hauteur
     && abs(pos.(i)-hauteur)<>abs(i-derniere)
     && pas_de_conflit_avec (i+1)
  in pas_de_conflit_avec 0;;
*)

let poser_reine tab colonne ligne =
  let n=A.length tab in
  let tab' = A.append (A.sub tab 0 colonne)
                      (A.make (n-colonne) ligne) in
  { reine_a_poser=colonne+1; pos=tab' };;
```

```
let queen_neighbors { reine_a_poser=colonne; pos=tab }=
   let n= A.length tab in
   if colonne=n then [] (* plus de reine à poser *)
   else let rec essayer ligne fistons =
      if ligne<0 then fistons else
      if possible tab colonne ligne
      then essayer (ligne-1)
              ((poser_reine tab colonne ligne)::fistons)
      else essayer (ligne-1) fistons in
   essayer (n-1) [];;

let queen_pbm n =
 { is_solution=( function { reine_a_poser=colonne; pos=tab }
                      -> colonne=A.length tab);
   neighbors = queen_neighbors;
   start_vertex = { reine_a_poser=0; pos=A.create n 0} };;

let solve_queen n =
     L.map (function path -> (L.hd path).pos)
           (fst (df_search (queen_pbm n))) ;;
(* on ne garde, dans les solutions, que le sommet terminal
de chaque chemin *)

let demo n=
   let (solutions,_)= df_search (queen_pbm n) in
   let echiquiers = L.map L.hd solutions in
   L.iter (function {pos=pos} ->
   A.iter (function x->
         Printf.printf "%2d " x) pos; Printf.printf "\n"
      ) echiquiers;;

demo 10;;
```

Remarques :

- il n'y a pas de risque de boucle avec le problème des reines ; on pourrait donc supprimer le test effectué dans la fonction étape. Exercice : ajouter un champ à la définition d'un problème, et en te tenir compte dans la fonction étape.

- les reines sont posées de "gauche à droite", *i.e.* de la colonne 0 à la colonne $n-1$. En fait, il vaudrait mieux calculer quelle est la colonne la plus contrainte, et poser la reine dans cette colonne. Cette optimisation est utilisée pour le coloriage d'un graphe (cf plus loin).

## 6.3 Application : le problème des moutons (chèvres)

Une file de moutons noirs rencontre une file de moutons blancs sur un chemin très étroit ; entre les deux files, il y a juste la place nécessaire pour un mouton. Le but est d'arriver à faire se croiser les deux files. Chaque mouton noir peut avancer d'une case vers la droite, et sauter par dessus le mouton qui est devant lui (une case à droite) si la case où il arrive est libre. Symétriquement, chaque mouton blanc peut avancer d'une case vers la gauche, et sauter par dessus le mouton qui est devant lui (une case à gauche) si la case où il arrive est libre. Les moutons ne peuvent pas reculer.

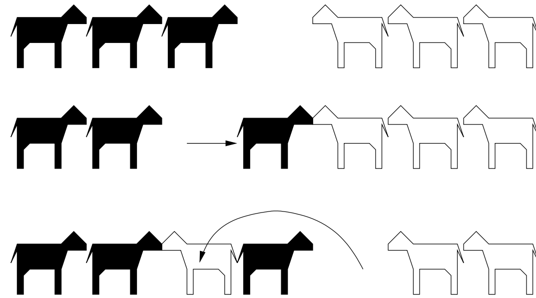

Fig. 1 –

Le chemin est représenté par un tableau de cases, une case contient soit un mouton noir (qui avance vers la droite), soit un mouton blanc (qui avance vers la gauche), soit est vide. Initialement la situation est : Noir, Noir, Noir, Vide, Blanc, Blanc, Blanc, et il faut passer à la situation Blanc, Blanc, Blanc, Vide, Noir, Noir, Noir. Il faut trouver toutes les situations intermédiaires. L'affichage (graphique) n'est pas demandé.

Le fichier chevres.ml contient une solution possible.

```
include Depthsearch;;

type config = {cases : int array;} ;;

(* determine la position de la case vide (0) *)
let rec c_vide cas i =
        if cas.cases.(i) = 0 then i
        else c_vide cas (i+1);;
(* échange deux places données i et j *)
let swap chem i j =
        let config={cases=Array.copy chem.cases} in
        let cour = config.cases.(i) in
        config.cases.(i)<-config.cases.(j);
        config.cases.(j)<-cour;
      config;;

let mouton_noir_avance cas =
       let i=c_vide cas 0 in i>0 && cas.cases.(i-1)=1 ;;
let mouton_blanc_avance cas =
          let i=c_vide cas 0 in
          i+1<A.length cas.cases && cas.cases.(i+1)=2 ;;
let mouton_noir_saute cas =
    let i=c_vide cas 0 in i-2>=0 && cas.cases.(i-2)=1;;
let mouton_blanc_saute cas = let i=c_vide cas 0 in
      i+2<A.length cas.cases && cas.cases.(i+2)=2 ;;

let config_voisines cas =
 if cas.cases=[|2;2;2;0;1;1;1|] then []
 else let i = c_vide cas 0 in
   (if (mouton_noir_avance cas) then [swap cas (i-1) i] else []) @
   (if (mouton_blanc_avance cas) then [swap cas (i+1) i] else []) @
   (if (mouton_noir_saute cas) then [swap cas (i-2) i] else []) @
```

```
    (if (mouton_blanc_saute cas) then [swap cas (i+2) i] else []);;

let pb_moutons nbblancs nbnoirs =
  let config_depart =
      A.concat [ A.make nbblancs 1; [| 0|]; A.make nbnoirs 2 ] in
  let config_finale=
      A.concat [ A.make nbnoirs 2; [| 0|]; A.make nbblancs 1 ] in
  let config_is_solution {cases = cases} =
      (cases=config_finale) in
  { neighbors= config_voisines;
  is_solution = config_is_solution;
  start_vertex = {cases=config_depart} } ;;

let demo nbblancs nbnoirs =
      let pb=pb_moutons nbblancs nbnoirs in
      let (paths,_)=df_search pb in
      let print_cases { cases=cases } =
            for i=0 to A.length cases-1 do
                Printf.printf "%d" cases.(i)
            done;
            Printf.printf "\n"; flush stdout
      in
      L.iter (function path-> L.iter print_cases path;
                 Printf.printf "------------------------------\n")
            paths;;

if not !Sys.interactive then ( demo 3 3; demo 3 5; demo 6 7 );;
```

### 6.4 Application : le transvasement

Soient 3 récipents de contenances maximales : 3, 5, 8 litres. Initialement ils contiennent 0, 0 8 litres. Il faut arriver à la situation 0, 4, 4 litres. Les récipients sont non gradués : on ne peut donc que verser un récipient $A$ dans un récipient $B$, jusqu'à ce que $A$ soit vide ou que $B$ soit plein. Voici un programme solution (fichier : transvasement.ml).

```
include Depthsearch;;

let rec uniq l = match l with [] -> []
| t::q -> if L.mem t q then uniq q else t::(uniq q);;

let verser maximum contenus i j=
   let delta = min contenus.(j) (maximum.(i) - contenus.(i)) in
   let tab = A.copy contenus in
   tab.(i) <- tab.(i) + delta; tab.(j) <- tab.(j)-delta;
   tab;;

(* avec des boucles et des affectations
let transvase_voisins maximum contenus=
   let l=ref [] in
   let n= A.length contenus-1 in
   for i=0 to n do for j=0 to n do
     if i<>j then l:=verser maximum contenus i j :: !l
   done done;
```

```
    uniq !l;;
*)
(* sans boucle et sans affectation : *)
let transvase_voisins maximum contenus=
   let n=A.length contenus in
   let rec auxi i liste =
      let rec auxj j liste =
         if j=n then liste
         else auxj (j+1) ((verser maximum contenus i j)::liste)
      in if i=n then liste else auxi (i+1) (auxj 0 liste)
   in uniq (auxi 0 []);;

let pbm_transvasement maximum contenus_initiaux contenus_finaux=
   { neighbors = transvase_voisins maximum;
     is_solution  = (function contenus->contenus=contenus_finaux);
     start_vertex = contenus_initiaux };;

let pbm358= pbm_transvasement [| 3;5;8|] [|0;0;8|] [|0;4;4|];;

let demo pbm=
   let (solutions,_)=df_search pbm in
   L.iter (function solution ->
            L.iter (function tab -> A.iter
                              (function c->Printf.printf "%2d" c)
                              tab;
                            Printf.printf "\n") solution;
             Printf.printf "------------------------------\n"
           ) solutions;;
demo pbm358;;
```

## 6.5 Application : le compte est bon

Un célèbre jeu télévisé... C'est le fichier : compteestbon.ml.

```
include Depthsearch;;

type op = Plus | Moins | Mult | Div | Initial of int ;;
type plaque = { valeur : int; def : op list  };;

let initial x = { valeur = x; def = [Initial x] };;
let plus a b = { valeur = a.valeur + b.valeur;
                 def = a.def @ b.def @ [Plus] };;
let moins a b = { valeur = a.valeur - b.valeur;
                 def = a.def @ b.def @ [Moins] };;
let mult a b = { valeur = a.valeur * b.valeur;
                 def = a.def @ b.def @ [Mult] };;
let div a b = { valeur = a.valeur / b.valeur;
                 def = a.def @ b.def @ [Div] };;

let rec allpairs liste f =  match liste with
| [] -> []
| t::q ->  (L.map (f t) q) @ (allpairs q f) ;;
(*
#   allpairs [1;2;3;4] (fun x y -> x,y) ;;
```

```
- : (int * int) list = [(1, 2); (1, 3); (1, 4);
    (2, 3); (2, 4); (3, 4)]
*)

let fils plaques = L.flatten (
  allpairs plaques (fun a b ->
    let plaques_ab = L.filter
        (function p-> not(p == a || p == b)) plaques in
    let plaques_filles =
    [ plus a b;  mult a b;
      if a.valeur - b.valeur > 0 then moins a b else moins b a; ]
    @ (if b.valeur <> 0 && a.valeur mod b.valeur = 0
       then [ div a b] else [] )
    @ (if a.valeur <> 0 && b.valeur mod a.valeur = 0
       then [ div b a] else [] ) in
    L.map (function p -> p::plaques_ab) plaques_filles));;

let pbm_compte_est_bon but nb_initiaux=
{  neighbors = fils;
   is_solution = (function plaques ->
     L.exists (function plaque->plaque.valeur=but) plaques );
   start_vertex = L.map initial nb_initiaux };;

let pb2= pbm_compte_est_bon 560 [1;2;10;25;6;7];;

let pb3= pbm_compte_est_bon 561 [1;2;10;25;6;7];;

(* df_search est un peu lent, car l'espace exploré est grand.
Pour l'accélérer, on peut remarquer qu'il n'y a pas de risque
de boucle (comme pour les reines d'ailleurs) puisque le
nombre de plaques diminue avec la profondeur dans l'arbre de
recherche. Dans la fonction étape, il est donc inutile de
tester que les voisins du sommet courant n'ont pas déjà été
atteints. Ajouter un champ au pbm, disant s'il y a risque de
boucle ou non. Modifier ensuite la fonction etape pour tenir
compte de ce champ. *)
```

## 6.6 Coloriage de graphes

Le coloriage d'un graphe (non orienté) peut être effectué par une méthode de recherche arborescente, *i.e.* un parcours en profondeur. Les coloriages voisins d'un coloriage colorent un sommet non coloré jusqu'ici, qui est le plus contraint (qui a le moins de couleurs possibles). En particulier, si un coloriage est tel qu'un sommet ne peut plus être coloré, alors ce coloriage n'a pas de voisin. Ce choix élague efficacement l'arbre de recherche.

```
(* coloriage d'un graphe *)
(* un exemple d'utilisation est : sudoku.ml *)

include Depthsearch;;

type 'couleur sommet_colorie = Colorie of 'couleur
         | Possible of 'couleur list;;
```

```
type 'couleur coloriage =
      { voisins: int list array ;
        couleurs: ('couleur sommet_colorie) array };;
(* voisins.(i) = les voisins du sommet i;
le graphe est supposé correct (càd symmetrique) *)

let rec iaj i j = if i=j then [i] else i::(iaj (i+1)j);;

(* rend une variante de coloriage, où le sommet s est
colorié en coul_s; cela met à jour les couleurs possibles
pour les voisins de s. On ne vérifie pas qu'il n'y a pas
de conflit ; c'est garanti par ailleurs *)

let colorier coloriage s coul_s =
   let coloriage'= { coloriage with
                     couleurs=A.copy coloriage.couleurs } in
   coloriage'.couleurs.(s) <- Colorie coul_s;
   L.iter (function v-> match coloriage'.couleurs.(v) with
      | Colorie cv -> assert (cv <> coul_s)
      | Possible l -> coloriage'.couleurs.(v) <-
           Possible (L.filter (function col->col<>coul_s) l)
      ) coloriage.voisins.( s);
   coloriage';;

(* rend le sommet du coloriage qui a le moins
de couleurs possibles, éventuellement aucune.
Rend celui d'indice min, quand il y en a plusieurs *)
let sommet_le_plus_contraint ({couleurs=couleurs} as coloriage)=
 let rec examiner s result =
     if s=A.length couleurs then result
     else match couleurs.(s) with
        | Colorie _ -> examiner (s+1) result
        | Possible l -> let ll = L.length l in
          match result with None -> examiner (s+1) (Some (s,ll))
           | Some (sommet,lsommet) ->
             if ll<lsommet then examiner (s+1) (Some( s, ll))
             else examiner (s+1) result
   in examiner 0 None ;;

let coloriage_termine coloriage =
      (None = sommet_le_plus_contraint coloriage);;

let fils_coloriage ({couleurs=couleurs } as coloriage) =
   match sommet_le_plus_contraint coloriage with
   | None -> []
   | Some (s,ls) -> match couleurs.(s) with
      | Colorie _ -> failwith "bug_in_coloriage"
      | Possible coul_s ->
         L.map (function col-> colorier coloriage s col) coul_s;;

let pbm_coloriage coloriage=
   { is_solution = coloriage_termine; neighbors = fils_coloriage;
     start_vertex = coloriage } ;;
```

Remarques :

- avec le coloriage, il n'y a pas de risque de boucle.

- un coloriage est un tableau, avec une case par sommet ; ce tableau est copié (en totalité) lorsqu'on passe d'un coloriage à un autre, alors que seule la couleur d'un sommet est modifiée. Cela peut être optimisé, en stockant les valeurs $T_0, T_1 \dots T_{n-1}$ dans les feuilles d'un arbre binaire ; lorque les $T_i$ sont copiés dans $T'$, et que seul $T_k$ est modifié, il est possible de partager beaucoup de sous arbres entre $T$ et $T'$ ; en fait seuls $\log n$ noeuds de $T$ doivent être copiés (les noeuds qui sont ancètres de la feuille $T_k$), tous les autres sont partagés. Ainsi colorier un sommet se fait en $O(\log n)$ au lieu de $O(n)$.

## 6.7 Application du coloriage : le sudoku

Le programme suivant (fichier : sudoku.ml) permet de résoudre les problèmes du sudoku.

```
include Coloriage;;

let rec iaj i j = if i=j then [i] else i::(iaj (i+1)j);;

(* pour le graphe du  sudoku *)
let n_of_xy x y = x + 9*y;;
let x_of_n n = n mod 9;;
let y_of_n n = n / 9;;
let voisins_horizontaux n =
      assert ( 0 <= n && n < 81);
      let y= y_of_n n in
      let l=L.map (function x-> n_of_xy x y) (iaj 0 8)
      in L.filter (function x-> x <> n) l;;
let voisins_verticaux n =
      assert ( 0 <= n && n < 81);
      let x= x_of_n n in
      let l=L.map (function y-> n_of_xy x y) (iaj 0 8)
      in L.filter (function x-> x <> n) l;;
let rec allpairs fonc resultat l1 l2 =
   match l1 with [] -> resultat
   | e1::q1 ->
      allpairs
         fonc
         (L.fold_left (fun result e2-> fonc result e1 e2)
                             resultat l2)
         q1 l2;;
(*
# allpairs (fun l x y -> (x,y)::l) [] (iaj 1 2) (iaj 10 11) ;;
- : (int * int) list = [(2, 11); (2, 10); (1, 11); (1, 10)]
*)
let voisins_du_carre n =
      assert ( 0 <= n && n < 81);
      let x= x_of_n n in let y= y_of_n n in
      let xx= x/3 in let yy=y/3 in
      let tx=xx*3 in let ty=yy*3 in
      let l=allpairs (fun liste x y ->
                                    (n_of_xy x y)::liste) []
                         (iaj tx (tx+2)) (iaj ty (ty+2)) in
      L.filter (function k -> k<> n) l;;
```

```
let rec uniq_sorted l = match l with [] | [_] -> l
   | x1::x2::q -> if x1=x2
                    then uniq_sorted (L.tl l)
                    else x1::(uniq_sorted (L.tl l));;
let graphe_du_sudoku  =
   let vs= A.make 81 [] in
   for i=0 to 80 do vs.(i) <- uniq_sorted (
      L.sort ( - ) ((voisins_horizontaux i)
                          @ (voisins_verticaux i)
                          @ (voisins_du_carre i)))
   done;
   vs;;
let grille_non_contrainte ()=
   let possibles = A.make 81 (iaj 1 9) in
   {
      voisins = graphe_du_sudoku;
      couleurs= A.make 81 (Possible (iaj 1 9))
   };;
let fixe_couleur grille (i,j,coul) =
    colorier grille (n_of_xy i j) coul;;
let fixe_couleurs grille liste =
    L.fold_left fixe_couleur grille liste;;

(* grilles prises dans Quadrature 62,
"sudokus et algorithmes de recuit", par Renaud Sirdey,
page 12 *)

let grille1 = fixe_couleurs (grille_non_contrainte()) [
0,0,1; 0,2,7; 0,8,6;
1,0,9; 1,7,1; 2,4,2; 2,6,8;
3,1,6; 3,5,1; 4,1,2; 4,4,3;
5,1,9; 5,3,8; 5,7,5; 5,8,3;
6,0,6; 6,8,7; 7,4,6; 7,6,3; 7,7,9; 7,8,8;
8,3,9; 8,6,1 ];;

let grille2 = fixe_couleurs (grille_non_contrainte())
[ 1,1,5; 2, 0, 2; 3, 1, 9; 4, 2, 6; 5, 0, 4; 6, 0, 9;
6, 1, 7; 8, 1, 1; 0, 3, 9; 1, 5, 2; 2, 4, 5;
3, 3, 6; 3, 5, 5; 5, 3, 3; 5, 5, 7; 6, 4, 4;
7, 3, 5; 8, 5, 8; 0, 7, 6; 2, 7, 3; 2, 8, 9; 3, 8, 1;
4, 6, 8; 5, 7, 9; 7, 7, 7; 8, 6, 4 ];;

let grille3= fixe_couleurs (grille_non_contrainte())
[ 0,0,8; 0,2,3; 0,5,6;
1,1,6; 1,4,2; 1,6,8; 1,7,7; 1,8,5;
2,0,7; 3,1,3; 3,4,5; 4,6,5;
5,1,4; 5,2,2; 5,4,1; 5,7,3; 5,8,7;
6,2,8; 6,4,4; 6,5,9; 6,7,2;
7,6,1; 7,8,3;
8,0,4; 8,5,2 ];;

(* pour obtenir toutes les grilles, à une permutation pres
des lignes et des colonnes, on impose le carre de 3 par 3 en
bas à gauche, ainsi que la 1ere ligne et la 1ere colonne *)
```

```
let grille4= fixe_couleurs (grille_non_contrainte()) [
0,0,1; 0,1,2; 0,2,3; 0,3,4; 0,4,5; 0,5,6; 0,6,7; 0,7,8; 0,8,9;
1,0,4; 1,1,5; 1,2,6;
2,0,7; 2,1,8; 2,2,9;
3,0,2; 4,0,5; 5,0,8; 6,0,3; 7,0,6; 8,0,9 ];;

let print_grille {voisins=_; couleurs=couleurs} =
   let trait()=
      Printf.printf " -------------------------------\n" in
   for ligne=8 downto 0 do
      if ligne mod 3 = 2 then trait();
      for col=0 to 8 do
         if col mod 3 = 0 then Printf.printf " |";
         let n= n_of_xy col ligne in
         match couleurs.(n) with
         | Colorie c -> Printf.printf " %d" c
         | Possible l-> if l=[] then Printf.printf " #"
                        else Printf.printf " ?"
      done;
      Printf.printf " |\n";
   done; trait(); Printf.printf "\n\n";;

let solve grille =
   let grilles_solutions= L.map L.hd
      (fst( Depthsearch.df_search (pbm_coloriage grille))) in
   L.iter print_grille grilles_solutions;
   grilles_solutions;;

let lister grille=
   let pb = pbm_coloriage  grille in
   let rec etape state =
      let state' =next_solution pb state in
      if state'==state then ()
      else (
         let solution =L.hd( L.hd (fst state)) in
         print_grille solution; print_newline();
         etape state'
      )
in etape (next_solution pb (init_search pb));;

let demo()=
   ignore( solve grille1);
   ignore( solve grille2);
   ignore( solve grille3);
   lister grille4;;
demo();;
```

# 7 Calcul des racines réelles d'un polynôme

1) La base de Bernstein permet de résoudre l'équation algébrique $f(x) = 0$ avec $x$ dans $[0..1]$.

Les racines de $f(x)$ dans $[1, +\infty]$ sont les inverses des racines du polynôme

$g(X)=0$ avec $X=1/x$. Si $f(x)=f_0+f_1x+\dots f_nx^n$, alors $g(X)=f_0x^n+f_1X^{n-1}+\dots f_n$ (trivial).

Les racines négatives de $f(x)$ sont les opposées des racines de $h(X)$ avec $X=-x$. Les coefficients de $h$ sont : $h_{2i}=f_{2i}$ et $h_{2i+1}=-f_{2i+1}$.

Il suffit donc de résoudre $f(x)=0$ avec $x$ dans $[0..1]$ ; on convertit d'abord la forme canonique dans la forme de Bernstein, en utilisant le fait que ($n$ st le degré du polynôme) :

$$x^k=\sum_{i=0}^{i=n}C(k,i)/C(k,n)B_i^n(x)$$

où $C(i,n)$ est le nombre de façons de choisir $i$ objets parmi $n$ : $C(0,n)=1$, $C(i>n,n)=0$, $C(1,n)=n$, $C(i,n)=C(i-1,n)\times(n-i+1)/i$.

Soit $f(x)=\sum p_iB_i^{(n)}(x)$ la forme de Bernstein de $f$ ; alors la courbe de Bézier a pour points de contrôle : $(i/n,p_i)$ pour $i$ de 0 à $n$.

La méthode isoler( $p_i,x_0,x_1$), pour isoler les racines de $f$, avec initialement $x_0=0$ et $x_1=1$ est alors : si l'intervalle des $p_i$ ne contient pas 0, $f$ n'a pas de racine dans l'intervalle courant $x_0,x_1$. Si l'intervalle courant $x_0,x_1$ est d'amplitude inférieure à $\epsilon=10^{-6}$, alors $x_0+x_1)/2$ est (une approximation d'une) racine de $f$ ; sinon on coupe la courbe de Bézier en 2 par la méthode de Casteljau : elle fournit les points de contrôle $p_i'$ de la courbe entre 0 et 1/2 ($x$ entre $x_0$ et $x_m=(x_0+x_1)/2$, et les points de contrôle $p_i''$ de la courbe entre 1/2 et 1 ($x$ entre $_m$ et $x_1$). isoler( $p_i',x_0,x_m$) et isoler( $p_i'',x_m,x_1$) sont appelées récursivement.

Au fur et à mesure, les racines trouvées sont stockées.

2) Les racines de $f(x)=0$ sont dans l'intersection entre l'enveloppe convexe des points de contrôle $P_i=(x_i=i/n,y_i=p_i)$ et l'axe $y=0$. On ne coupe plus bêtement en 2 la courbe de Bézier. Donner une méthode de calcul de l'enveloppe convexe en 2D.

3) Cette méthode peut calculer l'intersection entre une droite

$$x=x_0+at,y=y_0+bt,z=z_0+ct$$

et une surface implicite algébrique d'équation :

$$F(x,y,z)=\sum_\alpha\sum_\beta\sum_\gamma F_{\alpha\beta\gamma}x^\alpha y^\beta z^\gamma=0$$

Après substitution de $x,y,z$ :

$$\begin{aligned}
&F(x,y,z)=\sum_\alpha\sum_\beta\sum_\gamma F_{\alpha,\beta,\gamma}x^\alpha y^\beta z^\gamma\\
&=\sum_\alpha\sum_\beta\sum_\gamma F_{\alpha,\beta,\gamma}\left(\sum_{i=0}^{\alpha}C(i,\alpha)x_0^{\alpha-i}a^it^i\right)\left(\sum_{j=0}^{\beta}C(j,\beta)y_0^{\beta-j}b^jt^j\right)\left(\sum_{k=0}^{\gamma}C(k,\gamma)z_0^{\gamma-k}c^kt^k\right)\\
&=\sum_\alpha\sum_\beta\sum_\gamma\sum_{i=0}^{\alpha}\sum_{j=0}^{\beta}\sum_{k=0}^{\gamma}F_{\alpha,\beta,\gamma}C(i,\alpha)C(j,\beta)C(k,\gamma)x_0^{\alpha-i}y_0^{\beta-j}z_0^{\gamma-k}a^ib^jc^kt^{i+j+k}\\
&=f(t)=\sum_{d=0}^{n}f_dt^d=0
\end{aligned}$$

en utilisant le fait que : $\sum_i P_i \sum_j Q_j = \sum_i \sum_j P_i Q_j$. Initialiser les $f_d$ à 0, pour $d$ de 0 à $n$ (le degré de la surface), et pour chaque monôme $F_{\alpha,\beta,\gamma}x^\alpha y^\beta z^\gamma$ de $F$, faire : pour $i = 0 \ldots \alpha$, pour $j = 0 \ldots \beta$, pour $k = 0 \ldots \gamma$, incrémenter le coefficient $f_{i+j+k}$ du monome $t^{i+j+k}$ du polynome $f(t)$ de la quantité :

$$F_{\alpha,\beta,\gamma} C(i,\alpha) C(j,\beta) C(k,\gamma) x_0^{\alpha-i} y_0^{\beta-j} z_0^{\gamma-k} a^i b^j c^k$$

Ensuite, résoudre l'équation en $t$ par la méthode précédente.

```
let rec choices i n =
   if i=0 then 1
   else if i>n then 0
   else if i=1 then n
   else ((choices (i-1) n) * (n - i + 1)) / i ;;

type polynome = { coeffs: float array };;

let to_bernstein a_i =
   let n = Array.length a_i - 1 (* n est le degre *) in
   let b_i = Array.make (n+1) 0. in
   (* p(x)= sum a_i x^i *)
   (* x^k = sum_(i=0..n) C(k,i)/C(k,n) B(i,n)(x) *)
   for k=0 to n do
      for i=k to n do
         let c_k_i = float_of_int (choices k i) in
         let c_k_n = float_of_int (choices k n) in
         b_i.(i) <- b_i.(i) +. a_i.(k) *. c_k_i /. c_k_n
      done
   done;
   b_i;;

let milieu x0 x1 = (x0 +. x1)/. 2.;;

(* rend: (left,right) où left est le tableau des coefficients
de la moitié gauche, et right est le tableau des coefficients
de la moitié droite. Utilise la méthode de Casteljau
En entrée, coeffs est le tableau des coeffs dans la base de
Bernstein *)

let deCasteljau coeffs =
   let n = Array.length coeffs - 1 (* n est le degre *) in
   let left= Array.make (n+1) 0. and right= Array.make (n+1) 0. in
   let layers = Array.make (n+1) coeffs in
   for level=1 to n do
      layers.(level) <- Array.make (n+1-level) 0.;
      for k=0 to n-level do
        layers.(level).(k) <-
         milieu layers.(level-1).(k)
                                layers.(level-1).(k+1))
      done;
   done;
   for k=0 to n do left.(k) <- layers.(k).(0);
       right.(k) <- layers.(n-k).(k)
```

```
        done;
    left, right;;

let minmax tableau =
    let mini = ref tableau.(0) and maxi= ref tableau.(0) in
    for i=1 to Array.length tableau - 1 do
        mini := min !mini tableau.(i);
        maxi := max !maxi tableau.(i)
    done;
    !mini, !maxi;;

(* b est le tableau des coeffs, dans la base de Bernstein *)
(* epsilon est la précision souhaitée *)
let roots_0_1_bernstein epsilon b =
    let rec search x0 x1 coeffs known_roots =
        let xm = milieu x0 x1 in
        let (mi,ma)=minmax coeffs in
    (* Printf.printf "x0=%f x1=%f in [%f %f]\n" x0 x1 mi ma; *)
        if ma < 0. || mi > 0. then known_roots
        else if x1 -. x0 < epsilon then xm::known_roots
             else let left,right=deCasteljau coeffs in
             search x0 xm left (search xm x1 right known_roots)
    in
    let x0= 0. and x1=1. in
    let roots= search x0 x1 b [] in  roots;;

deCasteljau (to_bernstein [| 1.; 0.; 0. |] );;
let ai= [| 0.; 0.; 1. |];;
let bi= to_bernstein ai;;
to_bernstein [| 1.; 0.; 0. |];;
to_bernstein [| 0.; 1.; 0. |];;

(*
# let pol_ca = [| -0.5; 0.; 1. |];;
val pol_ca : float array = [|-0.5; 0.; 1.|]
# let pol_be = to_bernstein pol_ca;;
val pol_be : float array = [|-0.5; -0.5; 0.5|]
# roots_0_1_bernstein 0.00001 pol_be;;
- : float list = [0.707103729248046875]
*)

let roots_0_1_canonic epsilon a =
    roots_0_1_bernstein epsilon (to_bernstein a);;
let roots_1_infini_canonic epsilon a =
    let a' = Array.of_list (List.rev (Array.to_list a)) in
    let roots' = roots_0_1_canonic epsilon a' in
    List.rev (List.map (function x-> 1. /. x) roots');;

let roots_pos_canonic epsilon pol =
    roots_0_1_canonic epsilon pol
      @ roots_1_infini_canonic epsilon pol;;
let roots_neg_canonic epsilon pol =
    let pol'= Array.copy pol in
    for i=1 to Array.length pol - 1 do
```

```
        if i mod 2 = 1 then pol'.(i) <- 0. -. pol'.(i)
      done;
  let roots' = roots_pos_canonic epsilon pol' in
  List.map (function x-> 0. -. x ) roots' ;;

(* le polynome est dans la base canonique :
   polynome.(i)=coeff de x^i *)
let solve epsilon polynome =
  roots_neg_canonic epsilon polynome
    @ roots_pos_canonic epsilon polynome;;

(*
let eval_pol polynome x =
  let n= Array.length polynome - 1 in
  let accu = ref polynome.(n) in
  for i=n-1 downto 0 do
    accu := !accu *. x +. polynome.(i)
  done;
  !accu;;
*)

let eval_pol polynome x =
  let n= Array.length polynome - 1 in
  let rec aux i accu =
    if i<0 then accu
    else aux (i-1) (accu *. x +. polynome.(i))
  in aux (n-1) polynome.(n);;

let derivative polynome =
  let n=Array.length polynome - 1 in
  let der= Array.make n 0. in
  for i=0 to n-1 do der.(i) <- (float_of_int (i+1)) *. polynome.(i+1)
  done;
  der;;

(* un newton pour les polynomes : *)
(* newton 0.000001 [| -2.; 0. ; 1. |] 10. ;;
   rend 1.41421356237309537 = sqrt(2.) *)
let newton epsilon pol x0 =
  let pol' = derivative pol in
  let f x = x -. (eval_pol pol x) /. (eval_pol pol' x) in
  let rec iter x =
    let x'=f x in
    if abs_float (x' -. x) < epsilon then x' else iter x'
  in iter x0;;

(* un newton pour les fonctions : *)
(*
# newton_f epsilon (function x -> x *. x -. 2. ) 10.;;
- : float = 1.4142135623730967
*)
let newton_f epsilon f x0 =
  let f' x = (f (x +. epsilon) -. f x) /. epsilon in
  let f x = x -. (f x) /. f' x in
```

```
let rec iter x =
          let x'=f x in
          if abs_float (x' -. x) < epsilon then x' else iter x'
    in iter x0;;
```