

Tour de cavaliers

February 20, 2022

Attention ce qu'on va faire ici n'est pas facile. Je vous demande d'abord de reprendre les deux exemples des huit reines puis du Sudoku https://marcdefalco.github.io/chap_algorithmique_force_brute.html et de faire marcher le code ici dans un autre notebook. C'est en comprenant bien les exemples du cours qu'on peut construire la suite dessus.

On rappelle qu'un cavalier, aux échecs, se déplace en L : une case dans une direction, horizontale ou verticale, deux case dans l'autre.

Dans ce TP on va résoudre le problème du cavalier d'Euler : étant donné un échiquier rectangulaire $n \times m$ et une position initiale d'un cavalier en (x_0, y_0) :

- existe-t-il une succession de mouvements du cavalier permettant de visiter chaque case de l'échiquier une fois et une seule ? On parle de *tour* du cavalier.
- existe-t-il un tour du cavalier qui permettent avec un mouvement supplémentaire de revenir au point de départ (x_0, y_0) ? On parle de *tour fermé* du cavalier.

Un tour ouvert (non fermé) :

Un tour fermé :

On va représenter un échiquier par le type `OCaml` :

```
[1]: type echiquier = int option array array
```

```
[1]: type echiquier = int option array array
```

- L'idée est qu'une valeur `None` signifie que la case n'a pas été visitée.
- Une valeur `Some k` signifie que le cavalier a visité cette case en kème position dans son tour, en commençant à numéroter à 1 pour la case initiale.

Exemple de numérotation finale d'un tour fermé d'un échiquier 6x6

```
1| 6|15|18|35|26|
8|19|36|25|14|17|
5| 2| 7|16|27|34|
20| 9| 4|31|24|13|
 3|30|11|22|33|28|
10|21|32|29|12|23|
```

On va faire le choix d'énumérer par colonnes, ce qui permet d'écrire `e.(x).(y)` pour avoir le point de coordonnées (x, y) . Ainsi, l'échiquier

```

  |   |   |
1| 2| 3|

```

sera représenté par la valeur

```
[| [| None; Some 1 |]; [| None; Some 2 |]; [| None; Some 3 |] |]
```

```
[4]: let libre (e:echiquier) (x,y) : bool =
      (* vérifie si on peut se déplacer en (x,y) *)
      e.(x).(y) = None
```

```
[4]: val libre : echiquier -> int * int -> bool = <fun>
```

```
[5]: let dans_echiquier (e:echiquier) (x,y) : bool =
      (* vérifie que les coordonnées (x,y) sont valides pour cet échiquier *)
      let w = Array.length e in
      let h = Array.length e.(0) in
      0 <= x && x < w && 0 <= y && y < h
```

```
[5]: val dans_echiquier : echiquier -> int * int -> bool = <fun>
```

```
[6]: let valide (e:echiquier) (x,y) : bool =
      (* vérifie que (x,y) est une coordonnée valide et que la case est libre *)
      dans_echiquier e (x,y) && libre e (x,y)
```

```
[6]: val valide : echiquier -> int * int -> bool = <fun>
```

```
[7]: let deplacements (x,y) : (int * int) list =
      (* renvoie les 8 déplacements possible d'un cavalier en (x,y)
      quitte à avoir des coordonnees invalides *)
      [ (x-2,y-1); (x-2,y+1);
        (x-1,y-2); (x-1,y+2);
        (x+1,y-2); (x+1,y+2);
        (x+2,y-1); (x+2,y+1) ]
```

```
[7]: val deplacements : int * int -> (int * int) list = <fun>
```

Rappel `List.filter` permet de ne sélectionner dans une 'a list que les éléments vérifiant un prédicat 'a -> bool.

Plus d'info ici : https://ocaml.org/api/List.html#1_Listsearching

```
[8]: List.filter (fun x -> x mod 2 = 1) [0;1;2;3;4;5]
```

```
[8]: - : int list = [1; 3; 5]
```

```
[8]: let deplacements_valides (e:echiquier) (x,y) : (int * int) list =  
  (* renvoie les deplacements valides depuis (x,y) dans e *)  
  (* on pourra utiliser List.filter *)  
  List.filter (fun (x,y) -> valide e (x,y))  
    (deplacements (x,y))
```

```
[8]: val deplacements_valides : echiquier -> int * int -> (int * int) list = <fun>
```

Pour la fonction suivante, on aura besoin de s'assurer que les nombres sont affichés avec deux chiffres. Pour cela, on peut écrire %2d dans le format de Printf.printf.

```
[13]: Printf.printf "%d|%d|\n|%2d|%2d|\n" 3 64 3 64;  
flush stdout (* nécessaire pour être sur de faire l'affichage *)
```

```
|3|64|  
| 3|64|
```

```
[13]: - : unit = ()
```

On vous demande d'afficher l'échiquier sous la forme suivante :

```
| 5|12|17|22|  
13|18|23| 6|11|  
 4| 1| 8|21|16|  
19|14| 3|10| 7|  
 2| 9|20|15|  |
```

avec un alignement des | et des cases vides si on n'est pas passé dessus.

Pour vous simplifier la vie, vous pourrez faire un flush stdout avant la fin de votre fonction. flush c'est la chasse d'eau, ça envoie vraiment tous les print sur l'affichage.

```
[10]: let print_echiquier (e:echiquier) : unit =  
  (* Affiche l'échiquier *)  
  let w = Array.length e in  
  let h = Array.length e.(0) in  
  for y = 0 to h-1 do  
    for x = 0 to w-1 do  
      match e.(x).(y) with  
      | None -> Printf.printf "  |"  
      | Some k -> Printf.printf "%2d|" k  
    done; print_newline ()  
  done;  
  flush stdout
```

```
[10]: val print_echiquier : echiquier -> unit = <fun>
```

Pour chercher un tour, on va adapter la méthode du Sudoku : une exception `Solution` et une fonction auxiliaire qui effectue des essais de déplacement en lançant `Solution` si on a trouvé un tour.

Le principe va donc être celui-là quand on appelle `aux n (x,y)` pour dire que `(x,y)` est le `n` mouvement du cavalier :

- On marque `Some n` dans l'échiquier à cette case
- Si `n` marque le dernier mouvement (facile avec le nombre de cases), on lance `Solution`
- Sinon, on itère `aux (n+1)` sur tous les déplacements valides.
 - Soit, on a une `Solution` lancée pendant cette itération et donc *on sort*
 - Soit, c'est que ce déplacement ne permet pas d'aboutir à une solution, donc on repasse à `None` la case `(x,y)`

```
[1]: exception Solution

let tour w h (x0, y0) : echiquier =
  let e = Array.make_matrix w h None in
  try
    let rec aux n (x,y) =
      failwith "C'est à vous !"
    in
    aux 1 (x0,y0);
    raise Not_found (* pas de solution ! *)
  with Solution -> e (* on renvoie e avec la solution *)
```

```
[1]: exception Solution
```

```
File "[1]", line 3, characters 24-33:
3 | let tour w h (x0, y0) : echiquier =
      ~~~~~
```

Error: Unbound type constructor echiquier

```
[20]: print_echiquier (tour 5 5 (0,0)) (* doit permettre d'afficher le tour *)
```

```
Exception: Failure "C'est à vous !".
Raised at file "stdlib.ml", line 29, characters 22-33
Called from file "[18]", line 9, characters 8-21
Called from unknown location
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

```
[21]: let tour_ferme w h (x0,y0) : echiquier =
      (* Cherche un tour fermé *)
      failwith "C'est à vous !"
```

```
[21]: val tour_ferme : 'a -> 'b -> 'c * 'd -> echiquier = <fun>
```

Deux propositions d'extensions :

- énumérer tous les tours pour un petit nombre de cases
- trouver des parcours qui forment sur l'échiquier des carrés magiques ou pseudo-magiques.
Rappel : un carré est pseudo-magique si la somme des valeurs sur les lignes et les colonnes est la même, un carré est magique s'il est pseudo-magique et a la même somme sur les diagonales.

```
[1]: let l = [(3,4);(5,7)]
```

```
[1]: val l : (int * int) list = [(3, 4); (5, 7)]
```

```
[2]: let a = Array.of_list l
```

```
[2]: val a : (int * int) array = [| (3, 4); (5, 7) |]
```

```
[7]: let e = Array.make_matrix 5 3 None
```

```
[7]: val e : '_weak2 option array array =
      [| [|None; None; None|]; [|None; None; None|]; [|None; None; None|];
        [|None; None; None|]; [|None; None; None|] |]
```

```
[11]: Array.length e, Array.length e.(0)
```

```
File "[11]", line 1, characters 13-14:
1 | Array.length e, Array.length e.(0)
      ^
Error: Unbound value e
```

Façon Python avec une liste :

```
for x in l do
  f x
done
```

en OCaml

```
List.iter f l
```

```
[15]: List.iter print_int [1;2;3];  
      print_newline(); flush stdout
```

```
123
```

```
[15]: - : unit = ()
```

Sinon on peut le convertir en tableau si ce n'est pas trop grand (ici ≤ 8 déplacements) :

```
let a = Array.of_list l in  
for i = 0 to Array.length a - 1 do  
  (* Ici on a a.(i) c'est le ieme de l *)  
done
```