Backtracking - n queens

OPTION INFORMATIQUE - TP n° 3.1 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ implémenter un algorithme d'exploration de solutions par la force brute
- ☞ implémenter un algorithme d'exploration de solutions par retour sur trace
- ☞ coder une fonction sur une liste de manière récursive en utilisant le filtrage de motif et les fonctions auxiliaires
- ☞ utiliser module List pour coder des équivalents aux codes récursifs (iter, map, filter, fold)

A Le problème des n reines

On cherche à placer sur un échiquier de $n \times n$ cases n reines sans que celles-ci s'attaquent les unes les autres. On rappelle que la reine peut attaquer une pièce sur la ligne, la colonne et les diagonales qui partent de sa position.

B Modélisation de l'échiquier pour les n reines

Il est possible de représenter un échiquier à l'aide de différentes structures de données. La première qui vient à l'esprit, certainement à cause de la visualisation de l'échiquier, est le tableau à deux dimensions. Néanmoins, lorsqu'on observe de plus près la répartition des reines pour des configurations solutions, il apparaît clairement qu'une même ligne ne peut accueillir qu'une seule reine. C'est pourquoi il est possible d'implémenter l'échiquier par une liste : l'indice de la liste représente le numéro de la ligne sur laquelle se situe la reine. Le ième élément de la liste représente la colonne sur laquelle se trouve la reine. S'il n'y a pas de reine sur la ligne i, le ième élément de la liste vaut -1.

Par exemple, l'échiquier représenté sur la figure 1 est encodé par la liste `board=[7;3;0;2;5;1;6;4]`. On note que le premier élément de `board` vaut 7, ce qui signifie qu'il y a une reine sur la première ligne et la huitième colonne. Un échiquier `[3;-1;2;-1]` est un échiquier de 4x4 avec une reine en $(0,3)$ et une autre en $2,2$.

C Résolution par force brute

C1. Combien y-a-t-il de solutions au problème des quatre reines [1] ?

1. On peut vérifier le résultat en comparant avec la séquence A000170

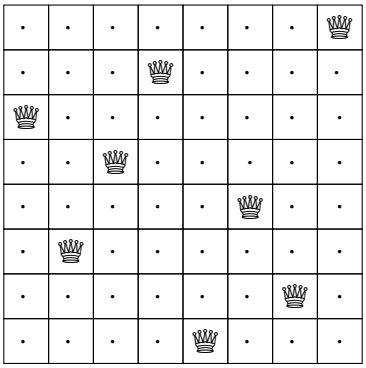


FIGURE 1 – Exemple d'échiquier 8x8 solution du problème des huit reines.

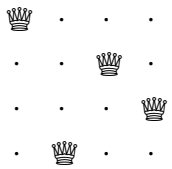
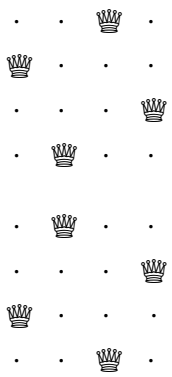


FIGURE 2 – Résultat sur la console de la fonction `show board` avec `board=[0;2;1;3]`

Solution : Il y a deux solutions :



C2. On souhaite afficher sur la console l'échiquier avec les reines comme sur la figure 2. La commande `print_string "\u{2655}"` permet d'imprimer le symbole UTF-8 de la reine d'un jeu d'échec.

(a) Écrire une fonction dont le signature est `show : int list -> unit` qui affiche sur la console l'échiquier comme sur les figures 2 et 3. **On utilisera la fonction `iter` du module List.**

(b) Écrire une fonction **récursive** dont le signature est `rec_show : int list -> unit` qui affiche sur la console l'échiquier comme sur les figures 2 et 3.

C3. On souhaite placer une reine sur la ligne r et la colonne c. D'autres reines sont déjà présentes sur l'échiquier.

(a) Écrire une fonction de signature `same_col : int -> int list -> bool` qui teste si une reine

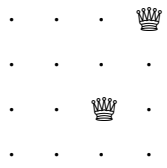


FIGURE 3 – Résultat de la fonction `show` sur la liste `[3;-1;2;-1]`

située sur la colonne c est attaquée par une autre reine présente sur la même colonne. **On écrira une version récursive et une autre version qui utilise la fonction `mem` module List.**

(b) Écrire une fonction de signature `same_row : int -> int list -> bool` qui teste si une reine située sur la ligne r est attaquée par une autre reine présente sur la même ligne. Si elle est attaquée, cela signifie que l'élément r de la liste est un nombre différent de -1. **On écrira une version récursive et une autre version qui utilise la fonction `nth` du module List.**

(c) Écrire une fonction de signature `down_diag : int -> int -> int list -> bool` qui teste si une reine située en (r,c) (donnés en paramètres) est attaquée par une autre reine présente sur la diagonale allant du haut vers le bas de l'échiquier.

(d) Écrire une fonction de signature `up_diag : int -> int -> int list -> bool` qui teste si une reine située en (r,c) (donnés en paramètres) est attaquée par une autre reine présente sur la diagonale allant du bas vers le haut.

C4. Écrire une fonction de signature `under_attack : int -> int -> int list -> bool` qui vérifie si la reine en (r,c) est attaquée par une autre reine présente sur l'échiquier. Il est nécessaire de masquer la présence éventuelle de cette reine avant de tester l'échiquier[2].

C5. Écrire une fonction de signature `valid_solution : int list -> bool` teste si une configuration (liste) est une configuration valide (c'est-à-dire aucune reine n'est attaquée).

C6. Résoudre le problème des quatre reines en écrivant un code qui trouve toutes les solutions par la force brute.

(Questions pour les 5/2) Pour améliorer la compacité du code, on peut remarquer qu'avec la représentation sous forme de liste de l'échiquier, une configuration valide est forcément une permutation de la liste `[0;1;2;3;4;5;6;7]`. On peut donc se contenter de tester toutes les permutations de cette liste. On se propose donc d'écrire un code qui génère toutes les permutations d'une liste.

C7. (a) Écrire une fonction récursive de signature `rm : 'a -> 'a list -> 'a list` qui supprime toutes les occurrences d'un élément spécifié d'une liste et renvoie la liste ainsi modifiée.

(b) Écrire une fonction de signature `create_board : int -> int list` qui renvoie la liste des entiers des 0 à $n-1$.

(c) Écrire une fonction récursive de signature `permutations : 'a list -> 'a list list` qui génère la liste de toutes les permutations d'une liste. On pourra raisonner comme suit :

1. si `my_list` est vide, renvoyer une liste vide,
2. si `my_list` possède un seul élément, renvoyer la liste qui contient cet élément,
3. sinon pour chaque élément de `my_list`, créer toutes les permutations de `my_list` sans cet élément et insérer l'élément en tête de ces listes. Concaténer toutes les listes obtenues ainsi.

2. une sous fonction serait adaptée à la création de cette liste masquée.

C8. Écrire une fonction de signature `brute_force_permutation : int -> unit` qui teste la validité sur l'échiquier de toutes les permutations d'une liste à n éléments. Cette fonction affiche les échiquiers solutions et le nombre de solutions sur la console.

C9. Cette fonction est-elle efficace pour huit reines? Peut-on résoudre le problème pour des n plus grands que 8?

Solution : Pour $n = 9$, on dépasse déjà la capacité de la pile d'exécution. (Stack overflow)

Solution :

Code 1 – Résolution par la force brute du problème des n reines

```
let show board =
  let print_row v =
    for i = 0 to (List.length board) - 1 do
      print_string (if i=v then "\u{2655} " else ". ");
    done;
    print_newline()
  in List.iter print_row board; print_newline();;

let rec_show board =
  let print_row col =
        for c = 0 to (List.length board) - 1 do
            if col = c then print_string "\u{2655} " else print_string ". " done
                ; print_newline()
  in let rec aux b = match b with
              | [] -> print_newline()
              | col::t -> print_row col; aux t;
  in aux board;;

let same_col c board = List.mem c board;;

let rec rec_same_col c board = match board with
    | [] -> false
    | head::tail -> if c = head then true else rec_same_col c tail;;

let same_row r board = (List.nth board r) != -1;;

let rec_same_row r board =
    let rec aux b i = match b with
        | [] -> false
        | head::tail -> if i = r && head != -1 then true else aux tail (i + 1)
        in aux board 0;;

let up_diag r c board =
  let rc = r + c
    in let rec diag b i = match b with
        | [] ->  false
        | head::tail -> if head != -1 then if (i + head) = rc then true else
            diag tail (i + 1) else diag tail (i + 1)
    in diag board 0;;

let down_diag r c board =
```

```
    let rc = r - c
      in let rec diag b row = match b with
          | [] ->  false
          | col::tail -> if col != -1 then if (row - col) = rc then true else diag
              tail  (row + 1) else diag tail (row + 1)
    in diag board 0;;

  let under_attack r c board =
    let masked = List.mapi (fun index elem -> if r = index && c = elem then -1
        else elem) board
        in same_row r masked || same_col c masked || up_diag r c masked  ||
            down_diag r c  masked;;

  let rec_under_attack r c board =
    let masked =
     let rec aux b row = match b with
          | [] -> []
          | col::tail -> if (row,col) = (r,c) then -1::(aux tail (row + 1)) else
              col::(aux tail (row + 1))
          in aux board 0
    in same_row r masked || same_col c masked || up_diag r c masked  || down_diag
        r c  masked;;

  let valid_solution board =
    let res = List.mapi (fun r c -> under_attack r c board) board
      in not (List.mem true res);;

  let rec_valid_solution board =
      let rec check row b = match b with
          | [] -> true
          | col::tail -> if under_attack row col board then false else check (row
              + 1) tail
      in  check 0 board;;

  let raw_force_4_queens () =
    let n = 4 and board = []
    in for i=0 to n - 1 do
          let bi = i::board in
          for j=0 to n - 1 do
            let bij = j::bi in
            for k=0 to n - 1 do
              let bijk = k::bij in
              for l=0 to n - 1 do
                let bijkl = l::bijk in
                    if valid_solution bijkl then (Printf.printf "[%i,%i,%i,%i]\n"
                        i j k l; rec_show bijkl;)
              done;
            done;
          done;
        done;;

  let rec rm x my_list = match my_list with
    | [] -> []
    | h::t -> if h=x then rm x t else  h::rm x t;;
```

```
let rm x my_list = List.filter ((!=) x) my_list;;

let create_board s =
    let rec aux l i = if s=i then l else i::(aux l (i + 1))
    in aux [] 0;;

let create_board s = List.init s (fun x -> x);;

(* Fixed-head solution*)
let rec permutations my_list =  match my_list with
  | [] -> []
  | x::[] -> [[x]]
  | l -> List.fold_left (fun acc x -> acc @ List.map (fun p -> x::p) (
      permutations (rm x l))) [] l;;


let brute_force_permutation s =
  let sol_nb = ref 0
    in let print_if_valid b = if valid_solution b then (incr sol_nb;
        print_string "Solution #"; print_int !sol_nb; print_newline(); rec_show
        b)
    in let perms = permutations (create_board s)
    in List.iter print_if_valid perms;;


(* TESTS *)
let n = 4;;
raw_force_4_queens();;
brute_force_permutation 9;;
```

D Résolution par retour sur trace

L'algorithme de retour sur trace 1 construit au fur et à mesure les solutions partielles du problème et les rejette dès qu'il découvre une impossibilité.

Algorithme 1 Algorithme de retour sur trace

```
1 : Fonction EXPLORER(v)                          ▷ v est un nœud de l'arbre de recherche
2 :     si v est une feuille alors
3 :         renvoyer Vrai
4 :     sinon
5 :         pour chaque fils u de v répéter
6 :             si u peut compléter une solution partielle au problème 𝒫 alors
7 :                 EXPLORER(u)
8 :     renvoyer Faux
```

On utilise la modélisation de l'échiquier précédente, c'est-à-dire qu'on construit la liste des colonnes occupées. On procède par ligne en positionnant d'abord une reine puis une autre sur la deuxième ligne…Ainsi de suite, la liste augmente de taille jusqu'à atteindre la taille n.

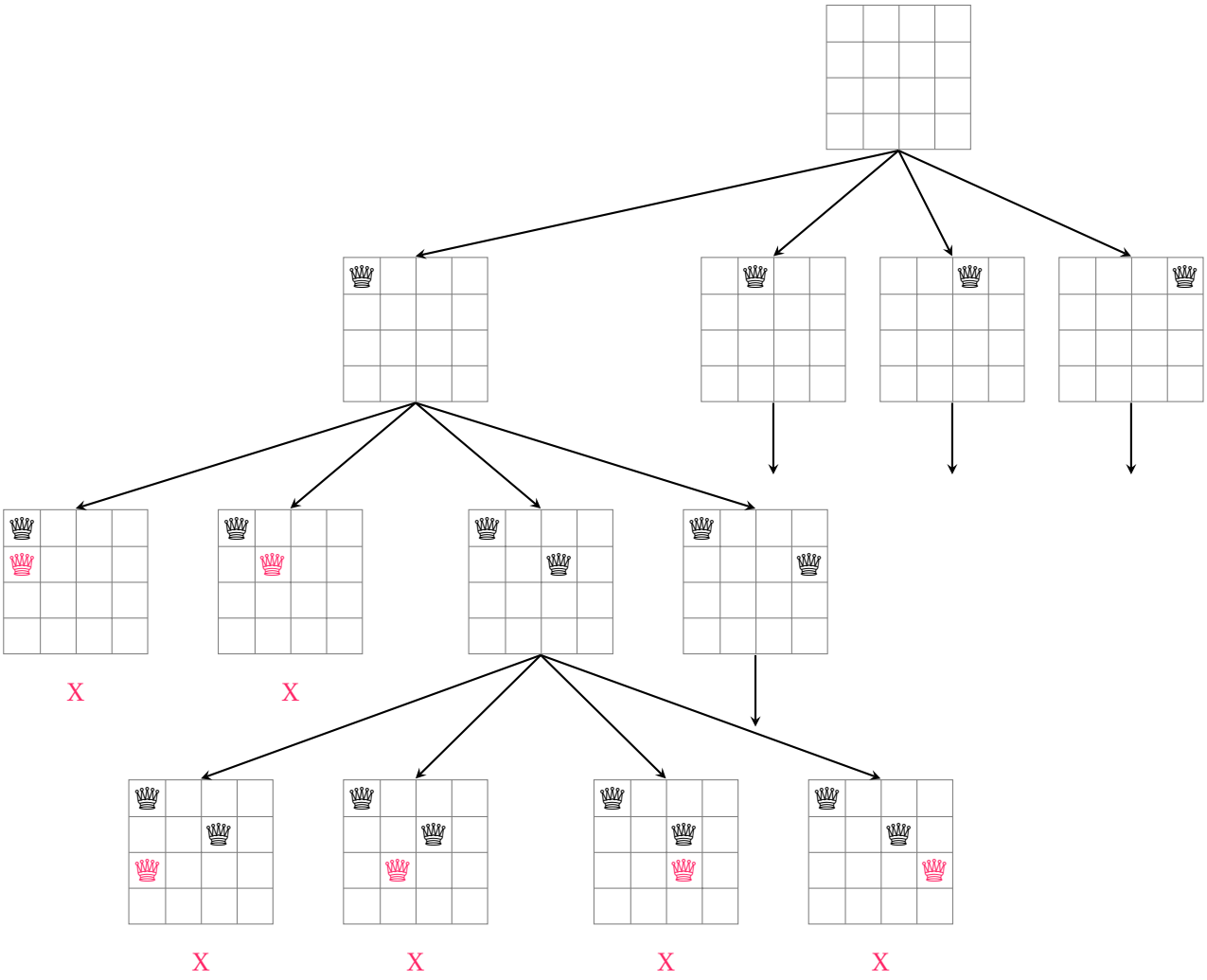


FIGURE 4 – Exemple d'arbre de recherche structurant l'algorithme de retour sur trace. Application au problème de quatre reines.

On n'a plus besoin de traiter le cas où la colonne est vide[3] car à chaque fois qu'on envisage une solution partielle (une liste partielle), les lignes qui précèdent possèdent nécessairement une reine sur une des colonnes de l'échiquier.

D1. Comment coder «v est une feuille» en OCaml?

> **Solution :** Lorsque la taille de la liste qui représente l'échiquier dépasse n, alors on a rempli l'échiquier. Si l'indice de la ligne considéré est n alors on est arrivé sur une feuille.

D2. Comment coder «u peut compléter une solution partielle au problème $\mathcal{P}$»?

3. que l'on avait dû traiter dans le cas de la force brute avec -1

Solution : On considère la nouvelle position u `(row,col)`. Alors, il faut considérer la solution si cette reine n'est pas attaquée.

```
if not (under_attack row col board) then
```

D3. Implémenter un algorithme de retour sur trace pour le problème des quatre reines. Vous pourrez vous inspirer du code de départ ci-dessous : seule la fonction `n_queens : int -> int list list` est à compléter. Celle-ci renoive la liste des solutions.

Code 2 – N queens backtracking

```
let show_sol board =
    let print_row col = for c = 0 to (List.length board) - 1 do
                            if col = c then print_string "\u{2655} " else
                                print_string ". "
                        done; print_newline() in
        let rec aux b = match b with
            | [] -> print_newline()
            | col::t -> print_row col; print_newline(); aux t;
        in print_newline(); aux board
  ;;

let under_attack row col board =
  let up_diag =
    let rec aux i b = match b with
    | [] -> false
    | j::t -> if (i + j) = (row + col) then true else aux (i + 1) t
    in aux 0 board
  in let down_diag =
      let rec aux i b = match b with
        | [] -> false
        | j::t -> if (i - j) = (row - col) then true else aux (i + 1) t
    in aux 0 board
  in List.mem col board || up_diag || down_diag
;;

let synth solutions =
  List.iter show_sol solutions;
  Printf.printf "Nombre de solutions #%d\n" (List.length solutions);;

let n_queens n =
  let rec build_solutions row current_board =
      if row = n  (* On a trouvé une solution *)
      then [current_board] (* On la renvoie dans une liste *)
      else (* Trouver toutes les solutions possibles en construisant toutes les
           nouvelles lignes possibles *)
        let rec build_line_with c solutions =
          (* Si c = n : on a testé toutes les colonnes possibles  pour l'ajout
             d'une reine *)
          (* Alors on renvoie la liste des solutions possibles *)
          (* Sinon *)
               (* Si on peut mettre une reine en c *)
               (* Alors construire toutes les solutions possibles avec une
                  reine en c et les ajouer à solutions *)
               (* Sinon essayer en c + 1 *)
```

```
        in build_line_with 0 []
      in build_solutions 0 [];;


  synth (n_queens 4);;  (* montre et compte les solutions *)
```

D4. Tester ce programme sur le problème des n reines et comparer les résultats avec l'algorithme de force brute. Vérifier que vous retrouvez les mêmes résultats.

D5. Compiler le programme. Quelle valeur de n engendre un temps d'exécution supérieur à la minute?

D6. Implémenter cet algorithme en Python et comparer les performances.

Solution : Sur ma machine, trois fois plus vite en compilé. OCaml plus rapide que Python non compilé.

Code 3 – N queens backtracking

```
let show_sol board =
    let print_row col = for c = 0 to (List.length board) - 1 do
                            if col = c then print_string "\u{2655} " else
                                print_string ". "
                        done; print_newline() in
        let rec aux b = match b with
            | [] -> print_newline()
            | col::t -> print_row col; print_newline(); aux t;
        in print_newline(); aux board
  ;;

  (* rec version *)
let under_attack row col board =
  let up_diag =
    let rec aux i b = match b with
    | [] -> false
    | j::t -> if (i + j) = (row + col) then true else aux (i + 1) t
    in aux 0 board
  in let down_diag =
      let rec aux i b = match b with
        | [] -> false
        | j::t -> if (i - j) = (row - col) then true else aux (i + 1) t
    in aux 0 board
  in List.mem col board || up_diag || down_diag
;;

let n_queens n =
  let rec build_solutions row current_board =
      if row = n
      then [current_board]
      else
        let rec build_line_with c solutions =
          if c = n
          then solutions
          else
            if not (under_attack row c current_board)
            then let new_solutions = build_solutions (row+1) (current_board@[c])
```

```
                    in build_line_with (c+1) (solutions @ new_solutions)
              else  build_line_with (c + 1) solutions
        in build_line_with 0 []
    in build_solutions 0 [];;

let synth solutions =
    List.iter show_sol solutions;
    Printf.printf "Nombre de solutions #%d\n" (List.length solutions);;


synth (n_queens 4);;
synth (n_queens 5);;
synth (n_queens 6);;
synth (n_queens 7);;
synth (n_queens 8);;
(*synth (n_queens 12);;*)
```

Code 4 – N queens backtracking

```
# Traduction du code OCaml en Python

def show_sol(board):
    def print_row(col, n):
        for c in range(n):
            if col == c:
                print("\u2655 ", end="")
            else:
                print(". ", end="")
        print()
    for col in board:
        print_row(col, len(board))
    print()

def under_attack(row, col, board):
    up_diag = any((i + j) == (row + col) for i, j in enumerate(board))
    down_diag = any((i - j) == (row - col) for i, j in enumerate(board))
    return col in board or up_diag or down_diag

def n_queens(n):
    def build_solutions(row, current_board):
        if row == n:
            return [current_board]
        solutions = []
        for c in range(n):
            if not under_attack(row, c, current_board):
                new_solutions = build_solutions(row + 1, current_board + [c])
                solutions.extend(new_solutions)
        return solutions

    return build_solutions(0, [])

def synth(solutions):
    for sol in solutions:
        show_sol(sol)
    print(f"Nombre de solutions: {len(solutions)}")
```

```
    # Tester avec différentes tailles
    for i in range(4, 9):
        synth(n_queens(i))
```