

Backtracking - n queens

MPSI/MP OPTION INFORMATIQUE - TP n° 3.1 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ effectuer un filtrage de motif (pattern matching)
- ☞ coder une expression conditionnelle `if ... then (...) else (...)`
- ☞ coder une boucle pour en OCaml `for i=0 to n-1 do (...) done`
- ☞ garantir qu'une fonction retourne toujours le même type et éventuellement `unit ()`.
- ☞ coder une fonction sur une liste de manière récursive
- ☞ utiliser l'API List pour coder une fonction sur une liste (`map`, `mapi`, `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 i ème 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 i ème 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 4×4 avec une reine en (0, 3) et une autre en 2, 2.

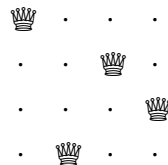
C Résolution par force brute

C1. Combien y-a-t-il de solutions au problème des quatre reines¹ ?

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

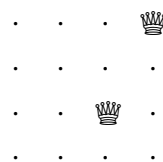


FIGURE 1 – Exemple d'échiquier 8x8 solution au problème des huit reines.

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

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.

- 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 les fonctions du module `List`.
- 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.

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

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

- Écrire une fonction de signature `same_col : 'a -> 'a list -> bool` qui teste si une reine qui serait située sur la colonne c serait attaquée par une autre reine présente sur la même colonne.
- Écrire une fonction de signature `same_row : int -> int list -> bool` qui teste si une reine qui serait située sur la ligne r serait 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 .

- (c) Écrire une fonction de signature `down_diag : int -> int -> int list -> bool` qui teste si une reine qui serait située en (r, c) serait 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 qui serait située en (r, c) serait 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 sera nécessaire de masquer la présence de cette reine avant de tester l'échiquier.
- C5. Écrire une fonction de signature `valid_solution : int list -> bool` teste si une configuration (liste) est une solution valide.
- C6. Résoudre le problème des quatre reines en écrivant un code qui trouve toutes les solutions par la force brute.
- 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 permutation d'une liste.
- C7. (a) Écrire une fonction récursive de signature `rm : 'a -> 'a list -> 'a list` qui supprime les éléments spécifiés 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 des de toutes les permutations de 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 renvoyer la liste des permutations de la liste, c'est à dire l'agrégation de toutes les listes créées ainsi : pour chaque élément de `my_list`, créer les listes qui commencent par cet élément concaténé à toutes les permutations de cette liste sans cet élément.
- C8. Écrire une fonction de signature `brute_force_permutation : int -> unit` qui teste la validité toutes les permutations d'une liste à n éléments et 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 grand que 8?

Solution :

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

```
1 let n = 4;;
2 let test_board = [2;0;3;2];;
3 let rec_show board =
4   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
```

```

5      let rec aux b = match b with
6      | [] -> print_newline()
7      | col::[] -> print_row col; print_newline()
8      | col::t -> print_row col; aux t;
9      in aux board
10   ;;
11   rec_show test_board;;
12   let show board =
13     let pr v =
14       for i = 0 to (List.length board) - 1 do
15         print_string (if i=v then "\u{2655} " else ". ");
16       done;
17       print_newline()
18     in List.iter pr board;;
19   show test_board;;
20   show [3;-1;2;-1];;
21   let same_col c board = List.mem c board;;
22
23   let same_row r board = (List.nth board r) != -1;;
24
25   let up_diag r c board =
26     let rc = r + c
27     in let diag board = List.mapi (fun index elem -> if elem != -1 then Some (
28       index + elem) else None) board
29     in List.mem (Some rc) (diag board);;
30
31   let down_diag r c board =
32     let rc = r - c
33     in let diag board = List.mapi (fun index elem -> if elem != -1 then Some (
34       index - elem) else None) board
35     in List.mem (Some rc) (diag board);;
36
37   let attack_test_board = [2;-1;3;-1];;
38
39   let under_attack r c board =
40     let masked = List.mapi (fun index elem -> if r = index && c = elem then -1
41       else elem) board
42     in same_row r masked || same_col c masked || up_diag r c masked ||
43       down_diag r c masked;;
44
45   let valid_solution board =
46     let res = List.mapi (fun r c -> under_attack r c board) board
47     in List.for_all (fun x -> not x) res;;
48
49   valid_solution attack_test_board;;
50   valid_solution test_board;;
51   let valid_board = [2;0;3;1];;
52   valid_solution valid_board;;
53   let raw_force_4_queens () =
54     let board = []
55     in for i=0 to n - 1 do
56       let bi = i::board in
57       for j=0 to n - 1 do
58         let bij = j::bi in

```

```

56         for k=0 to n - 1 do
57             let bijk = k::bij in
58             for l=0 to n - 1 do
59                 let bijkl = l::bijk in
60                 if valid_solution bijkl then rec_show bijkl;
61             done;
62         done;
63     done;
64 done;;
65 raw_force_4_queens();;
66 let rec rm x my_list = match my_list with
67 | [] -> []
68 | h::t -> if h=x then rm x t else h::rm x t;;
69 let rm x my_list = List.filter ((<>) x) my_list;;
70
71
72 let rec permutations my_list = match my_list with
73 | [] -> []
74 | x::[] -> [[x]]
75 | l -> List.fold_left (fun acc x -> acc @ List.map (fun p -> x::p) (
    permutations (rm x l))) [] l;;
76
77 let create_board s = List.init s (fun x -> x);;
78
79 let brute_force_permutation s =
80     let sol_nb = ref 0
81     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)
82     in let perms = permutations (create_board s)
83     in List.iter print_if_valid perms;;
84
85 brute_force_permutation 4;;
86 brute_force_permutation 5;;
87 brute_force_permutation 6;;
88 brute_force_permutation 7;;
89 brute_force_permutation 8;;
90 brute_force_permutation 9;;
91 brute_force_permutation 12;;

```

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.

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 .

On n'a plus besoin de traiter le cas où la colonne est vide² car à chaque fois qu'on envisage une solution partielle (une liste partielle), les lignes qui précèdent possèdent nécessairement une reine sur

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

Algorithme 1 Algorithme de retour sur trace

```

1: Fonction RETOUR_SUR_TRACE( $v$ )                                ▷  $v$  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  $\mathcal{P}$  alors
7:         RETOUR_SUR_TRACE( $u$ )
8:   renvoyer Faux

```

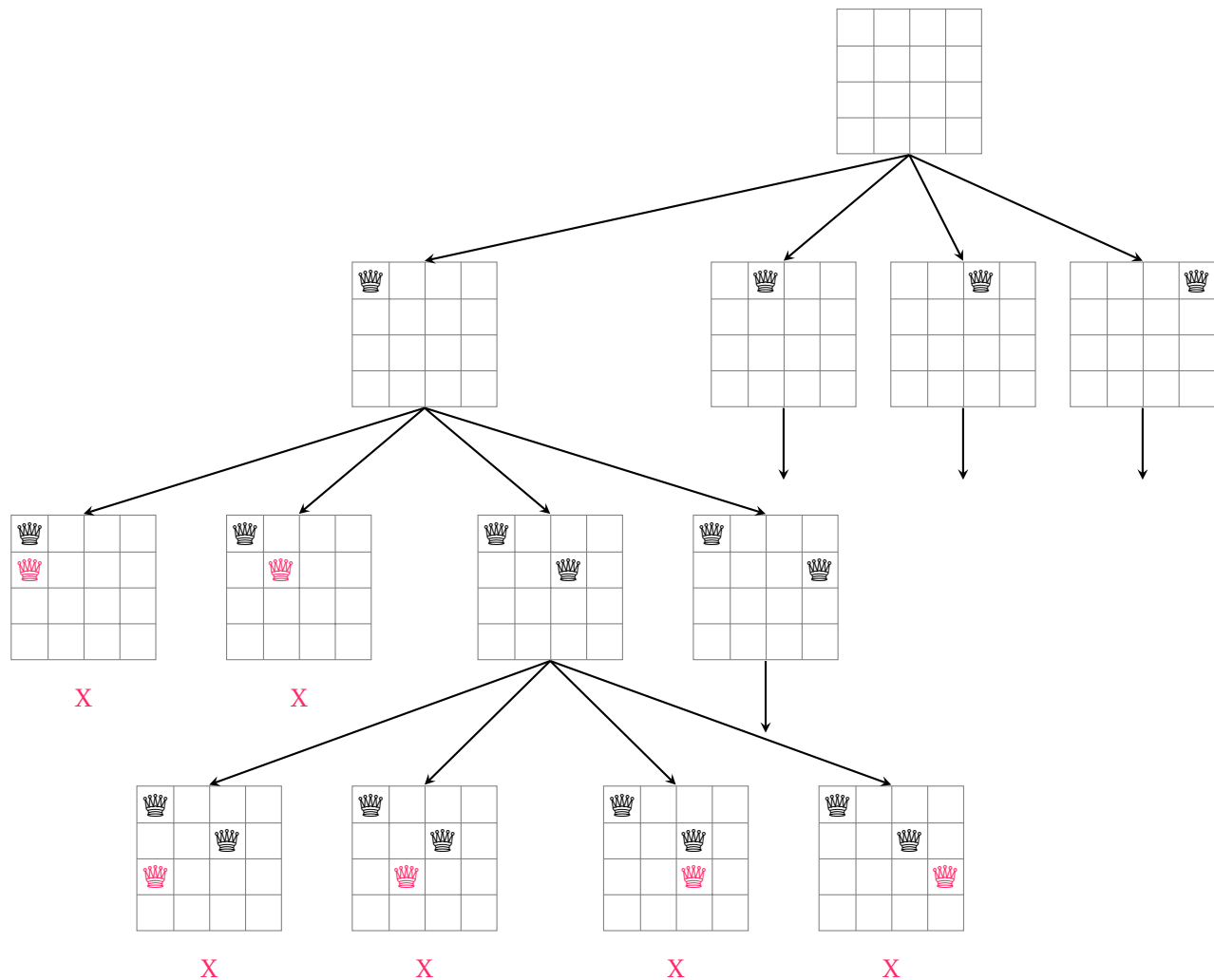


FIGURE 4 – Exemple d'arbre de recherche structurant l'algorithme de retour sur trace. Application au problème de quatre reines.

une des colonnes de l'échiquier.

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

- D2. Comment coder « u peut compléter une solution partielle au problème \mathcal{P} »?
- D3. Implémenter un algorithme de retour sur trace pour le problème des quatre reines.
- 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 2 – N queens backtracking

```

1 let show_sol board nb =
2   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
3     let rec aux b = match b with
4       | [] -> print_newline()
5       | col::[] -> print_row col; print_newline()
6       | col::t -> print_row col; aux t;
7     in print_string "Solution #"; print_int nb; print_newline(); aux board
8   ;;
9   (* rec version *)
10  let under_attack row col board =
11    let up_diag =
12      let rec aux i b = match b with
13        | [] -> []
14        | j::t -> (i + j)::(aux (i + 1) t)
15      in aux 0 board
16    in let down_diag =
17      let rec aux i b = match b with
18        | [] -> []
19        | j::t -> (i - j)::(aux (i + 1) t)
20      in aux 0 board
21    in List.mem col board || List.mem (row + col) up_diag || List.mem (row - col)
      down_diag
22  ;;
23  (* List module version *)
24  let under_attack row col board =
25    let up_diag = List.mapi (fun index elem -> index + elem) board
26    in let down_diag = List.mapi (fun index elem -> index - elem) board
27    in List.mem col board || List.mem (row + col) up_diag || List.mem (row - col)
      down_diag
28  ;;
29  let n_queens n =
30    let sol_nb = ref 0
31    in let rec build_solutions row board =
32      if row = n then
33        (incr sol_nb; show_sol board !sol_nb)
34      else
35        for c = 0 to n - 1 do
36          if not (under_attack row c board) then
37            build_solutions (row+1) (board@[c])
38        done

```

```

39         in build_solutions 0 [];;
40
41 n_queens 4;
42 n_queens 5;
43 n_queens 6;
44 n_queens 7;
45 n_queens 8;
46 n_queens 12;
47 n_queens 13;

```

Code 3 – N queens backtracking

```

1 # coding: utf8
2 # nb solution --> https://oeis.org/A000170/list
3 n = 12
4 on_board = [None for _ in range(n)]
5 lower_diag = []
6 upper_diag = []
7 solutions_number = 0
8
9 def print_board():
10     for row in on_board:
11         for i in range(len(on_board)):
12             if i == row:
13                 print('\u2655', end=" ") # Queen symbol utf-8
14             else:
15                 print('.', end=" ")
16         print()
17     print("-----")
18
19
20 def under_attack(r, c):
21     return c in on_board or r + c in lower_diag or r - c in upper_diag
22
23 def bt_n_queens(r=0):
24     global solutions_number # be explicit since is modified
25     if r == n:
26         return True
27     else:
28         for c in range(n):
29             #print(r,c)
30             if not under_attack(r, c):
31                 on_board[r] = c # Set Queen
32                 lower_diag.append(r + c) # danger aware !
33                 upper_diag.append(r - c) # danger aware !
34                 if bt_n_queens(r + 1):
35                     solutions_number += 1
36                     print("New solution !", solutions_number)
37                     print_board()
38                 on_board[r] = None # Unset Queen
39                 lower_diag.pop() # remove danger
40                 upper_diag.pop() # remove danger
41         return False
42
43

```



```
44 if __name__ == "__main__":  
45     print_board()  
46     bt_n_queens()
```
