

Projet de programmation fonctionnelle

Maîtrise d'informatique – Institut Galilée

Problème des n reines

Maël Le Berre

Janvier 2002

Projet de programmation fonctionnelle :

Le problème des n reines.

1. Problématique de l'algorithme.

Le problème des n reines consiste à disposer n reines sur un échiquier de taille $n \times n$ sans qu'elles puissent se manger entre elles selon les règles des échecs. Les reines doivent donc être chacune placées sur des lignes différentes, des colonnes différentes et des diagonales différentes. Un algorithme triviale résolvant ce problème serait de tester toutes les combinaisons de positionnements des reines sur l'échiquier en validant ou pas pour chacune d'elles son caractère de solution. La complexité en temps de cet algorithme au facteur près du temps de test (de l'ordre de $O(n^2)$ serait en $O(C(n^2, n))$ (nombre de combinaison possible des n reines sur les n^2 cases de l'échiquier.) soit en $O(n^2 / n!(n^2-1))$, complexité beaucoup plus grande que $O(n!)$. Un certain nombre de contraintes du problème peuvent nous permettre de considérablement réduire cette complexité.

1.1. Contraintes de lignes et de colonnes.

Tout d'abord, on peut remarquer qu'il faut disposer les n reines sur un damier contenant n lignes et n colonnes. Comme chaque reines doit être sur une ligne et une colonne différente, chaque ligne et chaque colonne de l'échiquier doit contenir exactement une reine. Ipso facto, le problème se réduit à savoir comment distribuer l'indice de ligne de chaque reine sur chaque colonne (l'indice de ligne de chaque reines pouvant être fixé au départ). Autrement dit, chaque colonne possédant une et une seul reines, leur indice de ligne doit être différent pour chaque colonne. Le problème se résume donc à tester toutes les distributions possibles des n reines numérotées sur un tableau de n cases. Cette contrainte réduit la complexité en temps de ce nouvel algorithme au facteur près du temps de test à $O(n!)$. Mais il est encore possible d'allé plus loing.

1.2. Contraintes de diagonales.

Les contraintes de diagonales peuvent s'exprimer plus formellement de la manière suivante : Les distances indicielles de ligne et de colonne entre deux reines doivent être différentes (sinon, cela signifie que les deux reines sont sur une même diagonale). Ainsi, chaque reine positionnée sur l'échiquier impose l'interdiction de certains couples d'indices pour les autres reines. En reprenant l'algorithme du paragraphe précédent, chaque reine indicée positionnée sur l'échiquier peut interdire au autres cases du tableau d'accepter certains indices de reines. Plus intuitivement, cela correspond sur l'échiquier, à s'interdire au moment de placer une nouvelle reine de la placée sur les lignes, colonnes ou diagonales des reines déjà placées. En retournant l'astuce, on peut définir pour chaque reine indicé les colonnes du tableau ou elles peuvent encore se positionner et ainsi déceler des situation sans avenir quand une reine ne peut plus se positionner nul part (ce qui correspondent à des situations ou une colonne libre est entièrement interdite par les reines déjà placée). Ce nouvel algorithme est conditionné dans son développement par toutes les contraintes du problème. Par voie de conséquence, quand une combinaison est trouvée, elle est obligatoirement solution du problème et rend inutile les tests. La complexité du problème, difficile à évaluée car

dépendante du déroulement de l'algorithme, est maintenant réduite à $o(n!)$ mais reste cependant très importante.

2. Le programme.

Cet algorithme s'adapte bien à une implémentation fonctionnel. En effet, il s'agit de construire un arbre des différentes combinaisons de colonnes pour chaque ligne en ajoutant les reines, les branches de cet arbre étant taillées au fur et à mesure que l'algorithme évolue en fonction des contraintes des reines déjà placée. La fonction principale sera donc une fonction récursive appelée à chaque feuille de l'arbre pour placer de nouvelles reines. La méthode ainsi que la description du type CAML utilisé (?? je n'ai pas trouvé l'utilité d'en créer plus d'un.) est expliquée en détail dans le listing en annexe.

Le programme est composé d'un unique fichier « Nreines.ml » . Une fois compilé, il prend en paramètre la taille du problème et retourne sur le flux de sortie standard les combinaisons de colonnes solutions numérotées et leur représentation sur un damier.

3. Jeux d'essais.

On peut remarquer que le nombre de solutions devient très vite gigantesque avec la taille du problème. Le tableau suivant donne le nombre de solution trouvée pour les treize premières tailles :

Taille du problème	Nombre de solution
1	1
2	0
3	0
4	2
5	10
6	4
7	40
8	92
9	352
10	724
11	2680
12	14200

Exemple des solutions du problème des 6-reines calculé par le programme :

```
Solution no 1 : [5|3|1|6|4|2]
# # 0
#0# #
0 # #
# # 0
# #0#
 0 # #
Solution no 2 : [4|1|5|2|6|3]
# #0#
0# # #
# # 0
 0 # #
# # #0
#0# #
Solution no 3 : [3|6|2|5|1|4]
# 0 #
# # 0
#0# #
# #0#
0 # #
# 0 #
Solution no 4 : [2|4|6|1|3|5]
#0# #
# 0 #
# # #0
0# # #
# 0 #
# #0#
4 solutions trouvees.
```

Annexe : Listing du programme.

```
(*****
(*****      Projet de CAML :      *****)
(*****      PROBLEME DES N-REINES *****)
(*****      *****)
(*****      Mael Le Berre          *****)
(*****      Janvier 2002          *****)
(*****      Maitrise d'informatique *****)
(*****      Institut Galilee      *****)
(*****      *****)

(* Definition des TYPES : *)

(*
Type reine : type permettant la creation d'une liste de reines munies pour
chacune d'elles
de leur numero de ligne et d'une liste des numeros de colonnes ou elles
peuvent encore etre
placees.
reine est :
- Null      si la fin de la liste de reine est atteinte.
- Reine pour chaque reine qui contient :
  - Un entier representant le numero de ligne de la reine.
  - Une reine : la reine suivante dans la liste.
  - Une liste d'entiers contenant les numeros de colonnes ou la reine
peut encore etre place.
- Sans si la situation est sans solution. (cad : une reine ne peut plus
etre place.)
*)

type reine = Null | Reine of int*reine*int list | Sans;;

(* Fonctions d'INITIALISATION : *)

(*
Fonction rec_faire_liste et faire_liste :
rec_faire_liste est une fonction recursive generant une liste d'entier
allant de 1 a n.
faire_liste dispense du parametre liste l'appel de rec_faire_liste.
*)

let rec rec_faire_liste liste n = if n=0 then liste else rec_faire_liste
(n::liste) (n-1);;
let faire_liste = fonction n->rec_faire_liste [] n;;

(*
Fonction rec_init_reines et init_reines :
rec_init_reines est une fonction initialisant la structure de donnee
initiale de l'algorithmme.
Soit : une liste de n reines placees sur les n lignes contenant chacune la
liste des n colonnes.
init_reines dispense des parametres liste et i l'appel de rec_init_reines.
*)

let rec rec_init_reines liste n i = if i = 0 then liste else
rec_init_reines (Reine(i,liste,faire_liste n)) n (i-1);;
```

```

let init_reines = fonction n->rec_init_reines Null n n;;

(* Fonctions d'AFFICHAGE : *)

(*
Fonction rec_affiche_liste et affiche_liste :
fonctions (appel par affiche_liste) affichant une liste a l'ecran.
*)

let rec rec_affiche_liste liste = match liste with
  x::y->
    print_int x;
    if y <> [] then print_char '|';
    rec_affiche_liste y;|
  []->
    print_char '|';;

let affiche_liste = fonction x->
  print_char '[';
  rec_affiche_liste x;;

(*
Fonction affiche_reines :
fonction de debugage affichant l'etat de la structure de donnee.
*)

let rec affiche_reines liste = match liste with
  Reine(no,suivantes,liste)->
    Printf.printf "Reine no %d : " no;
    affiche_liste liste;
    print_char '\n';
    affiche_reines suivantes;
  |Null->
    Printf.printf "Fin\n";
  |Sans->Printf.printf "Fin(Sans)\n";;

(*
Fonction affiche_solution :
fonction affichant les solutions trouvees sur un damier.
Pour des raison de commodite, cette fonction utilise des fonctions
imperatives.
*)

let rec affiche_solution p n liste = match liste with
  a::b->let it = ref p in
    for i = 1 to a-1 do
      if !it = 1 then
        (print_char '#';
         it := 0)
      else
        (print_char ' ';
         it := 1)
    done;
    print_char '0';
    if !it = 1 then it := 0 else it := 1;
    for i = 1 to n-a do
      if !it = 1 then
        (print_char '#';
         it := 0)
      else
        (print_char ' ';

```

```

        it := 1)
    done;
    print_char '\n';
    if p = 1 then affiche_solution 0 n b else affiche_solution 1 n b;
|[]->();;

(* ALGORITHME : *)

(*
Fonction condition :
Cette fonction determine si une case peut etre encore utilisee quand une
nouvelle reine est placee.
parametres :
- x et y : ligne et colonne de la reine placee.
- x2 et y2 : ligne et colonne de la case a tester.
la case est eliminee si :
- elle a le meme numero de colonne que que la reine a placer.
- la difference entre son numero de ligne et le nuumero de ligne de la
reine a placer est egal
a la difference entre son numero de colonne et le nuumero de colonne de la
reine a placer.
(reines sur la meme diagonnale.)
*)

let condition x y x2 y2 =
  if y=y2 then false else
  if ( y2-y = x2-x ) or ( y2-y = -(x2-x) ) then false
  else true;;

(*
Fonction contrainte :
Cette fonction contraint les reines restant a placer en fonction d'une
nouvelle reine placee.
Soit, elle elimine pour chaque reines les colonnes qui lui sont desormais
interdites
en utilisant la fonction condition.
parametres :
- reines : liste des reines restant a placer.
- x et y : ligne et colonne de la reine placee.
Si une reine ne peut plus etre placee nul-part, la fonction renvoie Sans
pour interompre
cette branche de l'arbre de recherche.
*)

let rec constraintent reines x y = match reines with
  Reine(no,suivantes,liste)-> let liste_filtree = List.filter (condition
x y no) liste in
  if (liste_filtree <> []) then
    Reine(no,constraintent suivantes x y ,liste_filtree)
  else Sans
|Null->Null
|Sans->Sans;;

(*
Fonction ajouter_reine :
Fonction principale de l'algorithmme qui place une nouvelle reine a partir
de l'echiquier
plus ou moins rempli et des reines restant a placer.
parametres :
n : taille du probleme.

```

echiquier : liste des colonnes des reines déjà placées (cette liste est exhaustive car les reines sont placées par numéro de ligne croissant donc implicite.)
 reine : liste des reines restant à placer. (et leurs contraintes déjà calculées.)
 nb_solution : nombre de solutions déjà trouvées avant l'appel de la fonction.
 (la fonction retourne le nombre de solutions trouvées après l'appel de la fonction.)

Pour chaque colonne où la reine peut être placée, la fonction essaie de placer une nouvelle reine qui, le cas échéant, essaiera de placer une nouvelle reine et ainsi de suite. À l'ajout de chaque reine, les autres sont contraintes par la nouvelle reine placée. L'algorithme génère ainsi un arbre de recherche des solutions de profondeur n et de multiplicité $< (n - \text{profondeur du nœud})$.

Si la fonction contrainte renvoie Sans (une reine ne peut plus être placée) alors ajouter_reine ne fait rien. Si la liste des reines est finie, soit ajouter_reine est appelée avec Null, alors une solution a été trouvée.
 *)

```
let rec ajouter_reine n echiquier reine nb_solution = match reine with
  Reine(no,suivantes,liste)-> (* Il faut placer cette
  reine *)
  (
    match liste with
      a::b->
        ajouter_reine n echiquier (Reine(no,suivantes,b)) (* essaye
de place la reine sur une autre colonne *)
        (ajouter_reine n (a::echiquier) (contraintes suivantes no
a) nb_solution); (* essaye de placer d'autres reines *)
      []->nb_solution)
  |Null-> (* Toute les reines sont placees :
solution. *)
    Printf.printf "Solution no %d : " (nb_solution+1);
    affiche_liste echiquier;
    print_char '\n';
    affiche_solution 1 n echiquier;
    (nb_solution+1) (* la fonction retourne une
solution de plus.*)
  |Sans->nb_solution;; (* plus de solution possible dans
cet arbre : on ne fait rien.*)

(* PROGRAMME PRINCIPAL : *)

let arg = int_of_string Sys.argv.(1);; (* retrait du parametre :
taille du probleme. *)

let jeux = init_reines arg;; (* initialisation de la structure
de donnee. *)

let nb = ajouter_reine arg [] jeux 0;; (* Application de
l'algorithme. *)
```

```
Printf.printf "%d solutions trouvees.\n" nb;;
```