Backtracking - n queens

MPSI/MP OPTION INFORMATIQUE - TP nº 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 nxn 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 au problème des huit reines.

```
♕ . . .
. . ♕ .
. . . ♕
. ♕ . .
```

FIGURE 2 – Résultat sur la console de la fonction `imp_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.

(a) Écrire une fonction impérative dont le prototype est `imp_show board` où `board` est une liste représentant un échiquier avec des reines. Cette fonction affiche sur la console l'échiquier comme sur la figure 2 et 3.

(b) Écrire une fonction récursive dont le prototype est `rec_show board` où `board` est une liste représentant un échiquier avec des reines. Cette fonction affiche sur la console l'échiquier comme sur la figure 2 et 3.

```
. . . ♕
. . . .
. . ♕ .
. . . .
```

FIGURE 3 – Résultat de la fonction `imp_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.

(a) Écrire une fonction impérative 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.

(b) Écrire une fonction impérative 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 impérative 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 impérative 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 impérative 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 impérative 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 impérative 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 impérative de prototype `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?

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

```
Fonction RETOUR_SUR_TRACE(v)                                   ▷ v est un nøeud de l'arbre de recherche
    si v est une feuille alors
        renvoyer Vrai
    sinon
        pour chaque fils u de v répéter
            si u peut compléter une solution partielle au problème 𝒫 alors
                RETOUR_SUR_TRACE(u)
    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.

On n'a plus besoin de traiter le cas où la colonne est vide[2] 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 ?

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.

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

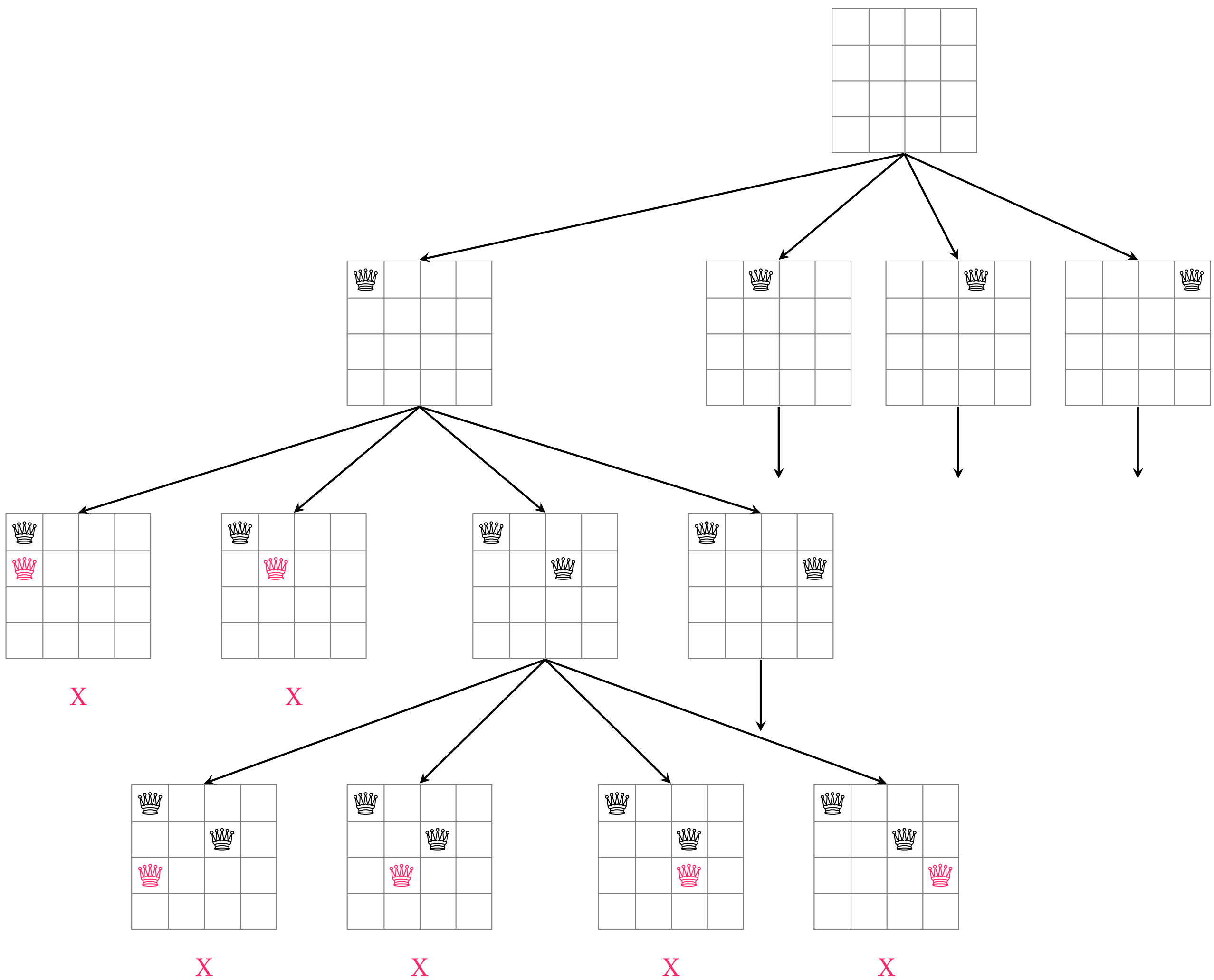

FIGURE 4 – Exemple d'arbre de recherche structurant l'algorithme de retour sur trace. Application au problème de quatre reines.