Problèmes d'échecs

1 Introduction

Au-delà du jeu d'échecs, qui fourmille de problèmes complexes à résoudre, l'échiquier et les déplacement des pièces sont à l'origine de centaines de problèmes de combinatoire qui ont alimenté la réflexion de mathématiciens depuis des siècles.

Si les problèmes ont généralement été posés pour un échiquier de taille normale (8×8 cases), il est aisé de moduler la difficulté de ces problèmes en ajustant la taille de l'échiquier.

On s'intéresse dans cette séance de travaux pratiques à deux problèmes très populaires, que l'on va essayer de résoudre, en OCaml, en utilisant le principe du retour sur trace : le problème des N reines et le tour du cavalier d'Euler.

On utilisera fréquemment, dans ces problèmes, le type OCaml `'a option` qui est **déjà** défini comme un type somme :

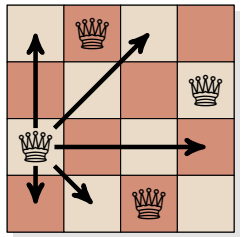
```
type 'a option = None | Some of 'a
```

Ce type permet à une fonction de retourner un résultat (sous la forme `Some x`) si elle est capable d'en trouver un, ou `None` si elle n'en trouve pas. Il s'agit d'un mécanisme un peu plus simple et efficace que d'utiliser des exceptions en cas d'échec à trouver une solution.

2 Problème des N reines

Dans ce premier problème, on cherche à placer N reines sur un échiquier de taille $N \times N$ de façon à ce qu'aucune reine ne menace aucune autre (les reines se déplacent sur l'échiquier d'un nombre quelconque de cases dans les quatre principales directions de l'échiquier ou en diagonale).

De façon évidente, les N reines doivent chacune être sur une ligne (et une colonne) différente. Il n'y a pas de solution pour $N = 2$ et $N = 3$, mais la solution ci-dessous convient pour $N = 4$:



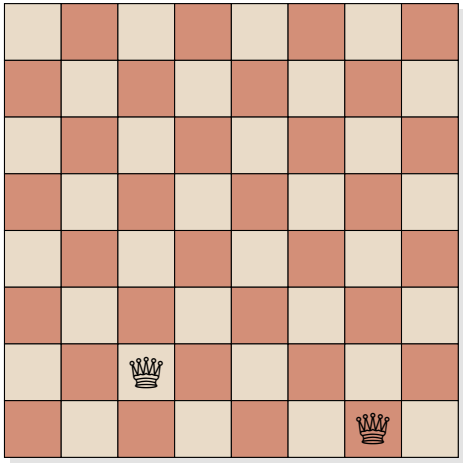
On représentera une solution sous la forme d'une liste d'index de colonnes, correspondant à chaque reine pris dans l'ordre de leurs lignes. La solution ci-dessus, par exemple, sera représentée par la liste `[1; 3; 0; 2]` (la reine sur la première ligne se trouve dans la colonne d'index 1, celle sur la seconde ligne dans la colonne d'index 3, et ainsi de suite).

On fournit la fonction suivante pour afficher, à partir d'une liste représentant une (possible) solution, une représentation du placement des pièces :

```
let affiche lst =
  let n = List.length lst in
  List.iter (fun i ->
    let rec aux = function
      | 0 -> print_newline ()
      | j -> print_char (if i+j=n then 'X' else '.'); aux (j-1)
    in aux n) lst
```

Pour construire une solution, nous allons utiliser le principe du retour sur trace, en construisant la liste de droite à gauche (donc en plaçant les reines de haut en bas).

Supposons que l'on travaille sur un échiquier de taille 8×8, et que les positions des reines sur les *deux dernières lignes* sont décrites par la liste `[2, 6]`, c'est-à-dire positionnées comme sur le graphe ci-dessous :



1. Quelles sont les positions autorisées (index de colonne) pour placer une reine sur l'antépénulptième ligne?

2. Proposer une fonction `permis` de type `int list -> int -> bool` prenant en argument une liste de k index de colonnes, correspondant aux positions des reines sur les k dernières lignes du plateau (numérotées de `n-k` à `n-1`), et un index de colonne `j` et

renvoyant un booléen indiquant s'il est permis de mettre une reine dans la colonne `i` de la ligne `n-k-1`. On pourra tester la fonction avec l'exemple de la question précédente. On pourra remarquer que la position j est permise si et seulement si :

- le premier terme de la liste doit être différent de $j-1$, j et $j+1$;
- le second terme de la liste doit être différent de $j-2$, j et $j+2$;
- et ainsi de suite.

On pourra donc utiliser une fonction auxilière pour parcourir la liste afin de pouvoir disposer de j et de la position de l'élément considéré dans la liste, par exemple sur ce modèle :

```
let permis lst j =
  let rec teste k reste = match reste with
    | [] -> ...
    | t::q -> ... && teste (k+1) q
  in teste 1 lst
```

3. En déduire une fonction `possibles` de signature `int -> int list -> int list` prenant en argument la taille `n` de l'échiquier, une liste de `k` k index de colonnes, correspondant aux positions des reines sur les k dernières lignes du plateau, et retournant la liste des index de colonne permis pour le placement d'une reine à la ligne les précédant.

4. En déduire une fonction `solution` de signature `int -> int list option` utilisant le principe du retour sur trace pour prendre en argument la taille de l'échiquier `n` et renvoyer `None` s'il n'y a pas de solution, et `Some lst` s'il existe au moins une solution, `lst` étant une telle solution. On propose une ébauche de structure, utilisant une fonction auxiliaire récursive prenant deux arguments, un nombre de reines déjà placées (sur les lignes en partant du bas) et la liste de leurs index de colonne :

```
let solution n =
  let rec explore nb_placees lst =
    if nb_placees = ... then ... else
    let possibilites = possibles n lst in
    let rec essaie = function
      | [] -> ...
      | t::q -> ...
    in essaie possibilites
  in explore 0 []
```

5. Jusqu'à quel taille d'échiquier peut-on résoudre le problème en un temps raisonnable ?

6. Modifier la fonction précédente pour qu'elle retourne le *nombre* de solutions au problème.

3 Problème du cavalier d'Euler

On s'intéresse à présent à un problème proposé par Léonard Euler. Il consiste à essayer de trouver un chemin, pour un cavalier d'échec, lui permettant de passer une fois et une seul par toutes les cases. On se propose à nouveau d'utiliser une approche par retour sur trace pour des raisons d'efficacité. On supposera par ailleurs, pour simplifier [1], que le cavalier débute sur la case $(0,0)$ en haut à gauche de l'échiquier.

Pour mémoriser les cases qui ont été visitées et celles qui ne l'ont pas été, nous allons utiliser un tableau d'entiers (`int array array`) de taille $N \times N$, dont les cases contiendront -1 si elles n'ont pas encore été visitées, et k si elles ont été visitées après k déplacements du cavalier (la case $(0,0)$ contiendra la valeur 0).

7. Proposer une fonction `init` de signature `int -> int array array` qui prend en argument une taille n et retourne le tableau correctement initialisé (-1 dans toutes les cases, à l'exception de la case $(0,0)$ qui contient 0).

Pour les tests, on fournit une fonction affichant un tableau d'entiers passé en argument :

```
let affiche =
  Array.iter (fun lgn ->
    Array.iter (fun x -> Printf.printf "%4d " x) lgn;
    print_newline ())
```

8. Proposer une fonction `possibles` de signature `int array array -> int * int -> (int * int) list` qui, pour un tableau et une position (i,j) donné (i indiquant le numéro de ligne sur l'échiquier et j le numéro de colonne), retourne la liste des positions accessibles qui n'ont pas encore été visitées.

9. Écrire une fonction `explore` qui de signature `int -> int * int -> int array array -> int array array option` qui prend en argument le nombre k de déplacements déjà effectués par le cavalier, sa position (i,j) et le tableau contenant les positions visitées lors des étapes *précédentes* (la case correspondant à (i,j) contient encore -1 pour le moment) et :

- place k dans la case (i,j) ;
- si $k = n^2 - 1$, retourne `Some tab` où `tab` est le tableau, normalement rempli avec des valeurs de 0 à n^2-1
- sinon,
  - détermine la liste des cases où un déplacement est possible ;
  - appelle, successivement pour chacun des déplacements possibles, récursivement la fonction `explore` avec les bons paramètres, et si un appel retourne `Some sol`, alors retourne `Some sol` (sans envisager les autres déplacements) et, si tous les appels retournent `None`, place -1 dans la case (i,j) et retourne `None`.

1. Ce n'est généralement pas une contrainte importante car, sur un échiquier de taille standard, il existe des chemins cycliques, donc le point de départ n'a pas d'importance

10. Justifier que la fonction précédente permet de trouver une solution au problème si une telle solution existe, et en déduire une fonction `tour` de signature `int -> int array array option` qui prend en argument la taille n de l'échiquier et retourne une solution s'il en existe une, et `None` sinon.

11. Pour quelles tailles d'échiquier l'approche précédente donne-t-elle des résultats satisfaisants ?

L'ennui, c'est que l'espace à explorer reste trop grand, et les solutions trop rares, pour que l'on puisse résoudre ce problème pour des n même modestes. Il convient donc d'être un peu plus malins.

Pour l'instant, le retour sur trace effectue une exploration en profondeur de l'arbre des possibilités, mais les enfants de chacun des nœuds de l'arbre ne sont pas ordonnés. Une idée intéressante est d'utiliser une *heuristique* pour explorer en priorité les solutions les plus prometteuses.

Pour évaluer la qualité d'un coup, on va écrire une fonction `heur` de signature `int array array -> int * int -> int` prenant en argument un tableau représentant les cases visitées et un couple (i, j) représentant une position et retournant un entier, dont on souhaite qu'il soit d'autant plus grand que possible si le coup semble intéressant.

On se propose, dans un premier temps, d'utiliser l'heuristique suivante : un déplacement est d'autant plus intéressant que le nombre de déplacements disponible pour le coup suivant est *faible*. L'idée est ici d'aller tout de suite dans les cases pour lesquelles il sera plus difficile de repartir ensuite.

12. Proposer une fonction `heur tab (i,j)` retournant l'opposé du nombre de possibilités de déplacement disponibles depuis la case (i, j).

Pour trier une liste de possibilités en fonction de l'heuristique, par ordre décroissant, on utilisera la fonction suivante :

```
let ordonne heur tab =
  List.sort (fun x y -> compare (heur tab y) (heur tab x))
```

Cette fonction, de signature `(int array array -> int * int -> int) -> int array array -> (int * int) list -> (int * int) list` prend en argument une fonction heuristique, le tableau indiquant les déplacements effectués, et une liste de déplacements possibles, et retourne la liste de déplacements possibles ordonnés par ordre décroissant de leur heuristique.

13. Modifier[2] la fonction `explore` ainsi que la fonction `tour`, pour qu'elles acceptent un argument supplémentaire, une heuristique de type `int array array -> int * int -> int`, et qu'elles l'utilisent pour trier les déplacements possible par ordre décroissant d'intérêt avant d'itérer sur ceux-ci.

Note : pour des raisons d'efficacité, on peut accélérer le calcul de l'heuristique en mémorisant, à tout instant, dans un second tableau, le nombre de déplacements possibles qu'il reste pour chaque case. On ne se souciera pas ici de cette optimisation.

14. Vérifier que cela permet de résoudre le problème pour de plus grands n. Quelle limite peut-on atteindre[3] ?

15. On a choisi pour heuristique d'aller vers les cases avec le *moins* de possibilités. Modifier la fonction `heur` pour privilégier les cases avec le *plus* de possibilités (on pensera à redéfinir les fonctions `possibles_ordonnes`, `explore` et `tour` qui en dépendent) et regarder si cette approche fonctionne.

16. Proposer une heuristique qui privilégie les coups en fonction de leur distance aux bords (plus une case est proche d'un bord, plus elle est intéressante). Est-elle efficace ? Comparer les résultats avec l'heuristique liée au nombre de possibilités disponibles.

17. Est-il possible de trouver un chemin où les déplacements dans les seize cases centrales sont sont toutes congrues à la même valeur modulo 4 (par exemple, on ne pénètre dans les cases centrales qu'aux déplacements 2, 6, 10, 14, etc.) ?

Pour aller plus loin, on pourra réfléchir à d'autres heuristiques, ou à imposer des contraintes supplémentaires sur le chemin, toutes possibles notamment pour $n = 8$:

- chemin fermé ;
- chemin avec des symétries ;
- chemin tel que le tableau retourné est un quasi-carré magique (il est impossible d'obtenir un vrai carré magique sur un échiquier 8×8, mais il est possible d'obtenir un tableau où chaque ligne et chaque diagonale ont la même somme, et même où chaque demi-ligne et chaque demi-colonne ont la même somme ; il faudra envisager un point de départ quelconque)

2. On sera avisé de conserver l'ancienne version à titre de référence.

3. On peut montrer que pour de très grands n, l'heuristique devient moins bonne.