Problèmes d'échecs

1 Introduction

Au-delà de l'authentique 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 nous allons nous efforcer de résoudre, en OCaml, en utilisant le principe du retour sur trace : le problème des « N reines » et le tour du cavalier.

On utilisera fréquemment, dans ces problèmes, le type OCaml `'a option` qui, rappelons-le, est défini comme un type somme (c'est un type du langage, il n'est *pas* à redéfinir, sa définition est donnée ici à titre de rappel) :

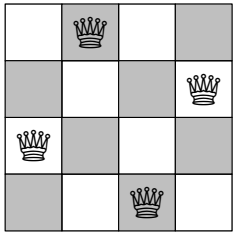
```
type 'a option = None | Some of 'a
```

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, par exemple, des exceptions en cas d'incapacité à trouver la solution requise.

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 ainsi qu'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]`.

On fournit la fonction suivante pour afficher, à partir de la taille n de l'échiquier et une liste représentant une possible solution, une représentation du placement des pièces :

```
let display n =
  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);;
```

1. Proposer une fonction `conflit` de type `(int*int) -> (int*int) -> bool` prenant en argument deux couples de coordonnées (ligne, colonne) et retournant un booléen indiquant si deux reines placées en ces positions se menacent mutuellement (on retournera `true` si les deux couples sont égaux).

2. Proposer une fonction `verifie` de signature `int list -> bool` prenant en argument une liste de longueur n contenant des entiers entre 0 et $n-1$, représentant un possible placement des reines (une reine par ligne, comme précédemment décrit), et retournant un booléen indiquant si cette solution est correcte.

Pour trouver une solution qui existe, on va dans un premier temps essayer toutes les possibilités (les listes de taille n contenant des entiers entre 0 et $n-1$), utiliser la fonction précedente pour les tester, et retourner la première qui convient.

3. Proposer une fonction `suivant` de signature `int -> int list -> int list option` qui prend en argument n et une liste contenant des entiers de $[\![0\,..\,n-1]\!]$, et qui retourne la liste suivante pour l'ordre lexicographique *de la liste retournée* s'il en existe et `None` sinon. Par exemple :

```
# suivant 4 [1; 0; 3; 2];;
- : int list option = Some [2; 0; 3; 2]

# suivant 4 [3; 3; 1; 2];;
- : int list option = Some [0; 0; 2; 2]

# suivant 4 [3; 3; 3; 3];;
- : int list option = None
```

4. En déduire une fonction de signature `int -> int list option` prenant en argument

un entier n et retournant une liste d'index de colonnes représentant une disposition acceptable de n reines sur un échiquier de taille $n \times n$ si une telle disposition existe, et `None` sinon.

5. Vérifier qu'il n'existe aucune solution pour $n = 2$ et $n = 3$, et que l'on trouve bien une solution pour $n = 4$. Pour quelle valeur de n cette approche devient-elle problématique ?

Pour gagner en efficacité, nous allons utiliser une approche de retour sur trace.

6. Proposer une fonction `possibles` de signature `int -> int list -> int list` qui, à partir de n et d'une liste de longueur $k < n$ contenant les positions des reines sur les k *dernières* lignes (de $n-k$ à $n-1$), retourne la liste des positions possible pour la reine sur la $n-k-1^{\text{e}}$ ligne (on retournera naturellement une liste vide si aucune position ne convient).

7. Écrire une fonction `explore` récursive, de signature `int -> int -> int list -> int list option`, prenant en argument un entier `n`, un entier `k` et une liste `lst` de taille k contenant des positions valides pour les k dernières lignes, et :

- retourne « `Some lst` » si $k = n$ (on a trouvé une solution) ;
- sinon, détermine la liste des positions possibles pour la ligne $n-k-1$, et :
  - retourne « `None` » si cette liste est vide
  - et sinon s'appelle récursivement pour toutes les listes où chacune des possibilités est rajoutée à gauche de la liste `lst` ; si l'un des appels retourne `Some x`, on retournera la liste complétée correspondante, toujours sous la forme d'un objet de type `'a option` (immédiatement, sans effectuer les autres appels), et si tous les appels retournent `None`, alors on retourne `None`

8. Justifier qu'un appel à `explore` avec les bons paramètres permet de trouver une solution au problème s'il en existe une, et en déduire une fonction `resout` de signature `int -> int list option` qui prend en argument n et retourne une solution s'il en existe une, et `None` sinon.

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

3 Problème du cavalier

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 rappelle qu'un cavalier se déplace de deux cases dans une direction et d'une case dans la direction orthogonale. 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 ainsi la valeur 0, la case $(1,2)$ ou la case $(2,1)$ contiendra la valeur 1, et ainsi de suite).

10. Proposer une fonction `init` de signature `int -> int array array` qui prend en argument une taille n et retourne le tableau correctement initialisé, correspondant à l'état où le cavalier est sur sa position initiale et n'a encore visité aucune autre case (soit -1 dans toutes les cases, à l'exception de la case $(0,0)$ qui contient 0).

Pour les tests, on fournit la fonction suivante qui affiche un tableau d'entiers passé en argument :

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

11. 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 depuis la case (i, j) qui n'ont pas encore été visitées.

12. Écrire une fonction `explore` qui de signature `int -> int * int -> int array array -> int array array option` qui prend en argument un entier k identifiant le numéro du déplacement en train d'être effectué par le cavalier (pour le premier déplacement, $k = 1$), la position (i, j) vers laquelle il se déplace, ainsi que 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 arr` où `arr` 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 renvoie `Some sol`, alors renvoie `Some sol` (sans envisager les autres déplacements) et, si tous les appels retournent `None`, replace -1 dans la case (i, j) et renvoie `None`.

13. 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.

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

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

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.

15. Proposer une fonction `heur board (i,j)` retournant 8 auquel on retranche le nombre de possibilités de déplacement disponibles depuis la case (i, j).

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

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

16. À partir de `possibles` et `ordonne`, construire une fonction `possibles_ordonnes` 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 ordonnées par ordre décroissant de leur heuristique.

17. Substituer à `possibles` la fonction `possibles_ordonnes` dans la fonction `explore` (on pensera égaement à redéfinir la fonction `tour` qui dépend de `explore`), et regarder si cela permet de résoudre le problème pour de plus grands n. Quelle limite peut-on atteindre[2] ?

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

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.

18. 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.

19. 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.

20. 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)...