TP #22 — Problème du ♘

Le *problème du cavalier* en dimension $m \times n$ consiste à trouver un ordre de parcours des cases d'un échiquier de dimension $m \times n$ tel que :

— toute case est parcourue exactement une fois ;
— la $i+1^{\text{ème}}$ case à être parcourue est accessible depuis la $i^{\text{ème}}$ case en utilisant les règles de déplacement du cavalier (♘). De façon équivalente, le problème consiste à trouver un « chemin hamiltonien » dans le graphe défini par les déplacements possibles du ♘. (Pour plus de contexte et de jolies animations, on pourra consulter la page wikipédia sur le sujet.)

Ce problème se résout facilement par *backtracking*, et comporte un très grand nombre de solutions même pour des dimensions modestes, cf. https://oeis.org/A165134 (mais pas *trop* modestes, cf. https://xkcd.com/839/).
L'objectif de cet exercice est d'être capable de calculer une liste de solutions distinctes (dans la limite d'une taille maximale donnée en argument) ; ceci peut se faire en modifiant de façon minimale une fonction de résolution par backtracking trouvant *une* solution.

Représentation des données. Dans tout l'exercice on utilisera un `type cb = int option array array` pour représenter les solutions (éventuellement partielles) : une solution au problème en dimension $m \times n$ sera une valeur `b` de type `cb` telle que `b` a m lignes et n colonnes et exactement une case de valeur `Some t` pour $0 \leq t < mn$ (en particulier, `b` n'a aucune case valant `None`), et où la numérotation respecte les contraintes du problème (c'est à dire qu'une case de valeur `Some t` doit être accessible depuis la case de valeur `Some (t-1)` (si elle est définie) par un déplacement de cavalier (pour l'association naturelle entre les cases de `cb` et celles d'un échiquier)). Une *solution partielle* est définie de façon identique à la différence que certaines cases peuvent être à `None` ; les valeurs prises par t doivent rester consécutives, mais en revanche il n'est *pas* nécessaire qu'une solution partielle puisse-t'être *étendue* en une solution (non partielle) qui lui est identique là où elle est définie.
On supposera également définies des constantes globales `dimX` et `dimY` égales à m et n respectivement.
On donne ci-dessous un exemple de solution et de solution partielle (qui ne peut pas être étendue en solution complète) en dimension 5×5 :

```
[|[|Some  0; Some 19; Some  8; Some 13; Some  2|];
  [|Some  9; Some 14; Some  1; Some 18; Some 23|];
  [|Some 20; Some  7; Some 22; Some  3; Some 12|];
```

```
   [|Some 15; Some 10; Some  5; Some 24; Some 17|];
   [|Some  6; Some 21; Some 16; Some 11; Some  4|]|]

 [|[|Some  0; Some 19; Some  8; Some 13; Some  2|];
   [|Some  9; Some 14; Some  1; Some 18; None   |];
   [|Some 22; Some  7; Some 20; Some  3; Some 12|];
   [|Some 15; Some 10; Some  5; None   ; Some 17|];
   [|Some  6; Some 21; Some 16; Some 11; Some  4|]|]
```

1. On donne la fonction `reachable` ci-dessous, qui pour une position `(i, j)` s'évalue en la liste des positions de l'échiquier accessibles par un ♘.

```
let reachable (i, j:int * int) : (int * int) list =
    let reach_base = [(i-2,j+1) ; (i-2,j-1) ;
                      (i-1,j+2) ; (i-1,j-2) ;
                      (i+1,j+2) ; (i+1,j-2) ;
                      (i+2,j+1) ; (i+2,j-1)]
    in
    let within_bounds (x,y) =
        (x >= 0) && (x < dimX) &&
        (y >= 0) && (y < dimY) in
    List.filter within_bounds reach_base
```

   Modifiez cette fonction en une fonction :

   `reachable_clear : cb -> int * int -> (int * int) list`

   qui filtre additionnellement le résultat tel que calculé par `reachable` pour en supprimer les cases de son premier argument qui ont déjà été visitées (c'est à dire, qui ne sont pas à `None`).

Une *heuristique* pour la résolution du problème du cavalier consiste, depuis une case `c`, à choisir comme prochaine case à visiter celle parmi les cases accessibles depuis `c` depuis laquelle le moins de cases sont accessibles (et libres).

2. Écrivez une fonction OCaml :

   `heuristic_reorder : cb -> (int * int) list -> (int * int) list`

   telle que pour `b` une solution partielle et `x` une liste de cases, un appel à `heuristic_reorder b x` s'évalue en une permutation de `x` dont les éléments sont triés par ordre croissant du nombre de cases libres dans `b` qui leur sont accessibles.
   Écrivez également une fonction OCaml `reachable_heuristic` qui la compose avec `reachable_clear`.
   (Il peut être utile d'utiliser ici les itérateurs `List.map` et `List.sort`.)

3. Écrivez une fonction OCaml récursive :

   `_kt : cb -> (int * int) list -> int -> cb option`

telle que pour une solution partielle `b` de `t` cases de valeur autre que `None` et une liste `dest` (possiblement vide) de cases accessibles & libres depuis la dernière case visitée, `_kt b dest t` utilise une approche par backtracking pour :

- s'évaluer à `None` s'il n'existe aucune solution complète qui étende `b` et pour laquelle la case de valeur `Some t` se trouve dans celles données par `dest` ;
- s'évalue à `Some b'` avec `b'` une telle solution, s'il en existe une.

Remarque. Prenez garde à correctement gérer la mutabilité du type `cb`.

4. Écrivez une fonction OCaml `kt : int -> int -> cb option` qui utilise `_kt` pour calculer une option sur une solution au problème du cavalier commençant à la case de coordonnées fournies en arguments.
5. Utilisez votre fonction `kt` pour trouver des solutions en dimension carrée 5, 8, 10, 70… Constatez sur de petits cas l'amélioration de performance apportée par l'utilisation de l'heuristique ci-dessus.
6. Écrivez une nouvelle fonction qui modifie *a minima* votre fonction `_kt` en :

```
_ktl : cb -> -> (int * int) list -> int ->
       cb list -> int -> cb list
```

   qui prend comme arguments supplémentaires une liste de solutions `sols` et une taille limite `lim` pour cette liste, et qui s'il existe des solutions étendant `b` avec `dest` (dans le même sens que pour `_kt`) s'évalue en une liste complétant `sols` avec ces solutions (dans la limite d'une taille totale `lim`), et sinon en `sols`.
7. Écrivez une fonction OCaml `ktl : int -> int -> int -> cb list` telle que `ktl si sj lim` s'évalue en une liste d'au plus `lim` solutions au problème du cavalier commençant à la case `(si, sj)`.
8. Vérifiez que votre fonction trouve le bon nombre de solutions pour les 25 cases de départ d'un échiquier carré de dimension 5, comme indiqué dans l'exemple donné sur https://oeis.org/A165134.