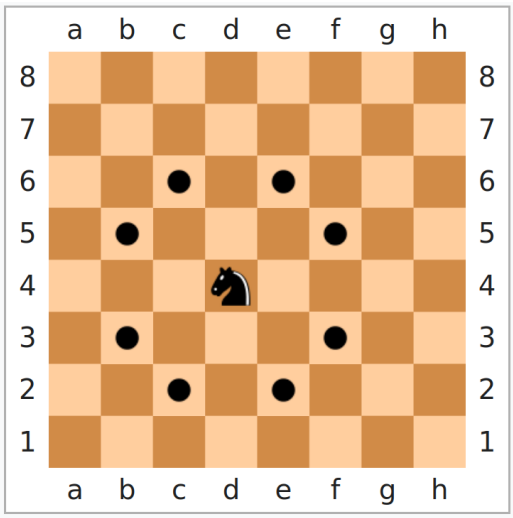
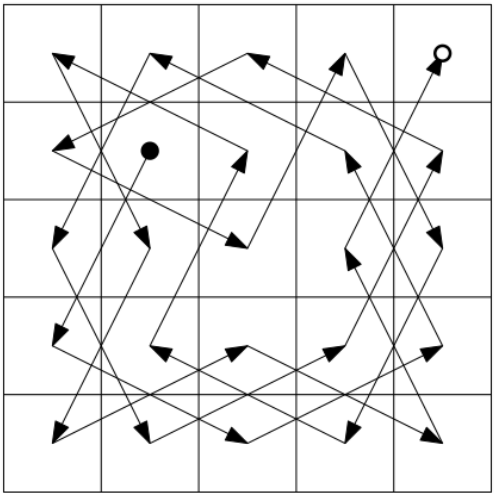
TRAVAUX PRATIQUES XII

Cavalier d'Euler

L'objectif de ce TP est de résoudre le problème du cavalier d'Euler. Pour rappel, le problème est de savoir si un cavalier (d'échecs) partant de la case (i, j) sur un échiquier $n \times n$ peut se déplacer sur l'échiquier de sorte à parcourir chaque case une et une seule fois. Le cavalier ne peut non plus pas faire un mouvement qui le sortirait du plateau.



(a) Les 8 mouvements possibles d'un cavalier (pour peu que les cases d'arrivée existent)



(b) Résolution du cavalier d'Euler sur un échiquier 5×5 en partant de $(1, 1)$

FIGURE XII.1 – Explications du cavalier d'Euler

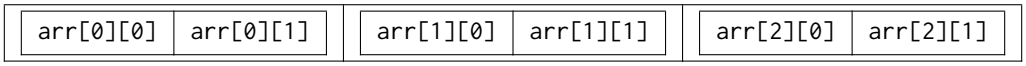
Dans ce TP, on programme en C.

A Le plateau

Le but de cette section est de construire le plateau. Deux variantes sont proposées. Dans les deux cas, il s'agit de représenter le tableau comme une matrice en maths (mais en partant de $(0, 0)$ et non de $(1, 1)$), et d'implémenter cette matrice comme le tableau de ses lignes, c'est à dire comme un tableau de tableaux.

$$\begin{pmatrix} (0,0) & (0,1) \\ (1,0) & (1,1) \\ (2,0) & (2,1) \end{pmatrix}$$

(a) Une matrice M de dimension 3×2



(b) Le tableau `arr` correspondant à M

FIGURE XII.2 – Correspondance entre la représentation matricielle et l'implémentation par tableau de tableaux

Dans la case initiale du cavalier sera stockée 0, dans la première case où va le cavalier sera stocké 1, dans la seconde 2, etc. Dans une case où le cavalier n'est jamais allé, on mettra une valeur spéciale, `vide` .

0. Dans `cavalier.c`, ajoutez la déclaration globale `int const vide = -1` .

A.1 Version non-étoilée

On créera les tableaux comme des tableaux statiques. Par exemple, `int arr[8][5]` crée un tableau de 8 sous-tableaux contenant chacun 5 entiers.

On se propose pour simplifier de fixer la dimension du plateau dans le code source. Pour cela, au début de `cavalier.c` (juste avant/après `vide`), on écrit :

```c
#define DIM 6
```

1. Faites-le !

Cela crée `DIM` qui « vaut » 6. Plus précisément, cela fait qu'à la compilation, `gcc` va chercher-remplacer toutes les occurences de `DIM` par 6. Ainsi, ce n'est *pas* une variable !

Pour créer un échiquier de `DIM` × `DIM` entiers, il suffit donc de faire :

```c
int echiquier[DIM][DIM];
```

2. Créez une fonction `bool cavalier(int n, int i, int j)`. Cette fonction doit créer un échiquier de taille `DIM` × `DIM`, initialiser toutes ses cases avec la valeur `vide` sauf la case de coordonnées (i, j) qui est initialisée à `0`. *Cette fonction n'est pas encore finie, on la complètera plus tard.*

   Vous pouvez utiliser la fonction `print_plateau` pré-codée qui affiche un plateau pour tester votre fonction.
3. Ajoutez un `assert` au début de votre fonction qui vérifie que `n` = `DIM`. Pour rappel, cette fonction (rangée dans `assert.h`) a pour prototype `void assert(bool condition)`, et ne fait rien si `condition` est vraie mais interrompt le programme sinon. Elle sert à garantir qu'une condition est vraie.

A.2 Version étoilée

On allouera les différents tableaux avec `malloc`. Ainsi, le plateau est un `int** arr` tel que `arr[i]` est un `int*` qui pointe vers les valeurs de la ligne numéro `i`.

4. Écrivez une fonction `int** cree_plateau(int n, int i, int j)` qui crée un plateau de dimensions `n` × `n`. Elle initialise toutes ses cases avec la valeur `vide` sauf la case de coordonnées (i, j) qui est initialisée à `0`, puis renvoie le plateau.
5. Écrivez une fonction `void free_plateau(int** plateau, int n)` qui libère toute la mémoire allouée d'un plateau.
6. Créez une fonction `bool cavalier (int n, int i, int j)` qui crée un plateau (puis saute des lignes) puis le libère. *Cette fonction n'est pas encore finie, on la complètera plus tard.*

   Vous pouvez utiliser `print_plateau_star` qui affiche un plateau pour vérifier la validité du tout, ainsi que l'option de compilation `-fsanitize=address` qui détecte des fuites de mémoire.

B Le retour sur trace

B.1 Choix successifs

Le retour s'applique s'applique à des problèmes où la solution peut-être exprimée comme le résultat de choix successifs. Ici, les choix successifs sont le premier déplacement du cavalier, puis le second, etc. Pour avoir une solution valide, il faut se déplacer $n^2 - 1$ fois sans jamais passer deux fois par la même case.

Les options possibles pour un choix sont les cases où le cavalier peut aller.

B.2 Rejet

Pour rejeter un choix, on va tester si la case ciblée existe (c'est à dire si elle est dans le plateau) et est disponible (c'est à dire si le cavalier n'y est pas encore passée, autrement dit si elle contient `vide`).

7. Écrivez une fonction `bool disponible(plateau, int n, int x, int y)` qui renvoie `true` si et seulement si la case (x, y) existe et est libre. Le type de `plateau` dépend de la variante du TP :
   - Non-étoilée : `bool disponible(int plateau[DIM][DIM], int n, int x, int y)`
   - Étoilée : `bool disponible(int const** plateau, int n, int x, int y)`
8. Testez-la ! testez notamment toutes les façons de renvoyer `true` ou `false`.

B.3 Validation

Il n'y a en fait pas de fonction de validation à écrire ! En effet, si une solution est complète et n'a jamais été rejetée, cela signifie que le cavalier a marché $n^2 - 1$ fois mais qu'à aucun de ces déplacements il n'a marché sur une case précédente : c'est ce que l'on attend.

B.4 Le retour sur trace en lui-même

9. Écrivez une fonction `bool backtrack(plateau, int i, int j)` qui fait un retour sur trace pour résoudre le problème du cavalier. Le cavalier est actuellement en position (i, j) ; notez que cela implique que `plateau[i][j]` contient le nombre de déplacements effectués jusqu'à présent.
La fonction doit renvoyer `true` si le cavalier peut parcourir les cases non-encore parcourues sans passer deux fois par la même case, et `false` sinon.

⚠ L'étape difficile est d'énumérer les choix possibles pour le prochain déplacement. Trois possibilités :

- Utiliser un gros `else if` qui couvre les 8 options.
- Exprimer les 8 déplacements possibles sous la forme de vecteurs (dx, dy) ; faire un tableau qui stocke tous les dx et un autre tous les dy, et parcourir simultanément ces deux tableaux pour parcourir tous les (dx, dy)
- Toute autre façon qui marche et vous semble plus intuitive.

10. Modifiez la fonction `cavalier` pour qu'elle appelle `backtrack`. Elle doit :

- Si `backtrack` renvoie `true`, afficher « Le cavalier d'Euler sur un échiquier $n \times n$ en partant de (i, j) est soluble. » puis afficher une solution.
- Sinon, afficher « Le cavalier d'Euler sur un échiquier $n \times n$ en partant de (i, j) n'est PAS soluble. ».

11. Testez ! Testez pour différentes valeurs de `n` et de (i, j). Pour plus de confort, vous pouvez utiliser `scanf` pour demander (i, j) et `n` à l'utilisateur (attention, en version non-étoilée vous ne pouvez pas demander n puisqu'il s'agit de `DIM` qui est écrit « en dur » dans le code).

12. Faites varier l'ordre dans lequel les différents mouvements sont testés. Certains sont bien plus rapides que d'autres. Si vous les parcourez dans un « mauvais » ordre, un 8×8 peut-être lent ; dans un « bon » ordre un 200×200 se fait en quelques secondes.
Indice sur une méthode efficace en bas de page [1].

1. Indice : privilégiez les mouvements qui « ont le moins d'options de prochain déplacement possibles ». En effet, ceux-ci sont soit rejetés très vite, soit atteignent très vite une solution valide puisqu'il y a peu d'options pour les choix à faire sur le chmin.