[CI2] Cours 9: Backtracking II

Daniela Petrişan
Université Paris Cité, IRIF

Le problème du cavalier

Dans le jeu d'échecs, le cavalier peut se déplacer deux cases dans une direction, puis une case dans une direction perpendiculaire.

Le problème du cavalier

Dans le jeu d'échecs, le cavalier peut se déplacer deux cases dans une direction, puis une case dans une direction perpendiculaire.

Le problème du cavalier est le suivant :

Un cavalier posé sur une case quelconque d'un échiquier de dimension `n × n` doit en visiter toutes les cases sans passer deux fois par la même. Le parcours peut être ouvert, c'est à dire, il n'est pas nécessaire de revenir à la case de départ.

Le problème du cavalier

Dans le jeu d'échecs, le cavalier peut se déplacer deux cases dans une direction, puis une case dans une direction perpendiculaire.

Le problème du cavalier est le suivant :

Un cavalier posé sur une case quelconque d'un échiquier de dimension `n × n` doit en visiter toutes les cases sans passer deux fois par la même. Le parcours peut être ouvert, c'est à dire, il n'est pas nécessaire de revenir à la case de départ.

Écrire un algorithme basé sur le backtracking construisant tous les parcours ouverts d'un échiquier de dimension `n`.

Une classe pour modéliser les mouvements du cavalier

```
class Vec2 {
    public int x;
    public int y;

    public Vec2(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public static Vec2 add(Vec2 a, Vec2 b) {
        return new Vec2(a.x + b.x, a.y + b.y);
    }

    public String toString() {
        return "(" + x + ", " + y + ")";
    }
}
```

Une classe pour modéliser les mouvements du cavalier

```
public class KnightMoves {

    /* à partir d'une position donnée (x,y), le cavalier peut se
     * déplacer vers la position (x+u, y+v), où (u,v) est l'un des
     * vecteurs du tableau */
    static Vec2[] knightMoves = {
        new Vec2(+2, +1),
        new Vec2(+2, -1),
        new Vec2(-2, +1),
        new Vec2(-2, -1),
        new Vec2(+1, +2),
        new Vec2(-1, +2),
        new Vec2(+1, -2),
        new Vec2(-1, -2),
    };
```

Les mouvements du cavalier

```java
//on vérifie qu'une position donnée pos est sur l'échiquier de taille n
public static boolean isLegalPosition(Vec2 pos, int n) {
    return pos.x >= 0 && pos.x < n && pos.y >= 0 && pos.y < n;
}
```

Les mouvements du cavalier

```java
//on vérifie qu'une position donnée pos est sur l'échiquier de taille n
 public static boolean isLegalPosition(Vec2 pos, int n) {
     return pos.x >= 0 && pos.x < n && pos.y >= 0 && pos.y < n;
 }
```

```java
//L'échiquier est initialisé avec 0 en toutes les positions qui ne
 //sont pas sur le parcours. Si une case a déjà été visitée, elle
 //contiendra un nombre strictement positif
 public static boolean isOccupiedPosition(Vec2 pos, int [][]board) {
     return board[pos.x][pos.y] != 0;
 }
```

La méthode backtracking

```
static void knightBacktrack(int[][] board, int n, Vec2 currentPos, int v) {
    /* on ajoute au parcours la position courante en modifiant la
     *  valeur sur la case correspondante de l'échiquier avec la
     *  valeur v, la taille du parcours jusqu'à ce moment
     */
     board[currentPos.x][currentPos.y] = v;


     ....
```

La méthode backtracking

```
static void knightBacktrack(int[][] board, int n, Vec2 currentPos, int v) {
    /* on ajoute au parcours la position courante en modifiant la
     *  valeur sur la case correspondante de l'échiquier avec la
     *  valeur v, la taille du parcours jusqu'à ce moment
     */
     board[currentPos.x][currentPos.y] = v;

    if (v >= n * n) { // Si le parcours est complet
        System.out.println("Solution:");
        printBoard(board, n);
        System.out.println("");
    } else {

      .....
```

La méthode backtracking

```
static void knightBacktrack(int[][] board, int n, Vec2 currentPos, int v) {
    // on ajoute au parcours la position courante
    board[currentPos.x][currentPos.y] = v;

    if (v >= n * n) { // Si le parcours est complet
        System.out.println("Solution:");
        printBoard(board, n);
        System.out.println("");
    } else {
      /* Si le parcours n'est pas complet, on vérifie pour chaque mouvement possible
       * si on atteint une position valide et dans ce cas  on appelle alors la méthode
       * récursive à partir de cette nouvelle position.*/
        for (Vec2 move : knightMoves) {
            Vec2 nextPos = Vec2.add(currentPos, move);
            if (isLegalPosition(nextPos, n) && !isOccupiedPosition(nextPos, board)) {
                knightBacktrack(board, n, nextPos, v + 1);
        }}} ....
```

La méthode backtracking

```
static void knightBacktrack(int[][] board, int n, Vec2 currentPos, int v) {
    // on ajoute au parcours la position courante
    board[currentPos.x][currentPos.y] = v;
    if (v >= n * n) { // Si le parcours est complet
        System.out.println("Solution:");
        printBoard(board, n);
        System.out.println("");
    } else {
      /* Si le parcours n'est pas complet, on appelle alors la méthode récursive à partir
       * des positions suivantes valides*/
        for (Vec2 move : knightMoves) {
            Vec2 nextPos = Vec2.add(currentPos, move);
            if (isLegalPosition(nextPos, n) && !isOccupiedPosition(nextPos, board)) {
                knightBacktrack(board, n, nextPos, v + 1);
        }}}
    // retour sur la trace
    board[currentPos.x][currentPos.y] = 0;
}
```

La méthode backtracking

```
/* Nous appelons la méthode backtracking pour un échiquier de
 * taille n et en supposant que le cavalier commence en position
 * (0,0). */
static void knight(int n) {
     int[][] board = new int[n][n];
     knightBacktrack(board, n, new Vec2(0, 0), 1);
}

public static void main(String[] args){
  knight(5);
}
```