51IF2IK3 – Examen Session 1

13 novembre 2009 – durée 2h – aucun document autorisé

Dans le jeu d'échec, le cavalier est la pièce ayant le déplacement le plus original. Il se déplace en L, c'est-à-dire de deux cases dans une direction puis d'une perpendiculairement (voir l'échiquier de gauche de la figure 1). C'est la seule pièce du jeu qui ne soit pas bloquée dans son déplacement par les autres pièces.

Nous allons utiliser cette pièce pour jouer un jeu plus simple que celui des échecs. Il s'agit de trouver le chemin d'un cavalier placé sur un échiquier pour atteindre un point donné de l'échiquier sans passer deux fois par la même case. L'échiquier à droite de la figure 1 donne un exemple d'instance de ce jeu : le cavalier est placé dans la case d6 et il faut le déplacer vers la case g3.

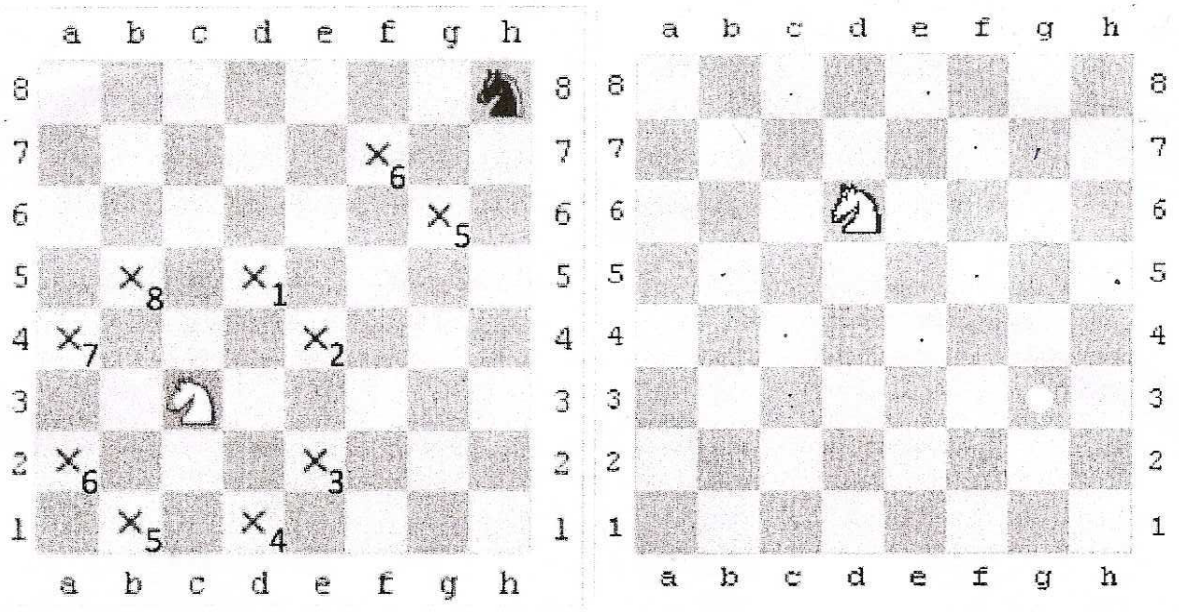

FIG. 1 – Les déplacements d'un cavalier sur un échiquier et une instance du jeu.

Question 1) Donner au moins deux façons de déplacer le cavalier pour résoudre le problème de la figure 1 (droit) en listant les cases par lesquelles le cavalier doit passer. Quel est le nombre minimum de déplacements pour résoudre ce problème ? Y-a-t-il un nombre maximum de déplacements ? La même question si on lève l'interdiction de ne pas repasser par une case. Justifier vos réponses en donnant des exemples de déplacements.

Les questions qui suivent vous aident à obtenir un algorithme qui résout ce jeu pour toute instance (case initiale, case finale) du jeu. Cet algorithme doit être un algorithme de type *backtrack*. Dans les annexes, vous trouverez l'interface `Jeu` et la classe `Backtrack` données en cours.

Pour faciliter l'implémentation de cet algorithme, on fait les deux choix suivants. Premièrement, les déplacements du cavalier sur l'échiquier laissent des traces sur celui-ci : à chaque case visitée par le cavalier est affecté un entier qui indique le nombre de déplacements faits pour y arriver depuis la case initiale. Ainsi, la case initiale est notée par 0, la case obtenue après le premier déplacement a le numéro 1, etc. Deuxièmement, on associe un numéro à

chaque déplacement possible du cavalier autour de la case dans laquelle il se trouve, comme indiqué dans l'échiquier à gauche de la figure 1.

Ainsi, une solution du jeu peut être représentée par une suite d'entiers $d_0, d_1, \cdots$, chaque entier $d_i \in [1, 8]$ indiquant le déplacement effectué dans la $i$ème case visitée par le cavalier (la case initiale correspond à l'indice 0). Par exemple, pour déplacer le cavalier de la case d6 à la case d8 on donne la suite 2,7.

Question 2) Exprimer les solutions construites à la question 1) en utilisant une suite d'entiers représentant des déplacements.

Question 3) En utilisant les choix d'implémentation ci-dessus, donner une modélisation de ce jeu sous forme d'un problème de recherche de solution par *backtrack*. Plus précisément, vous devez définir le vecteur solution, l'alphabet de valeurs de ce vecteur et la relation d'ordre entre ces valeurs.

Question 4) Dérouler à la main la partie de l'algorithme itératif de *backtrack* qui amène à une des solutions que vous avez trouvées à la question 1). Indiquer, à chaque pas, le contenu de la pile et les résultats attendus pour les méthodes appelées.

Dans la suite, vous devez écrire le code Java permettant d'obtenir une implémentation de l'algorithme de recherche de solution. Ce code doit être écrit le plus proprement possible et commenté si besoin.

Question 5) Pour commencer, définir une classe `JeuCavalier` qui implémente l'interface `Jeu`. Préciser quels sont les champs de cette classe. Implémenter un constructeur qui reçoit comme arguments la case initiale et finale du jeu qui sont des valeurs de la classe `Case` (voir sa définition dans l'annexe).

Question 6) Implémenter une méthode `deplacement` qui reçoit en arguments un objet c de classe `Case` et un entier d dans $[1, 8]$ et qui calcule la case correspondant au $d$ème déplacement possible du cavalier autour de c en respectant les conventions de notation de la figure 1.

Question 7) Implémenter une méthode `depl2case` qui reçoit en arguments un entier i et renvoie la case correspondant au $i$ème déplacement du cavalier depuis le début du jeu.

Question 8) En utilisant la modélisation faite à la question 3), indiquer quelles sont les classes qui typent les objets `position` et `valeur`, puis donner une implémentation des méthodes de l'interface `Jeu`.

Question 9) Ecrire le code qui applique l'algorithme `iteratif` de la classe `Backtracking` au jeu du cavalier.

Question 10) Modifier le code de l'algorithme `iteratif` de la classe `Backtracking` afin de calculer le plus petit nombre de déplacements pour résoudre le jeu du cavalier.

Annexes

```
package backtracking;
import java.util.Stack;
/**
 * Interface pour la modelisation des problemes à resoudre par
 * backtracking.
 */
public interface Jeu {
  /**
   * @return position initiale du jeu à remplir
   */
  public Object positionInitiale();
  /**
   * @return valeur initiale possible pour la position donnée,
   *         ou null si pas de valeur possible
   */
  public Object valeurInitialePossible(Object position);
  /**
   * @return valeur suivante possible pour la position donnée,
   *         ou null si pas de valeur possible
   */
  public Object valeurSuivantePossible(Object valeur, Object position);
  /**
   * @return true si la pile est une solution, false sinon
   */
  public boolean estSolution(Stack<Object> pile);
  /**
   * @return true si la position est à remplir dans le jeu, false sinon
   */
  public boolean positionValide(Object position);
  /**
   * @return la position suivante à remplir
   */
  public Object positionSuivante(Object position);
  /**
   * @return la position précédente remplie
   */
  public Object positionPrecedente(Object position);
  /**
   * affiche la solution
   * @param pile
   */
  public void affiche(Stack<Object> pile);
  /**
   * empile la valeur dans la pile et actualise l'état du jeu
   * @return la nouvelle pile
   */
  public Stack<Object> empiler(Stack<Object> pile, Object valeur);
  /**
   * depile et actualise l'état du jeu
   * @return le haut de la pile
   */
  public Object depiler(Stack<Object> pile);
}
```

```java
package backtracking;
import java.util.Stack;
/**
 * Classe avec l'algorithme iteratif de backtracking.
 */
public class Backtracking {
  Jeu jeu;

  public Backtracking(Jeu jeu) {
    this.jeu = jeu;
  }

  public void iteratif(){
    Stack<Object> pile = new Stack<Object>();
    Object position = this.jeu.positionInitiale();
    Object valeurCourante = this.jeu.valeurInitialePossible(position);

    while(this.jeu.positionValide(position)) {
      while(valeurCourante != null) {
        this.jeu.empiler(pile, valeurCourante);
        if(this.jeu.estSolution(pile)) {
          this.jeu.affiche(pile);
          valeurCourante = this.jeu.valeurSuivantePossible(this.jeu.depiler(pile),position);
        } else {
          position = this.jeu.positionSuivante(position);
          valeurCourante = this.jeu.valeurInitialePossible(position);
        }
      }
      position = this.jeu.positionPrecedente(position);
      if (!pile.empty()) {
        valeurCourante = this.jeu.valeurSuivantePossible(this.jeu.depiler(pile),position);
      }
    }
  }
}

/**
 * Classe pour les coordonnees des cases.
 */
public class Case {
  public int x;
  public int y;

  Case (int x, int y){
    this.x = x; this.y = y;
  }
  public String toString() {
    return "(" + x + "," + y + ")";
  }
}
```