POLYTECHNIQUE MONTRÉAL

LE GÉNIE EN PREMIÈRE CLASSE

INF8215
Intelligence Artificielle

Travaille Pratique 1:
RushHour

Présenté à:
Daniel Aloise

Réalisé par:
Cynthia Castillo 1878153

Date de remise:
18 février 2017

Implementations au code:

(Explication plus détaillée de chaque fonction et ses variables dans les commentaires du code)

- **Class State:**
  - **public State(State s, int c, int d)**
    - Il crée un nouvel état en considérant:
      - L'état précédent.
      - Le nombre de mouvements de l'état précédent et le mouvement fait (***n += s.n + 1***)
      - La valeur estimée pour éteindre l'état final (***f = n + estimee1()*** **ou** ***f = n + estimee2()***)
  - **public boolean success()**
    - Retourne si l'état est considérée final. (***pos[0] == 4***)
  - **public int estimee1()**
    - Returne la distance entre la voiture rouge et l'état final (***4 - pos[0]***)
  - **public int estimee2()**
    - Calcule la distance entre le voiture rouge et l'état final (***int distanceSortie = 4 - pos[0]***;)
    - Calcule le nombre de voitures qui bloque le chemin de la voiture rouge:
      - En premier, vérifient si la voiture se trouve déjà à l'état final (***pos[0] == 4***)
      - Sinon, vérifiant s'il y a des voitures dans le chemin:
        - **Horizontal:**
          -
```
if (rh.moveon[i] == 2)
    numVoitures++;
```
        - **Vertical:**
          -
```
if (pos[i] == 0 && rh.len[i] == 3)
    numVoitures++;

else if (pos[i] == 1 || pos[i] == 2)
    numVoitures++;
```
    - Retourne la distance et le nombre de voitures.

- **Class Rush Hour:**
  - **void initFree(State s)**
    - Remplit la matrice avec des " true's " (sans voitures)
    - Commence a changer par "false" où il y a des voitures:

- **Horizontal**

```
//Block the spaces where there is a horizontal cars
if (horiz[i] == true)
{
    for (int lenght = 0; lenght < len[i]; lenght++)
    {
        free[moveon[i]][s.pos[i]+lenght] = false;
    }
}
```

- **Vertical**

```
//Block the spaces where there is a vertical cars
else
{
    for (int lenght = 0; lenght < len[i]; lenght++)
    {
        free[s.pos[i]+lenght][moveon[i]] = false;
    }
}
```

- **public ArrayList<State> moves(State s)**
  - Remplit une liste avec tous les états qui peut s'atteindre dès le état donné en vérifiant les cases suivants:
    - **Voitures length 2**

```
if (mov < 4)
    mov += 2;
```

    - **Voitures length 3**

```
if (mov < 5 && len[i] == 3)
    mov++;
```

    - **Avancer en Horizontal**

```
//MOVING FORWARD (right)
if (free[moveon[i]][mov] == false)
    ;
else //if the next square is free
{
    newState = new State(s, i, 1);
    l.add(newState);
}
```

    - **Retourner en Horizontal**

```
//MOVING BACKWARDS (left)
mov = s.pos[i]; //initial position
if (mov > 0) //Making 'mov' a squar
    mov--;

//if the previous square is free sa
if (free[moveon[i]][mov] == false)
    continue;
else
{
    newState = new State(s, i, -1);
    l.add(newState);
}
```

- Le même mais adapté pour le case vertical.

- **public State solve(State s)**
  - Remplit la Queue avec les états non visitées:
    - Vérifier si l'état a été visitée, sinon, ajouter l'etat au visités et au Queue:

```
if (visited.contains(movements.get(i)))
    ;
else
{
    //add unexplored state to the lists
    Q.add(movements.get(i));
    visited.add(movements.get(i));
}
```

- **public State solveAstar(State s)**
  - Exactement le même que solve **mais** utilisant une heuristique et une priority queue.
- **void printSolution(State s)**
  - Imprime récursivement les voitures:
    - Cas base: imprimer le nombre de mouvements totales.

```
if (s == null)
    System.out.println(nbMoves-1);
```

    - Autrement:
      - Augmenter le nombre de mouvements et appeler la fonction avec un cas précédent:

```
nbMoves++;
printSolution(s.prev);
```

      - Quand les appels récursifs finalisent, imprimer les voitures:

```
if (s.prev != null)
{
    for (int i = 0; i < nbcars; i++)
    {
        if (s.pos[i] == s.prev.pos[i])
            ;
        else if (s.pos[i] > s.prev.pos[i])
        {
            if (horiz[i] == true)
                System.out.println("Voiture " + co
            else
                System.out.println("Voiture " + co
        }
        else if (s.pos[i] < s.prev.pos[i])
        {
            if (horiz[i] == true)
                System.out.println("Voiture " + co
            else
                System.out.println("Voiture " + co
        }
    }
}
```

➢ **Class Test:**

- **Main:**
  - Modifications d'empreinte pour montrer les résultats.
- **static void solve22()**
  - Des preuves de temps pour comparer l'exécution des algorithmes.
  - Ceci commentés pour pas interférer avec le résultat imprimé.

```
//TIME COMPARISONS!
/*
//   ****   NO HEURISTIC   ********
System.out.println("    Aucun heuristique");
long startTime = System.currentTimeMillis();
s = rh.solve(s);
long endTime = System.currentTimeMillis();
System.out.println("Temps en miliseconds: " + (endTime -
*/

/*
//  ****   USING ESTIMEE1 / ESTIMEE2   ****
//Changes for using estimee1 or estimee2 must be done in
System.out.println("    ESTIMEE2 heuristique");
long startTime = System.currentTimeMillis();
s = rh.solveAstar(s);
long endTime = System.currentTimeMillis();
System.out.println("Temps en miliseconds: " + (endTime -
*/
```

- Pour les tester, est indispensable commenter et enlever les commentaires de:
  - **solve22()**
    - Commenter:
```
s = rh.solveAstar(s);
rh.printSolution(s);
```
    - Enlever le commentaire soit
```
//   ****   NO HEURISTIC   ********
System.out.println("    Aucun heuristique");
long startTime = System.currentTimeMillis();
s = rh.solve(s);
long endTime = System.currentTimeMillis();
System.out.println("Temps en miliseconds: " +
*/
```
    ou
```
/*
//  ****   USING ESTIMEE1 / ESTIMEE2   ****
//Changes for using estimee1 or estimee2 must be
System.out.println("    ESTIMEE2 heuristique");
long startTime = System.currentTimeMillis();
s = rh.solveAstar(s);
long endTime = System.currentTimeMillis();
System.out.println("Temps en miliseconds: " + (e
*/
```
  - **RushHour → Solve(State s)**
    - Enlever les commentaires en (pour imprimer le nombre d'états visités):
```
if (Q.peek().success())//{
    //System.out.println("Nombre de etats visite
    return Q.remove();
//}
```
  - **RushHour → SolveAstar(State s)**
    - Enlever les commentaires en (pour imprimer le nombre d'états visités):
```
if (Q.peek().success())//{
    //System.out.println("Nombre de etats visitees: "
    return Q.remove();
//}
```

Résultats:

➢ **Sans utiliser un heuristique:**

```
run:
    Aucun heuristique
Nombre de etats visitees: 4968
Temps en miliseconds: 40
```

➢ **En comptant la distance entre la voiture rouge et l'état final:**

```
run:
    ESTIMEE1 heuristique
Nombre de etats visitees: 4454
Temps en miliseconds: 37
BUILD SUCCESSFUL (total time: 0 seconds)
```

➢ **En comptant la distance entre la voiture rouge et l'état final et le nombre des voitures qui bloquent le chemin:**

```
run:
    ESTIMEE2 heuristique
Nombre de etats visitees: 4080
Temps en miliseconds: 34
BUILD SUCCESSFUL (total time: 0 seconds)
```