

1 Introduction

Le but de ce projet est de produire un algorithme efficace pour résoudre le casse-tête "Parking Escape" également connu sous le nom "Rush Hour". Le jeu simule une congestion automobile dans un parking et l'objectif est de faire sortir une certaine voiture (goal/rouge) alignée avec la sortie. Les autres véhicules bloquent la sortie et ne peuvent se déplacer que le long de ses axes (horizontal ou vertical).

2 Solution proposé - Algorithme

Comme conseillé dans l'énoncé, on considère chaque état possible du parking comme un sommet dans un graphe, et chaque mouvement (déplacer une voiture de une position) un arc. Le problème devient alors très simple et il suffit de réaliser un BFS (Breadth First Search), qui visite (et construit) les noeuds en largeur. L'algorithme peut mettre fin à son exécution dès qu'une solution a été trouvée, puisque on est sûr d'avoir trouvé la plus courte et il ne faut pas vérifier des autres solutions.

Plus spécifiquement, on regarde tous les mouvements possibles dans un sommet/parking de départ s . Pour chaque mouvement, un autre Parking est créé, et il n'est gardé que s'il n'a pas été visité avant. Après avoir construit tous les états possibles avec ces mouvements on passe au premier et recommence, en regardant tous les mouvements possibles pour ce parking.

Finalement, si une solution a été trouvée, les étapes sont données dans l'ordre avec une méthode recursive, qui remonte jusqu'au sommet de départ grâce aux liens entre les arcs et les sommets.

Une autre approche serait de construire le graphe avec tous les cas possibles, les relier de la même façon (avec des arcs/mouvements - on aura des cycles) et utiliser l'algorithme de Dijkstra pour trouver le chemin le plus court entre un sommet initial et un sommet marqué comme solution. Dans ce cas, puisque tous les arcs ont le même poids, cette approche n'est pas choisie.

2.1 Complexité

Dans le pire des cas, cet algorithme a une complexité de $O(|S| + |A|)$, si S est le nombre de sommets et A est le nombre d'arcs, car tous les sommets et tous les arcs sont visités. L'espace de travail est le même.

Dans notre cas, on connaît pas tous les sommets, donc il est plus adéquat d'exprimer la complexité comme : $O(b^m)$ où b est le nombre maximum d'arcs sortants d'un sommet (aussi nombre de sommet "fils") et m la hauteur maximale.

2.2 Classes implémentées

- Escaper : Classe principale, qui contient l'algorithme de visite et instancie la class Parking au fur et à mesure. Cette classe s'occupe aussi du parsing du fichier et de communiquer la solution finale.
- Parking : État du parking dans un moment donné. Plusieurs objets de cette classe forment le graphe.
- Car : Cette classe représente une voiture (ses coordonnées, ID, voiture Goal ou pas, orientation, etc).
- Move : Contient toute l'information pour savoir quel a été le mouvement réalisé et quel était l'état du parking juste avant. Permet de revenir en arrière facilement. Ces mouvements sont les arcs du graphe.

2.3 Parsing fichier input

Le fichier input passé comme argument au programme est "parsé" avec deux méthodes :

1. `parseFile()` : Utilise la classe `Scanner` et `FileReader` pour l'IO et se sert de la librairie `java.util.regex` pour établir un modèle de coordonnées et faire du "Pattern Matching".
2. `extractCar()` : Renvoie un objet de type `Car` avec les bonnes coordonnées.

2.4 Contraintes

- Toutes les voitures doivent être de taille 2.
- Les fichier input ne doit pas comporter d'erreur.

2.5 Structures de données

Des listes de type `List<TYPE>`, `ArrayList()` sont utilisés dans la plus part des cas. Une file `Queue<TYPE>`, `LinkedList()` est aussi utilisé pour stocker les sommets a visiter dans l'algorithme de visite. En effet, celui ci se base sur cette file pour accéder aux sommets en largeur ("breadth first").

3 Améliorations

- Même si le paradigme OOP a été suivi, beaucoup de propriétés des objets restent publiques, pour faciliter l'accès aux autres classes, et ne pas gonfler les classes avec des accesseurs. Néanmoins, une meilleure encapsulation peut être achevée.
- On pourrait considérer le déplacement de plusieurs positions d'une voiture un mouvement. Cela réduit le nombre des "mouvements effectifs", mais la solution reste la même.
- Pour réduire le temps d'exécution (proportionel au nombre de voitures cette fois), la vérification d'emplacements libres pourrait se faire quand on est sûr que ce parking est nouveau et n'a jamais été visité. Dans l'algorithme proposé, tout est fait au même temps dans la même méthode (`updateGrid()`).
- Une nouvelle class `Truck` pourrait être créé pour gérer les cas des vehicules qui occupent 3 positions.

4 Code listing

```

1  import java.io.*;
   import java.util.*;
3  import java.util.regex.*;

5  public class Escaper {
      public static void main(String[] args) {
7      // Assuming no error in argument number
      System.out.println("\nInput_file_given :_" + args[0]);
9      Parking myPark = parseFile(args[0]); // Get parking
      bfs(myPark);                          // Solve parking
11     }

13     public static Parking parseFile(String file) {
        Scanner in = null;
15         try {
            // Cheking if file is OK to read from
17             in = new Scanner((new FileReader(file)));
        } catch (IOException e) {
19             System.out.println("Wrong_input_file :_" + e.getMessage());
        }
21         // \A in the beginning of the file. Interprets the rest as one big string.
        String text = in.useDelimiter("\\A").next();
23         Pattern carPattern = Pattern.compile("\\Q[(\\E[0-4],[0-4]\\Q),_(\\E[0-4],[0-4]\\Q)]\\E");
        Matcher matcher = carPattern.matcher(text);

25         List<Car> carList = new ArrayList();
27         boolean goal = true;
        int carCount = 0;
29         while (matcher.find()) {
            // Goal car first. Should be set up like this in input.txt
31             Car tempCar = extractCar(matcher.group(), goal, carCount);
            carList.add(tempCar);
33             goal = false;
            ++carCount;
35         }
        Parking myPark = new Parking(carList);
37         return myPark;
    }

39     public static Car extractCar(String ugly, boolean goal, int carCount) {
41         String carId;
        int[] coords = new int[4];
43         int index = 0;
        for (int i = 0; i < ugly.length(); i++){
45             char c = ugly.charAt(i);
            if (Character.isDigit(c)) {
47                 coords[index] = Character.getNumericValue(c);
                index++;
49             }
        }
    }

```

```
51     if (carCount == 0) {
52         carId = "GG";
53     } else {
54         carId = "c" + carCount;
55     }
56
57     Car car = new Car(coords[0], coords[1], coords[2], coords[3], carId, goal);
58     return car;
59 }
60
61 public static void bfs(Parking park) {
62     Queue<Parking> toCheck = new LinkedList();
63     List<String[][]> visited = new ArrayList();
64
65     Parking treating = park;
66     visited.add(treating.grid);
67     toCheck.add(treating);
68
69     while(toCheck.size() != 0) {
70         treating = toCheck.remove();
71         for (int i = 0; i < treating.moves.size(); i++) {
72             Move nextMove = treating.moves.get(i);
73             Parking newParking = new Parking(nextMove);
74             if (isInList(visited, newParking.grid) == false) {
75                 if (checkSolved(newParking.grid)) {
76                     followPath(newParking, 0);
77                     System.out.println("\nDone.\n");
78                     System.exit(0);
79                 }
80                 visited.add(newParking.grid);
81                 toCheck.add(newParking);
82             }
83         }
84     }
85     noSol(treating);
86 }
87
88 public static boolean isInList(List<String[][]> list, String[][] candidate) {
89     // Comparing arrays, since the methods .equals() and
90     // .contains() do not work well with arrays
91     for (String[][] array : list) {
92         if (Arrays.deepEquals(array, candidate)) {
93             return true;
94         }
95     }
96     return false;
97 }
98
99 public static boolean checkSolved(String[][] grid) {
100     // Condition for the puzzle to be solved. Can be
101     // anything... Testing the string in the exit case is simple
102     return grid[Parking.exitX][Parking.exitY] == null ? false :
103         grid[Parking.exitX][Parking.exitY].equals("GG");
104 }
```

```
105     }
106
107     public static void noSol(Parking block) {
108         System.out.println("No_solution_found!_Have_a_look_at_the_situation");
109         block.printGrid();
110     }
111
112     public static void followPath(Parking solution, int step) {
113         // Recursive function that goes all the way up to the starting
114         // point and then indicates the moves to do to reach the
115         // solution.
116         Move movement = solution.comingFrom;
117         if (movement != null) {
118             // If not in the initial state, keep going up
119             followPath(movement.predParking, ++step);
120             System.out.print("étape_suivante :_");
121             Car toMove = movement.carList.get(movement.index);
122             String id = toMove.carId;
123
124             System.out.println("déplacer_voiture_" + id);
125             System.out.print(movement.predParking.carList.get(movement.index));
126             System.out.print("_->_");
127             System.out.println(toMove);
128
129             System.out.println("Le_Parking_doit_se_trouver_dans_cet_état :");
130             solution.printGrid();
131         } else {
132             // Printing some info at the beginning about the first
133             // state parking
134             System.out.println(solution);
135             System.out.println("Une_facon_de_sortir_du_Parking_en_" + step + "_
136             mouvements_a_été_trouvée");
137         }
138     }
139 }
```

Escaper.java

```
1 import java.util.*;
3
4 public class Parking {
5     // Parking properties
6     static int SIZE = 5;
7     public static int exitX = 2;
8     public static int exitY = 4;
9     List<Car> carList = new ArrayList();
10    public List<Move> moves = new ArrayList();
11    public String[][] grid = new String[SIZE][SIZE];
12    public Move comingFrom = null;
13
14    public Parking (List<Car> carList) {
15        // Default ctor
16        this.carList = carList;
17        updateGrid();
18    }
19
20    public Parking (Move move) {
21        // Overloaded ctor if parking changes state
22        this.comingFrom = move;
23        this.carList = move.carList;
24        Car toChange = carList.get(move.index);
25        if (move.axis.equals("y")) {
26            toChange.y1 += move.inc;
27            toChange.y2 += move.inc;
28        } else {
29            toChange.x1 += move.inc;
30            toChange.x2 += move.inc;
31        }
32        updateGrid();
33    }
34
35    public void addCar(Car car) {
36        // Fill carList. No need if carList is public
37        carList.add(car);
38    }
39
40    public String toString() {
41        // Basic info to begin the program with.
42        String rep = "Le_parking_a_une_dimension_" + SIZE + "_fois_" + SIZE + "\n";
43        rep = rep.concat("Il_contient_1_Goal_car_et_" + (carList.size() - 1) + "_autres_");
44        rep = rep.concat("voitures\n");
45        for (int i = 0; i < carList.size(); i++) {
46            Car car = carList.get(i);
47            if (i == 0) {
48                rep = rep.concat("La_voiture_Goal_se_trouve_en_position :");
49            } else {
50                rep = rep.concat("La_voiture_" + i + "_se_trouve_en_position :");
51            }
52            rep = rep.concat(car.toString() + "\n");
53        }
54    }
55 }
```

```
53     }
54     return rep;
55 }

57 public void printGrid() {
58     // Sends out the grid.
59     printGridPlus();
60     for (int i = 0; i < SIZE; i++) {
61         System.out.print("|");
62         for (int j = 0; j < SIZE; j++) {
63             if (j != 0) {
64                 System.out.print("_");
65             }
66             if (grid[i][j] != null) {
67                 System.out.print("_" + grid[i][j]);
68             } else {
69                 System.out.print("___");
70             }
71         }
72         if (i != exitX) {
73             System.out.print("|");
74         }
75         if (i != SIZE - 1) {
76             printGridNextLine();
77         }
78     }
79     printGridPlus();
80     System.out.println("");
81 }

83 public void printGridPlus() {
84     // Helper method
85     System.out.print("\n+");
86     for (int i = 0; i < SIZE; i++) {
87         System.out.print("----+");
88     }
89     System.out.println("");
90 }

92

94 public void printGridNextLine() {
95     // Helper method
96     System.out.print("\n+");
97     for (int k = 0; k < SIZE; k++) {
98         System.out.print("___+");
99     }
100     System.out.println("");
101 }

103 public List<Car> copyList() {
104     // Deep copy of carList
105     List<Car> newList = new ArrayList();
106     for (Car c : carList) {
```

```
107         newList.add(new Car(c.x1, c.y1, c.x2, c.y2, c.carId, c.goal));
108     }
109     return newList;
110 }
111 public void updateGrid() {
112     // Redraw every car in the grid and check for possible moves
113     // where there are whitespaces
114     for (int i = 0; i < carList.size(); i++) {
115         Car car = carList.get(i);
116         grid[car.x1][car.y1] = car.carId;
117         grid[car.x2][car.y2] = car.carId;
118     }
119     for (int i = 0; i < carList.size(); i++) {
120         Car car = carList.get(i);
121         if (car.horizontal) {
122             if (inboundsAndFree(car.x1, car.y1 - 1)) {
123                 moves.add(new Move(copyList(), i, "y", -1, this));
124             }
125             if (inboundsAndFree(car.x2, car.y2 + 1)) {
126                 moves.add(new Move(copyList(), i, "y", 1, this));
127             }
128         } else {
129             if (inboundsAndFree(car.x1 - 1, car.y1)) {
130                 moves.add(new Move(copyList(), i, "x", -1, this));
131             }
132             if (inboundsAndFree(car.x2 + 1, car.y2)) {
133                 moves.add(new Move(copyList(), i, "x", 1, this));
134             }
135         }
136     }
137 }
138
139 public boolean inboundsAndFree(int x, int y) {
140     // Helper method to check if a case (x, y) is within the grid
141     // and there is nothing on it.
142     if ((x >= 0 && x < SIZE) &&
143         (y >= 0 && y < SIZE)) {
144         if (grid[x][y] == null) {
145             return true;
146         }
147     }
148     return false;
149 }
150 }
151 }
```

Parking.java


```
1 public class Car {
    public String carId;
3    public int x1, y1, x2, y2;
    public boolean horizontal = false;
5    public boolean goal = false;

7    public Car (int x1, int y1, int x2, int y2, String carId, boolean goal) {
        this.goal = goal;
9        this.carId = carId;

11       this.x1 = x1;
        this.y1 = y1;
13       this.x2 = x2;
        this.y2 = y2;
15       if (x1 == x2) {
            this.horizontal = true;
17             if (y2 < y1) {
                this.y2 = y1;
19                 this.y1 = y2;
            }
21         } else if (x2 < x1) {
            this.x2 = x1;
23             this.x1 = x2;
        }
25     }

27
29     public boolean isGoal() {
        return goal;
    }

31
33     public String toString() {
        // Coords
        return String.format("[(%d,%d),_(%d,%d)]", this.x1, this.y1, this.x2, this.y2);
35     }
}
```

Car.java

```
import java.util.*;

2
4 public class Move {
    public List<Car> carList;
    public Parking predParking;
    public int index;
    public String axis;
    public int inc;

    public Move (List<Car> carList, int index, String axis, int inc, Parking predParking
    ) {
        this.carList = carList;
        this.index = index;
        this.axis = axis;
        this.inc = inc;
        this.predParking = predParking;
    }
18 }
```

Move.java