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 vehicules bloquent la sortie et ne peuvent se déplacer que le long de ses axes (horizontal ou vertical).

2 Solution proposée - 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 a son exécution dès qu'une solution a été trouvée, puisque on est sûr d'avoir trouvé la plus courte et il ne faut pas verifier 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. Aprés avoir construit tous les états possibles avec ces mouvements on passe au premier et recommence, en regardant tous le 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 grace aux liens entre les arcs et les sommets.

Une autre approche serait de construire un graphe non orienté 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'exprimmer la complexité comme : $O(b^m)$ où b est le nombre maximum d'arcs sortants d'un sommet (aussi nombre de sommets "fils") et m la hauteur maximale.

2.2 Classes implementées

- `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 realisé 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 et les coordonnées doivent être au format `[(x1,y1), (x2,y2)]`.
— La voiture Goal doit se trouver en première position dans le fichier input.

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 proprietés des objets restent publiques, pour faciliter l'accés 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 reduit le nombre des "mouvements effectifs", mais la solution reste la même.
— Pour reduire 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

```java
import java.io.*;
import java.util.*;
import java.util.regex.*;

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

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

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

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

```
51          if (carCount == 0) {
                carId = "GG";
53          } else {
                carId = "c" + carCount;
55          }

57          Car car = new Car(coords[0], coords[1], coords[2], coords[3], carId, goal);
            return car;
59      }

61      public static void bfs(Parking park) {
            Queue<Parking> toCheck = new LinkedList();
63          List<String[][]> visited = new ArrayList();

65          Parking treating = park;
            visited.add(treating.grid);
67          toCheck.add(treating);

69          while(toCheck.size() != 0) {
                treating = toCheck.remove();
71              for (int i = 0; i < treating.moves.size(); i++) {
                    Move nextMove = treating.moves.get(i);
73                  Parking newParking = new Parking(nextMove);
                    if (isInList(visited, newParking.grid) == false) {
75                      if (checkSolved(newParking.grid)) {
                            followPath(newParking, 0);
77                          System.out.println("\nDone.\n");
                            System.exit(0);
79                      }
                        visited.add(newParking.grid);
81                      toCheck.add(newParking);
                    }
83              }
            }
85          noSol(treating);
        }
87
        public static boolean isInList(List<String[][]> list, String[][] candidate) {
89          // Comparing arrays, since the methods .equals() and
            // .contains() do not work well with arrays
91          for (String[][] array  : list) {
                if (Arrays.deepEquals(array, candidate)) {
93                  return true;
                }
95          }
            return false;
97      }

99      public static boolean checkSolved(String[][] grid) {
            // Condition for the puzzle to be solved. Can be
101         // anything... Testing the string in the exit case is simple
            return grid[Parking.exitX][Parking.exitY] == null ? false  :
103             grid[Parking.exitX][Parking.exitY].equals("GG");
```

```
    }

    public static void noSol(Parking block) {
        System.out.println("No solution found ! Have a look at the situation");
        block.printGrid();
    }

    public static void followPath(Parking solution, int step) {
        // Recursive function that goes all the way up to the starting
        // point and then indicates the moves to do to reach the
        // solution.
        Move movement = solution.comingFrom;
        if (movement != null) {
            // If not in the initial state, keep going up
            followPath(movement.predParking, ++step);
            System.out.print("étape suivante : ");
            Car toMove = movement.carList.get(movement.index);
            String id = toMove.carId;

            System.out.println("déplacer voiture " + id);
            System.out.print(movement.predParking.carList.get(movement.index));
            System.out.print(" -> ");
            System.out.println(toMove);

            System.out.println("Le Parking doit se trouver dans cet état :");
            solution.printGrid();
        } else {
            // Printing some info at the beginning about the first
            // state parking
            System.out.println(solution);
            System.out.println("Une facon de sortir du Parking en " + step + " mouvements a été trouvée");
        }
    }
}
```

Escaper.java

```java
import java.util.*;


public class Parking {
    // Parking properties
    static int SIZE = 5;
    public static int exitX = 2;
    public static int exitY = 4;
    List<Car> carList;
    public List<Move> moves = new ArrayList();
    public String[][] grid = new String[SIZE][SIZE];
    public Move comingFrom = null;

    public Parking (List<Car> carList) {
        // Default ctor
        this.carList = carList;
        updateGrid();
    }

    public Parking (Move move) {
        // Overloaded ctor if parking changes state
        this.comingFrom = move;
        this.carList = move.carList;
        Car toChange = carList.get(move.index);
        if (move.axis.equals("y")) {
            toChange.y1 += move.inc;
            toChange.y2 += move.inc;
        } else {
            toChange.x1 += move.inc;
            toChange.x2 += move.inc;
        }
        updateGrid();
    }

    public void addCar(Car car) {
        // Fill carList. No need if carList is public
        carList.add(car);
    }


    public String toString() {
        // Basic info to begin the program with.
        String rep = "Le parking a une dimension " + SIZE + " fois " + SIZE + "\n";
        rep = rep.concat("Il contient 1 Goal car et " + (carList.size() - 1) + " autres voitures\n");
        for (int i = 0; i < carList.size(); i++) {
            Car car = carList.get(i);
            if (i == 0) {
                rep = rep.concat("La voiture Goal se trouve en position : ");
            } else {
                rep = rep.concat("La voiture " + i + " se trouve en position : ");
            }
            rep = rep.concat(car.toString() + "\n");
```

```
53          }
            return rep;
55      }

57      public void printGrid() {
            // Sends out the grid.
59          printGridPlus();
            for (int i = 0; i < SIZE; i++) {
61              System.out.print("|");
                for (int j = 0; j < SIZE; j++) {
63                  if (j != 0) {
                        System.out.print("␣");
65                  }
                    if (grid[i][j] != null) {
67                      System.out.print("␣" + grid[i][j]);
                    } else {
69                      System.out.print("␣␣␣");
                    }
71              }
                if (i != exitX) {
73                  System.out.print("|");
                }
75              if (i != SIZE - 1) {
                    printGridNextLine();
77              }
            }
79          printGridPlus();
            System.out.println("");
81      }

83      public void printGridPlus() {
            // Helper method
85          System.out.print("\n+");
            for (int i = 0; i < SIZE; i++) {
87              System.out.print("---+");
            }
89          System.out.println("");
        }
91

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

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

```
            newList.add(new Car(c.x1, c.y1, c.x2, c.y2, c.carId, c.goal));
        }
        return newList;
    }

    public void updateGrid() {
        // Redraw every car in the grid and check for possible moves
        // where there are whitespaces
        for (int i = 0; i < carList.size(); i++) {
            Car car = carList.get(i);
            grid[car.x1][car.y1] = car.carId;
            grid[car.x2][car.y2] = car.carId;
        }
        for (int i = 0; i < carList.size(); i++) {
            Car car = carList.get(i);
            if (car.horizontal) {
                if (inboundsAndFree(car.x1, car.y1 - 1)) {
                    moves.add(new Move(copyList(), i, "y", -1, this));
                }
                if (inboundsAndFree(car.x2, car.y2 + 1)) {
                    moves.add(new Move(copyList(), i, "y", 1, this));
                }
            } else {
                if (inboundsAndFree(car.x1 - 1, car.y1)) {
                    moves.add(new Move(copyList(), i, "x", -1, this));
                }
                if (inboundsAndFree(car.x2 + 1, car.y2)) {
                    moves.add(new Move(copyList(), i, "x", 1, this));
                }
            }
        }
    }


    public boolean inboundsAndFree(int x, int y) {
        // Helper method to check if a case (x, y) is within the grid
        // and there is nothing on it.
        if ((x >= 0 && x < SIZE) &&
            (y >= 0 && y < SIZE)) {
            if (grid[x][y] == null) {
                return true;
            }
        }
        return false;
    }
}
```

Parking.java

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

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

        this.x1 = x1;
        this.y1 = y1;
        this.x2 = x2;
        this.y2 = y2;
        if (x1 == x2) {
            this.horizontal = true;
            if (y2 < y1) {
                this.y2 = y1;
                this.y1 = y2;
            }
        } else if (x2 < x1) {
            this.x2 = x1;
            this.x1 = x2;
        }
    }


    public boolean isGoal() {
        return goal;
    }

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

Car.java

```java
import java.util.*;


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;
    }
}
```

Move.java