Aufgabe 1: Schiebeparkplatz

Team-ID: 00070

Team: Gaußgamz

Bearbeiter/-innen dieser Aufgabe:
Emil Krieger, Sinan Ayhan

19. Oktober 2021

Inhaltsverzeichnis

Lösungsidee

Als Lösungsidee haben wir uns eine zweiseitige Bruteforce-Methode überlegt, welche durch Ausprobieren die kürzeste Lösung für jedes Auto findet.

Umsetzung

Beim Start des Programms wird die Textdatei eingelesen. Die einzelnen Zeilen werden verarbeitet und in zwei Arrays abgespeichert.

Dabei dient uns ein Array für die Abspeicherung der auszuparkenden Autos und das Andere für die Abspeicherung der querstehenden Autos.

Im Array mit den querstehenden Autos werden leere Plätze mit „null“ und besetzte Plätze mit dem jeweiligen Buchstaben des Autos angegeben.

Der Kern des Programms ist die Methode „autosAusparken“. Diese ruft für alle auszuparkenden Autos die „selectShorterPath“-Methode, gefolgt von der „resetParallelCars“-Methode auf und gibt zuletzt das Ergebnis in der Konsole aus. Bevor die Ausführung startet, wird überprüft zu welchem Grad der Parkplatz überhaupt lösbar ist.

Definitionen:

Der Name „ParallelCars“ bezeichnet die querstehenden Autos. Der Name „StraightCar“ bezeichnet die auszuparkenden Autos.

Die „resetParallelCars“-Methode, wie der Name schon sagt, setzt die querstehenden Autos zurück. Sie löscht erst alle Werte des Arrays für die querstehenden Autos, und füllt sie anschließend wieder mit den Werten aus der Textdatei.

Die „selectShorterPath- Methode überprüft zuerst, ob hinter dem auszuparkenden Auto ein querstehendes Auto steht, was das Ausparken verhindern würde. Falls es kein querstehendes Auto gibt, kann problemlos ausgeparkt werden. Gibt es jedoch eine Blockade, wird versucht das Auto in beide Richtungen so weit zu verschieben, dass das Auto ausparken kann. Danach wird entschieden, welche Richtung weniger Züge benötigt. Mit Hilfe der Methode „calcSolution“ wird in einem Array die Züge für beide Richtungen eingetragen. Über die globale Variable „iterations“, welche speichert wie oft das jeweilige Auto nach rechts oder links verschoben wurde, wird dann entschieden, welche der beiden Lösungen weniger Züge benötigt und diese dann als finale Lösung gespeichert.

Für die Verschiebung der querstehenden Autos ist die Methode „moveParallelCar“ zuständig. Zuerst wird getestet, ob das Auto, welches verschoben werden soll, von einem anderen Auto oder der Wand blockiert wird. Falls es keine Blockade gibt, kann das Auto einfach verschoben werden. Existiert eine Blockade, werden alle im Weg stehenden Autos mittels Rekursion verschoben.

Die Änderung auf dem Array wird von der Methode: “calcMovement“ ausgeführt. Diese nimmt als Parameter die Distanz, um welche das Auto verschoben werden soll, die Richtung und den Index. Nach einer logischen Verknüpfung dieser Parameter werden die entsprechenden Indizes mit entweder dem Namen des Autos oder „null“ überschrieben.

Hilfsfunktionen:

„calcSecondChar“-Methode: Gibt den zweiten Buchstaben eines querstehenden Autos zurück.

„calcFreeSpace“-Methode: Gibt als Array den Platz zurück, welches ein querstehendes Auto zur rechten und linken Seite hat.

„noArrayBoundaries“-Methode: Testet, ob der Versuch ein Auto zu verschieben, außerhalb der Array-Grenzen enden würde.

„noCarBoundaries“-Methode: Identisch zur „noArrayBoundaries“-Methode, jedoch werden hier auch andere Autos berücksichtigt.

Beispiele

Hier ist ein Beispiel für einen Parkplatz, welcher nicht lösbar ist, da alle Plätze belegt sind.

```
Searching solution of: "./examples/parkplatz7.txt"

straight cars:  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
parallel cars:  A A C C E E G G I I K K M M O O Q Q S S U U W W Y Y

Note:
Based on the composition of the parallel cars, no Solution for the straight cars exist
```

Beispiel für einen Parkplatz, welcher nur zur Hälfte lösbar ist.

```
straight cars:  A B C D E F G H I J K L M N O P Q R S T U V W X Y
parallel cars:  A A C C E E G G I I K K M M O O Q Q S S U U W W #

Note:
Based on the composition of the parallel cars, not all of the straight cars have a Solution


Solution:

A: W 1 right, U 1 right, S 1 right, Q 1 right, O 1 right, M 1 right, K 1 right, I 1 right, G 1 right, E 1 right, C 1 right, A 1 right
B: No solution for this car
C: W 1 right, U 1 right, S 1 right, Q 1 right, O 1 right, M 1 right, K 1 right, I 1 right, G 1 right, E 1 right, C 1 right
D: No solution for this car
E: W 1 right, U 1 right, S 1 right, Q 1 right, O 1 right, M 1 right, K 1 right, I 1 right, G 1 right, E 1 right
F: No solution for this car
G: W 1 right, U 1 right, S 1 right, Q 1 right, O 1 right, M 1 right, K 1 right, I 1 right, G 1 right
H: No solution for this car
I: W 1 right, U 1 right, S 1 right, Q 1 right, O 1 right, M 1 right, K 1 right, I 1 right
J: No solution for this car
K: W 1 right, U 1 right, S 1 right, Q 1 right, O 1 right, M 1 right, K 1 right
L: No solution for this car
M: W 1 right, U 1 right, S 1 right, Q 1 right, O 1 right, M 1 right
N: No solution for this car
O: W 1 right, U 1 right, S 1 right, Q 1 right, O 1 right
P: No solution for this car
Q: W 1 right, U 1 right, S 1 right, Q 1 right
R: No solution for this car
S: W 1 right, U 1 right, S 1 right
T: No solution for this car
U: W 1 right, U 1 right
V: No solution for this car
W: W 1 right
X: No solution for this car
Y:
```

Beispiel an Textdatei „parkplatz0.txt“

```
Searching solution of: "./examples/parkplatz0.txt"

straight cars:  A B C D E F G
parallel cars:  # # H H # I I


Solution:

A:
B:
C: H 1 right
D: H 1 left
E:
F: H 1 left, I 2 left
G: I 1 left
```

Quellcode

Methode „moveParallelCar“

```
int secondChar = parallelAuto.calcSecondChar(parallelIndex);
int freePath = parallelAuto.calcFreeSpace(parallelIndex)[direction];
boolean noArrayBoundaries = parallelAuto.noArrayBoundaries(parallelIndex, distance)[direction];
try {
    if ((noArrayBoundaries) && (freePath >= distance)) {
        calcMovement(parallelIndex, distance, direction);
        calcSolution(parallelIndex, distance, straightIndex, direction);
        iterations[direction] += distance;
        return true;
    } else if ((noArrayBoundaries) && (freePath < distance)) {
        int newParallelIndex = (parallelIndex - (2 - secondChar) - freePath) * (1 - direction)
                + (parallelIndex + secondChar + 1 + freePath) * direction;
        int newDistance = 1;
        if (distance == 2) {
            newDistance = 2;
            newDistance -= freePath;
        }
        if (!moveParallelCar(newParallelIndex, newDistance, straightIndex, direction)) {
            return false;
        }
        calcMovement(parallelIndex, distance, direction);
        calcSolution(parallelIndex, distance, straightIndex, direction);
        iterations[direction] += distance;
        return true;
    }
    return false;
} catch (Exception e) {
    iterations[direction] = 0;
    return false;
}
```

Einlesen der Textdatei

```
//Reading File
try{
    File file = new File(filePath);
    Scanner scanner = new Scanner(file);
    while(scanner.hasNext()){
        lines.add(scanner.nextLine());
    }
} catch (FileNotFoundException e){
    System.out.println("ERROR: Invalid filename");
    return;
}
```

Die „selectShorterPath“-Methode

```
iterations[1] = 0;
iterations[0] = 0;
selectSolution[0] = null;
selectSolution[1] = null;
int secondChar = parallelAuto.calcSecondChar(straightIndex);
if (parallel[straightIndex] == null) { // exit path is empty
    finalSolution[straightIndex] = straight[straightIndex] + ": ";
    return true;
} else {
    for (int i = 1; i >= 0; i--) {// for-loop to get both directions
        moveParallelCar(straightIndex, (1 + secondChar) * (1 - i) + (2 - secondChar) * i, straightIndex, i);
        resetParallelCars();
    }
    //Short Decision tree which decides, which of the two direction took the least steps to move the car out
    if (selectSolution[1] == null && selectSolution[0] == null) {
        return false;
    } else if (iterations[0] == 0 || selectSolution[0] == null) {
        finalSolution[straightIndex] = selectSolution[1];
    } else if (iterations[1] == 0 || selectSolution[1] == null) {
        finalSolution[straightIndex] = selectSolution[0];
    } else if (iterations[1] <= iterations[0]) {
        finalSolution[straightIndex] = selectSolution[1];
    } else if (iterations[1] > iterations[0]) {
        finalSolution[straightIndex] = selectSolution[0];
    }
    return true;
}
```

Methode „AutosAusparken“

```
//Checks if a soulution is possible
if (canSolve() == 0.5) {
    System.out.println(
            "\nNote:\nBased on the composition of the parallel cars, not all of the straight cars have a Solution");
    System.out.println("\n\n" + "Solution: \n");
} else if (canSolve() == 1) {
    System.out.println("\n\n" + "Solution: \n");
}
//Calls the solving-method on all cars
for (int straightIndex = 0; straightIndex < straight.length; straightIndex++) {
    if (selectShorterPath(straightIndex)) {
        System.out.println(finalSolution[straightIndex]);
    } else {
        System.out.println(straight[straightIndex] + ": No solution for this car");
    }
    resetParallelCars();
}
```