Aufgabe 1: Schiebeparkplatz

Teilnahme-ID: 62344

Manuel Frohn

30.09.2021

Inhalt

Lösungsansatz

In diesem Abschnitt werde ich Ihnen meinen Lösungsansatz erklären. Zu Beginn möchte ich aber erst einmal das Problem ein wenig genauer darstellen. Das Problem ist grundlegend die Frage danach, wie man eine Menge Autos W verschieben muss um ein, zu diesen Orthogonal stehendes Auto N_i auszuparken. Dabei ist jedes Auto eine Längeneinheit breit und zwei Längeneinheiten lang. Eine Lösung, für dieses Problem soll für alle Autos N ermittelt werden, wobei immer nur ein Auto gleichzeitig ausgeparkt werden soll, also die Ausparkprozedur des einen Autos N_i hat keinen Einfluss, auf ein anderes Auto N_i.

Mein Lösungsansatz ist es nun für jedes einzelne Auto N_i eine Ausparkprozedur (sofern dies nötig ist) zu berechnen, indem ich jeweils feststelle welches Auto W_i das auszuparkende Auto N_i blockiert und dann wie man dieses bewegen müsste um das Ausparken zu ermöglichen für diese Bewegungsmöglichkeiten prüfe ich, dann wieder ob sie blockiert (und möglich, also nicht außerhalb des Parkplatzes gehend) sind und wen ja nehme ich mir das Auto das diese Bewegung blockiert und schaue dann wieder wie ich dieses Bewegen müsste um den Weg frei zu machen diese Bewegung prüfe ich dann wieder, so wiederholt sich die Prozedur solange bis die benötigte Bewegung nicht blockiert ist, wodurch sich ein Lösungsweg gefunden hat, beziehungsweise wiederholt sich die Prozedur bis eine Bewegung gefunden wurde die nicht ausführbar ist, da das Fahrzeug hierzu den Parkplatz verlassen müsste. Hierbei wurde dann kein Lösungsweg gefunden. Normalerweise werden hierdurch immer zwei Lösungen gefunden, da das erste Auto W_i, dass das auszuparkende Auto N_i blockiert, sowohl nach rechts als auch nach links ausweichen könnte. Sollten also zwei mögliche Ausparkprozeduren möglich sein, so wähle ich den, bei dem weniger Autos verschoben werden müssen.

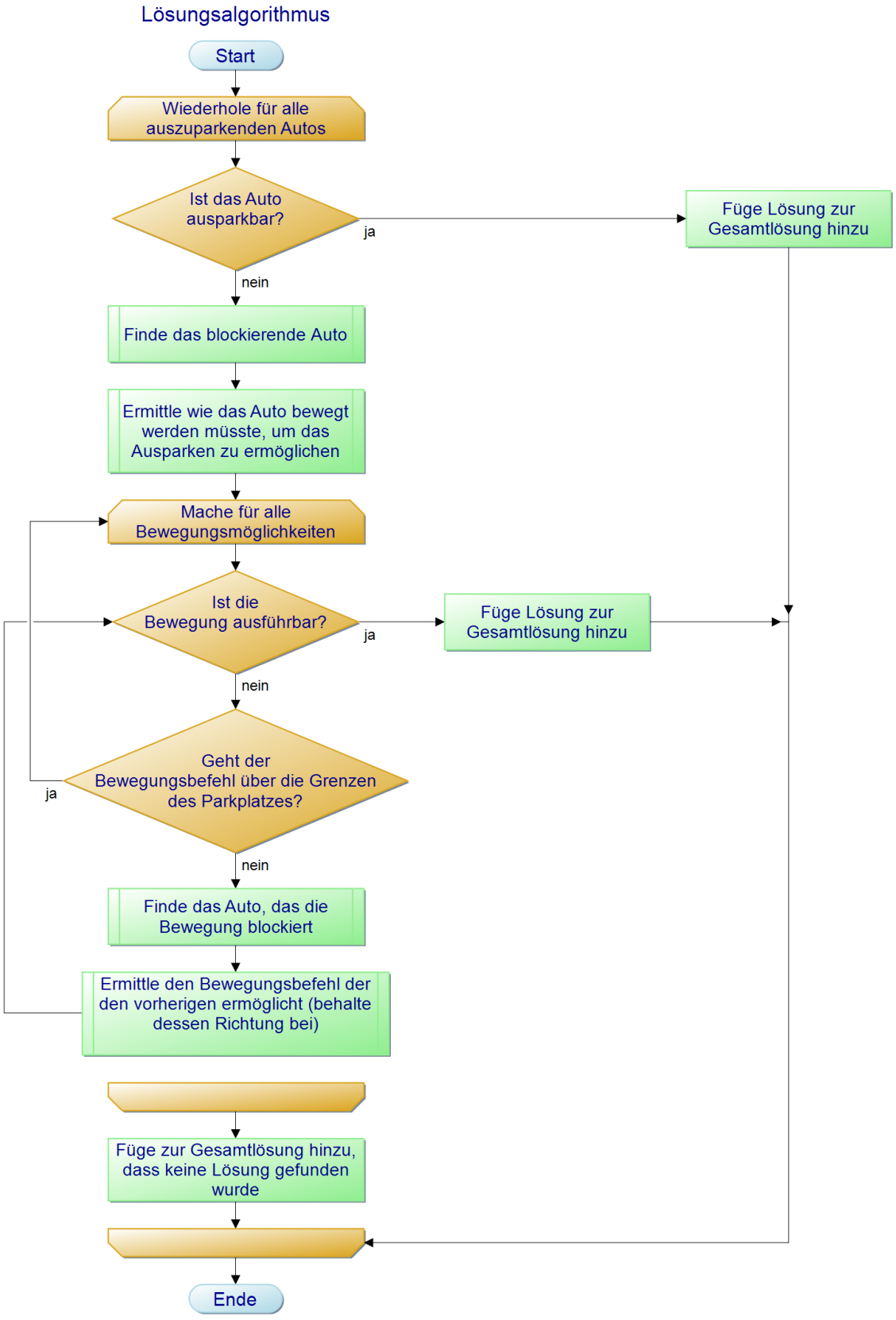

Abbildung 1: Flowchart des Lösungsalgorithmus

Umsetzung

Kommen wir nun dazu, wie der oben beschrieben Algorithmus im Programm umgesetzt wird, schauen wir uns hierzu erst einmal meine Modellierung des Problems im Code. Ich nutze hierzu vier verschiedene Klassen: Die *NormalCar*-Klasse, welche lediglich einen Namen und eine Position hat. Als diese werden die auszuparkenden Autos dargestellt. Die *WeirdCar*-Klasse, welche von ihrem Inhalt her identisch zur *NormalCar*-Klasse ist (der Positionswert gibt hier die erste Position von links aus, beginnend bei 0 an), die dazu dient die Autos darzustellen, die orthogonal zu den auszuparkenden Autos stehen, und dementsprechend verschoben werden müssen. Die dritte Klasse ist die des *MovementCommand*s, welche sowohl zur Speicherung des Verschiebungsprozesses insgesamt als auch zur Daten Speicherung während des Berechnungsprozesses dient. Hierzu werden in ihr das zu verschieben Auto, sowie der Vektor, der die Verschiebung beschreibt, gespeichert. Als letzte Klasse zur Modellierung des Problems verwende ich die *Parkinglot*-Klasse, welche den Parkplatz, auf dem sich das ganze geschehen abspielt repräsentiert. Diese Klasse wird mithilfe einer der Beispieldateien initialisiert. Dabei liest sie aus der Datei die Autos des Parkplatzes ein und speichert die Autos in ihrer jeweiligen Liste. Danach ist es möglich mit der *getSolution()*-Methode die Lösungen für alle Autos zu erhalten. Hierzu enthalt die Klasse noch einige Unter- und Hilfsmethoden. Des Weiteren gibt es auch noch eine Enumeration des Namens *ColisionType* diese dient aber lediglich dazu um die Art (ob auf den hinteren oder vorderen Teil des Autos) der ersten Kollision übersichtlicher festzuhalten und dann den ersten MovementComand dementsprechend erstellen zu können.

Da sie nun mit der Datenstruktur meines Programmes vertraut sind können wir uns nun mit der Umsetzung des Lösungsalgorithmus beschäftigen. Beginnen müssen wir bei der *getSolution()*-Methode diese gibt sobald die Berechnungen abgeschlossen sind die Lösung für den jeweiligen Parkplatz, in Form eines Strings zurück. Hierzu ruft die Methode iterativ, für alle im Parkplatz (in einer Liste) gespeicherten NormalCars die Überladung der *getSolution()*-Methode : die *getSolution(NormalCar normalCar)*-Methode auf und fügt deren Ergebnis einem Gesamtergebnis hinzu. Innerhalb dieser Methode wird die Einzellösung für das jeweils mitgegebene NormalCar berechnet. Dies wird bewerkstelligt, indem zuerst einmal geprüft wird ob das Auto überhaupt blockiert ist, durch das Aufrufen der *isColliding(NormalCar normalCar)*-Methode (Abbildung). Ist dem nicht so, dann wird bereits hier ein Ergebnis zurückgegeben, da es keine Verschiebungen benötigt. Ist das Auto aber blockiert, so bestimmt die Methode zunächst auf welche Weise das Auto blockiert wird, dazu ruft das Programm die *getColisionType(NormalCar normalCar)*-Methode auf, welche eine Enumeration zurück gibt, anhand welcher die zwei nötigen Bewegungsbefehle erstellt werden (front -2, 1; back: -1, 2(Negativwerte entsprechen hier einer Bewegung nach links)). Für beide dieser Befehle wird dann die *getSidePath(MovementCommand command, WeirdCar weirdCar, ArrayList<MovementCommand> path)*-Methode ausgeführt (für *WeirdCar weirdCar* wird das Auto mitgegeben, welches das zu bewegende NormalCar blockiert, und für *ArrayList<MovementCommand> path* eine leere ArrayList) und dessen Ergebnis (eine ArrayList mit MovementCommands, welche den auszuführenden Verschiebungsprozess

beschreibt) gespeichert. Danach wird in einer if-Verzweigung der bessere Pfad ausgewählt, in einen String umgewandelt und zuletzt zurückgegeben wird.

Nun müssen wir uns aber, um die vollständige Implementierung meines Algorithmus nachvollziehen zu können noch mit der *getSidePath(MovementCommand command, WeirdCar weirdCar, ArrayList<MovementCommand> path)*-Methode auseinandersetzen, in dieser nämlich finden die Berechnung der auszuführenden Bewegungen statt, die notwendig sind um das Auto, welches das Auszuparkende blockiert so bewegen zu können das es das auszuparkende Auto nichtmehr blockiert. Hierzu stellt die Methode zunächst fest, in welche Richtung der ihr mitgegeben Bewegungsbefehl gerichtet ist, durch eine if-Abfrage. Der Wert wird dann als Integer gespeichert (-1 für links, 1 für rechts). Danach wird, durch die *isCommandGoingOutOfBounds(MovementCommand command, WeirdCar weirdCar)*-Methode geprüft ob sich das Auto, auf das sich der mitgegebene MovementCommand bezieht, durch diesen aus dem Parkplatz rausbewegen würde ist dem so, wird *null* zurückgegeben. Sollte dem aber nicht so sein, dann wird durch die *isCommandColliding(command, weirdCar)*-Methode geprüft, ob das Auto beim Ausführen des MovementCommands auf ein anderes Auto auffahren würde. Ist dem nicht so wird der bisher errechnete Pfad zurück gegeben sonst wird, durch die *getColisionType(MovementCommand command, WeirdCar weirdCar)*-Methode die Art der potentiellen Kollision (1 für ein Auffahren hinten beziehungsweise forne und 2 für ein wortwörtliches Drauffahren) und das WeirdCar mit dem kollidiert werden würde festgestellt. Danach wird aus den vorher erhobenen Daten ein neuer MovementCommand generiert woraufhin ein Aufruf der hier beschriebenen Methode, nur jetzt mit dem neuen MovementCommand und dem als blockierend festgestellten Autos zurückgegeben.

Lösungen

```
Auto: A
Parke Aus

Auto: B
Parke Aus

Auto: C
Bewege Auto H um 2 nach links
Parke Aus

Auto: D
Bewege Auto H um 1 nach links
Parke Aus

Auto: E
Parke Aus

Auto: F
Bewege Auto H um 1 nach links
Bewege Auto I um 2 nach links
Parke Aus

Auto: G
Bewege Auto I um 1 nach links
Parke Aus
```

Abbildung 2: Lösung Beispiel 0

```
Auto: A
Parke Aus

Auto: B
Bewege Auto P um 1 nach rechts
Bewege Auto O um 1 nach rechts
Parke Aus

Auto: C
Bewege Auto O um 1 nach links
Parke Aus

Auto: D
Bewege Auto P um 1 nach rechts
Parke Aus

Auto: E
Bewege Auto O um 1 nach links
Bewege Auto P um 1 nach links
Parke Aus

Auto: F
Parke Aus

Auto: G
Bewege Auto Q um 1 nach rechts
Parke Aus

Auto: H
Bewege Auto Q um 1 nach links
Parke Aus

Auto: I
Parke Aus

Auto: J
Parke Aus

Auto: K
Bewege Auto R um 2 nach links
Parke Aus

Auto: L
Bewege Auto R um 1 nach links
Parke Aus

Auto: M
Parke Aus

Auto: N
Parke Aus
```

Abbildung 3: Lösung Beispiel 1

```
Auto: A
Parke Aus

Auto: B
Parke Aus

Auto: C
Bewege Auto O um 2 nach links
Parke Aus

Auto: D
Bewege Auto O um 1 nach links
Parke Aus

Auto: E
Parke Aus

Auto: F
Bewege Auto O um 1 nach links
Bewege Auto P um 2 nach links
Parke Aus

Auto: G
Bewege Auto P um 1 nach links
Parke Aus

Auto: H
Bewege Auto R um 1 nach rechts
Bewege Auto Q um 1 nach rechts
Parke Aus

Auto: I
Bewege Auto P um 1 nach links
Bewege Auto Q um 1 nach links
Parke Aus

Auto: J
Bewege Auto R um 1 nach rechts
Parke Aus

Auto: K
Bewege Auto P um 1 nach links
Bewege Auto Q um 1 nach links
Bewege Auto R um 1 nach links
Parke Aus

Auto: L
Parke Aus

Auto: M
Bewege Auto P um 1 nach links
Bewege Auto Q um 1 nach links
Bewege Auto R um 1 nach links
Bewege Auto S um 2 nach links
Parke Aus

Auto: N
Bewege Auto S um 1 nach links
Parke Aus
```

Abbildung 4: Lösung Beispiel 2

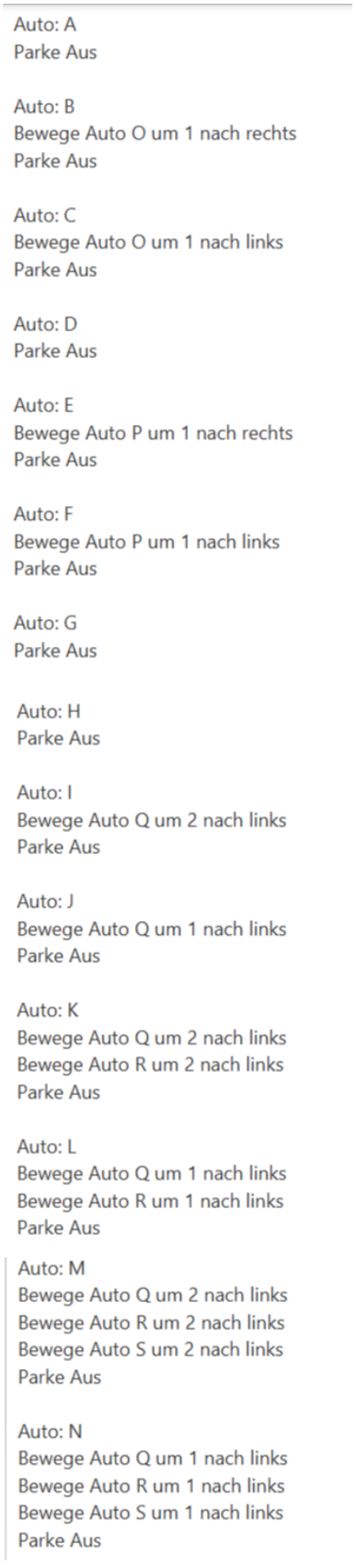

Abbildung 5: Lösung Beispiel 3

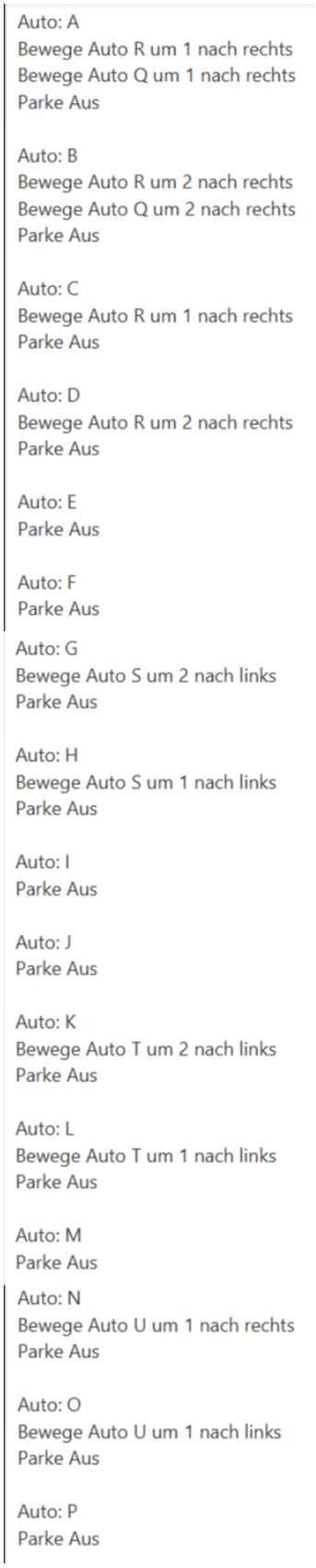

Abbildung 6: Lösung Beispiel 4

Abbildung 7: Lösung Beispiel 5

Code-Ausschnitte

```
public String getSolution(NormalCar normalCar)
{
    String carSolution = new String();
    carSolution += "Auto: " + normalCar.getName() + "\n";

    if(isColliding(normalCar))
    {
        MovementCommand initialMovementCommandLeft = null;
        MovementCommand initialMovementCommandRight = null;
        WeirdCar blockingCar = getBlockingCar(normalCar);

        switch(getColisionType(normalCar))
        {
        case front:
            initialMovementCommandLeft = new MovementCommand(blockingCar.getName(), -2);
            initialMovementCommandRight = new MovementCommand(blockingCar.getName(), 1);
            break;

        case back:
            initialMovementCommandLeft = new MovementCommand(blockingCar.getName(), -1);
            initialMovementCommandRight = new MovementCommand(blockingCar.getName(), 2);
            break;
        }

        ArrayList<MovementCommand> leftPath = getSidePath(initialMovementCommandLeft, blockingCar, new ArrayList<MovementCommand>());
        ArrayList<MovementCommand> rightPath = getSidePath(initialMovementCommandRight, blockingCar, new ArrayList<MovementCommand>());

        if(rightPath == null && leftPath == null)
        {
            carSolution += "Nicht ausparkbar\n";
        }
        else if(leftPath != null && (rightPath == null || leftPath.size() <= rightPath.size()))
        {
            for(int n = (leftPath.size() - 1); n > -1; n--)
            {
                carSolution += leftPath.get(n).toString();
                carSolution += "\n";
            }
            carSolution += "Parke Aus\n";
        }
        else if ( rightPath != null && (leftPath == null || rightPath.size() <= leftPath.size()))
        {
            for(int n = (rightPath.size() - 1); n > -1; n--)
            {
                carSolution += rightPath.get(n).toString();
                carSolution += "\n";
            }
            carSolution += "Parke Aus\n";
        }
    }
    else
    {
        carSolution += "Parke Aus\n";
    }

    return carSolution;
}
```

Abbildung 8: getSolution-Methode

```
private ArrayList<MovementCommand> getSidePath(MovementCommand command, WeirdCar weirdCar, ArrayList<MovementCommand> path)
{
    path.add(command);

    int movementDirection = 0;

    if(command.getMoveData() > 0)
    {
        movementDirection = 1;
    }
    else if(command.getMoveData() < 0)
    {
        movementDirection = -1;
    }

    if(isCommandGoingOutOfBounds(command, weirdCar))
    {
        return null;
    }
    else if (!isCommandColliding(command, weirdCar))
    {
        return path;
    }
    else
    {
        int movementStrenght = getColisionType(command, weirdCar);
        WeirdCar blockingCar = getBlockingCar(command, weirdCar);
        MovementCommand nextCommand = new MovementCommand(blockingCar.getName(), movementStrenght * movementDirection);

        return getSidePath(nextCommand, blockingCar, path);
    }
}
```

Abbildung 9: getSidePath-Methode