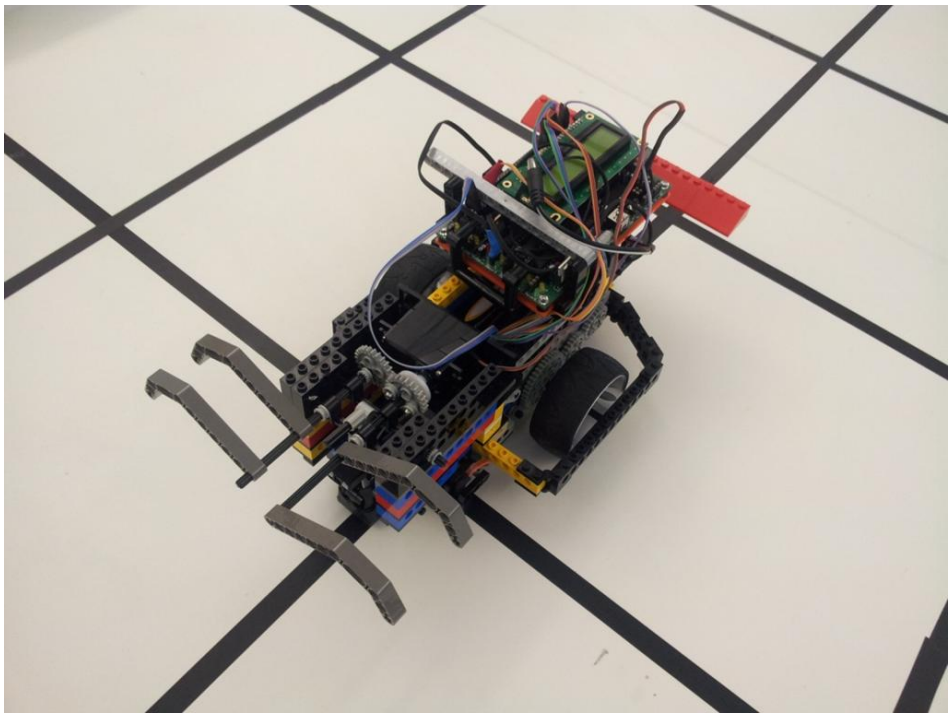
Fachhochschule Brandenburg
University of Applied Sciences

Dokumentation im Rahmen des Projekts - ROBI -

Wintersemester 2012/13



von Patrick Pohlmann,
Maximilian Orlowski
und Heiko Ruhm

Inhaltsverzeichnis

1 Hardware

1.1 Technisches Zubehör

1x Aksenboard

1x Akku

1x Servomotor

2x Motor

5x Optokoppler

1x Lichtsensor

1x Kontroll-LED

1.2 Verkabelung Aksenboard

Abbildung 1

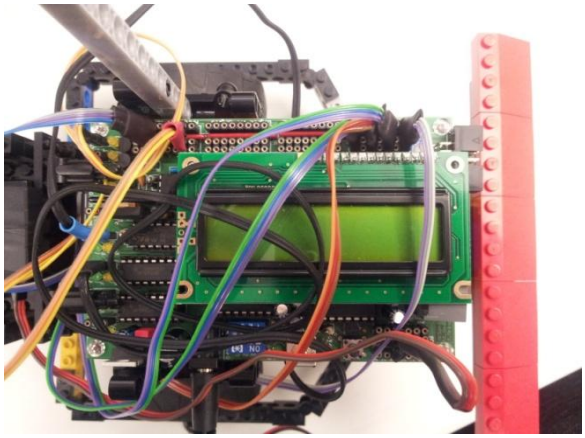


Abbildung 2

In Abbildung 1 sehen wir das reine Aksenboard.

Nun folgt die genaue Angabe der Verkabelung:

4x Optokoppler für die Streckenerkennung (An.: 08, 10, 12, 14)

1x Optokoppler für die "Personen"-erkennung (An.: 05)

1x Lichtsensor für das Startlichtsignal (An.: 01)

2x Motoren für den Antrieb (Motor: 2,3)

1x Servomotor für die Personenaufnahme (S: 0)

1x Kontroll-LED für die Überprüfung sämtlicher Prozesse (Led: 3)

1x Akku für Strom an den dafür vorgesehenen Anschluss (Abbildung 1 unten)

Auf Abbildung 2 ist die fertige Verkabelung zu sehen.

1.3 Aufbau Optokoppler

Die Optokoppler können zwischen hell und dunkel unterscheiden. Sie geben Werte zwischen 0 und 255 aus und diese kann man mühelos in den Code einfügen, um verschiedene Probleme zu bewerkstelligen. Da das "Spielfeld" mit schwarzen Linien auf weißen Hintergrund versehen ist, eignen sich die Optokoppler wunderbar dafür. Für den Aufbau der Optokoppler wurden 2x (3x3) 90° Winkelbausteine verwendet. Zwischen diesen wurde mithilfe von Klebeband der Optokoppler befestigt. Anschließend schiebt man die Konstruktion mit einem Abstandshalter auf eine Legoachse. Die fertige Konstruktion ist auf Abbildung 3 zu sehen.

Abbildung 3

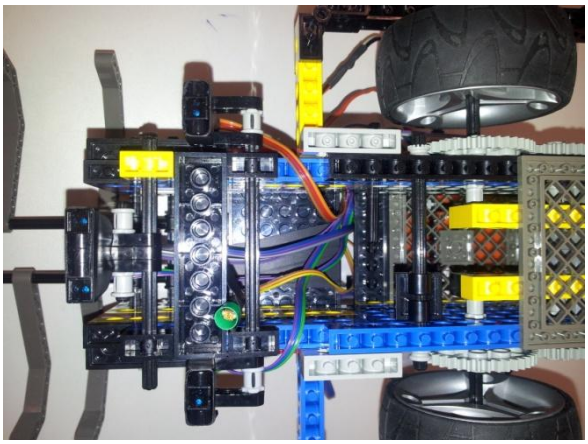


Abbildung 4

Auf Abbildung 4 sind die Positionen der vier Optokoppler zu sehen. Die vorderen beiden dienen zur Linienführung und die hinteren beiden, die etwas weiter außen sind, dienen zur Erkennung von Kreuzungen, damit ROBI abbiegen kann.

1.4 Funktion Greifarm

Auf Abbildung 5 ist die fertige Mechanik zu sehen. Der Servomotor ist mithilfe von Klebeband befestigt. Die erste Legoachse ist direkt an der Antriebswelle des Servomotors befestigt. Mithilfe von einem Zahnrad wird die zweite Legoachse endgegengesetzt angetrieben. Vorne werden die Greifarme raufgesteckt. Zur Stabilisierung wurden noch zwei Legoachsen mit Halterungen für die beiden darüber liegenden Antriebslegoachsen hinzugefügt, die quer darunterliegen. Damit der Greifarm weiß, wann er zugreifen soll, wird noch ein weiterer Optokoppler benötigt. Dieser befindet sich unter den Greifarmachsen in der Mitte (Abb. 6). Sobald die Person (der Ball) zu nah kommt, nimmt er eine Veränderung war und der Greifarm packt zu.

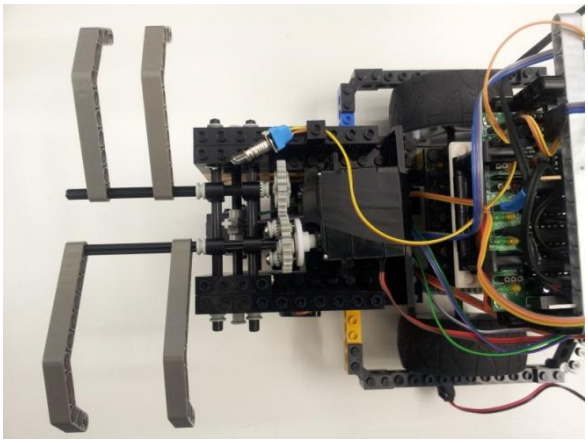


Abbildung 5

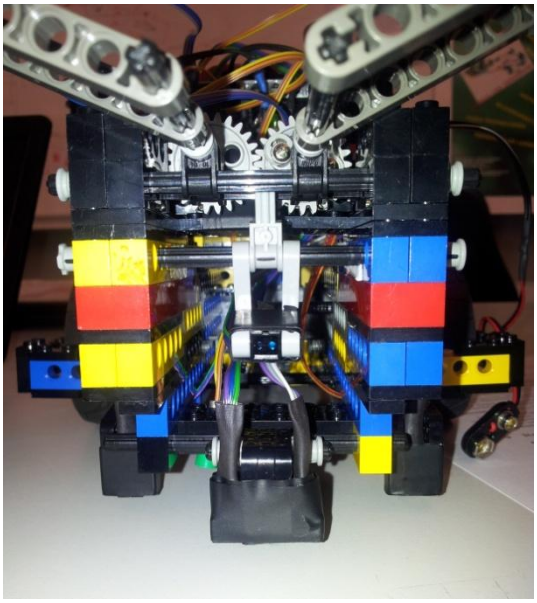


Abbildung 6

1.5 Grober Aufbau von ROBI

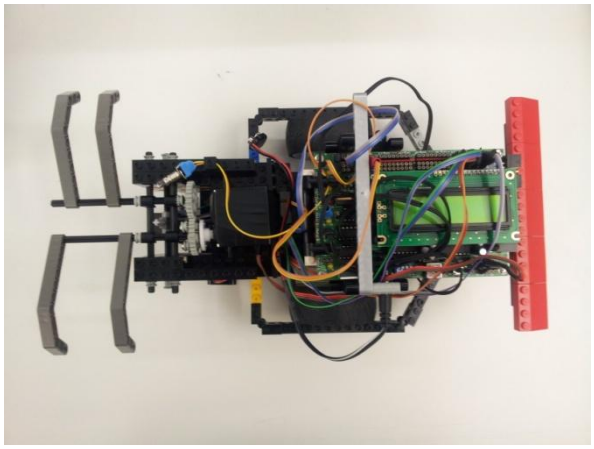


Abbildung 7

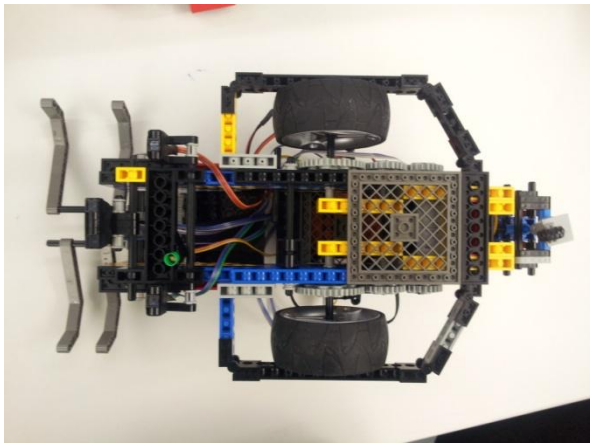


Abbildung 8

Auf Abbildung 7 ist ROBI von der Draufsicht zu sehen. Es ist deutlich der Greifarm zu erkennen, der sich an der Front befindet. Dahinter befindet sich die Mechanik für diesen. Unter der Mechanik befinden sich die fünf Optokoppler und der Lichtsensor für das Startsignal (Abb. 8). Der Abstand der äußeren beiden Optokopplern ist exakt der gleiche, wie der Abstand von dem Radachsenmittelpunkt zu den Rädern. Daraus resultiert eine perfekte 90° Kurve. Anschließend folgt der Rumpf, der aus zwei Rädern, den dazugehörigen Motoren, dem Akku (Abb. 9) und dem darüber liegenden Aksenboard besteht. Damit die Motoren das Gewicht bewegen können, wurde mithilfe von jeweils drei Zahnrädern eine Übersetzung von 1:5 geschaffen. Am Heck befindet sich lediglich ein kleines Rad für möglichst geringe Reibung beim Lenken und der Spoiler für das Gleichgewicht,

da er aufgrund des Greifarms sonst vorne überkippen würde.
Abbildung 10 zeigt, wie ROBI es geschafft hat eine Person aufzunehmen und zu befördern.

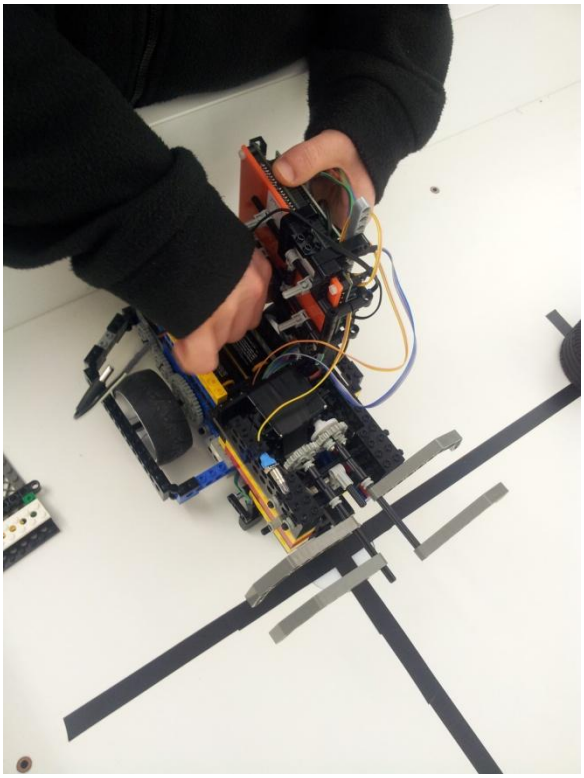


Abbildung 9

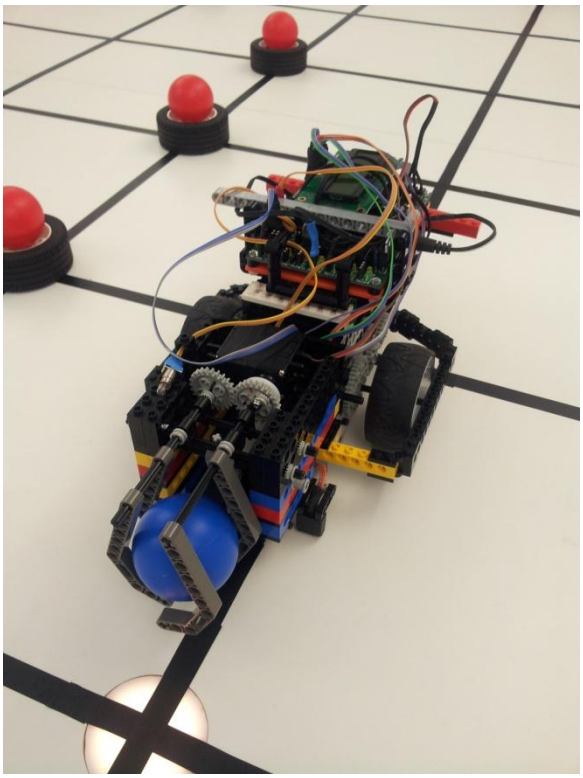


Abbildung 10

1.6 Probleme

Bei der Konstruktion des Greifarmes ergaben sich einige Probleme. Die erste Konstruktion war eine Art Gabel, die wie eine Schaufel eines Baggers fungierte (Abb. 11). Allerdings war somit der ganze Roboter zu lang und hat den Ball beim Einlenken schon herunter gehauen. Nachdem die komplette Konstruktion gekürzt wurde, hat er zwar die Kurve geschafft, aber die Chance den Ball aufzuheben war zu gering. So wurde alles verworfen und die aktuelle Konstruktion angebaut. Jetzt gab es diese Probleme nicht mehr.

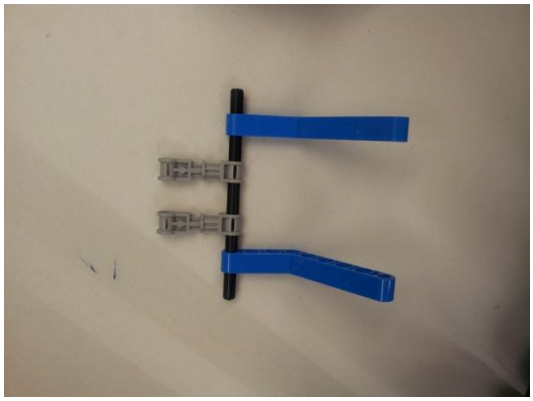


Abbildung 11

Die Länge von ROBI ist jedoch ein allgemeines Problem. Bis zuletzt hat die Gruppe es nicht geschafft, ihn optimal zu kürzen. Aus diesem Grund musste der Code dementsprechend angepasst werden. Es musste eine spezielle Funktion für das Lenken und Zurücksetzen eingepflegt werden.
Aber nicht nur die Länge ist ein Problem. Aufgrund der Größe ist auch das Gewicht beachtlich. So ergab sich ein höherer Akkuverbrauch und es hatte den Nachteil, dass ROBI je nach Akkustand anders gehandelt hat. Der Code musste je nach Akkustand immer wieder neu angepasst werden. Um dieses Problem zu beheben, hat es letztendlich an Zeit gefehlt. Ein Grund für die fehlende Zeit ist, dass die erste Version von ROBI noch größer und schwerer war, sodass alles verworfen wurde und ein neuer ROBI gebaut wurde. Ein weiterer Grund waren die defekten Servomotoren. Mittlerweile befindet sich der vierte Servomotor im System. Es war leider nicht immer gleich ersichtlich, ob der Servomotor defekt war oder nicht. Das hat viel Zeit gekostet.

2 Navigation

2.1 Funktionen

Hier folgt nun eine Auflistung der Funktionen die aus der Hauptroutine der Navigation ausgelagert wurden um eine Wiederverwendung zu ermöglichen. Die Hauptroutine wird in Abschnitt 2.2 gesondert behandelt.

2.1.1 void set_back()

Diese Funktion lässt nach Aufruf den Roboter solange rückwärtsfahren bis einer der beiden Optokoppler, die für die Kreuzungserkennung zuständig sind, auf Schwarz kommt. Dann wird der Roboter scharf gebremst, sodass er möglichst exakt auf der Kreuzung steht.

2.1.2 void drive_over()

Diese Funktion wird aufgerufen, wenn der Roboter eine Kreuzung einfach überfahren soll. Sie lässt den Roboter solange weiter geradeaus fahre, bis er von der Kreuzung wieder runter ist. Überprüft wird dies indem abgefragt wird ob einer der beiden Optokoppler der Kreuzungserkennung noch auf Schwarz ist.

2.1.3 void turn_right()

Diese Funktion dreht den Roboter um 90° nach rechts. Dafür dreht sich das linke Rad mit moderater Geschwindigkeit vorwärts, während das rechte Rad leicht gegensteuert um sich nicht mitzudrehen. Das rechte Rad ist dementsprechend auch der Drehpunkt der Drehung. Zu Beginn wird kurz blind von der aktuellen Position nach rechts gedreht, um von der gerade verfolgten Linie abzukommen. Die Drehung ist abgeschlossen und wird gestoppt, sobald der rechte der beiden Optokoppler, die für das Linienfolgen zuständig sind, wieder auf Schwarz kommt. Im Normalfall sollte er damit eine 90° Drehung vollführt haben.

2.1.4 void turn_left()

Analog zu *turn_right()*, nur dass sich jetzt das rechte Rad vorwärts dreht, während das linke leicht gegensteuert.

2.1.5 void robo_init()

Diese Funktion setzt alle Einstellungen des Roboters auf Startwerte. So wird z.B. der Servomotor auf 90° eingestellt, was zu einem geöffneten Greifer führt. Auch wird in dieser Funktion die Breitensuche aufgerufen, welche die Fahrwege berechnet und in Arrays schreibt.

2.1.6 void take_guest()

Diese Funktion wird aufgerufen, wenn der Roboter einen Fahrgast vor sich erkannt hat. Die Funktion setzt die Gradzahl des Servomotors auf 20°, was bedeutet dass sich der Greifer des Roboters schließt. Anschließen wird ein „has_guest“-Flag auf true gesetzt, welches bewirkt, dass der Roboter weiß, dass er gerade einen Fahrgast aufgenommen hat. Dies ist notwendig, damit der Roboter nicht während er den Fahrgast transportiert die Funktionen ausführt, die er normalerweise aufruft, wenn er einen Fahrgast erkennt.

2.1.7 void put_guest()

Diese Funktion ist das Gegenstück *take_guest()*. Sie ist dafür zuständig den Fahrgast wieder freizulassen, wenn der Roboter im Ziel angekommen ist. Dafür wird in dieser Funktion lediglich der Servomotor wieder auf 90° gestellt. Außerdem wird das „has_guest“-Flag wieder auf falsch gesetzt und der Greifer damit geöffnet.

2.2 Die Hauptroutine

Die Hauptroutine des Roboters _ROBI_ ist in der main-Funktion definiert. Zu Beginn des Programms werden die Funktion robo_init()(siehe Abschnitt 2.1.5) aufgerufen und der Roboter damit initialisiert. Das bedeutet, dass z.B. der Greifer geöffnet und die Breitensuche durchgeführt wird. Letztere befüllt eine Zahl globaler Arrays mit den Fahraufträgen, die der Roboter später abfahren muss, um zu einem Gast zu kommen und wieder zurück zum Start zu fahren. Den äußeren Rahmen des Programms bildet eine Endlosschleife. In dieser findet sich eine weitere Schleife, in der der aktuelle Fahrauftrag (aktuell ausgewähltes Array mit codierten Bewegungsanweisungen) elementweise abgearbeitet wird. In dieser Schleife wiederum befinden sich vier Logikeinheiten, die im Folgenden erläutert werden sollen.

2.2.1 Die Kreuzungsbehandlung

Im Wesentlichen handelt es sich hierbei um eine umfangreiche If-Abfrage. Es wird abgefragt, ob einer der beiden äußeren Optokoppler (siehe Kapitel zur technischen Beschreibung) auf eine

Schwarz gekommen ist. Ist das der Fall wird angenommen, dass ROBI soeben auf eine Kreuzung gerollt ist. Nun passiert Folgendes:
Aus dem Array des aktuellen Fahrauftrages wird ausgelesen, was an dieser Kreuzung zu tun ist. Mögliche Werte für das ausgelesene Element sind an dieser Stelle:

- 'g' - Wird ein g ausgelesen bedeutet dies, dass die Funktion *drive_over()* aufgerufen wird und die Kreuzung dadurch einfach überfahren wird.
- 'r' - Wird ein r ausgelesen, so ruft das Programm die Funktion *turn_right()* auf und ROBI dreht sich an der Kreuzung nach rechts.
- 'l' - Wird ein l ausgelesen, so wird die Funktion *turn_left()* aufgerufen und ROBI dreht sich an der Kreuzung nach links.
- 'z' - Ist das ausgelesene Element ein z, so weiß das Programm, dass der nachfolgende Knoten (ermittelt durch Breitensuche) ein Zielknoten und damit ein Fahrgast ist. Beim Auslesen eines z wird zusätzlich ermittelt um welche Fahranweisung es sich beim Folgeelement des Arrays handelt. Dieses kann wiederum ein Element aus der Menge {'g', 'l', 'r'} sein. Es wird dann die Fahranweisung des Folgeelementes ausgeführt und der Fahrgast aufgenommen. Zu guter letzt wird ROBI mittels Aufruf der Funktion *set_back()* an die vorherige Kreuzung zurückgesetzt.
- 'e' - Beim Auslesen dieses Elementes wird die innere Schleife sofort mittels break-Anweisung verlassen und der Roboter angehalten. Das e steht für Ende und signalisiert dem Roboter, dass sein Fahrauftrag komplett abgearbeitet ist.

Für die Elemente 'g', 'l' und 'r' gilt außerdem, dass nach ihrem Auslesen das nachfolgende Arrayelement überprüft wird. Falls es sich bei diesem Element um ein 'e' handeln sollte, so weiß der Roboter, dass er nach dem Ausführen der aktuellen Fahranweisung (Funktionen *drive_over()*, *turn_right()*, oder *turn_left()*) anhalten soll, da er dann mit dem Gesamtfahrauftrag fertig ist.

2.2.2 Das Linienfolgen

Das Linienfolgen bildet die wohl grundlegendste Funktionalität des Roboters. Auch hierbei handelt es sich um einen If-Anweisungsblock, in dem bei jeder Schleifeniteration geprüft wird, ob entweder der rechte oder linke Optokoppler (innere Optokoppler, siehe Kapitel zur technischen Beschreibung) auf Schwarz gekommen ist. Ist der rechte Optokoppler auf Schwarz gekommen, wird die Drehgeschwindigkeit des rechten Motors verringert um dem Ausbrechen des Roboters nach links entgegenzuwirken. Ebenso wird die Geschwindigkeit des linken Motors gedrosselt, wenn der linke Optokoppler auf Schwarz kommt. Sind beide Optokoppler auf Weiß, wird die Geschwindigkeit der beiden Motoren auf einen höheren Wert gesetzt um die Linie möglichst schnell entlangzufahren.

2.2.3 Die Greiflogik

Hat ROBI während der Kreuzungsbehandlung ein z ausgelesen, so steht er danach in der Regel vor einem Fahrgast. Ein If-Block zur Greiflogik sorgt nun dafür, dass ROBI noch solange vorwärts fährt bis der nach vorne ausgerichtete Optokoppler signalisiert, dass ein Objekt unmittelbar vor dem Roboter und damit direkt unter dem Greifer ist. Nun wird die Funktion *take_guest()* aufgerufen und der Fahrgast, ein blauer Plastikball von oben gegriffen. Anschließend wird die Funktion *set_back()* aufgerufen, welche ROBI zurück an die vorherige Kreuzung setzen lässt. Von dort aus wird mit der Kreuzungsbehandlung weitergemacht.

2.2.4 Weitersetzen des (Teil-)Fahrauftrages

Hat ROBI einen Fahrgast (erfolgreich oder nicht) am Ziel abgeliefert, wird in diesem If-Block der aktuelle Teil- Fahrauftrag weitergeschalten. Konkret bedeutet dies, dass ein Zeiger vom aktuell abzuarbeitenden Array auf das nächste umgesetzt wird. Außerdem wird eine Zählvariable inkrementiert in der gespeichert ist, bei welchem Array sich der Roboter im Moment befindet und der Index für den nächsten Auftrag zurückgesetzt, damit ROBI auch am Anfang beginnt.

2.3 Probleme

2.3.1 Linienfolgen im Rückwärtsgang

Ein großes Problem ergab sich aus dem Umstand, dass am Roboter nur Optokoppler in Fahrtrichtung angebracht sind. Beim Rückwärtsfahren kann dadurch die Linie nicht zuverlässig verfolgt werden. Dass liegt daran, dass nicht unterschieden werden kann, in welche Richtung der Roboter mit dem Heck ausgebrochen ist, wenn einer der Linienfolge-Optokoppler auf Schwarz gekommen ist. Es gibt sowohl für den rechten als auch für den linken Optokoppler die Möglichkeiten, dass der Roboter mit dem Heck nach rechts oder links ausgebrochen ist, aber das Programm kann dies nicht unterscheiden und dementsprechend kann nicht verlässlich gegengesteuert werden, um den Ausbruch zu korrigieren. Im schlimmsten Fall wird der Fehler dadurch sogar verschlimmert. Deshalb fiel die Entscheidung, beim zurückfahren keine Linienverfolgung einzusetzen, da dies teilweise schlimmere Auswirkungen hatte, als blind zu fahren.

2.3.2 Kreuzungserkennung

Zu Beginn der Implementierung der Kreuzungserkennung bestand das Problem, dass ROBI eine Kreuzung beim Überfahren mehrfach erkannte. Das lag daran, dass er den Kreuzungszähler imkrementierte sobald er Schwarz erkannte mit den entsprechenen Optokopplern. Durch die hohe Iterationsrate der Hauptschleife waren die Optokoppler jedoch während einer einzelnen Kreuzungsüberfahrt noch auf Schwarz während die Kreuzungserkennung bereits mehrmals durchgelaufen war. Dieses Problem wurde gelöst indem die Schleife nach erstmaligem erkennen der Kreuzung eine Weile angehalten wurde mittels der Sleep() Funktion.

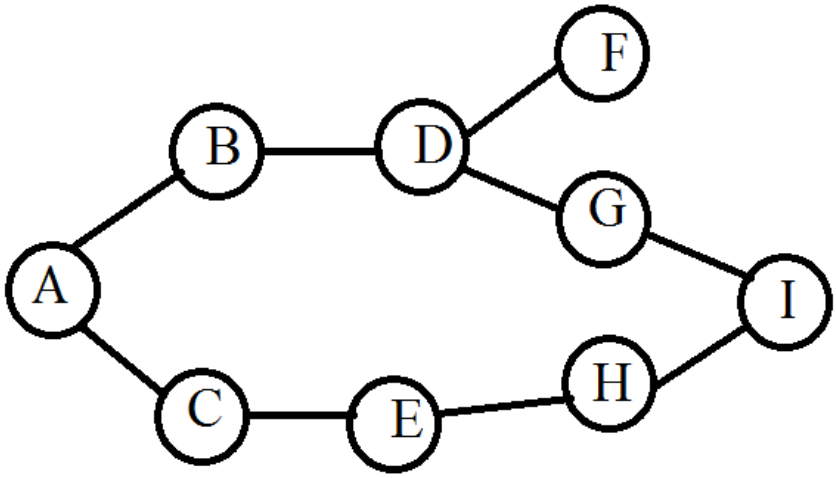
Ein weiteres großes Problem bestand darin, zu erkennen wann der Roboter an einer Kreuzung ist. Zunächst wurde hier abgefragt ob beide äußeren Optokoppler auf Schwarz seien und daraus geschlussfolgert, dass ROBI auf einer Kreuzung stehen müsse. Es stellte sich jedoch heraus, dass der Roboter teilweise so schräg über eine Kreuzung fuhr, dass einer der beiden Optokoppler bereits von der Linie runter war, als der andere gerade erst auf sie kam. Dadurch wurde die Und-Bedingung nicht wahr und ROBI übersah die Kreuzung einfach.

2.3.3 Berücksichtigung der Trägheit

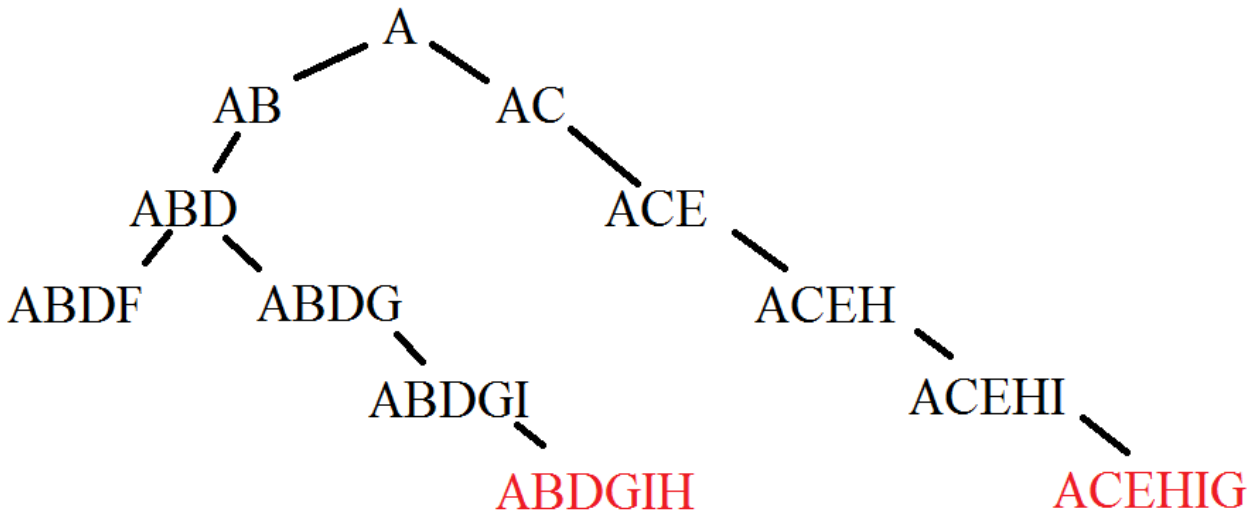
Bei annähernd allen Bewegungen ROBIs musste die reale Trägheit berücksichtigt werden. Diese führte anfangs häufig zu Problemen, da es ROBI z.B. beim Drehen an der Kreuzung von der Linie trug nachdem er sich gedreht hatte. Ähnliches galt für das Anhalten an einer Kreuzung. Hier fuhr ROBI meist etwas zu weit und stand deswegen in einer ungünstigen Position zum drehen. Diesen Problemen wurde Rechnung getragen, indem der Roboter nach jeder Bewegung scharf gebremst wurde. Dafür wurden die Motoren kurz und stark in die jeweils entgegengesetzte Richtung gedreht.

3 Die Breitensuche

Der Ansatz um den schnellsten und damit kürzesten Weg zu finden, bestand in der Nutzung der Breitensuche, ein Beispiel, wir haben eine Route: A-I seien dabei die zu besuchenden Orte.



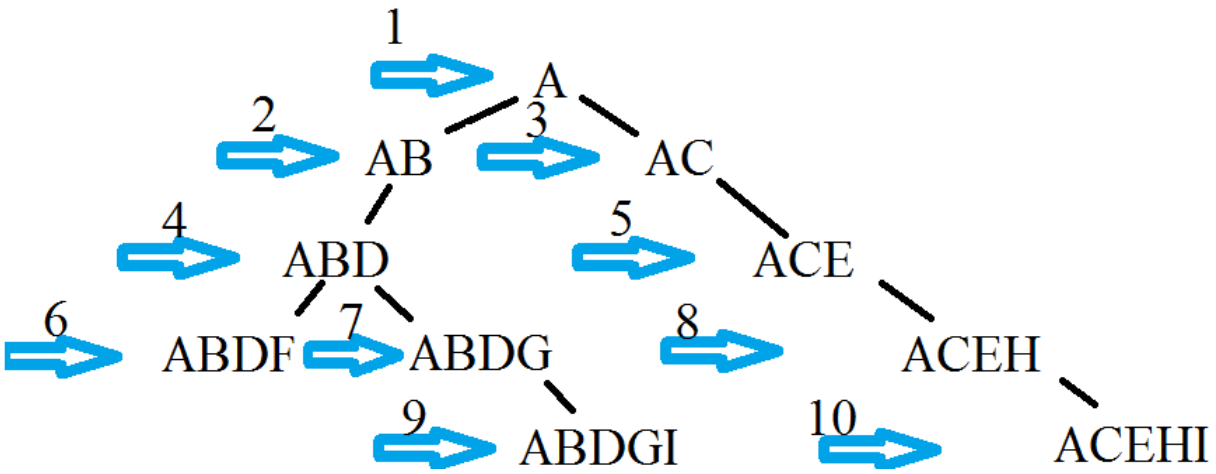
Die Kanten sind ungerichtet und als Verbindungen zwischen diesen Orten zu verstehen. Dabei ergibt sich folgender Suchbaum, wenn wir in der Annahme sind, dass A der Startpunkt sei:



Die roten Knoten können wir ignorieren, denn dort würde die Suche im Kreise verlaufen, wir kommen also schon zu längst besuchten Orten!

Nun zu der Breitensuche, sie beginnt bei dem festgelegten Startknoten, in dem Beispiel A, im folgenden Quellcode würde dieser entweder 68 oder 64 sein. Nun wird der Startknoten expandiert, dass heißt alle seine Nachfolgerknoten (von ihm aus zu erreichende Knoten) werden in der nächsten

Suchtiefe des Baums mit seinen Vorgängerknoten eingetragen (siehe Grafik). Die rot markierten Knoten deuten auf eine Endlosschleife hin, ab da wiederholt sich alles, dies kommt bei einem Suchbaum mit Schleifen vor. Der Vorteil bei der Breitensuche ist, dass sie immer den kürzesten Weg findet, vorausgesetzt die Kosten sind in jedem Knoten gleich oder anders gesagt die Entfernungen aller verbundenen Orte ist dieselbe. Nun zum Verfahren, stellen wir uns nun vor wir würden den Zielknoten H suchen:
(Die blauen Pfeile verdeutlichen die Suchrichtung, wie wir sehen suchen wir in der Breite)



Die Agenda (Suchverlauf) würde sich wie folgt entwickeln:

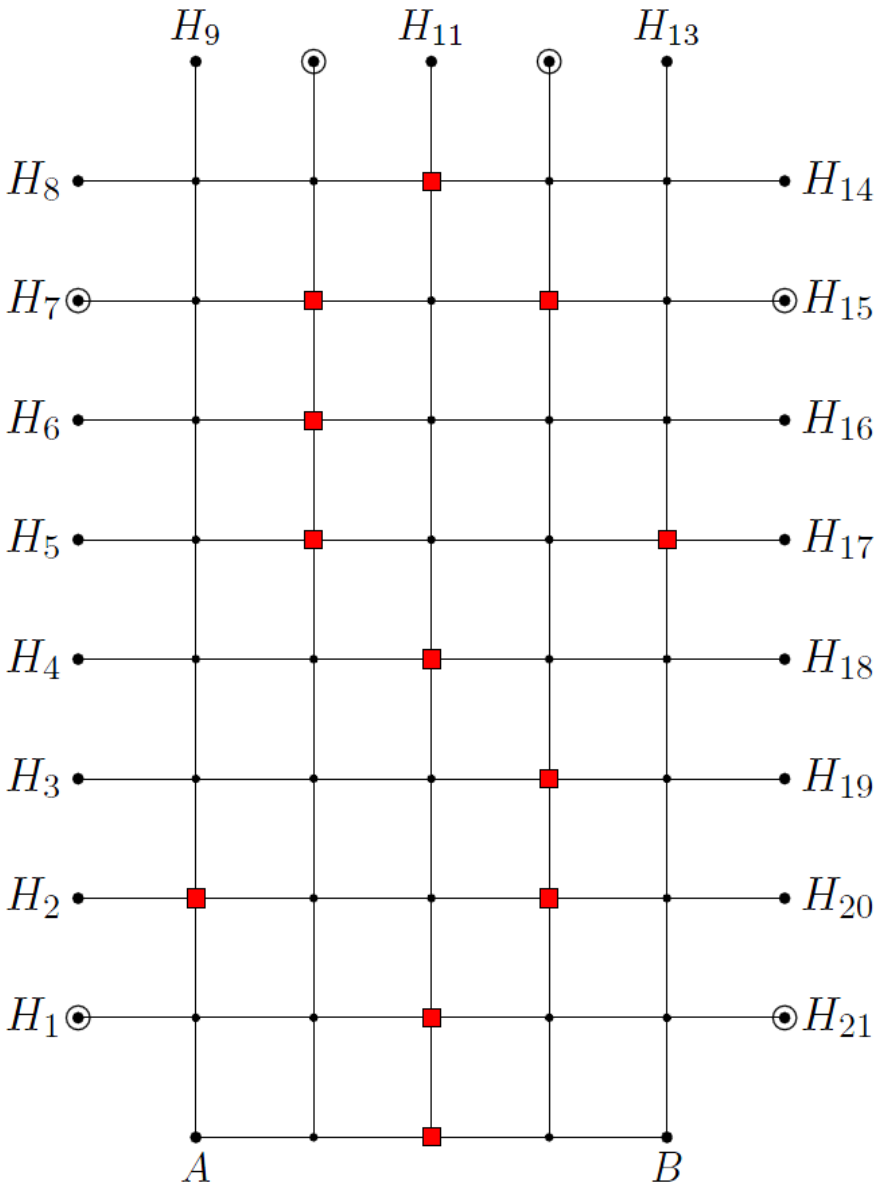
Expandiere	Agenda			
A	A			
AB	AB	AC		
AC	AC	ABD		
ABD	ABD	ACE		
ACE	ACE	ABDF	ABDG	
ABDF	ABDF	ABDG	ACEH	//hier ist schon Ende, H haben
ABDG	ABDG	ACEH		

Nun wären wir schon im 6ten Schritt fertig, da wir merken das sich ACE zu ACEH expandieren lässt und unseren Zielknoten enthält. In jedem Schritt wurde also der Knoten der zuerst in unserer Liste auftaucht expandiert und der/die neu dazu gekommene/n Knoten nach hinten angehängt. Die Breitensuche arbeitet also nachdem FIFO Prinzip, welches gewährleistet das eben in der Breite gesucht wird.

3.1 Die Repräsentation der Karte

3.1.1 Aufbau

Hier ist der Aufbau des Felds, wo die Roboter agieren:



[Masdar1213]

Es handelt sich also um ein 7*10 großes Feld, 70 Gitterpunkte also, wenn man die 4 Ecken mit betrachtet würde. Zuerst muss die Repräsentation der Karte im Programm gewählt werden, dabei war anfangs die Grundidee die Karte als 2 Dimensionales Array der Form

`char karte[7][10];` .Doch es wurde ein 1 Dimensionales Array vorgeschlagen `char karte[70];` bzw. `char schedule[70];`.

Wenn man sich nun einen Knoten, also ein Gitterpunkt betrachtet, sehen wir das wir den Knoten

nach Norden, Osten, Süden und Westen expandieren können, dabei stelle man sich vor das H11 im Norden liegt und die Startpositionen A, B dem zu Folge im Süden. Jedem Knoten wurde eine Nummer zugewiesen, oben links war die 0, A war die 64 und B, 4 Gitterpunkte weiter, die 68. Dies gilt, da das Array und die Karte bei 0 anfangen, demnach ist also die 68 eine 69.

So gestaltete sich der Zugriff des Arrays leicht, denn `schedule[68]` entsprach genau der Position 68 auf der Karte. Die obige Karte zB. wurde so abgebildet:

```
char schedule[71] ="xxFxFxxx..x..xF.x.x.Fx.x...xx.x..xxx..x..xx...x.xxx..x.xF..x..Fx..x..x";
```

Wobei eben nach jedem 7ten Zeichen, die neue Zeile beginnen würde. Wie sicherlich bemerkt wurde, ist es ein Array mit der Länge von 71 Zeichen, eine Fehlannahme wären 70 Zeichen, da es sich um einen String handelt und dabei darf das "*\0*" Stringterminierungszeichen nicht fehlen, welches das Ende eines Strings signalisiert.

Ein "*x*" steht für eine gesperrte Kreuzung (Gitterpunkt), "." stehen für offene Kreuzungen und ein *"F"* symbolisiert den Fahrgast. Wollen wir den letzten vorkommenden Fahrgast also aus der Karte wählen, müssten wir auf `schedule[62]` zugreifen.
Wir sehen in der Karte, dass es sich um den Fahrgast am rechten Rand in der vorletzten "Zeile" handelt. Wir können uns den Zugriff leicht vorstellen, in dem wir von der Position B=68 nach oben gehen, da jede Zeile 7 Kreuzungen umfasst, rechnen wir also 68-7 = 61, nun sehen wir das wir um 1 nach rechts gehen müssen, also 61 + 1 = 62.

3.1.2 Berechnung

Damit ergeben sich die Rechnung um einen Knoten zu expandieren wie folgt:
Norden = Knoten – 7, wenn der Knoten größer als 6 ist, sonst bekommen wir negative Werte und würde uns außerhalb des Felds befinden.

Osten = Knoten +1, wenn der Knoten+1 Modulo 7 nicht 0 ergibt, denn das sind alle Knoten, die sich im Westen befinden.

Westen = Knoten -1, dann muss noch die Bedingung geprüft werden, ob der Knoten-1 modulo 7 nicht 6 ist, denn das wären wiederum alle Knoten, die sich im Osten befinden.

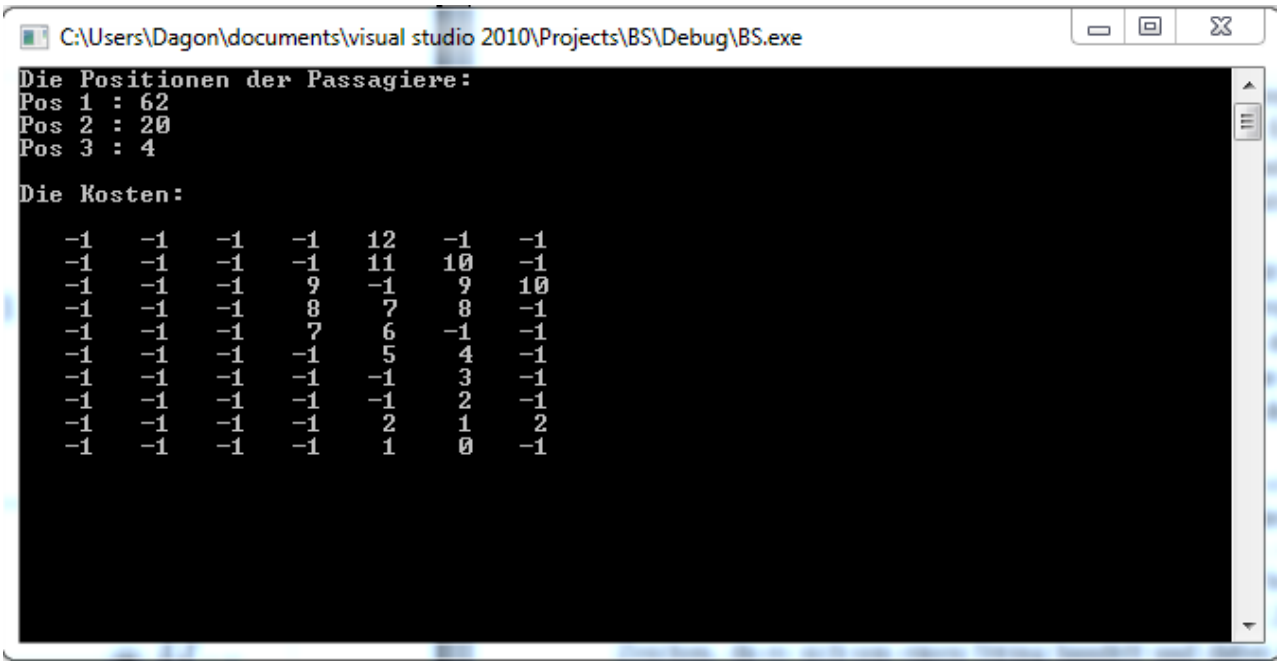
Süden = Knoten+7, wie die Expansion im Norden, kann der Knoten im Süden auch die Karte überschreiten, daher, dass die Anzahl der Knoten 70 (0-69) ist, müssen wir prüfen, dass der Knoten+7 kleiner als 70 ist!

3.1.3 Wertung dieser Implementierung und etwaige Schwächen

Im Nachhinein ist klar geworden, dass man die Prüfungen wie Knoten – 7 > -1 wegfallen lassen hätte können, denn die Bedingungen werden nur wahr, wenn eben am äußersten Rand der Karte steht, dies ist aber nie der Fall, da die Fahrgäste dort positioniert sind und von dort aus es verboten ist, diese Position einzunehmen, darüberhinaus werden diese Randgebiete sonst durch ein "*x*" begrenzt, daher hätte es genügt nach einem "*x*" oder einem "*F*" zu prüfen.

3.2 Kostenmatrix

Hierbei handelt es sich um eine Matrix, welche die Kosten für jeden Knoten abbildet.



[*Abbildung der Kostenmatrix*]

Die Kostenmatrix wird als erstes mit -1 initialisiert, dieser Wert sagt, dass es bisher keine Kosten gab, 0 kann nicht als Startwert genommen werden, denn diese Kosten trägt der Knoten, bei dem wir beginnen.
Da dieser Knoten auch bekannt ist, setzen wir zu Beginn die Kosten dieses Knotens, nachdem alle anderen mit -1 initialisiert worden sind, auf 0.

Oben in der Abbildung der Kostenmatrix ist eine Ausgabe mit den Positionen der Fahrgästen zu sehen. Die Kostenmatrix und die Passagierpositionen, sind die Ergebnisse der Breitensuche.

Die Kostenmatrix ist ebenfalls ein 1-Dimensionales Array, wobei von 0-69 die Kosten jedes Knoten eingetragen worden sind. Die Kosten übertragen sich
`cost_matrix[node_cur] = cost_matrix[agenda[0]] + 1;` in dieser Weise.
Der derzeitig gefundene Knoten k1 hat die Kosten des derzeitig expandierten Knoten k0 erhöht um 1. k0 ist an der ersten Stelle der Agenda (`agenda[0]`) zu finden, wobei k1 eben der gefundene Knoten aus der Expansion von k0 ist.

3.3 Der Weg zum Passagier

3.3.1 Die Idee

Anhand der Fahrgastpositionen und der Kostenmatrix wird nun der Weg ermittelt.
Direkt vom Startknoten aus ist dies nicht möglich, denn es gibt mehrere Punkte, welche die selben Kosten tragen und die Breitensuche liefert den kürzesten Weg jedoch nicht die Knoten, dazu müsste man im Vorhinein wissen, welcher Knoten zum Zielknotenführt um diese in einer Liste aufnehmen zu können.

Der Weg zum Fahrgast wird nun über die Fahrgastpositionen und seine Kosten ermittelt. Dieser Algorithmus ist ähnlich der Breitensuche, diese sucht nun auch nach Knoten in jeder Himmelsrichtung unter Berücksichtigung der Kosten und arbeitet sich den Weg von dieser Position aus zum Knoten mit den Kosten 0 zurück, in dem es immer vom aktuellen Knoten k, den nächsten Knoten k´ sucht, wobei Kosten(k´) = Kosten(k)-1 sind.

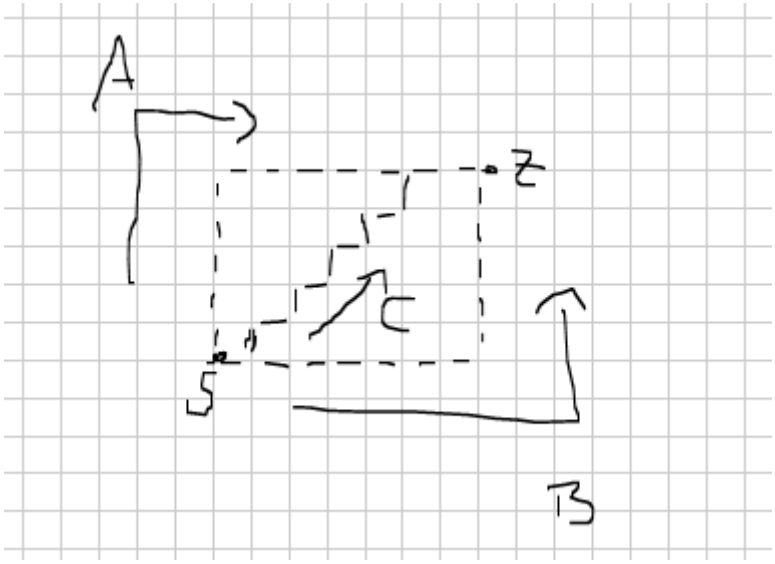
Die Kostenmatrix ergibt sich wie ein "*Diamant*" (Herr Boersch). Wenn man die Grafik betrachtet, wird ersichtlich wies.

```
        3
      3 2 3
    3 2 1 2 3
  3 2 1 0 1 2 3
    3 2 1 2 3
      3 2 3
        3
```

[*Abbildung Kostenmatrix, bei 0 ist der Beginn*]

3.3.2 Wertung dieser Implementierung

Man hätte noch zusätzlich den optimalsten Weg finden können, indem man die Anzahl der Rotationen mit berücksichtigt. Beispiel:



[Abbildung: 3 mögliche Wege zum Ziel, von S nach Z]

Die Wege A, B und C sind alle gleich lang, also haben die gleichen Kosten, jedoch haben die Wege A und B nur eine bis zwei Rotationen, je nachdem wie der Roboter anfangs ausgerichtet ist. Der Weg C auf der anderen Seite 10-11 Rotationen, welches zur Ungenauigkeit des Roboters beitragen kann.

Diese implementierte BS sucht aber jedoch nur nach den erst besten Weg den sie findet. Würde die BS bei S beginnen und die Knoten expandieren, wäre A der erst beste Weg, da die BS ja zuerst im Norden sucht, würde die BS zuerst im Osten suchen, würde diese den Weg C als erst besten Weg finden, da die Knoten auf der Strecke C immer zuerst expandiert werden würden.

3.4 Die Pathlist

Mit den Fahrgastpositionen und der Kostenmatrix wird nun das `char wp_list[70];` Array gefüllt. Dieses enthält die Knoten um zu dem Fahrgast zu kommen. Dieses Array wird jedoch ausgehend von der Fahrgastposition aus aufgebaut, daher enthält dieses Array die Wegpunktliste in umgekehrter Reihenfolge.

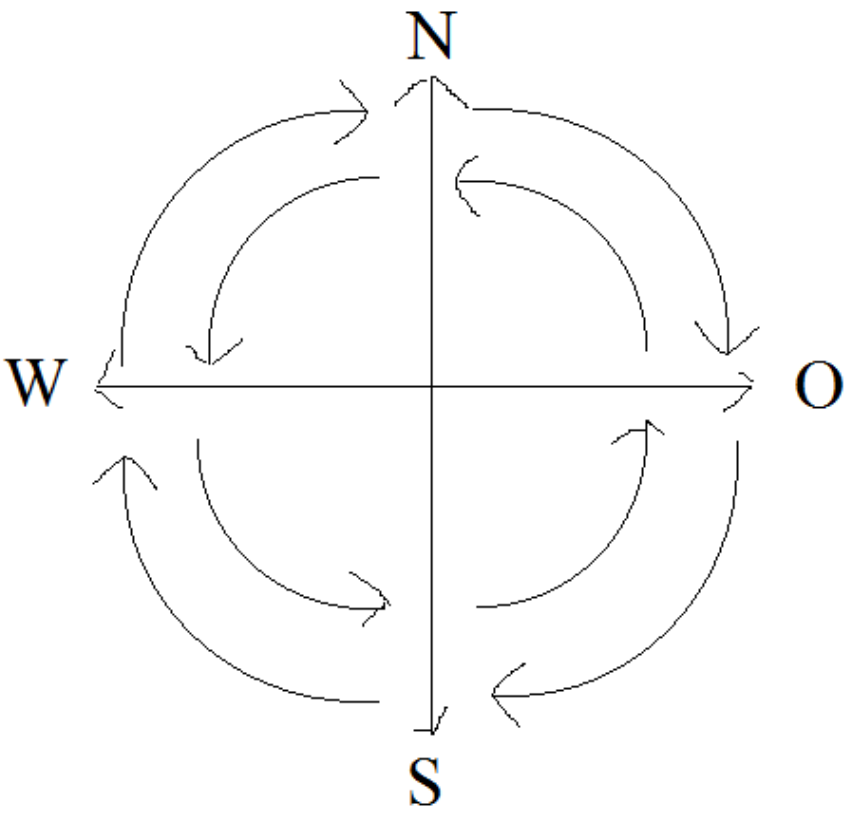
Nun wird die `char path_list[70]` gefüllt, diese entsteht dabei aus der Differenz der einzelnen Wegpunkte aus der `wp_list` dabei entsteht ein Array, welches aus der Menge {-7,-1,1 und 7} besteht, wobei -7 wieder Norden ist, äquivalent zur BS, welche auch nach Norden guckt.

Im nächsten Schritt wird bei der `path_list` diesen Werten neue erzeugt: -7 = 0, 1 = 1, -7 = 2, -1 = 3.

3.5 Ausrichtung des Roboters (Heading)

3.5.1 Die Idee

Im Folgenden wird auch hier mit dem `path_list` Array gearbeitet.
Der Roboter würde anfangs nach Norden gucken (Heading = 0), je nachdem wo der Roboter nun hin muss, würde relativ zu seiner Ausrichtung bestimmen werden, ob er links, rechts oder gerade aus fahren muss. Rückwärts in diesem Fall nicht, da er ja am untersten Rand beginnt und nie rückwärtsfahren muss, es sei denn er hat sich verfahren, welches aber durch die BS verhindert wird. Auch wenn er einen Passagier gefunden hat, wird er anders vorgehen als Rückwärts zu fahren.
Je nach seiner Ausrichtung gibt es also 3 Möglichkeiten: L,R,G.



[*Abbildung Orientierung*]

Werden als Integerwerte (N = 0, O =1, S = 3, W = 3) repräsentiert. Wir sehen das Gegen den Uhrzeigersinn alle links Wendungen und mit dem Uhrzeigersinn alle rechts Wendungen angegeben werden. Diese Transitionen, wann l oder r folgt, sehen wir anhand der Pfeile.
Geradeaus zu fahren bedeutet seine derzeitige Ausrichtung beizubehalten, dieser Übergang erfolgt wenn zwei gleiche Ausrichtungen gefunden werden (z.B. W1,W2).

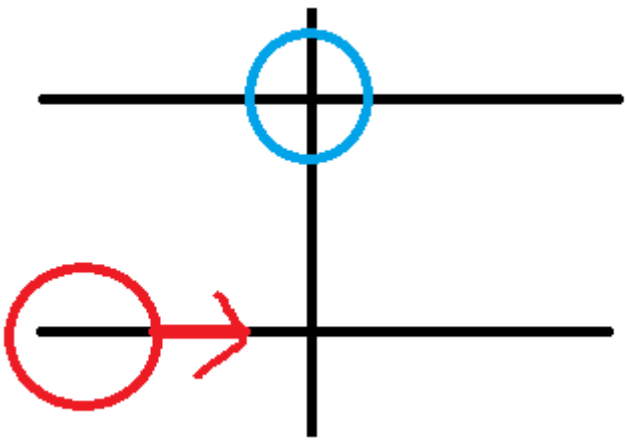
Wenn wir die Differenzen mit dem Uhrzeigersinn betrachten, e.g. Norden – Osten = 0 -1 = -1, erhalten wir überall die Differenz von -1, diese könnte man als rechts nehmen, jedoch gibt es bei W-N = 3-0 = 3 eine 3. Genauso verhält es sich links herum, überall wäre die Differenz +1, jedoch bei 0-3 wäre diese -3. Daher hätte man die Unterscheidungen zwischen rechts = -1 oder 3 bzw. Links = +1 oder -3 machen können. Die Beibehaltung der aktuellen Richtung wäre immer 0, da N-N= 0, O-O = 0, etc. Da sich dahinter immer die gleichen Zahlen verbergen.

Jedoch wurde diese Variante vorgeschlagen:

```
direction = ((heading +4) - path_list[i])%4;
```

[*Herr Boersch*]

`direction` enthält immer die Drehrichtung, welche durch die derzeitige Ausrichtung des Roboters und durch die Position des neuen Knotens ergibt:



[*Abbildung Wo soll ich hin*]

Rot = Roboter mit dem Heading (Pfeil) und dem Zielknoten (blau). Es muss eine Linksdrehung erfolgen, da der Knoten im "Norden" liegt und der Roboter nach "Osten" guckt.

Nun können keine negativen Differenzen mehr entstehen, da + 4 gerechnet wird nun entstehen vom Heading ausgehend folgende Werte 4,5,6,7, diese werden nun mit der Differenz von 0,1,2,3 gebildet. Da nun Werte im Bereich von 0,1,2,3 gewünscht wurden, rechnen wir zum Schluss modulo 4. 4-1 = 5-2 = 3, hier könnte man davon ausgehen, dass es Werte gibt, die sich überschneiden, aber da wir uns immer nur zum Nachfolger, Vorgänger oder gar nicht drehen, kann solch eine Rechnung 5-2 nicht entstehen. Auch das Heading muss auf den neuen Wert gesetzt werden, im obigen Beispiel, siehe [Abbildung Wo soll ich hin], also auf 0.

Das Ergebnis sieht wie folgt aus:

[*Abbildung neue path_list*]

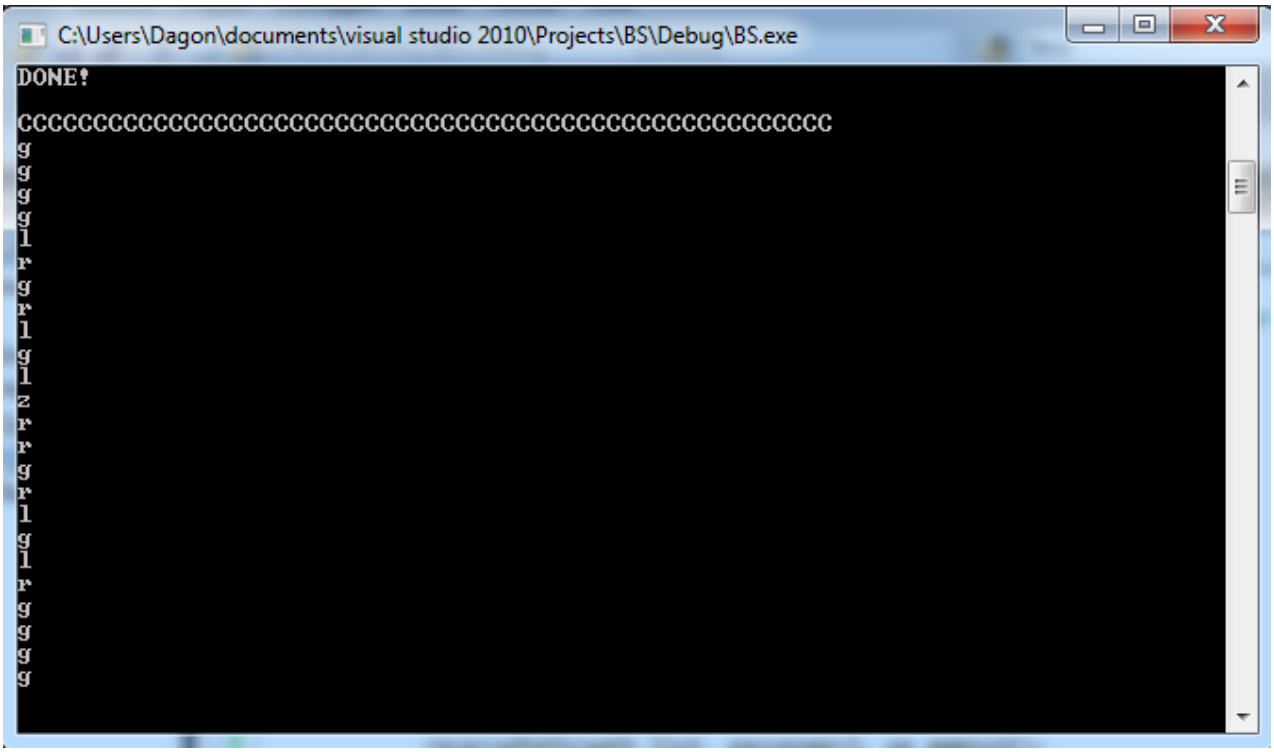
3.5.2 Fehler

Mit dem Heading war hier die größte Fehlerquelle, denn in dem obigen Bespiel wurde auf der Position 62 der erste Fahrgast gefunden, unter der Voraussetzung das `node_k0 = 68` ist. Das Heading war also dann 1, für Osten. Wurde nun der nächste Fahrgast gesucht, dann war der Roboter anfangs nicht mehr nördlich ausgerichtet, sondern dachte er würde nach Osten sehen. Daher gibt es die Funktion `void clearWPInfo();`, welche einige Variablen auf ihren Urzustand zurücksetzt.

Aus der `path_list` ist nun ein Array geworden, welches aus der Menge (r,l,g,z) besteht. Das z sagt uns an, dass wir uns ein Knoten vor dem Ziel befinden.

3.6 Der Weg zurück

Zuletzt wird der Weg zurück bestimmt, in dem die Werte aus der `path_list` {l,r} umgedreht werden: l=r, r =l. Dieses wird dann in eines dieser Arrays gespeichert: `passenger0,passenger1` oder `passenger2` abhängig davon, welche Route zu welchem Fahrgast wir ermitteln. Wobei nur die Richtungsangabe nachdem z nicht umgedreht wird, sondern beibehalten wird und gleichzeitig als erstes Element der umgedrehten Liste zu sehen ist.



[*Abbildung Ergebnis der reversePath()*]

Wenn man sich bei [Abbildung Ergebnis der reversePath()] anguckt und sich immer die gegenüberliegenenden Elemente anguckt (e.g. 1-24,2-23), dann sieht man, dass es sich immer um das umgedrehte handelt, außer bei "g". Das Element nach "z" ist wie erwähnt, das einzige welches nicht umgedreht werden muss und wird zwei Mal benutzt, ein Mal bei der Hinfahrt und ein weiteres Mal bei der Rückfahrt.

Hier stellt sich nun die Frage, was passieren würde, wenn so etwas kommen würde: (g,r,g,g,l,z,g) Dann wäre das Ergebnis sicherlich (g,r,g,g,l,z,g,r,g,g,l,g), dann wäre das erste "g" nachdem "r" falsch! Daher hat der Algorithmus noch einen kleinen Fehler.

Der Weg zurück war auch eines der fehlerbehaftetsten Abschnitte, vor allem deswegen:

```
dest[2*wp_ammount-3-i] = 'l';
```

2*wp_ammount, da sich ja der Weg verdoppelt, wenn man noch zurückfährt, aber da entfallen 3 Elemente, wovon anfangs 1 nicht berücksichtigt wurde. Das Array beginnt wieder bei der 0, daher entfällt das erst einmal, nun entfällt das "r", das brauchten wir nur ein Mal. Woran jedoch nicht gedacht wurde, ist das Element, welches nachdem "r" kommt, dieses wird nämlich zwei Mal benutzt, daher wird dieses auch nicht noch ein Mal aufgenommen. 2*wp_ammount-3, auch nicht 2*(wp_ammount-3), sonst entfallen diese ja doppelt.

3.7 Fazit BS

Es ist sehr zu empfehlen, von Anfang an seinen Code sauber zu halten, darunter fällt die richtige Variablenbenennung, Prüfungen von eventuellen Fehlerquellen und Kommentierung, dadurch wäre vieles von Anfang an leichter gefallen.

4 Quellen

[Masdar1213] *I.Boersch, J. Heihnsohn im WS 12/13 PDF über Masdarcity, Sammlungen der Karten*

5 Anhang

Eigenständiger, ausführbarer Quellcode der BS, nur dieser ist auch kommentiert. Die BS im Hauptprogramm wurde nicht kommentiert.

```
#include <stdio.h>
#include <conio.h>

char heading = 0;
        //die Ausrichtung des Roboters (norden 0, osten 1, süden 2, westen 3)
char node_k0 = 68;
        //hier beginnt unsere Breitensuche (Startpunkt des Roboters)
char node_cur;
        //jeder neue Knoten, welcher druch die BS gefunden wird, wird hier zwischen
gespeichert
char node_passenger;
        //hier ist der Knoten, an dem der Fahrgast gefunden wurde
char agenda_size = 0;
        //wieviele Knoten in unserer Agenda sind, ist diese 0 und kein Fahrgast wurde
gefunden endet die BS
char found_passenger = 0;
        //wenn ein Fahrgast gefunden wurde, auf 1 gesetzt, um die Breitensuche zu beenden
char passengers_left = 3;
        //wie viele Fahrgäste noch zu finden sind
char wp;
        //enthält den derzeitigen Knoten aus der Route, welche zum Fahrgast führt
char wp_ammount = 1;
        //die Größe dieser Route
char wp_complete = 0;
        //dies sagt uns, ob wir mit der Route fertig sind

char passenger_positions[3];
        //die Fahrgastpositionen sind alle hier zu verzeichnen

char schedule[71]
="xxFxFxxx..x..xF.x.x.Fx.x...xx.x..xxx..x..xx...x.xxx..x.xF..x..Fx..x..x";  //Die Karte
char cost_matrix[70];
        //Kosten jedes Knotens
char agenda[70];
        //die Agenda
char wp_list[70];
        //die Route zum Fahrgast als Kontenmenge in umgekehrter Reihenfolge
char path_list[70];
        //enthält zuerst -7,-1,1,7 dann 0,1,2,3 und zum Schluss l,g,r

char passenger0[70];
        //3 seperate Arrays, für jeden der Fahrgäste seine eigene Route
char passenger1[70];
char passenger2[70];

void search();
        //dies ruft die Funktionen setCostMatrix(), setAgenda(), expandAgenda() und
nodeList() auf
void setCostMatrix();
        //initialisiere die Kostenmatrix mit -1 und den Startknoten mit 0
void setAgenda();
        //den ersten Knoten in unsere Agenda aufnehmen

/*Debug Ausgaben*/
void outAgenda();
```

```
void outCostMatrix();

void outPassengerPositions();

void outWPList();

void outPathList(char c);

void outPassenger012(char* passenger012, char c);
void outPassenger(char* pass);

void expandAgenda();
        //dies ruft die Funktionen alle folgenden Funktionen auf sortAgenda(),
 expand_north(), expand_east(), expand_south() und expand_west()
void sortAgenda();
        //entfernt alle expandierten Knoten, wieviele das sind wird durch
void expand_north();
        //expandiere im Norden, etc.
void expand_east();

void expand_south();

void expand_west();

void checkPassenger();
        //prüft, ob der derzeitige gefundene Knoten in einer der Himmelsrichtungen einen
  Fahrgast gefunden hat

void nodeList();
        //aus dem Ergebnis der BS, welches die Kostenmatrix und die Position der
 Farhgäste ist, werden nun die Wege zu den gefundenen Fahrgästen erstellt
void nextWPNorth();
        //hier wird der Weg vom Fahrgast zum node_k0 gesucht
void nextWPEast();

void nextWPSouth();

void nextWPWest();

void checkWPComplete();
        //ist der Weg vervollständigt?
void clearWPInfo();
        //setze relevante Informationen auf Usrprung zurück (zB. heading)

void path();
        //ruft die foglenden Funktionen auf evalPath(),directions() und reverseWPList()
void evalPath();
        //werden die Himmelsrichtungen die vorher aus einer Differenz in der path()
 entstanden in {0,1,2,3} überführt (Norden,Osten,Süden,Westen)
void directions();
        //hier werden die Drehrichtungen bestimmt
void reverseWPList();
        //da der Pfad umgedreht ist, wird dieser hier richtig angeordnet (umgedreht da
 wir bei der Position des Fahrgasts beginnen)
void reversePath(char* src, char* dest, char wp_ammount);
        //Damit der Roboter vom Fahrgast zum Startpunkt wieder kommt, werden nun alle
```

```
 richtungen umgekehrt und diese hinten an den Pfad rangehängt

int main(int argc, char* argv[]){

    search();

    return 0;

}

void search(){

    setCostMatrix();
    setAgenda();
    expandAgenda();
    outPassengerPositions();
    outCostMatrix();
    nodeList();

}

void setCostMatrix(){

    char i;

    //alle Knoten haben die Kosten -1
    for(i = 0; i < 70; i++){

        cost_matrix[i] = -1;

    }

    //außer unser Anfangsknoten
    cost_matrix[node_k0] = 0;

}

void setAgenda(){

    //Nehme unseren Anfangsknoten als ersten Knoten in unsere Agenda auf und erhöhe die
    Anzahl der in der Agenda befindlichen Knoten um 1
    agenda[0] = node_k0;
    agenda_size++;

}

void outAgenda(){

    char i;

    printf("Die Agenda:\n");

    for(i=0; i < agenda_size; i++){

        printf("%d", agenda[i]);
        printf("\n");
```

```
    }
    getchar();
}
void outCostMatrix(){
    char i;
    printf("Die Kosten:\n");
    for(i=0; i < 70; i++){
        if(i % 7 == 0){
            printf("\n");
        }
        printf("%5d", cost_matrix[i]);
    }
    getchar();
}
void outPassengerPositions(){
    char i;
    printf("Die Positionen der Passagiere:\n");
    for(i = 0; i < 3; i++){
        printf("Pos %d : %d \n", i+1, passenger_positions[i]);
    }
    getchar();
}
void outWPList(){
    char i;
    printf("Die WPList(Wegpunkte):\n");
    for(i = 0; i < wp_ammount; i++){
        printf("%d \n", wp_list[i]);
    }
```

```
    getchar();

}

void outPathList(char c){

    char i;

    printf("Die PathList(Wegpunkte der BS):\n");

    if(c == 'c'){

        for(i = 0; i < wp_ammount; i++){

            printf("%c \n", path_list[i]);

        }

    }else{

        for(i = 0; i < wp_ammount; i++){

            printf("%d \n", path_list[i]);

        }

    }

    getchar();

}

void outPassenger012(char* passenger012, char c){

    char e;
    char i = 0;
    e = passenger012[0];

    if(c=='e'){

        while( e != 'e'){

            e = passenger012[i+1];
            printf("%c \n", passenger012[i]);
            i++;

        }

    }else{

        for(i=0; i<wp_ammount;i++){

            printf("%c \n", passenger012[i]);

        }
```

```
    }

    printf("\n");

}

void expandAgenda(){

    while(!found_passenger && agenda_size){

        //Expandiere unseren derzeitigen ersten Knoten in der Agenda
        expand_north();
        expand_east();
        expand_south();
        expand_west();

        //dann entferne diesen expandierten Knoten
        sortAgenda();

    }

    //Hat die agenda_size einen Wert von 0, konnte kein Knoten mehr gefunden werden, der einen Fahrgast enthält
    if(!agenda_size){

        printf("There are no Passengers left, which i can reach from my current Position!");
        getchar();

        //sicherheitshalber den derzeitigen Knoten auf -1 setzen
        node_cur = -1;

    }

}

void sortAgenda(){

    char i;

    //für alle unsere Knoten (agenda_size) müssen nun, wenn der erste Knoten entfernt wird, alle Nachfolger Knoten auf ihre Vorgängerknoten zugewiesen werden
    //da wir bei i=1 beginnen wird der expandierte Knoten k0 von seinem Nachfolger k1 gnadenlos überschrieben, damit k1 der neue zu expandierende Knoten werden kann
    for(i = 1; i < agenda_size; i++){

        agenda[i-1] = agenda[i];

    }

    //da ein Knoten entfernt worden ist, verringert sich die agenda_size um 1
    agenda_size--;

}
```

```
void expand_north(){

    //die Information des gefundenen potentiellen nächsten Knotens, welcher in der Agenda
     aufgenommen wird und sich nördlich des derzeitig expandierten Knoten befindet, ist
    in node_cur festgehalten
    node_cur = agenda[0] -7;

    //prüfe ob dieser in der Agenda aufgenommen werden kann: Kann man noch nach Norden
    expandieren, Ist die Krezung frei (kein x), wurde dieser Knoten noch nicht bewertet,
    es kein Fahrgast biesher gab und ober er weder im Westen, noch im Osten steht
    if((node_cur > -1) && (schedule[node_cur] != 'x') && (cost_matrix[node_cur] == -1) &&
     (!found_passenger) && (node_cur % 7 != 6) && (node_cur % 7 != 0)){

        cost_matrix[node_cur] = cost_matrix[agenda[0]] + 1; //dann setze die Kosten
        agenda[agenda_size] = node_cur; //hinten anstellen bitte! Den neuen Knoten nach
        dem FIFO Prinzip hinten in der Agenda anorden
        agenda_size++; //da wir einen neuen Knoten haben, erhöhen wir die Anzahl der in
        der Agenda befindlichen Knoten

        checkPassenger(); //nun prüfe ob sich auf dieser neuen Position node_cur ein
        Fahrgast verbrigt

    }

    //Bedingungen wie node_cur > -1, node_cur % 7 != 6 und node_cur % 7 != 0 hätten weg
    gelassen werden können, da der Roboter aufgrund der eingtragenen x um Rand der Karte
    nicht hinkommt

}

/*expand_[east|south|west] laufen ähnlich ab*/

void expand_east(){

    node_cur = agenda[0] +1;

    if((node_cur % 7 != 0 ) && (schedule[node_cur] != 'x') && (cost_matrix[node_cur] ==
    -1) && (!found_passenger)){

        cost_matrix[node_cur] = cost_matrix[agenda[0]] + 1;
        agenda[agenda_size] = node_cur;
        agenda_size++;

        checkPassenger();

    }

}

void expand_south(){

    node_cur = agenda[0] +7;

    if((node_cur < 70) && (schedule[node_cur] != 'x') && (cost_matrix[node_cur] == -1) &&
     (!found_passenger) && (node_cur % 7 != 6) && (node_cur % 7 != 0)){
```

```
        cost_matrix[node_cur] = cost_matrix[agenda[0]] + 1;
        agenda[agenda_size] = node_cur;
        agenda_size++;

        checkPassenger();

    }

}

void expand_west(){

    node_cur = agenda[0] -1;

    if((node_cur % 7 !=6 ) && (schedule[node_cur] != 'x') && (cost_matrix[node_cur] ==
    -1) && (!found_passenger)){

        cost_matrix[node_cur] = cost_matrix[agenda[0]] + 1;
        agenda[agenda_size] = node_cur;
        agenda_size++;

        checkPassenger();

    }

}

void checkPassenger(){

    //befindet sich auf der Position des aktuellen Knotens ein Fahrgast
    if(schedule[node_cur] == 'F'){

        //wenn ja nehme diese Postion auf
        passenger_positions[3 - passengers_left] = node_cur;
        passengers_left--;

        if(passengers_left == 0){

            //wenn 3 Fahrgäste gefunden sind, kann die BS enden oder, wenn die
            agenda_size 0 ist und keine (weiteren) gefunden worden sind
            found_passenger = 1;

        }

    }

}

void nodeList(){

    char i;

    //initialisiere anfangs alle Routen zu den Fahrgästen mit e, denn weniger als 3
    gefunden worden sind oder keiner, dann sagt uns das e das vorzeitige ende an
    passenger0[0] = 'e';
    passenger1[0] = 'e';
```

```
    passenger2[0] = 'e';

    for(i = 0; i < 3-passengers_left; i++){

        node_cur = passenger_positions[i];  //wieder wird mir node_cur gearbeitet,
        welcher den ersten Fahrgast aus der Liste nimmt
        wp_list[0] = node_cur;              //wp_list nimmt die Knoten auf, die zum
        Fahrgast direkt führen

        while(!wp_complete){

            //hier werden die Knoten für wp_list gefunden, ähnlich dem Prinzip der BS
            nextWPNorth();
            nextWPEast();
            nextWPSouth();
            nextWPWest();

        }   //at this point the list is completed and we have now to compute the
        directions using the path method

        path();

        if(i==0){

            reversePath(path_list, passenger0, wp_ammount);

        }else if(i==1){

            reversePath(path_list, passenger1, wp_ammount);

        }else if(i == 2){

            reversePath(path_list, passenger2, wp_ammount);
            printf("CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC\n");
            outPassenger012(passenger2,'e');
            getch();

        }

        clearWPInfo();

    }

}

void nextWPNorth(){

    //der Unterschied: der Knoten k1, welcher druch die Expansion des derzeitigen Knotens
     k0(node_cur) gefunden wird, ist in wp und nicht node_cur,
    wp = node_cur - 7;

    //der Knoten muss diesmal bewertet worden sein cost_matrix[wp] > -1 und die Kosten
    dieses Knotens k1 müssen geringer sein, als der von k0: cost_matrix[wp] < cost_matrix
    [node_cur] und !wp_complete
    if((wp > -1) && (cost_matrix[wp] > -1) && (cost_matrix[wp] < cost_matrix[node_cur])
    && (!wp_complete)){
```

```
        //nun wird node_cur auf diesen gefundenen Knoten gesetzt
        node_cur = wp;
        wp_list[wp_ammount] = wp;
        wp_ammount++;
        checkWPComplete();

    }

}

void nextWPEast(){

    wp = node_cur + 1;

    if((wp % 7 != 0 ) && (cost_matrix[wp] > -1) && (cost_matrix[wp] < cost_matrix
    [node_cur]) && (!wp_complete)){

        node_cur = wp;
        wp_list[wp_ammount] = wp;
        wp_ammount++;
        checkWPComplete();

    }

}

void nextWPSouth(){

    wp = node_cur + 7;

    if((wp < 70) && (cost_matrix[wp] > -1) && (cost_matrix[wp] < cost_matrix[node_cur])
    && (!wp_complete)){

        node_cur = wp;
        wp_list[wp_ammount] = wp;
        wp_ammount++;
        checkWPComplete();

    }

}

void nextWPWest(){

    wp = node_cur - 1;

    if((wp % 7 != 6) && (cost_matrix[wp] > -1) && (cost_matrix[wp] < cost_matrix
    [node_cur]) && (!wp_complete)){

        node_cur = wp;
        wp_list[wp_ammount] = wp;
        wp_ammount++;
        checkWPComplete();

    }
```

```
}

void checkWPComplete(){

    //wenn wir die Route vervollständigt haben, dann muss es unser Knoten sein, wo die BS
     ihren Anfang nahm
    if(wp == node_k0){

        wp_complete = 1; //sage das die Route zu diesem Fahrgast fertig ist und beginne
        eventuell mit dem Nächsten

    }

}

void clearWPInfo(){

    //wir müssen einige Bedingungen zurücksetzen, nämlich alles was wir verändert haben
    und für die Ausgangsposition relevant ist (die Ausrichtung ist anfangs immer Norden
    oder das die Route nicht vervollständigt ist)
    wp_complete = 0;
    wp_ammount = 1;
    heading = 0;

}

void path(){

    char i;

    //outWPList();
    //diese Differenz liefert Werte von -7,-1,1, 7, also Norden, Westen,Osten,Süden,
    wp_ammount-1, da wir ja für das Letzte Element keinen Nachfolger für die Subtraktion
    haben
    for(i=0; i < wp_ammount-1 ;i++){

        path_list[i] = wp_list[i] - wp_list[i+1];

    }
    //outPathList('d');
    //diese -7,-1,1, 7 Liste wollen wir nun umkehren
    reverseWPList();

    //
    evalPath();
    //outPathList('d');

    directions();
    //printf("Directions\n");
    //getchar();
    //outPathList('c');

    //now we want to call the drive function

}
```

```
//hier wird aus der -7,-1,1, 7 der path_list 0 Norden, 1 Osten, 2 Süden und 3 Westen
 zugeordnet
void evalPath(){

    char i;

    for(i=0; i < wp_ammount ;i++){

        if(path_list[i] == -7){

            path_list[i] = 0;

        }else if(path_list[i] == 1){ //hätte man dann auch weglassen können, da der Wert
        gleichbleibt

            path_list[i] = 1;

        }else if(path_list[i] == 7){

            path_list[i] = 2;

        }else if(path_list[i] == -1){

            path_list[i] = 3;

        }

    }

}

//reverseWPList (eher reverse Path List), die Route die wir in der wp_list haben beginnt
 mit der Postion des Fahrgasts, also umgekehrt, daher werden wir die Route in die
 Richtige Reihenfolge bringen
void reverseWPList(){

    char i;
    char mem[50];//memory, um zwischen zuspeichern

    for(i=0; i < wp_ammount-1; i++){

        /*
        wir beginnen hinten und da wir aus der Differenz bei path() -7,-1,1,7 erhalten
        haben und einer "entfallen ist" müssen wir von
        wp_ammount 2 abziehen, einer ist entfallen und das array beginnt bei 0, wir
        rechnen -i um das gegensetzliche Element zu erhalten
        */
        mem[i] = path_list[wp_ammount-2-i];

    }

    for(i=0; i < wp_ammount-1; i++){

        //nun wird path_list die richtige Reihenfolge zugewiesen
        path_list[i] = mem[i];
```

```
    }

    wp_ammount = wp_ammount-1;

}

void directions(){

    char i;
    char direction;
    char mem;

    printf("Drections{r,l,g}!%d\n",wp_ammount);
    getchar();

    //Mit der path_list wird nun ein Array geliefert, welches mit 0,1,2 und 3 gefüllt
    ist, je nachdem, wie der Roboter ausgerichtet ist, muss er sich dementsprechend nach
    Links, rechts drehen oder seine Fahrtrichtung beibehalten
    for(i=0; i<wp_ammount;i++){

        //hier die Richtung, entweder drehen oder beibehalten seiner aktuellen Richtung
        direction = ((heading +4) - path_list[i])%4;
        //printf("%d heading %d path_list %d \n und berechnet %d \n\n", i,heading,
        path_list[i],direction);

        if(direction == 3){

            path_list[i] = 'r';
            heading = (heading+1)%4;
            //printf("Ich gucke nach %d und drehe mich nach rechts zu %c\n", heading,
            path_list[i]);

        }

        if(direction == 1){

            path_list[i] = 'l';
            heading = (heading-1)%4;
            //printf("Ich gucke nach %d und drehe mich nach links zu %c\n", heading,
            path_list[i]);

        }

        if(direction == 0){


            path_list[i] = 'g';
            //printf("Ich gucke nach %d und fahre gerade aus zu %c\n", heading, path_list
            [i]);

        }

        if( i == wp_ammount -1){

            //printf("%d Das Ziel ist: %c \n", i, path_list[i]);
```

```
            mem = path_list[i];
            path_list[i] = 'z';
            path_list[i+1]=mem;

        }

    }

    wp_ammount++;
}

void reversePath(char* src, char* dest, char wp_ammount){

    char i=0;
    printf("################################################################################
    \n");

    for(i=0; i < wp_ammount; i++){

        dest[i] = src[i];

    }

    i = 0;

    //drehe den Pfad um, um vom Fahrgast zurück zu finden
    while(src[i] != 'z'){

        //printf("Src=%c ", src[i]);
        //printf("Wo = %d\n", 2*wp_ammount-3-i);

        if(src[i] == 'r'){

            /*das Array verdopplet sich fast, ein Element entfällt, jenes nachdem z,
            dieses müssen wir nicht doppelt aufnehmen,
            ein weiteres entfällt da wir bei 0 anfangen im Array und das Letzte entfällt,
             da wir das r auch ein zweites mal
            nicht aufnehmen, daher entfallen insgesamt 3
            */
            dest[2*wp_ammount-3-i] = 'l';

        }else if(src[i] == 'l'){

            dest[2*wp_ammount-3-i] = 'r';

        }else if(src[i] == 'g'){

            dest[2*wp_ammount-3-i] = 'g';

        }

        i++;

    }

    printf("DONE!\n");
```

```
    getchar();

    //Das e ganz hinten bedeutet, dass der Fahrgast "aussteigen" kann und der Weg
    abgefahren wurde
    dest[2*wp_ammount-2] = 'e';

}
```

```
//Autor: K.-U. Mrkor

//Standard-Include-Files
#include <regc515c.h>

//Diese Include-Datei macht alle Funktionen der
//AkSen-Bibliothek bekannt.
//Unbedingt einbinden!
#include <stub.h>

#define FA11
#include "fa.h"

//
DEFINES=============================================================================
=========================
#define LEFT_MOTOR  2
#define RIGHT_MOTOR 3
#define LEFT_OPTO   analog(8)
#define RIGHT_OPTO  analog(10)
#define LEFT_OPTO2  analog(12)
#define RIGHT_OPTO2 analog(14)
#define GREIFOPTO   analog(5)
#define LEFT_OPTOHECK   analog(7)
#define LICHTSCHRANKE   analog(0)
#define FOTOSENSOR      analog(1)
#define WEISS       110
#define SCHWARZ     140
#define MAX_LENGTH  70

//
GLOBALS=============================================================================
=========================
char heading = 0;
        //the direction our robot looks to (0=north, 3 = west)
char node_k0 = 68;
        //this is our node we will start on (64 or 68), later on it will be determined
through a trigger
char node_cur;
        //this will hold our new node, which is be found in the process of our
Breitensuche
char node_passenger;
        //this is the last node which the passenger is on
char agenda_size = 1;
        //this tells us, how many nodes our agenda currently holds
char found_passenger = 0;
        //this will tells us, if the closest passenger is found
char passengers_left = 3;
        //we have to find this ammount of passengers
char wp;
        //this will hold the current WP and store this one in the Route array
char wp_ammount = 1;
        //the size of the route, to be taken to each passenger
char wp_complete = 0;
        //check if all WPs were found
```

```
char passenger_positions[3];
        //We have to find 3 passengers, the positions (nodes) of them will be stored here


/*Mr. Boerschs Test-schdeules:
char schedule[71] =
 "xxFxFxxx..x..xF.x.x.Fx.x...xx.x..xxx..x..xx...x.xxx..x.xF..x..Fx..x..x";   //F1
 Standart
char schedule[71] =
 "xxFxFxxx..x..xF.x.x.Fx.x...xx.x..xxx..x..xx..x..xxx..x.xF..x..Fx..x..x";   //F2    One
 Crosssing Modified
char schedule[71] =
 "xxFxFxxx..x..xF.x.x.Fx.x...xx.x..xxx..x..xx..x..xxx.x..xF..x..Fx..x..x";   //F3    City
  Slalom
char schedule[71] =
 "xFxxxFxx.x.x.xF..x..Fx..x..xx..x..xx.x.x.xF..x..Fx..x..xx..x..xx.x.x.x";   //F4    Race
  on the Highway
char schedule[71] =
 "xFxxxFxx.x.x.xF..x..Fxx.x.xxx..x..xx.x.x.xF..x..Fxx.x.xxx..x..xx.x.x.x";   //F5    Big
 Slalom
char schedule[71] =
 "xxFxFxxF..x..Fx.x.x.xx..x..xF..x..Fx..x..xxx.x.xxx..x..xxx.x.xxx..x..x";   //F6
 Symmetry
char schedule[71] =
 "xxxFxxxx.....xx..x..xx..x..xF..x..Fx..x..xx..x..xxx.x.xxx..x..xx..x..x";   //F7
 Connected Game
char schedule[71] =
 "xxxFxxxF..x..Fx..x..xxx.x.xxF..x..Fx...x.xx..x..xx.x...xx..x..xx..x..x";   //F8
 Blocked Passenger
char schedule[71] =
 "xFxxxFxF..x..Fx.x...xx.x...xx.x...xx.xxx.xx...x.xx...x.xF...x.Fx..x..x";   //F9
 Outdoor - Free Land
char schedule[71] =
 "xFxxxFxF..x..Fx.x...xx.x.xxxx.x...xx.xxx.xx...x.xxxx.x.xF...x.Fx..x..x";   //F10   Long
  Distance Endurance
char schedule[71] =
 "xxxFxxxx.....xx..x..xx.....xxx...xxx.....xx.....xx.....xx.....xx..x..x";   //F11   VIP
*/

//char schedule[71]
 ="xxFxFxxx..x.AxF.x.x.Fx.x...xx.x..xxx..x..xx...x.xxx..x.xF..x..Fx..x..x";  //71 because
  of the String-Termination sign
//char _fa[71] =
 "xFxxxFxx.x.x.xF..x..Fx..x..xx..x..xx.x.x.xF..x..Fx..x..xx..x..xx.x.x.x";

char cost_matrix[MAX_LENGTH];
                //to store the costs of every node
char agenda[MAX_LENGTH];
                //the list of nodes, which will be found while our algorithm searching
 for the shortest way
char wp_list[MAX_LENGTH];
                //this is the reversed list of nodes, that will lead to the passengers
char path_list[MAX_LENGTH];
                //contains r=right, l=left, u=up, d=down thus, the directions for the
 robot to take to get to the next node
```

```
char passenger0[MAX_LENGTH];
char passenger1[MAX_LENGTH];
char passenger2[MAX_LENGTH];

char has_guest = 0;
//FUNCTION
 PROTOTYPES=============================================================================
 ==================
void search();
        //there we will begin to determine the node
void setCostMatrix();
        //to set in the beginning our costs
void setAgenda();
        //to set our start node in the agenda

void expandAgenda();
        //to expand our nodes we have in our agenda
void sortAgenda();
        //this will remove nodes, wich have allready been visited
void expand_north();
        //this will expand our current in the agenda and searching north of it
void expand_east();
        //this will expand our current in the agenda and searching east of it

void expand_south();
        //this will expand our current in the agenda and searching south of it
void expand_west();
        //this will expand our current in the agenda and searching west of it
void checkPassenger();
        //this notifies us, if our current node holds the passenger

void nodeList();
        //the nodes the rorbot have to take in order to get to the passenger
void nextWPNorth();
        //this will look north, and check if this node was a valid waypoint to be taken
 in order to get to the passenger
void nextWPEast();
        //this will look east, and check if this node was a valid waypoint to be taken
 in order to get to the passenger
void nextWPSouth();
        //this will look south, and check if this node was a valid waypoint to be taken
 in order to get to the passenger
void nextWPWest();
        //this will look west, and check if this node was a valid waypoint to be taken
 in order to get to the passenger
void checkWPComplete();
        //this will check if all wp are found in order to get to the player
void clearWPInfo();
        //to set up the original state in order to determine the next waypoints
 correctly

void path();
        //to generate the path list by subtracting neighboring waypoints
void evalPath();
        //this will determine from the values providing by the path_list, which
 direction they lie
```

```
        //
void reverseWPList();
void reversePath(char* src, char* dest, char wp_ammount);

void set_newpassenger();

//BEWEGUNGS
 FUNKTIONEN=================================================================================
 ==================================
void set_back(){
    //beide Motoren auf mittlerer Stärke anfahren lassen
    char right_power = 5;
    char left_power = 5;
    char right_direction = 1; //1 = zurück für rechten motor
    char left_direction = 1; //1 = zurück für linken motor
    char stopp = 0;

    lcd_cls();
    lcd_setxy(0,0);
    lcd_puts("going back:");

    while(1){

        if((LEFT_OPTO2 >= SCHWARZ) || (RIGHT_OPTO2 >= SCHWARZ)){ //Kreuzung erkannnt mit
        äußeren Optos?
            //dann stoppen durch kurze Gegensteuerung und schleife verlassen
            motor_richtung(LEFT_MOTOR, 0);
            motor_richtung(RIGHT_MOTOR, 0);
            motor_pwm(LEFT_MOTOR, 5);
            motor_pwm(RIGHT_MOTOR, 5);
            sleep(30);
            break;
        }

        //beide motoren zurücksetzen lassen
        motor_richtung(LEFT_MOTOR, left_direction);
        motor_richtung(RIGHT_MOTOR, right_direction);
        motor_pwm(LEFT_MOTOR, left_power);
        motor_pwm(RIGHT_MOTOR, right_power);

    }

}

void turn_left(){

    //lcd_cls();
    lcd_setxy(0,0);
    lcd_puts("turning left...");
    //follow_line();

    // inneres Rad kurz rückwärts
    motor_richtung(LEFT_MOTOR, 1);
    motor_richtung(RIGHT_MOTOR, 0);
    motor_pwm(LEFT_MOTOR, 2);
    motor_pwm(RIGHT_MOTOR, 4);
```

```
    sleep(400); // 0,5 sekunden blind von der Kreuzung drehen
    while(LEFT_OPTO <= WEISS); // solange weiterdrehen bis linker Opto nichtmehr auf Weiß

    //Motoren in die jeweils andere Richtung drehen zm zu stoppen
    motor_richtung(LEFT_MOTOR, 0);
    motor_richtung(RIGHT_MOTOR, 1);
    motor_pwm(LEFT_MOTOR, 3);
    motor_pwm(RIGHT_MOTOR, 5);
    sleep(30);

    motor_richtung(LEFT_MOTOR, 0);
    motor_richtung(RIGHT_MOTOR, 0);
    motor_pwm(LEFT_MOTOR, 0);
    motor_pwm(RIGHT_MOTOR, 0);

    lcd_cls();
}

void turn_right(){

    //lcd_cls();
    lcd_setxy(0,0);
    lcd_puts("turning right...");

    // inneres Rad kurz rückwärts
    motor_richtung(LEFT_MOTOR, 0);
    motor_richtung(RIGHT_MOTOR, 1);
    motor_pwm(LEFT_MOTOR, 5);
    motor_pwm(RIGHT_MOTOR, 2);
    sleep(400); // 0,5 sekunden blind von der Kreuzung drehen
    while(RIGHT_OPTO <= WEISS); // solange weiterdrehen bis linker Opto nichtmehr auf
    Weiß

    //Motoren in die jeweils andere Richtung drehen zm zu stoppen
    motor_richtung(LEFT_MOTOR, 1);
    motor_richtung(RIGHT_MOTOR, 0);
    motor_pwm(LEFT_MOTOR, 5);
    motor_pwm(RIGHT_MOTOR, 3);
    sleep(45);

    motor_richtung(LEFT_MOTOR, 0);
    motor_richtung(RIGHT_MOTOR, 0);
    motor_pwm(LEFT_MOTOR, 0);
    motor_pwm(RIGHT_MOTOR, 0);

    lcd_cls();
}

void drive_over() {
    //kurze Kontrollausgabe
    lcd_setxy(0,0);
    lcd_puts("over...");

    //Vorwärtsfahren solange nicht beide OPTOs von der Kreuzung runter sind
    motor_richtung(LEFT_MOTOR, 0);
    motor_richtung(RIGHT_MOTOR, 0);
```

```
    motor_pwm(LEFT_MOTOR, 5);
    motor_pwm(RIGHT_MOTOR, 5);
    while((LEFT_OPTO2 >= SCHWARZ) || (RIGHT_OPTO2 >= SCHWARZ));
    sleep(20);
    lcd_cls();
}

void take_guest(){
    lcd_setxy(0,0);
    lcd_puts("nehme gast...  ");
    servo_arc(0,20);
    sleep(1000);
}

void put_guest(){
    servo_arc(0,90);
}

//Initialisiert die Robotereinstellungen und ruft die Breitensuche auf
void robo_init(){
    //Zählvariable i für Ausgabe auf LCD
    char i = 0;
    servo_arc(0,90);
    lcd_setxy(1,0);

    //Auf welcher Startposition wird begonnen?
    if(dip_pin(0) == 0){
        node_k0 = 64;
        lcd_puts("linke startpos");
    }
    else{
        node_k0 = 68;
        lcd_puts("rechte startpos");
    }

    sleep(1000);
    lcd_cls();

    //Breitensuche starten -> füllt Arrays mit den Wegen
    search();

    //Schleifenblöcke zur Ausgabe der Fahraufträge und dem Signal dass es losgehen kann
    while((i < 16)){
        lcd_putchar(passenger0[i]);
        i++;
    }
    i = 0;
    sleep(2000);
    lcd_cls();
    while((i < 16)){
        lcd_putchar(passenger1[i]);
        i++;
    }
    i = 0;
    sleep(2000);
    lcd_cls();
```

```
    while((i < 16)){
        lcd_putchar(passenger2[i]);
        i++;
    }
    i = 0;
    sleep(2000);
    lcd_cls();


    lcd_setxy(1,0);
    lcd_puts("Fahrauftrag: ");
    lcd_ubyte(_fa_nr);
    sleep(2000);
    lcd_cls();
    lcd_setxy(0,0);
    lcd_puts("Warte auf Lampe...");

    //Warten auf Lampe
    while(FOTOSENSOR >= 50);
    lcd_cls();
}

//BREITENSUCHE
 FUNKTIONEN=============================================================================
 ======================
void search(){

    setCostMatrix();
    setAgenda();
    expandAgenda();
    nodeList();

}

void setCostMatrix(){

    char i;

    //all nodes will have start costs of -1
    for(i = 0; i < 70; i++){

        cost_matrix[i] = -1;

    }

    //except fpr one, our node, where we will start our journey, is set to 0
    cost_matrix[node_k0] = 0;

}

void setAgenda(){

    agenda[0] = node_k0;

}
```

```
void expandAgenda(){

    while(!found_passenger && agenda_size){

        expand_north();
        expand_east();
        expand_south();
        expand_west();

        sortAgenda();

    }

    if(!agenda_size){

        //to tell, that there is no other node to be found, we set the node_cur (current node) to -1
        node_cur = -1;

    }

}

void sortAgenda(){

    char i;
    for(i = 1; i < agenda_size; i++){

        agenda[i-1] = agenda[i];

    }

    agenda_size--;

}

void expand_north(){

    node_cur = agenda[0] -7;

    if((node_cur > -1) && (_fa[node_cur] != 'x') && (cost_matrix[node_cur] == -1) && (!found_passenger) && (node_cur % 7 != 6) && (node_cur % 7 != 0)){

        cost_matrix[node_cur] = cost_matrix[agenda[0]] + 1;
        agenda[agenda_size] = node_cur;
        agenda_size++;

        checkPassenger();

    }

}

void expand_east(){

    node_cur = agenda[0] +1;
```

```
    if((node_cur % 7 != 0 ) && (_fa[node_cur] != 'x') && (cost_matrix[node_cur] == -1) &&
     (!found_passenger)){

        cost_matrix[node_cur] = cost_matrix[agenda[0]] + 1;
        agenda[agenda_size] = node_cur;
        agenda_size++;

        checkPassenger();

    }

}

void expand_south(){

    node_cur = agenda[0] +7;

    if((node_cur < 70) && (_fa[node_cur] != 'x') && (cost_matrix[node_cur] == -1) && (!
    found_passenger) && (node_cur % 7 != 6) && (node_cur % 7 != 0)){

        cost_matrix[node_cur] = cost_matrix[agenda[0]] + 1;
        agenda[agenda_size] = node_cur;
        agenda_size++;

        checkPassenger();

    }

}

void expand_west(){

    node_cur = agenda[0] -1;

    if((node_cur % 7 !=6 ) && (_fa[node_cur] != 'x') && (cost_matrix[node_cur] == -1) &&
    (!found_passenger)){

        cost_matrix[node_cur] = cost_matrix[agenda[0]] + 1;
        agenda[agenda_size] = node_cur;
        agenda_size++;

        checkPassenger();

    }

}

void checkPassenger(){

    if(_fa[node_cur] == 'F'){

        passenger_positions[3 - passengers_left] = node_cur;
        passengers_left--;

        if(passengers_left == 0){
```

```
            found_passenger = 1;

        }

    }

}

void nodeList(){

    char i;

    passenger0[0] = 'e';
    passenger1[0] = 'e';
    passenger2[0] = 'e';

    for(i = 0; i < 3-passengers_left; i++){

        node_cur = passenger_positions[i];
        wp_list[0] = node_cur;

        while(!wp_complete){

            nextWPNorth();
            nextWPEast();
            nextWPSouth();
            nextWPWest();

        }   //at this point the list is completed and we have now to compute the directions using the path method

        path();

        if(i==0){

            reversePath(path_list, passenger0, wp_ammount);

        }else if(i==1){

            reversePath(path_list, passenger1, wp_ammount);

        }else if(i == 2){

            reversePath(path_list, passenger2, wp_ammount);

        }

        clearWPInfo();

    }

}

void nextWPNorth(){
```

```
    wp = node_cur - 7;

    if((wp > -1) && (cost_matrix[wp] > -1) && (cost_matrix[wp] < cost_matrix[node_cur])
    && (!wp_complete)){

        node_cur = wp;
        wp_list[wp_ammount] = wp;
        wp_ammount++;
        checkWPComplete();

    }

}

void nextWPEast(){

    wp = node_cur + 1;

    if((wp % 7 != 0 ) && (cost_matrix[wp] > -1) && (cost_matrix[wp] < cost_matrix
    [node_cur]) && (!wp_complete)){

        node_cur = wp;
        wp_list[wp_ammount] = wp;
        wp_ammount++;
        checkWPComplete();

    }

}

void nextWPSouth(){

    wp = node_cur + 7;

    if((wp < 70) && (cost_matrix[wp] > -1) && (cost_matrix[wp] < cost_matrix[node_cur])
    && (!wp_complete)){

        node_cur = wp;
        wp_list[wp_ammount] = wp;
        wp_ammount++;
        checkWPComplete();

    }

}

void nextWPWest(){

    wp = node_cur - 1;

    if((wp % 7 != 6) && (cost_matrix[wp] > -1) && (cost_matrix[wp] < cost_matrix
    [node_cur]) && (!wp_complete)){

        node_cur = wp;
        wp_list[wp_ammount] = wp;
        wp_ammount++;
```

```
        checkWPComplete();

    }

}

void checkWPComplete(){

    if(wp == node_k0){

        wp_complete = 1;

    }

}

void clearWPInfo(){

    wp_complete = 0;
    wp_ammount = 1;
    heading = 0;

}

void reverseWPList(){

    char i;
    char mem[50];

    for(i=0; i < wp_ammount-1; i++){

        mem[i] = path_list[wp_ammount-2-i];

    }

    for(i=0; i < wp_ammount-1; i++){

        path_list[i] = mem[i];

    }

    wp_ammount = wp_ammount-1;

}

void path(){

    char i;

    for(i=0; i < wp_ammount-1 ;i++){

        path_list[i] = wp_list[i] - wp_list[i+1];

    }

    reverseWPList();
```

```
    evalPath();

    directions();

    //now we want to call the drive function

}

void evalPath(){

    char i;

    for(i=0; i < wp_ammount ;i++){

        if(path_list[i] == -7){

            path_list[i] = 0;

        }else if(path_list[i] == 1){

            path_list[i] = 1;

        }else if(path_list[i] == 7){

            path_list[i] = 2;

        }else if(path_list[i] == -1){

            path_list[i] = 3;

        }

    }

}

void directions(){

    char i;
    char direction;
    char mem;

    for(i=0; i<wp_ammount;i++){

        direction = ((heading +4) - path_list[i])%4;

        if(direction == 3){

            path_list[i] = 'r';
            heading = (heading+1)%4;

        }

        if(direction == 1){

            path_list[i] = 'l';
```

```
            heading = (heading-1)%4;

        }

        if(direction == 0){


            path_list[i] = 'g';

        }

        if( i == wp_ammount -1){

            mem = path_list[i];
            path_list[i] = 'z';
            path_list[i+1]=mem;

        }

    }

    wp_ammount++;
}

void reversePath(char* src, char* dest, char wp_ammount){

    char i=0;

    for(i=0; i < wp_ammount; i++){

        dest[i] = src[i];

    }

    i = 0;

    while(src[i] != 'z'){

        if(src[i] == 'r'){

            dest[2*wp_ammount-3-i] = 'l';

        }else if(src[i] == 'l'){

            dest[2*wp_ammount-3-i] = 'r';

        }else if(src[i] == 'g'){

            dest[2*wp_ammount-3-i] = 'g';

        }

        i++;

    }
```

```
    dest[2*wp_ammount-2] = 'e';

}


//                                                                              ↵
 MAINFUNCTION=================================================================↵
 ==================================
void AksenMain(void){

    char i = 0;
    char right_power = 6;
    char left_power = 5;
    char right_direction = 0; //0 = geradeaus für rechten motor
    char left_direction = 0; //0 = geradeaus für linken motor
    char direction;
    char has_guest = 0;
    char passenger_count = 0;

    char *cur_passenger = passenger0; //Initialisiere den aktuellen Teilfahrtplan auf den↵
     Ersten.

    robo_init();

    while(1){

        while(i < MAX_LENGTH){
            lcd_setxy(1,0);
            lcd_putchar(cur_passenger[i]);
            lcd_puts("   ");
            // Kreuzung?
            if ((LEFT_OPTO2 >= SCHWARZ) || (RIGHT_OPTO2 >= SCHWARZ)) { // auf Kreuzung?
                direction = cur_passenger[i];       //aktueller richtungsanweisung
                i++;

                if(direction == 'r'){
                    if(cur_passenger[i] == 'e'){
                        lcd_setxy(1,0);
                        lcd_puts("Ende gefunden");

                        //Motoren hart bremsen
                        motor_richtung(RIGHT_MOTOR, 1);
                        motor_richtung(LEFT_MOTOR, 1);
                        motor_pwm(RIGHT_MOTOR,2);
                        motor_pwm(LEFT_MOTOR,2);
                        sleep(50);
                        motor_pwm(RIGHT_MOTOR,0);
                        motor_pwm(LEFT_MOTOR,0);
                        sleep(50);
                        //Gast ablegen und wenden
                        turn_right();
                        put_guest();
                        has_guest = 0;
                        passenger_count++;
                        set_back();
                        turn_right();
```

```
                    break;
                }else{
                    sleep(50);
                    turn_right();
                }
            }else if(direction == 'l'){
                if(cur_passenger[i] == 'e'){
                    lcd_setxy(1,0);
                    lcd_puts("Ende gefunden");

                    //Motoren hart bremsen
                    motor_richtung(RIGHT_MOTOR, 1);
                    motor_richtung(LEFT_MOTOR, 1);
                    motor_pwm(RIGHT_MOTOR,2);
                    motor_pwm(LEFT_MOTOR,2);
                    sleep(50);
                    motor_pwm(RIGHT_MOTOR,0);
                    motor_pwm(LEFT_MOTOR,0);
                    sleep(50);
                    //Gast ablegen und wenden
                    turn_left();
                    put_guest();
                    has_guest = 0;
                    passenger_count++;
                    set_back();
                    turn_left();
                    break;
                }else{
                    sleep(50);
                    turn_left();
                }
            }else if(direction == 'g'){
                if(cur_passenger[i] == 'e'){
                    lcd_setxy(1,0);
                    lcd_puts("Ende gefunden");
                    put_guest();
                    has_guest = 0;
                    passenger_count++;
                    //Motoren hart bremsen
                    motor_richtung(RIGHT_MOTOR, 1);
                    motor_richtung(LEFT_MOTOR, 1);
                    motor_pwm(RIGHT_MOTOR,2);
                    motor_pwm(LEFT_MOTOR,2);
                    sleep(50);
                    motor_pwm(RIGHT_MOTOR,0);
                    motor_pwm(LEFT_MOTOR,0);
                    sleep(50);
                    //Gast ablegen und wenden
                    turn_right();
                    set_back();
                    motor_richtung(RIGHT_MOTOR, 0);
                    motor_richtung(LEFT_MOTOR, 0);
                    motor_pwm(RIGHT_MOTOR,3);
                    motor_pwm(LEFT_MOTOR,3);
                    sleep(100);
                    turn_right();
```

```
                    break;
                }else{
                    drive_over();
                }
            }else if(direction == 'z'){
                if(cur_passenger[i] == 'l'){
                    sleep(50);
                    turn_left();
                }else if(cur_passenger[i] == 'r'){
                    sleep(50);
                    turn_right();
                }else if(cur_passenger[i] == 'g'){
                    drive_over();
                }
            }else if(direction == 'e'){
                break;
            }else{
                motor_richtung(LEFT_MOTOR, 1);
                motor_richtung(RIGHT_MOTOR, 0);
                motor_pwm(LEFT_MOTOR, 3);
                motor_pwm(RIGHT_MOTOR, 3);
                sleep(3000);
                break;
            }
        }
        //GreifLogik
        if((GREIFOPTO <= 120) && (!has_guest)){
            motor_richtung(LEFT_MOTOR, 1);
            motor_richtung(RIGHT_MOTOR, 1);
            motor_pwm(LEFT_MOTOR, 3);
            motor_pwm(RIGHT_MOTOR, 3);
            has_guest = 1;
            sleep(20);
            motor_pwm(LEFT_MOTOR, 0);
            motor_pwm(RIGHT_MOTOR, 0);
            take_guest();
            set_back();

            if((cur_passenger[i] == 'g') && (node_k0 == 68)){
                motor_richtung(LEFT_MOTOR, 0);
                motor_richtung(RIGHT_MOTOR, 0);
                motor_pwm(LEFT_MOTOR, 4);
                motor_pwm(RIGHT_MOTOR, 4);
                sleep(130);
                turn_left();
                set_back();
                turn_left();
                i++;
            }else if((cur_passenger[i] == 'g') && (node_k0 == 64)){
                motor_richtung(LEFT_MOTOR, 0);
                motor_richtung(RIGHT_MOTOR, 0);
                motor_pwm(LEFT_MOTOR, 4);
                motor_pwm(RIGHT_MOTOR, 4);
                sleep(130);
                turn_right();
                set_back();
```

```
                    turn_right();
                    i++;
                }
            }
            // Liniefolgen Logik
            if((RIGHT_OPTO >= SCHWARZ) && (LEFT_OPTO <= SCHWARZ)){  //kommt der rechte
            Optokoppler auf Schwarz...
                right_power = 2;            //dann Stärke des rechten Motors senken um
                gegenzusteuern
                left_power = 6;

            }else if((LEFT_OPTO >= SCHWARZ) && (RIGHT_OPTO <= WEISS)){  //Kommt linker
            Optokoppler auf Schwarz...
                left_power = 2;             //dann Stärke links verringern um
                gegenzusteuern
                right_power = 5;
            }else{
                right_power = 6;            //sonst Geschwindigkeit für beide Motoren
                erhöhen um
                left_power = 6;             //Zeit zu sparen
            }

            motor_richtung(LEFT_MOTOR, left_direction);
            motor_richtung(RIGHT_MOTOR, right_direction);
            motor_pwm(LEFT_MOTOR, left_power);
            motor_pwm(RIGHT_MOTOR, right_power);
        }

        //Logik zum Weitersetzen des (Teil-)Fahrauftrags auf den nächsten Passenger
        if(passenger_count == 1)
            cur_passenger = passenger1;
        else if(passenger_count == 2)
            cur_passenger = passenger2;
        else
            break;
        if(cur_passenger[0] == 'e')
            break;
        //setze zum Schluss den Index des aktellen Arrays auf 1 zurück um denn nächsten
        Gast einzusammeln
        i = 1;


     }
    motor_pwm(LEFT_MOTOR, 0);
    motor_pwm(RIGHT_MOTOR, 0);
    while(1);
}
```