Dokumentation

AMS-Prokekt im WS 2014/15
MASDAR CITY - Personal Rapid Transit

Aleksej Manaev, Mykhaylo Levental

Inhaltsverzeichnis

Strategien

Pfadfindung

Die Idee war den Roboter immer den kürzesten Weg zwischen 2 Knoten zurücklegen zu lassen, weil die zu erfüllenden Aufgaben zeitlich begrenzt sind. Im Studium wurden 2 Algorithmen behandelt, die immer den kürzesten Weg zurückliefern, falls es einen gibt: Breitensuche und der A*-Algorithmus. Jedoch bietet der A*-Algorithmus den Vorteil, dass auch Kosten miteinbezogen werden. So ist es möglich den optimalen Weg mit den wenigsten Abbiegevorgängen zu finden, da diese als zusätzliche Kosten hinzugerechnet werden können. Dies ist insofern wichtig, da es mehrere gleich lange Pfade geben kann. Dabei sind die Pfade mit einer höheren Anzahl an Abbiegevorgängen unerwünscht, da sie Zeitverlust bedeuten und das Risiko bergen von der schwarzen Linie abzukommen, siehe Abbildung 1. Aus zeitlichen Gründen ist die Miteinbeziehung der Kosten für das Abbiegen nicht implementiert worden.

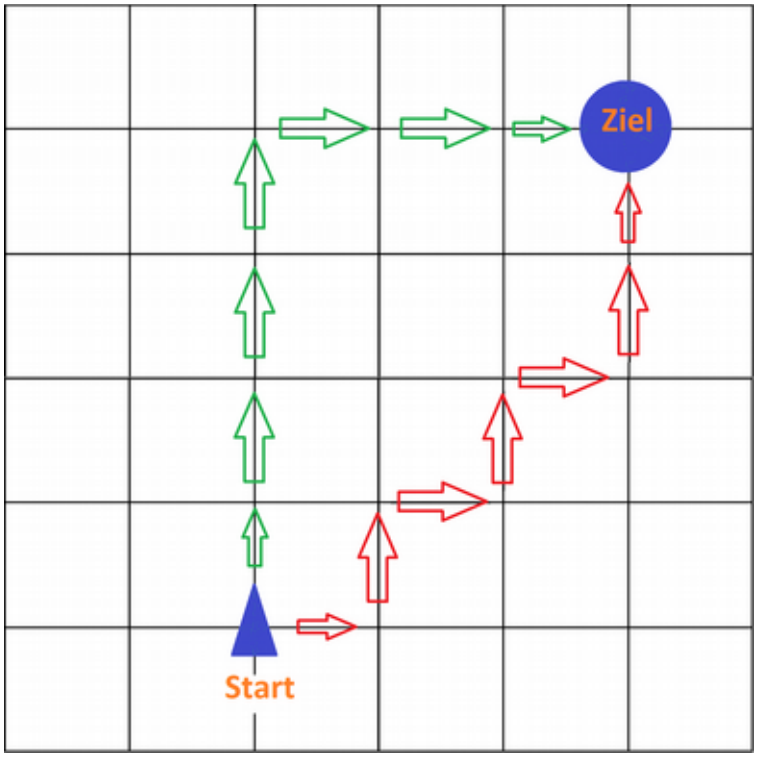

Abbildung 1: Beispiel für 2 gleich lange Pfade, aber mit unterschiedlicher Anzahl an Abbiegevorgängen

Pfad für Hin -und Rückweg

Für Hin -und Rückweg wird der gleiche Pfad verwendet, was zum Vorteil hat, dass der Weg nur ein Mal berechnet werden muss. Wäre das Ziel den optimalen Rückweg zu erhalten, dann müsste eine zusätzliche Berechnung durchgeführt werden bei der das Absetzen des Fahrgastes hinter der AB-Strecke ausreichen würde. Dadurch wäre es möglich, dass kürzere Rückwege entstehen, als den Fahrgast zur Startposition zu bringen.

Priorisierung der Fahrgastabholungen

Bei der Bestimmung der Fahrgäste wird die Karte zeilenweise von rechts nach links ausgelesen, beginnend in der rechten unteren Ecke. In der Auslesereihenfolge werden die Fahrgäste abgeholt. Somit werden bei Karten, wo es keinen Pfad von A nach B gibt, die am nächsten liegenden Fahrgäste zuerst abgeholt, bei anderen Karten nicht.

Roboter, die es nicht schaffen in 120 Sekunden alle Fahrgäste abzuholen und abzuliefern, erzielen mit dieser Strategie am meisten Punkte, was bei diesem Roboter der Fall ist. Außerdem steigt mit der Dauer des Wettbewerbs die Wahrscheinlichkeit, dass der Roboter einen Fehler macht und vom Weg abkommt.

Aufteilung der Arbeit im Team

A. Manaev war hauptsächlich an der Entwicklung des Roboters beteiligt, wohingegen M. Levental den A*-Algorithmus implementierte. Die restlichen Aufgaben wurden gemeinsam erledigt.

Chronologischer Ablauf der Arbeitsschritte

1. Vorüberlegungen zum Lösungsweg/Algorithmus und Roboterarchitektur
2. Bau des Fahrgestells inklusive Getriebe und Rädern
3. Einbau der Motoren und Anpassung des Fahrgestells
4. Einbau von Akku und AKSEN-Board
5. Einbau der Sensoren zum Erkennen der schwarzen Linie
6. Erstellung der Funktionen zum Folgen der Linie und Erkennen der Kreuzung
7. Erstellung der Funktionen zum Abbiegen
8. Implementierung des A*-Algorithmus
9. Bau des Greifers
10. Erstellung der Funktionen zum Greifen und Absetzen des Fahrgastes
11. Erstellung der Funktion zum Umwandeln des Pfads in Roboterbefehle
12. Erstellung der Funktion zum Umwandeln von Befehlen in Roboterbewegungen
13. Bau und Programmierung der Ampel
14. Umsetzung der Startlichterkennung

Vorstellung

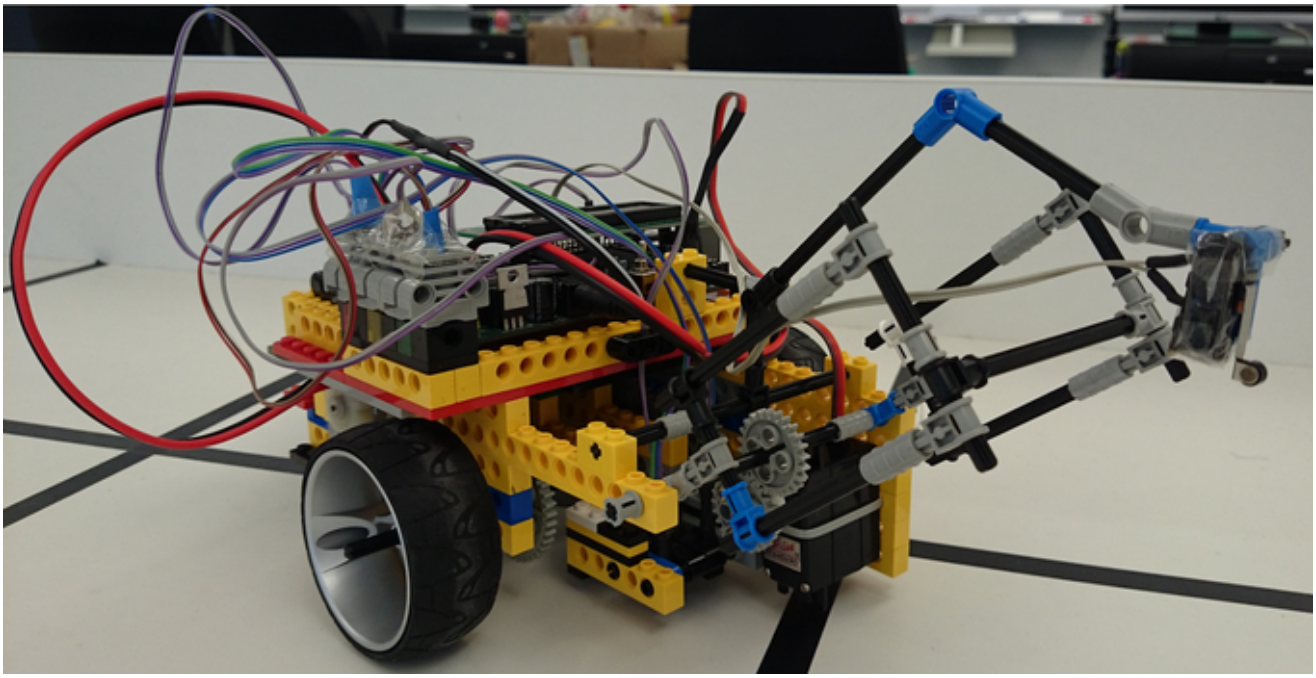

Abbildung 2: Fertig gestellter Roboter

Hardware

Es wird ein differentieller Antrieb verwendet, welcher es ermöglicht den Roboter auf der Stelle drehen zu lassen. Dieser hat aber den Nachteil, dass der Roboter nicht immer gerade aus fährt und deshalb eine Korrektur stattfinden muss. Zum Einsatz kommt eine 1:125 - Getriebeübersetzung. Vorne mittig knapp über dem Boden sind 2 Optokoppler angebracht, welche zum Erkennen der schwarzen Linie genutzt werden. Zwischen den beiden Optokopplern befindet sich der Photosensor, dessen Aufgabe es ist das Startsignal wahrzunehmen. Der Bumper ist die vordere Spitze des Roboters und erkennt die Wand. Unter der Halterunung, an der der Bumper befestigt ist, befindet sich der Greifer in Form eines auf -und zuklappbaren Käfigs, welcher von einem Servomotor angetrieben wird. Der NiMH-Akku befindet sich hinten und das AKSEN-Board oberhalb des Getriebes. Die Ampel ist neben dem AKSEN-Board und besteht aus 3 Lämpchen, die jeweils von einem unterschiedlich farbigen und halbtransparenten Legobauteil umhüllt sind.

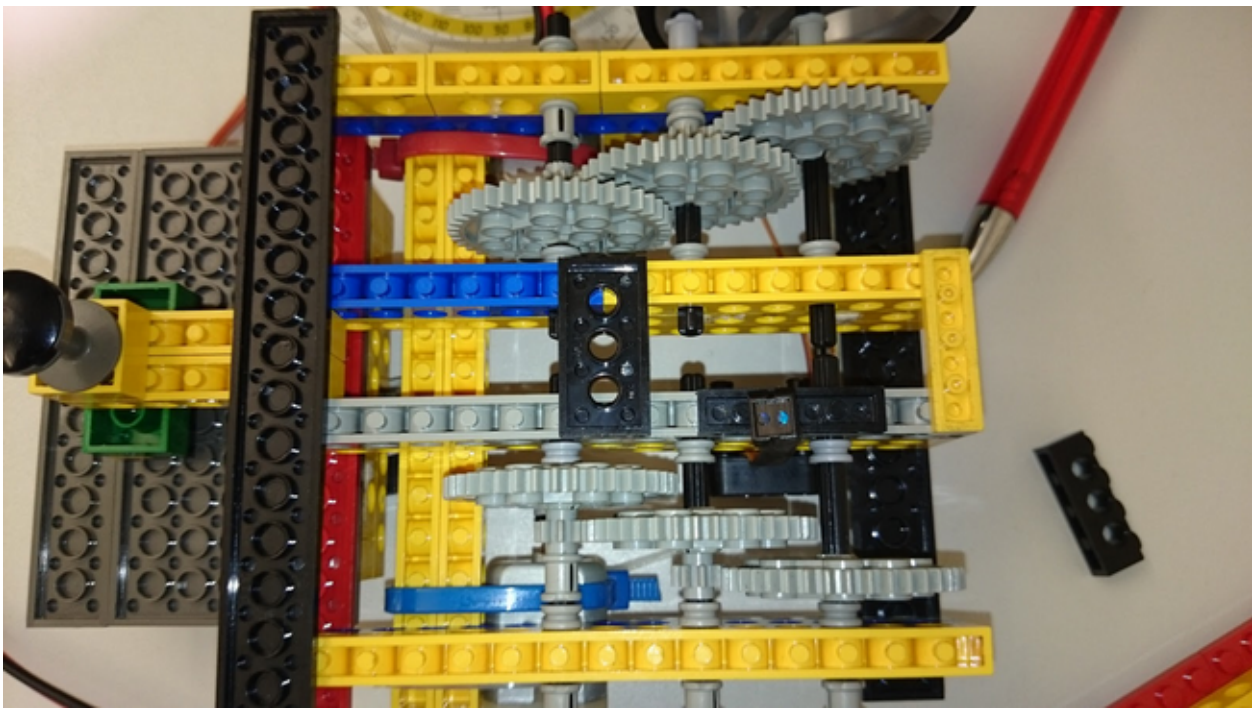

Abbildung 3: Das Getriebe

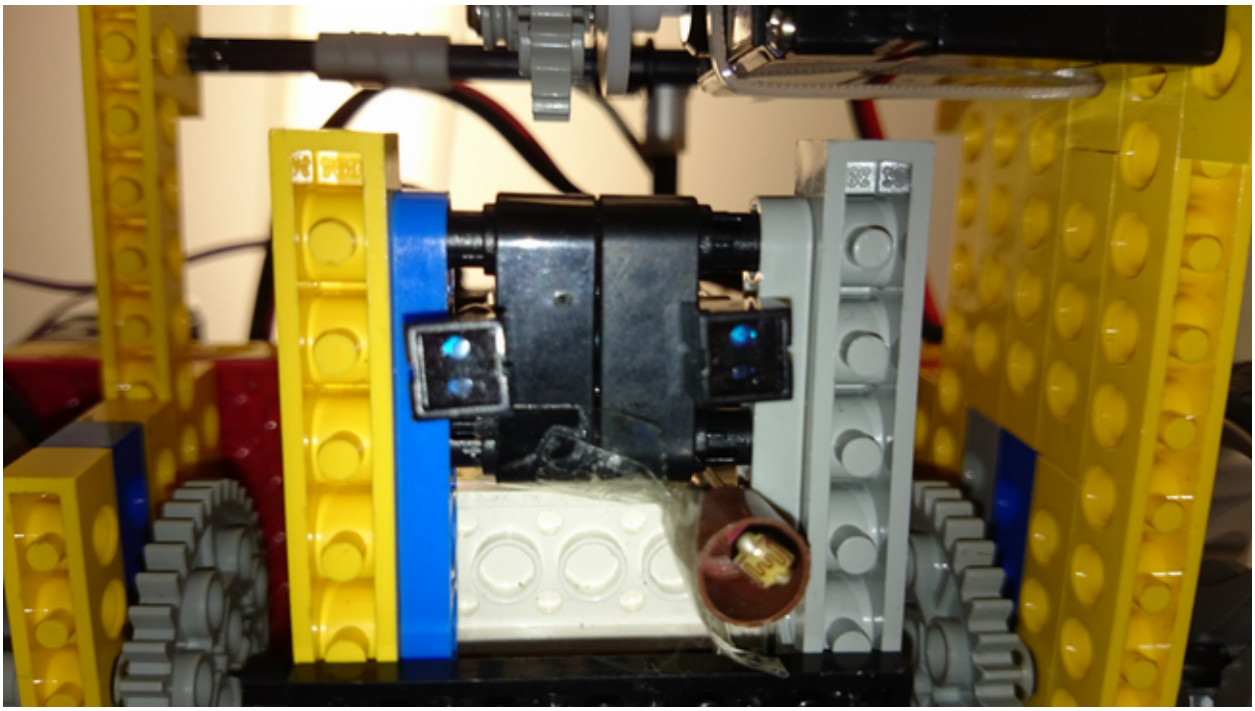

Abbildung 4: Die beiden Optokoppler und der Photosensor

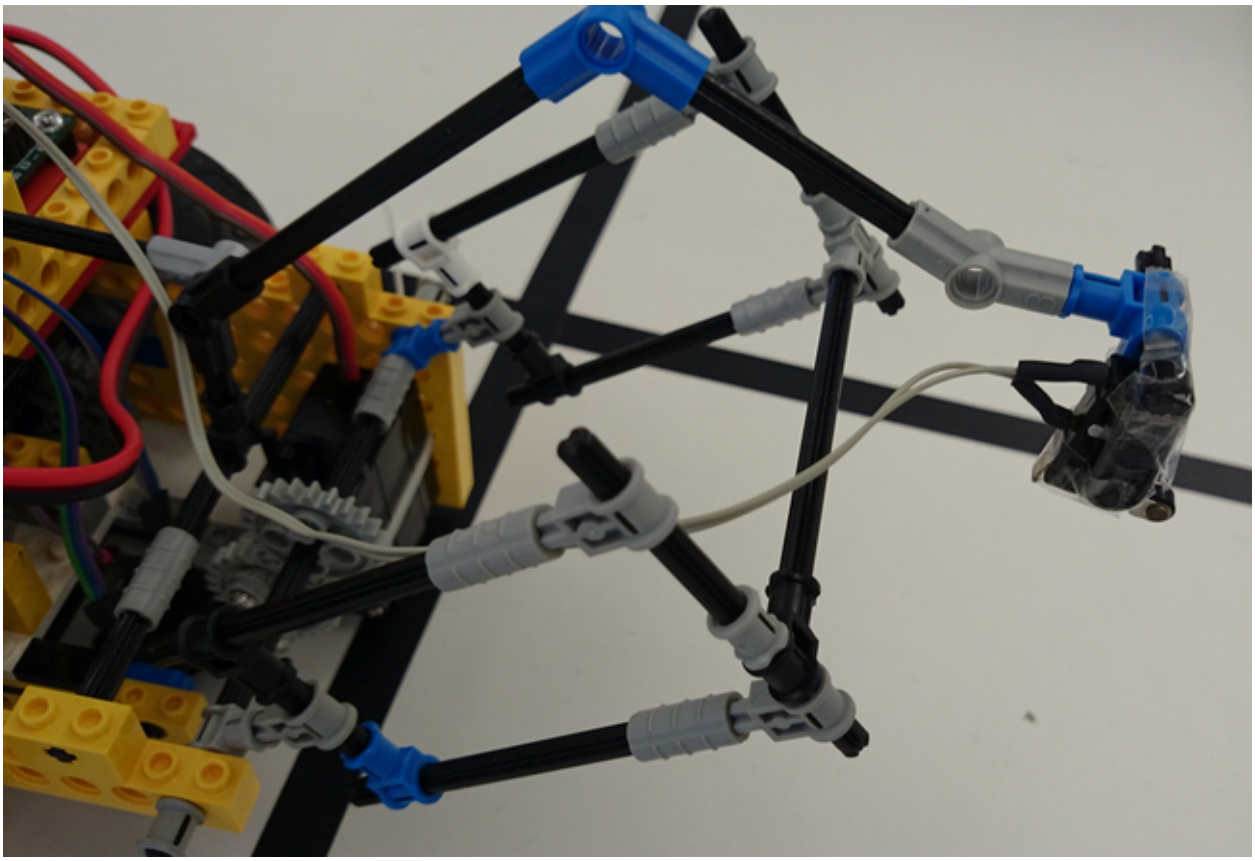

Abbildung 5: Der Bumper und der Greifer

Software

Beim Programmstart zeigt die Ampel rot und es wird auf den Fahrauftrag gewartet. Nachdem der Fahrauftrag eingelesen wurde, zeigt die Ampel gelb und der durch einen String repräsentierte Fahrauftrag wird in ein zweidimensionales Array von Strukturen durch die stringToMap()-Methode umgewandelt, die einen Knoten der Karte darstellen. Diese Methode hat zusätzlich die Aufgabe alle Fahrgäste auszulesen und in einem Array zu speichern.

Danach werden die Fahrgäste durchiteriert und es wird geprüft, ob diese erreichbar sind. Ob ein Fahrgast erreichbar ist, wird bestimmt, indem versucht wird einen Pfad von dem Startpunkt zum Fahrgast zu finden. Dies geschieht mit der Methode shortestPath(), welche den A*-Algorithmus implementiert. Als Ergebnis liefert die Methode den Pfad als eine einfach verkettete Liste, falls es einen gibt.

Ist ein Fahrgast erreichbar, dann wird die verkettete Liste in ein Pfad-Array umgewandelt. Da der selbe Pfad für Hin -und Rückweg benutzt wird, kann dieses Array in 2 Richtungen ausgelesen werden. Um aus dem Pfad die Roboterkommandos für den Hinweg abzuleiten, wird das Array mit calculateAllCommandsForwards() in eine Richtung und für den Rückweg mit calculateAllCommandsBackwards() in die entgegengesetzte durchiteriert. Die Roboterkommandos werden durch Buchstaben repräsentiert und in das Array für die Roboterkommandos eingereiht, welche anschließend in der executeCommands()-Methode auf tatsächliche Roboterbewegungen abgebildet werden (siehe Tabelle 1).

Buchstabe	Methode
l	turnLeft()
t	turnAround()
r	turnRight()
s	straightOnTillNextCrossing()
g	grabPassenger()
d	dropPassenger()

Tabelle 1: Abbildung der Roboterkommandos auf Roboterbewegungen

Das Ableiten der Kommandos funktioniert wiefolgt. Das Pfad-Array wird durchlaufen, es wird der aktuelle Knoten und der nachfolgende betrachtet. Damit der Roboter zum nächsten Knoten fahren kann, muss er vor dem Fahren eventuell abbiegen. Um dies zu entscheiden, wird die Methode calculateDirection() benutzt, um festzustellen in welche Richtung der Roboter fahren muss.
Das passiert, indem zuerst von der x-Koordinate des aktuellen Knotens die des nachfolgenden subtrahiert wird. Ist die Differenz 0, dann findet keine Änderung in x-Richtung statt, demzufolge muss eine Änderung in y-Richtung stattfinden. Die Änderung in y-Richtung wird durch die Differenz der y-Koordinaten der beiden Knoten bestimmt. Bei einer positiven Differenz ergibt sich als Zielrichtung Norden, bei einer negativen Süden. (siehe Abb. 6)
Ist Differenz der x-Koordinaten nicht 0, dann findet eine Änderung x-Richtung statt. Wenn diese positiv ist, dann ergibt sich als Zielrichtung Westen, bei einer negativen Osten. (siehe Abb. 7)
Nachdem die Zielrichtung bestimmte wurde, wird diese mit der aktuellen Richtung in Beziehung gesetzt, und zwar mithilfe der Formel (Zielrichtung - Aktuelle Richtung + 4) mod 4, wobei die Richtungen durch die ganzzahligen Werte von 0 bis 3 repräsentiert werden (siehe Abb. 8).

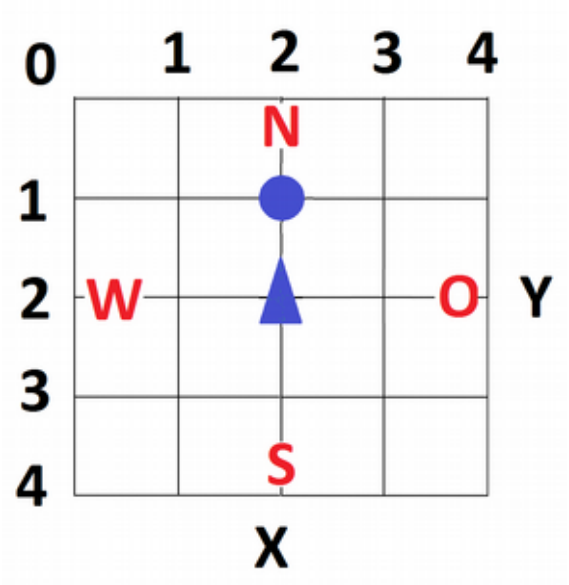

Abbildung 6: Roboter(2,2) mit Ziel(2,1) und Differenz (0,1), somit Zielrichtung Norden

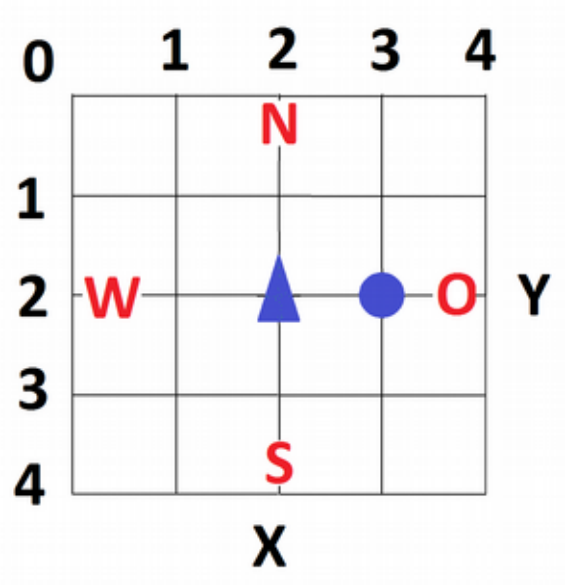

Abbildung 7: Roboter(2,2) mit Ziel(3,2) und Differenz (-1,0), somit Zielrichtung Osten

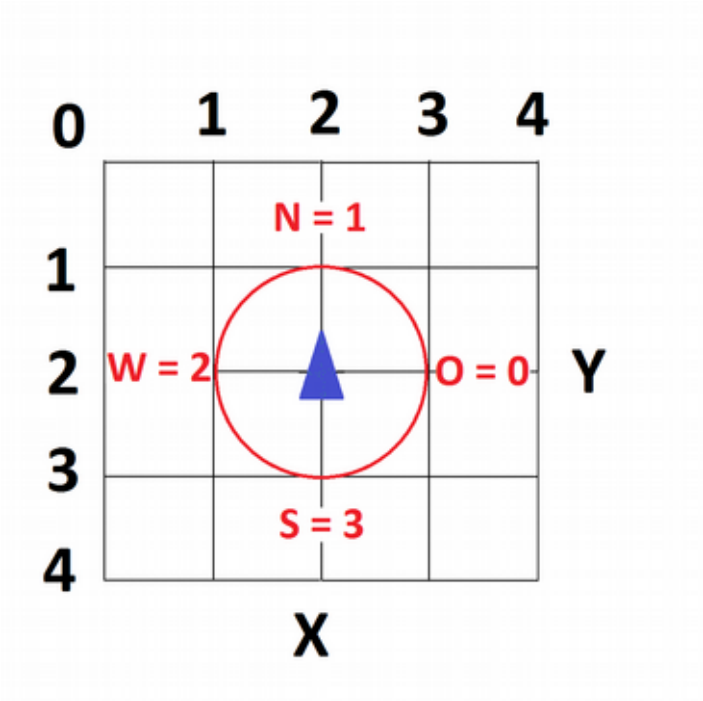

Abbildung 8: Abbildung der Richtungen auf Zahlenwerte

Das Ergebnis der Rechnung wird nach der Tabelle 2 auf ein Roboterkommando abgebildet. Nach dem optionalen Abbiegen findet normalerweise das Geradeausfahren bis zur nächsten Kreuzung statt. Allerdings gibt es dabei 2 Ausnahmen. Die erste tritt auf beim Hinweg, wenn der nachfolgende Knoten ein Fahrgast ist, in diesem Fall wird der Buchstabe “g”, anstatt “s” in das Kommandos-Array eingereiht. Die zweite tritt auf beim Rückweg, wenn der nachfolgende Knoten der letzte ist. Da der Fahrgast aber hinter dem letzten Knoten, also außerhalb der Karte abgeliefert werden muss, erfolgt an der Stelle eine Spezialbehandlung: der Roboter soll durch Abbiegen nach Süden ausgerichtet werden, wenn er es nicht bereits ist, danach wird der Fahrgast abgesetzt. Das Absetzen entspricht dem Buchstaben “d”.

Wert	Buchstabe
0	keine Richtungsänderung nötig
1	l
2	t
3	r

Tabelle 2: Abbildung der Rechenergebnisse auf Buchstaben

Nachdem der Hin -und Rückweg für alle erreichbaren Fahrgäste berechnet wurde, stehen die auszuführenden Kommandos fest und die Ampel zeigt grün. Sobald das Startlicht leuchtet, wird der Timer gestartet und die Kommandos in der executeCommands()-Methode werden nacheinander abgearbeitet. Vor Abarbeitung jedes Kommandos wird geprüft, ob die Zeit bereits abgelaufen ist. Wenn dies der Fall ist oder es keine abzuarbeitenden Kommandos mehr gibt, hält der Roboter an.

Abschließend wird die Implementierung der elementaren Roboterbefehle beschrieben.
Für das Geradeausfahren ist die Methode straightOnTillNextCrossing() verantwortlich. Als Erstes wird dabei geprüft, ob sich der Roboter bereits auf einer Kreuzung befindet. Wenn das der Fall ist, dann wird die Kreuzung überquert und danach bis zur nächsten Kreuzung gefahren. Beim Fahren wird ständig die Abweichung des Roboters von der schwarzen Linie mithilfe der Methode correctPathDeviation() korrigiert. Dies passiert, indem solange ein Optokoppler über der schwarzen Linie ist, wird der Motor auf dessen Seite langsamer.

Das Abbiegen besteht aus 3 Schritten und hat als Voraussetzung, dass die beiden Optokoppler über der schwarzen Linie sind. Hier erklärt anhand des Linksabbiegens:

1. Der Roboter dreht sich solange nach links bis sich der linke Optokoppler über dem weißen Untergrund befindet.
2. Der Roboter fährt 0,375 Sekunden nach vorne, damit sich auch der rechte Optokoppler über weißem Untergrund befindet.
3. Der Roboter dreht sich solange nach links bis sich der rechte Optokoppler über der schwarzen Linie befindet.

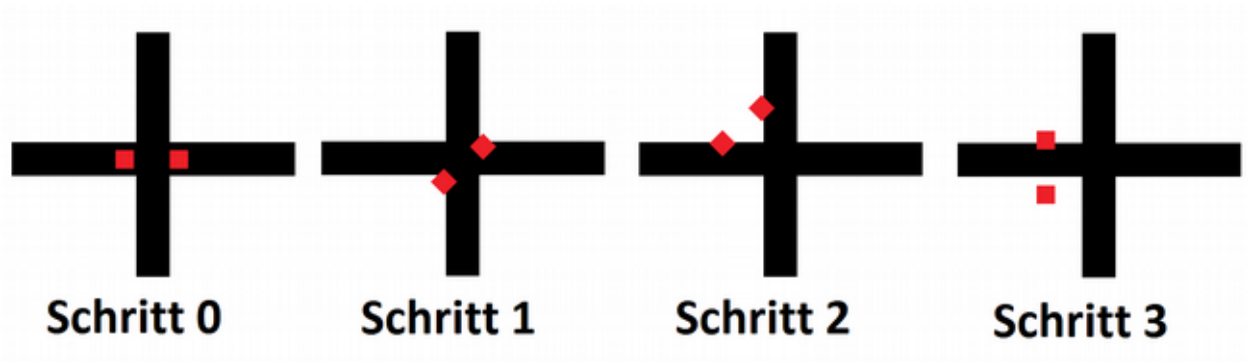

Abbildung 9: Skizze des Abbiegevorgangs

Das Greifen (grabPassenger()) und Absetzen (dropPassenger()) des Fahrgasts funktioniert sehr ähnlich. In beiden Fällen fährt der Roboter solange gerade aus bis der Bumper an eine Wand stößt. Danach klappt der Käfig zu bzw. öffnet sich. Daraufhin dreht sich der Roboter um 180° und fährt zurück bis zur Kreuzung. Die Drehung findet in der Methode turnAround() statt und funktioniert folgendermaßen. Zuerst dreht sich der Roboter eine Sekunde lang nach links, damit sich der rechte Optokoppler über weißem Untergrund befindet. Danach wird weitergedreht bis der rechte Sensor über schwarzem Untergrund ist.

Verworfene Ideen

In Abbildung 10 sieht man die erste Version des Greifers. Dieser wurde verworfen, weil er zu viel Präzisionsarbeit erfordern würde und die Befestigung des Servomotors zu umständlich war.

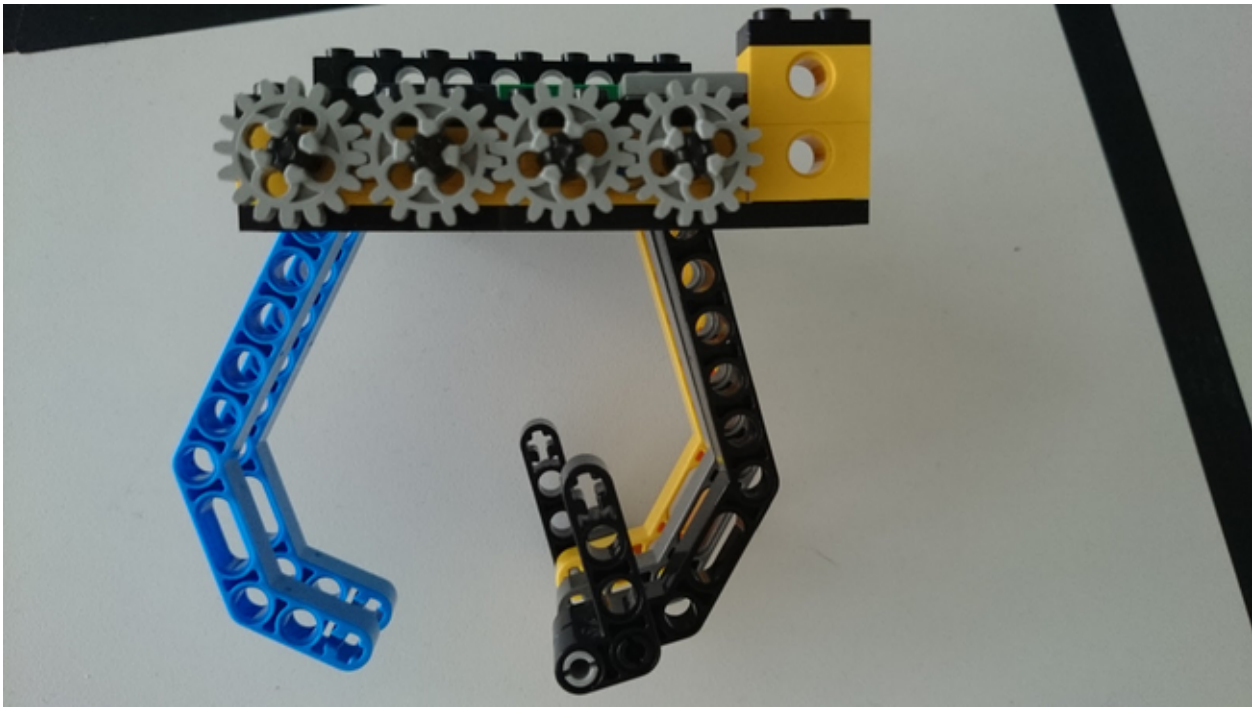

Abbildung 10: Erste Version des Greifers

Das Abbiegen bestand am Anfang aus 4 Schritten, wobei die ersten beiden Schritte identisch waren mit der aktuellen Umsetzung. Beim Linksabbiegen bestanden die beiden letzten Schritte daraus zuerst den Roboter solange nach links drehen zu lassen bis der linke Optokoppler über schwarzem Untergrund und danach bis dieser über weißem Untergrund war. Dabei kam es oft zu Fehlern.

Nach dem Greifen und Absetzen war die Idee den Roboter rückwärts fahren zu lassen bis zur vorherigen Kreuzung und dann eine 180° Drehung durchzuführen. Das wurde verworfen, weil ein Rückwärtsfahren ohne zusätzliche hintere Optokoppler zu ungenau war.

Um den Roboter nach 120 Sekunden zum Stehen zu bringen, wurde überlegt, dass ein Thread in regelmäßigen Abständen prüft, ob die Zeit schon abgelaufen ist und wenn dieser Fall eintritt den Main-Thread und danach sich beendet. Eine erste Version beeinflusste negativ das Fahrverhalten und es kam zu Stacküberläufen, deren Grund uns sich nicht erschließt. Eine weitere Einarbeitung wurde als zu zeitintensiv eingestuft.

Bewertung

Das Folgen der Linie funktioniert sehr gut. Weniger zuverlässig gestaltet sich ab und zu das abbiegen. Das Greifen ist zuverlässig, weil der Käfig groß und somit tolerant gegenüber Abweichungen ist. Da der Roboter relativ viel wiegt und eine große Übersetzung hat, ist er langsam. Die große Übersetzung hat aber den Vorteil, dass der Roboter auch bei geringer Akkuladung losfahren kann. Durch den großen Abstand zwischen den Rädern ist der Roboter weniger wendig.

Anhang

structures.h:

```
#ifndef STRUCTURES_H
#define STRUCTURES_H

   struct node{
      char character;
      char x;
      char y;
      char coveredDistance;
      char heuristicDistance;
      char totalCosts;
      struct node* previousNode;
   };

#endif
```

main.c:

```
#include "structures.h"
#define FA1
#include "fa.h"

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

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

#define EAST 0
#define NORTH 1
#define WEST 2
#define SOUTH 3
#define MAX_PASSENGERS 10
#define START_A map[1][9]
#define START_B map[5][9]

//Roboter Variablen
char currentDirection = NORTH;
char commands[360];
int commandsPos = 0;
struct node* pathArray[50];
char pathLength = 0;

unsigned char sensorLeft;
unsigned char sensorRight;
```

```
//Roboter Methodendeklarationen
void pathNodeToArray();
void calculateAllCommandsForwards();
void calculateAllCommandsBackwards();
void executeCommands();

void turnLeft();
void turnRight();
void printOptocouplerValues();
void correctPathDeviation();
void straightOnTillNextCrossing();
void stop();
void grabPassenger();
void dropPassenger();
void turnAround();

//A-Stern Variablen
struct node map [7][10];
struct node* openList [70];
char openListPos = 0;
struct node* closedList [70];
char closedListPos = 0;
struct node* startNode;
struct node* destinationNode;
unsigned char* mapString;
struct node* path;
struct node* passengers [MAX_PASSENGERS];

//A-Stern Methodendeklarationen
void stringToMap(unsigned char* mapString);
struct node* shortestPath(struct node* startNode, struct node* destinationNode);
void copyNode(struct node* to, struct node* from);
void expandOneDirection(char x, char y, struct node* previousNode);
void addNodeToList(struct node* list [], char* listPos, struct node* node);
void addNodeToSortedList(struct node* list [], char* listPos, struct node* node);
void moveNodesRight(char startPos, struct node* list [], char* listPos);
void removeFirstNodeFromOpenList();
char listContainsNode(struct node* list [], char listPos, struct node* node);
char calculateDistance(struct node* nodeA, struct node* nodeB);
void resetVariables();
char calculateDirection(struct node* startNode, struct node* destinationNode);
```

```
//Aksen-Main
void AksenMain(void){
   int i;
   servo_arc(1, 110);  //Käfig auf

   //Ampel rot
   led(0,1);

   //Fahrauftrag einlesen
   mapString = _fa;
   //Ampel gelb
   led(0,0);
   led(1,1);

   //Planung
   stringToMap(mapString);
   startNode = &START_A; //Anfangspunkt setzen
   for (i = 0; i < MAX_PASSENGERS; i++){
      if (passengers[i] != 0){
         destinationNode = passengers[i];
         path = shortestPath(destinationNode, startNode);
         if (path == startNode){
            pathNodeToArray();
            calculateAllCommandsForwards();
            calculateAllCommandsBackwards();
         }
         resetVariables();
      }
   }

   //Ampel grün
   led(1,0);
   led(2,1);

   //Warten auf Startlicht
   while(analog(8) > 50);

   //Kommandos ausführen
   led(2,0);        //Licht aus
   clear_time();    //Zeit starten
   motor_richtung(0,1); //linker Motor nach vorne
   motor_richtung(3,1); //rechter Motor nach vorne
   executeCommands();  //alle Kommandos ausführen
   stop();     //anhalten
}
```

```
//Kopieren des Inhalts einer Node zu einer anderen
void copyNode(struct node* to, struct node* from){
   to -> character = from -> character;
   to -> x = from -> x;
   to -> y = from -> y;
   to -> totalCosts = from -> totalCosts;
   to -> coveredDistance = from -> coveredDistance;
   to -> heuristicDistance = from -> heuristicDistance;
   to -> previousNode = from -> previousNode;
}

//String, der die Karte symbolisiert wird in eine 2D-Darstellung umgewandelt und Fahrgäste
extrahieren
void stringToMap(unsigned char* mapString){
   char stringPos = 69;
   int i, j, k = 0;
   struct node currentNode;

   for(i = 9; i >= 0; i--){
      for(j = 6; j >= 0; j--){
         //Node aus aktuellem Charakter erstellen und zur Map hinzufügen
         currentNode.character = mapString[stringPos];
         currentNode.x = j;
         currentNode.y = i;
         currentNode.totalCosts = currentNode.coveredDistance =
                                                          currentNode.heuristicDistance = 0;
         currentNode.previousNode = 0;
         copyNode(&map[j][i], &currentNode);
         stringPos--;

         //Fahrgäste extrahieren
         if(currentNode.character == 'F'){
            passengers[k++] = &map[j][i];
         }
      }
   }
}
```

```
//Es wird versucht den kürzesten Pfad zwischen 2 Punkten zu finden. Wenn dieser existiert,
dann wird dieser zurückgeliefert, sonst die zuletzt betrachtete Node (führt nicht zum Ziel).
struct node* shortestPath(struct node* startNode, struct node* destinationNode){
   //Initialisiere aktuelle Node und openList
   struct node* currentNode = startNode;
   currentNode ->heuristicDistance = calculateDistance(currentNode, destinationNode);
   currentNode ->totalCosts = currentNode->heuristicDistance;
   addNodeToList(openList, &openListPos, currentNode);

   while (openListPos > 0 && currentNode != destinationNode){
      //Kopiere expandierte Node in closedList
      addNodeToList(closedList, &closedListPos, openList[0]);
      //Lösche expandierte Node aus openList
      removeFirstNodeFromOpenList();

      expandOneDirection(currentNode ->x, currentNode ->y - 1, currentNode);
      expandOneDirection(currentNode ->x + 1, currentNode ->y, currentNode);
      expandOneDirection(currentNode ->x, currentNode ->y + 1, currentNode);
      expandOneDirection(currentNode ->x - 1, currentNode ->y, currentNode);

      //expandiere Node mit kleinsten Gesamtkosten, falls openList nicht leer
      if(openListPos > 0) currentNode = openList[0];
   }

   return currentNode;
}

/*Es wird geprüft, ob Node:
           innerhalb der Map,
           eine befahrbare Node oder das Ziel,
           nicht bereits in der closedList bzw. openList ist.
  Wenn ja, dann füge Node zur openList hinzu.
*/
void expandOneDirection(char x, char y, struct node* previousNode){
     if (x > -1 && x < 7 && y > -1 && y < 10 &&
       (map[x][y].character == '.' || &map[x][y] == destinationNode)){

        if (!listContainsNode(closedList, closedListPos, &map[x][y]) &&
        !listContainsNode(openList, openListPos, &map[x][y])){

           map[x][y].previousNode = previousNode;
           map[x][y].heuristicDistance = calculateDistance(&map[x][y], destinationNode);
           //1 = Kosten von Vorgängernode zu der neuen Node.
           map[x][y].coveredDistance = (map[x][y].previousNode) ->coveredDistance + 1;
           map[x][y].totalCosts= map[x][y].coveredDistance + map[x][y].heuristicDistance;
           addNodeToSortedList(openList, &openListPos, &map[x][y]);
        }
     }
}
```

```
//Die heuristische Distanz zwischen 2 Nodes wird berechnet
char calculateDistance(struct node* nodeA, struct node* nodeB){
   return abs(nodeA -> x - nodeB -> x) + abs(nodeA -> y - nodeB -> y);
}

//1 = Liste enthält Node, 0 = Liste enthält Node nicht
char listContainsNode(struct node* list [], char listPos, struct node* node){
   char i = 0;

   for(i; i < listPos; i++){
      if(list[i] == node){
         return 1;
      }
   }

   return 0;
}

//Der struct node-Pointer "node" wird ans Ende der Liste "list" angehangen
void addNodeToList(struct node* list [], char* listPos, struct node* node){
   list[(*listPos)++] = node;
}

//Node zur sortierten Liste an richtige Stelle hinzufügen
void addNodeToSortedList(struct node* list [], char* listPos, struct node* node){
   int i;

   for (i = 0; i < *listPos; i++){
      if ((*node).totalCosts < (*list[i]).totalCosts){
         moveNodesRight(i, list, listPos);
         (*listPos)++;
         list[i] = node;
         return;
      }
   }

   addNodeToList(list, listPos, node);
}

//Nodes der openList ab "startPos" werden um eine Stelle nach rechts geschoben
void moveNodesRight(char startPos, struct node* list [], char* listPos){
   int i;

   for(i = 0; *listPos - i > startPos; i++){
      int j = *listPos - i;
      int k = *listPos - i-1;
      list[j] = list[k];
   }
}
```

```
//Der struct node-Pointer der Liste "list" an der Position "nodePos" wird NULL gesetzt
void removeFirstNodeFromOpenList(){
   int i;
   openList[0] = 0;
   openListPos--;

   //Nodes der openList werden nach ganz links geschoben
   for(i = 0; i < openListPos; i++){
      int j = i + 1;
      openList[i] = openList[j];
   }
   openList[openListPos] = 0;
}

//Alle nötigen Variablen für die nächste Pfadsuche zurücksetzen
void resetVariables(){
   int i;

   //openList und closedList zurücksetzen
   for(i = 0; i < 70; i++){
      openList[i] = closedList[i] = 0;
   }

   //Listenpositionen zurücksetzen
   openListPos = closedListPos = 0;

   //Pfad zurücksetzen
   for(i = 0; i < 50; i++){
      pathArray[i] = 0;
   }
   pathLength = 0;
}

//Ausgehend von 2 benachbarten Nodes wird die nächste Richtung bestimmt. BEACHTE:
Reihenfolge der Parameter spielt eine Rolle
char calculateDirection(struct node* startNode, struct node* destinationNode){
   if (startNode ->x - destinationNode->x == 0){
      if(startNode ->y - destinationNode->y == 1){
         return NORTH;
      }
      else{
         return SOUTH;
      }
   }
   else{
      if(startNode ->x - destinationNode->x == 1){
         return WEST;
      }
      else{
         return EAST;
      }
   }
}
```

```
//Aus der Pfadnode ein Array erstellen
void pathNodeToArray(){
   char i = 1;
   pathArray[0] = path;
   for (i; path->previousNode != 0; i++){
      pathArray[i] = path->previousNode;
      path = path->previousNode;
   }
   pathLength = i;
}

//Kommandos für den Hinweg berechnen
void calculateAllCommandsForwards(){
   int i;
   for(i = 0;  i < pathLength -1; i++){
      int j = i+1;
      char targetDirection = calculateDirection(pathArray[i], pathArray[j]);
      switch( ((targetDirection - currentDirection)+4 ) % 4 ){
         case 1: commands[commandsPos++] = 'l'; break; //links abbiegen
         case 2: commands[commandsPos++] = 't'; break; //180° Drehung
         case 3: commands[commandsPos++] = 'r';        //rechts abbiegen
      }
      currentDirection = targetDirection;

      if(pathArray[j]->character == 'F'){
         commands[commandsPos++] = 'g';   //Fahrgast greifen
         currentDirection = (targetDirection + 2) % 4;
      }
      else{
         commands[commandsPos++] = 's'; //gerade aus
      }
   }
}

//Kommandos für den Rückweg berechnen
void calculateAllCommandsBackwards(){
   int i;
   for(i = 0; pathLength -2-i >= 1; i++){
      int k = pathLength -2-i;
      int j = pathLength -3-i;
      char targetDirection = calculateDirection(pathArray[k], pathArray[j]);
      switch( ((targetDirection - currentDirection)+4 ) % 4 ){
         case 1: commands[commandsPos++] = 'l'; break; //links abbiegen
         case 2: commands[commandsPos++] = 't'; break; //180° Drehung
         case 3: commands[commandsPos++] = 'r';        //rechts abbiegen
      }

      commands[commandsPos++] = 's'; //gerade aus
      currentDirection = targetDirection;

      if(pathArray[j]->x == startNode->x && pathArray[j]->y == startNode->y){
         if(currentDirection == EAST){
            commands[commandsPos++] = 'r';
         }
```

```
        else if(currentDirection == WEST){
           commands[commandsPos++] = 'l';
        }
        commands[commandsPos++] = 'd';   //Fahrgast absetzen
        currentDirection = NORTH;
     }
   }
}

//Kommandos ausführen und Roboter anhalten, wenn Zeit abgelaufen ist
void executeCommands(){
   int i;
   for (i = 0; i < commandsPos; i++){
      if(akt_time() >= 120000){
         return;
      }
      switch (commands[i]){
         case 's': straightOnTillNextCrossing(); break;
         case 'l': turnLeft(); break;
         case 'r': turnRight();  break;
         case 't': turnAround(); break;
         case 'g': grabPassenger(); break;
         case 'd': dropPassenger();
      }
   }
}

void turnLeft(){
   motor_richtung(0,0);
   motor_pwm(0,10);
   motor_pwm(3,10);

   //abbiegen bis linker Sensor auf weiß
   if (analog(8) < 51){
      sleep(375);
   }
   else{
      while (analog(0) > 50);
   }

   //geradeaus bis rechter Sensor auf weiß
   motor_richtung(0,1);
   sleep(375);

   //nach links drehen bis rechter Sensor schwarz
   motor_richtung(0,0);
   while (analog(7) < 51 );

   //Motor wieder in Ausgangsrichtung (nach vorne) setzen
   motor_richtung(0,1);
}
```

```
void turnRight(){
   motor_richtung(3,0);
   motor_pwm(0,10);
   motor_pwm(3,10);

   //abbiegen bis rechter Sensor auf weiß
   if (analog(8) < 51){
      sleep(375);
   }
   else{
      while (analog(7) > 50);
      sleep(100); //noch etwas weiter nach rechts abbiegen
   }


   //geradeaus bis linker Sensor auf weiß
   motor_richtung(3,1);
   sleep(375);

   //nach rechts drehen bis linker Sensor schwarz
   motor_richtung(3,0);
   while (analog(0) < 51 );

   //Motor wieder in Ausgangsrichtung (nach vorne) setzen
   motor_richtung(3,1);
}

void printOptocouplerValues(){
   lcd_cls();

   lcd_setxy(0,0);
   lcd_puts("SensorLeft: ");
   lcd_ubyte(sensorLeft);

   lcd_setxy(1,0);
   lcd_puts("SensorRight: ");
   lcd_ubyte(sensorRight);
}

void correctPathDeviation(){
   if(sensorLeft > 50){
      motor_pwm(0,3);
   }
   else{
      motor_pwm(0,10);
   }

   if(sensorRight > 50){
      motor_pwm(3,3);
   }
   else{
      motor_pwm(3,10);
   }
}
```

```
void straightOnTillNextCrossing(){
   motor_pwm(0,10);
   motor_pwm(3,10);
   sensorLeft = analog(0);
   sensorRight = analog(7);

   //Wenn Roboter genau auf der Kreuzung steht, dann Kreuzung überfahren
   while(sensorLeft > 200 && sensorRight > 200){
      sensorLeft = analog(0);
      sensorRight = analog(7);

      printOptocouplerValues();
   }

   //Nachdem Roboter über Kreuzung gefahren, bis zur nächsten Kreuzung fahren
   while(1){
      sensorLeft = analog(0);
      sensorRight = analog(7);

      printOptocouplerValues();

      //Kreuzung erkennen
      if(sensorLeft > 200 && sensorRight > 200){
         return;
      }

      correctPathDeviation();
   }
}

void stop(){
   motor_pwm(0,0);
   motor_pwm(3,0);
}

void grabPassenger(){
   motor_pwm(0,10);
   motor_pwm(3,10);

   //Bis zur Wand fahren, Fahrgast greifen, umdrehen, zur Kreuzung fahren
   while(1){
      sensorLeft = analog(0);
      sensorRight = analog(7);
      printOptocouplerValues();

      //Wand erkennen mit Bumper
      if (digital_in(0) == 0){
         stop();                        //Motoren ausschalten
         servo_arc(1, 170);               //Fahrgast greifen
         sleep(100);                      //kurz warten

         //umdrehen und bis zur Kreuzung fahren
         turnAround();
```

```
         straightOnTillNextCrossing();

         return;
      }

      correctPathDeviation();
   }
}

void dropPassenger(){
   motor_pwm(0,10);
   motor_pwm(3,10);

   //Bis zur Wand fahren, Fahrgast greifen, umdrehen, zur Kreuzung fahren
   while(1){
      sensorLeft = analog(0);
      sensorRight = analog(7);
      printOptocouplerValues();

      //Wand erkennen mit Bumper
      if (digital_in(0) == 0){
         stop();                        //Motoren ausmachen
         servo_arc(1, 110);               //Fahrgast absetzen
         sleep(100);                      //kurz warten

         //umdrehen und bis zur Kruezung
         turnAround();
         straightOnTillNextCrossing();

         return;
      }

      correctPathDeviation();
   }
}

void turnAround(){
   motor_richtung(0,0);
   motor_richtung(3,1);
   motor_pwm(0,10);
   motor_pwm(3,10);

   //zeitlich drehen damit beide Sensoren auf weiß sind
   sleep(1000);

   //solange drehen bis rechter Sensor auf schwarz
   while (analog(7) < 51 );

   //Motor wieder in Ausgangsrichtung (nach vorne) setzen
   motor_richtung(0,1);
}
```