Dokumentation zum AMS - Projekt MASDAR City - Personal Rapid Transit

Tobias Franz, Franziska Krebs
20100047, 20101552

20. Januar 2013

Inhaltsverzeichnis

1 Aufgabenstellung und Motivation

Ziel des Projekts ist die Konstruktion eines Roboters und die Implementierung eines Suchverfahrens im Sinne einer Routenberechnung. Nachempfunden ist diese Idee dem Personal Rapid Transit (PRT) - System. Dabei handelt es sich ein Personentransportsystem, welches ohne Fahrer und ohne Fahrplan Personen mittels Transportkabinen über ein Schienennetz ans Ziel bringt. Das Ein - und Aussteigen ist dabei nur an Haltestellen möglich, die bei vielen Systemen auf Nebengleisen gelegen sind.

Abbildung 1: PRT - Kabine *ULTra* am London Heathrow Airport
[1]

Der Namensgeber dieses Projekts ist *Masdar City* - eine geplante Ökostadt in Abu Dhabi, deren Verkehr durch ein 33,1 km langes PRT -System umgesetzt werden soll. Momentan kommen 10 Kabinen probeweise zum Einsatz, bis zum Jahre 2015 sollen 2500 Kabinen zur Verfügung stehen. [2] [1]
Dem Original nachempfunden befinden sich die Haltestellen an den Rändern. Hindernisse bzw. unbefahrbare Kreuzungen werden auf dem Feld durch rote Bälle repräsentiert, Fahrgäste durch blaue Bälle.

2 Lösungsweg

Im Folgenden werden Ideen vorgestellt, die die bauliche Umsetzung des Roboters betreffen, dargestellt. Unter dem Punkt *Hardware* erfolgt eine genaue Vorstellung des fertig gestellten Roboters.
An die Konstruktionsphase schloss sich die Programmierung des Suchverfahrens an, die unter dem Punkt *Software* genauer betrachtet wird.
Die folgende Graphik zeigt den zeitlichen Ablauf des Projekts.

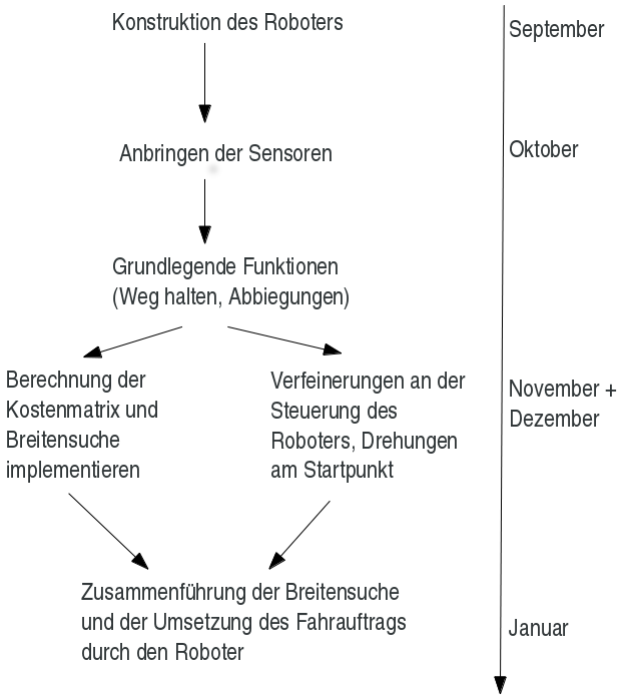


Abbildung 2: zeitliche Achse

Unsere zu Beginn festgelegten Ziele für die Konstruktion des Roboters umfassten das Entwerfen eines stabilen Gerüsts, verbunden mit einem kraftvollen und zugleich agilen Getriebe.

Bei der ersten Designentscheidung wählten wir die Unterbringung von Aksen-Board und Akkumulator. Dabei befestigten wir ersteres durch einen Bügel auf dem Roboter, die Batterie wird unterhalb des Boards in einem Lego-Gitter gehalten.

Nach einigem Herumprobieren mit den Lego-Steinen fiel die Entscheidung auf einen Differentialantrieb, da dieser einfach zu konstruieren ist und eine Drehung um die eigene Achse möglich ist. Im Verlaufe der Programmierung des Roboters zeigte sich ein Nachteil des Differentialantriebs: eine Kurvenfahrt dauerte deutlich länger, als erwartet. Da der Roboter sich sonst sehr flink über das Feld bewegen konnte, sahen wir über diesen Aspekt hinweg.

Die beiden Räder des Roboters liegen auf einer Achse, werden aber getrennt durch 2 Lego – Motoren angetrieben. Die Antriebsachse befindet sich am Kopf des Roboters, sodass weiterhin die Entscheidung getroffen werden musste, ob ein drittes Rad oder ein Alternative zum Nachlauf genutzt wird. Zunächst fiel die Wahl auf einen Ball, der in einem Gitter gehalten werden sollte. Dies bereitete uns jedoch erhebliche Probleme, da nun eine deutliche stärkere Reibung auf dem Boden verursacht wurde und dies die Geschwindigkeit minderte.

Die Idee, ein Nachlaufrad zu verwenden, musste aufgrund von Ressourcenknappheit verworfen werden. Letztendlich wurde ein Art Stütze angebracht, die das Problem der starken Reibung beseitigte und aufgrund des leichten Gewichts des Roboters die Geschwindigkeit kaum negativ beeinflusste.

Der bis zu diesem Zeitpunkt noch namenlose Roboter wurde durch zahlreiche kleinere Umbauarbeiten (Veränderungen am Getriebe) entschlackt und erhielt aufgrund der verbleibenden gelb- und schwarz-

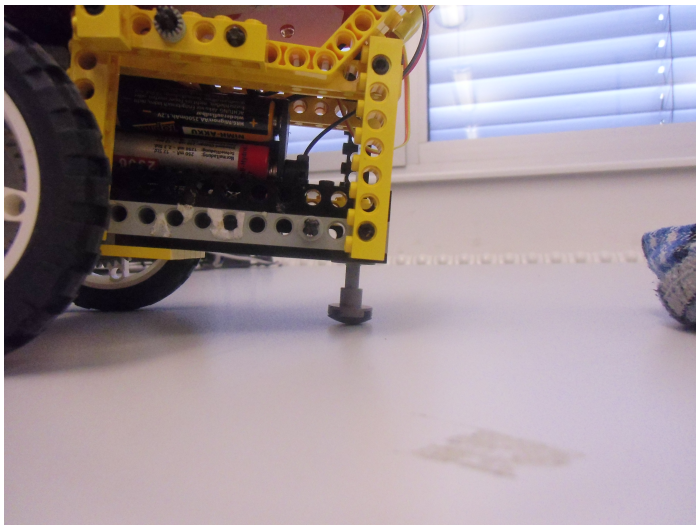


Abbildung 3: Nachlauf des Roboters

gefärbten Steine den Namen „Crawling Horned“.

Das nächste Hindernis eröffnete sich bei der Erweiterung des Roboters um eine Fangkonstruktion für die Passagiere. Von Beginn an sollte dies durch einen Fangbügel, der durch einen Servomotor gesteuert wird, realisiert werden. Zum Erkennen, ob sich ein Passagier in Greifweite befindet, sollte eine Lichtschranke, bestehend aus einem Infrarotsender und -empfänger eingesetzt werden. Um den Servomotor unterzubringen, musste der Roboter nach vorne hin um eine Platte als Befestigungspunkt erweitert werden. Der Fangbügel wurde zu Beginn zu groß gewählt, sodass er beim Einfangen oft durch die hinter den Passagieren gelegene Wand behindert wurde. Zusätzlich mussten sich die Infrarotsensoren innerhalb des Fangbügels befinden, sodass sich dieser beim Einfangen an den Sensoren vorbei bewegen kann. Durch den zunächst gestreckten parabelförmigen Bügel musste der Abstand zwischen den Sensoren entsprechend gering gehalten werden. Dies führte beim Fangen dazu, dass die Bälle meist von ihrer Station gestoßen wurden.

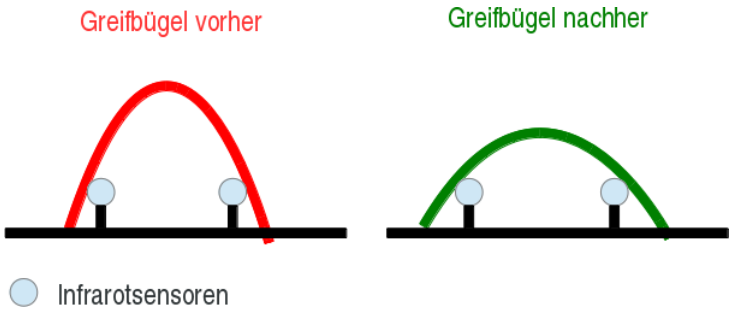


Abbildung 4: Veranschaulichung der Fangvorrichtung

Daraufhin haben wir uns entschlossen, die Form des Bügels so zu stauchen, dass der Abstand zwischen den Sensoren großzügig genug gewählt werden konnte, um die Bälle kollisionsfrei zu fangen.

3 Vorstellung

Bei der Konstruktion des Roboters gingen wir nach dem sogenannten “KISS“ Prinzip vor. KISS bedeutet soviel wie “ keep it small and simple“. So verbauten wir nur die notwendigsten Sensoren und Elemente, um eine leichte Bauweise und eine unkomplizierte Steuerung zu erhalten. Durch eine Entschlackung haben wir auch unnötige Bausteine entfernt, was zu einer erneuten Gewichtsreduzierung führte und ihm sein charakteristisches Äußeres gab.

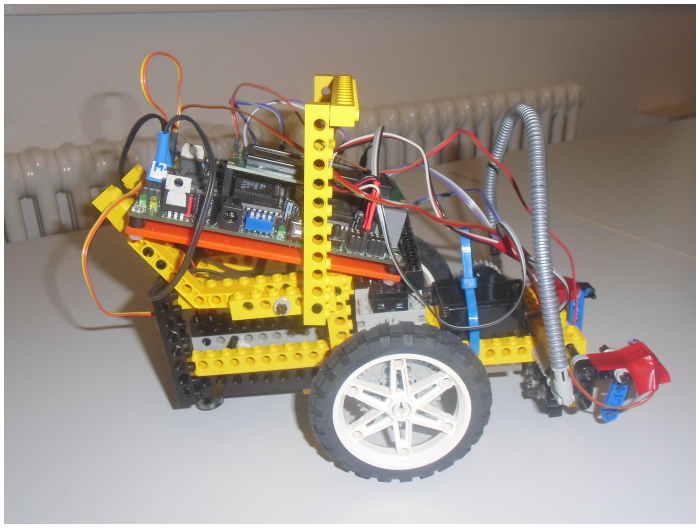


Abbildung 5: Roboter: “Crawling Horned“

Besonders gelungen ist das Drehverhalten. Schnell, aber präzise fährt die “Crawling Horned“ um jede Kurve. Auch die Breitensuche ist eine Erwähnung wert. Ohne Zeitverlust werden die korrekten Wege berechnet und zu einer großen Fahrsequenz vereint.
Erhebliche Probleme wurden durch die Drehung versucht, die der Roboter nach dem Ablegen des Balles am Startpunkt vollführt. Immer noch etwas wacklig auf den Rädern schafft er diese jedoch. Der Bügel, den wir zum Fangen des Balles einsetzen, machte ebenfalls Schwierigkeiten. So stieß er immer wieder an die Wand des Spielfeldes, was so viel Reibung verursachte, dass ein Wenden kaum möglich war. Dieses Problem wurde einerseits durch die bereits erwähnte Stauchung des Bügels, als auch durch ein kurzes Zurücksetzen des Roboters gelöst.

4 Hardware

4.1 Antrieb

Angetrieben wird die “Crawling Horned“ über einen Differentialantrieb. Das bedeutet, das zwei unabhängig voneinander laufende Motoren jeweils ein Rad antreiben. Ein Rad erhält die Kraft eines ganzen Motors. Zudem verbauten wir eine kleines Getriebe, was durch die große Kraft der eingesetzten Lego Motoren möglich war und das Gewicht und die Länge gering hielt. Durch diese Bauweise verschwindet relativ wenig Kraft durch die Reibung eines großen Getriebes, weswegen auch der Stromverbrauch des Antriebs gering ist. Sogar wenn die Räder blockiert werden, ist der Verbrauch im grünen Bereich. Bei der Wahl der Räder haben wir uns für schmale Exemplare mit einem hohen Durchmesser entschieden, da so die Reibung minimiert wird und bei einer einzigen Umdrehung mehr Strecke zurückgelegt wird.

4.2 Wendigkeit und Steuerung

Der Roboter überzeugt mit seiner hohen Geschwindigkeit, dem sicheren Wenden und Überfahren von Kreuzungen. Auch bei halber Kraft bekommen wir eine überzeugende Geschwindigkeit. Allerdings gibt es zunehmend Probleme, wenn wir die Kraft erhöhen. Die “ Crawling Horned ”’wird dann zunehmend unkontrollierbar, da sie einfach viel zu schnell ist. Eine Kurvenfahrt gleicht dann mehr einem “ Drift “, als einer kontrollierten 90° Drehung. Deswegen läuft der Roboter auch mit mehr als halber Kraft, denn das gewähprogrammingrleistet eine hohe Wendigkeit sowie eine optimale Steuerbarkeit und ist schnell genug, um die Aufgabe innerhalb der geforderten Zeit zu meistern.
Um eine saubere Kurvenfahrt zu realisieren,wird ein Rad angehalten und das andere angetrieben, bis der entsprechende Optokoppler (siehe Sensoren) wieder eine schwarze Linie erkennt. Durch den Schwung, welchen der Roboter während der Drehung aufgebaut hat, rutscht er über die schwarze Linie und richtet sich wieder korrekt aus. Ein Problem, welches bei Kurvenfahrten auftrat, war, dass das angehaltene Rad nicht vollständig stehen blieb, sondern von dem anderen Rad mitgezogen wurde. Unsere Lösung bestand darin, das entsprechende Rad mit minimaler Kraft zurückdrehen zu lassen - so “ krallte “ es sich fest und wurde nicht mehr mitgezogen.

4.3 Sensoren

Um eine sichere Fahrt und eine gute Erkennung der Kreuzungen zu gewährleisten verbauten wir 2 Optokoppler weit vorne am Roboter.

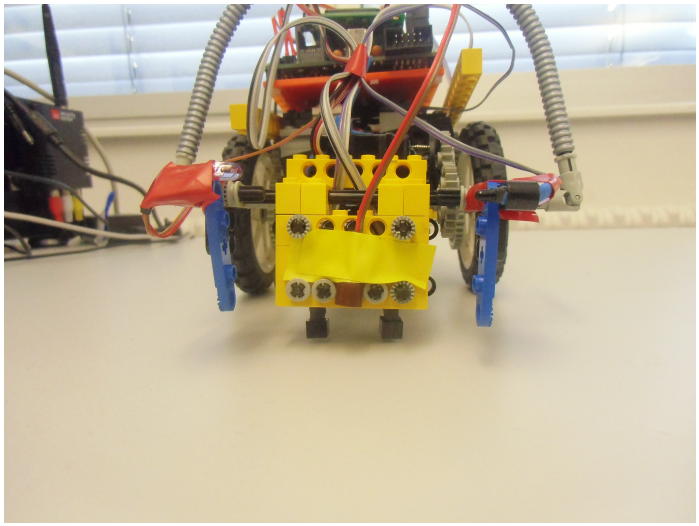


Abbildung 6: Darstellung der Optokoppler

Das sichere Streckenverhalten erhält der Roboter durch das Zusammenspiel der Optokoppler. Sobald einer der Optokoppler schwarz registriert, wird ein Rad gestoppt, während das andere weiter angetrieben wird. Das veranlasst den Roboter zu einer leichten Drehung, die solange fortgesetzt wird, bis der entsprechende Optokoppler wieder weiß registriert.

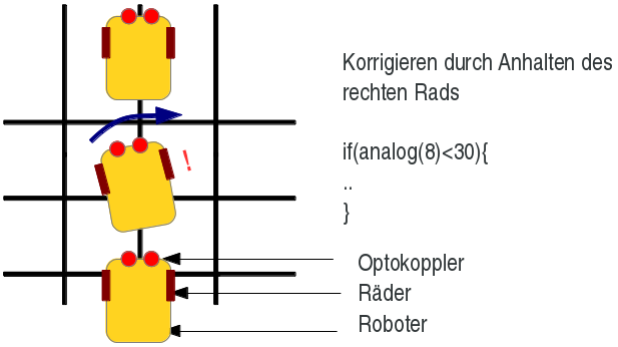


Abbildung 7: Korrigierverhalten des Roboters

Für das Erkennen eines Fahrgastes wurde eine Infrarot-Schranke benutzt. Eine Infrarot LED sendet dabei ein kontinuierliches Signal an einen Sensor. Solange er dieses Signal erhält, geschieht nichts. Erst, wenn ein Gegenstand das Signal blockiert, wird der Fangbügel aktiviert und der Fahrgast eingesammelt. Diese Schranke wird allerdings erst funktionsfähig, sobald der Roboter bemerkt, das er kurz vor einer Kugel ist. Auf dem Hin- und Rückweg ist sie inaktiv, alles andere wäre Stromverschwendung.

4.4 Besonderheiten

Eine der größten Besonderheiten ist die Konstruktion des Fangbügels. Solch ein Bügel war einfach an unserem Roboter zu installieren und er hat wenig Gewicht, damit der schwache Servomotor nicht überlastet wird. Auch optisch passt dieser Bügel gut zum charakteristischen Aussehen.
Die größte Schwierigkeit war es, den Bügel zu strecken, um der Lichtschranke mehr Platz zu geben und das Berühren der Spielfeld- Wand zu verhindern.

4.5 Anordnung der Komponenten

Zum Entlasten der Antriebsachse verlegten wir den Schwerpunkt nach hinten. Der Akkumulator sitzt in einem Fach im Heck der “ Crawling Hornet “ und drückt diese nach unten. Das Aksenboard liegt in einer Halterung auf dem Roboter und drückt die hintere Stütze nach unten und gibt den Rädern eine gute Traktion. Weiter vorne sitzt der Antrieb und zieht den Roboter hinter sich her. Daran angeschlossen ist der Servomotor mit dem Fangbügel. Die Front der Konstruktion beinhaltet die Infrarotschranke. Sender und Empfänge der Schranke wurden relativ weit auseinander installiert, um genug Platz zur Erkennung des Balls zu gewährleisten.

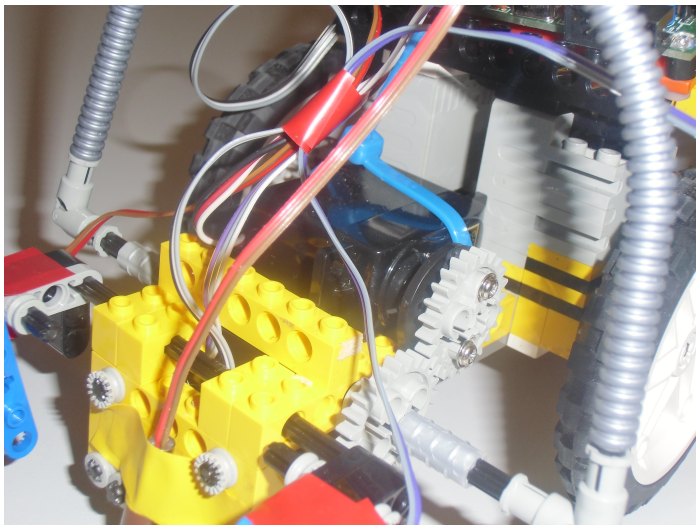


Abbildung 8: Anbringung des Servomotors

5 Software

5.1 Implementierung des Suchverfahrens

Um den Weg aus dem gegebenen Feld zu bestimmen, entschieden wir uns für die Implementierung einer Breitensuche, die zunächst die aus dem String vorstellbare Matrix mit Kosten für jeden Knoten (bzw. jede Kreuzung) vom Startpunkt aus füllte.

Der folgende Ausschnitt zeigt die Darstellung des Feldes als String im Programm. Die Abbildung visualisiert die daraus resultierende 7x10er Matrix.

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

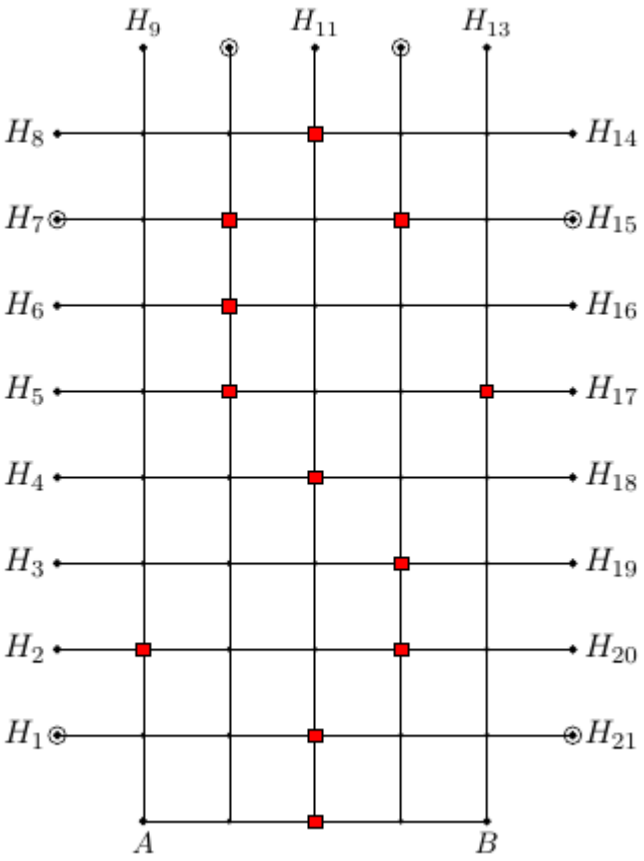


Abbildung 9: Darstellung des Fahrauftrags

Ein 'x' steht dabei für eine nicht befahrbare Kreuzung, ein 'F' stellt einen Fahrgast dar, der auf seine Abholung wartet.

Füllen der Kostenmatrix

Zunächst werden für die jeweilige Startposition die Kosten 0 im Feld eingetragen. Dieser Startpunkt wird anschließend in die Agenda übernommen. Anschließend beginnt die eigentliche Expandierung der Knoten, die durch eine while-Schleife gesteuert ist.

Betrachten wir das Anfangsszenario, so ist die Agenda gefüllt mit dem Index des Startknotens. Neben dem Index wird weiterhin die Anzahl der bisher in diesem Durchlauf aufgenommenen Knoten hinzugefügt. Können beispielsweise vom Startpunkt aus 2 Knoten erreicht werden, wird für den ersten Knoten der Index und eine '0' in die Agenda eingetragen. Der zweite Knoten wird mit einer '1' aufgenommen. Dadurch soll sicher gestellt werden, dass bei einer Expandierung die Kosten erst dann inkrementiert werden, wenn die in einem Durchgang hinzugefügten Knoten abgearbeitet wurden. Ist dies nicht der Fall, werden die Kosten dekrementiert.

```
if(agenda[0][1]!=0){
  costs--;
}
```

Nach der Aufnahme des Startknotens werden die Kosten um 1 hochgezählt. Der Algorithmus überprüft nun, ob dieser Knoten Nachbarpunkte hat, die erreicht werden können. Liefert diese Abfrage eine "1", befindet sich in mindestens einer der Richtungen ein '.'. Gleichzeitig führt die Abfrage dazu, dass ein eindimensionales Array mit 4 Einträgen entweder mit Nullen oder Einsen gefüllt wird.

Ausgangszustand des Arrays:

```
char nachbarn[] = "0000";
```

Findet die Methode einen erreichbaren Punkt im Norden, wird an die erste Stelle eine '1' geschrieben. Befindet sich an dieser Stelle ein Fahrgast, eine blockierte Kreuzung oder wurde dieses Feld schon mit Kosten versehen und beinhaltet folglich eine Zahl, wird die erste Stelle mit einer '0' gefüllt. Die zweite Stelle des Arrays gibt an, ob sich im Westen ein zu expandierender Punkt befindet. Dementsprechend repräsentieren der 3. und 4. Eintrag die Himmelsrichtungen Süden und Osten.

Hat unser Startpunkt nun beispielsweise im Norden und im Westen einen erreichbaren Punkt, liefert die Methode als Rückgabewert eine '1' und das Nachbar-Array ist wie folgt gefüllt:

```
1 1 0 0
```

Anschließend werden die entsprechend erreichbaren Punkte mit den Kosten gefüllt. Für unseren Startpunkt mit dem Index 68 ergibt sich nach dem 2.Durchgang folgende Darstellung:

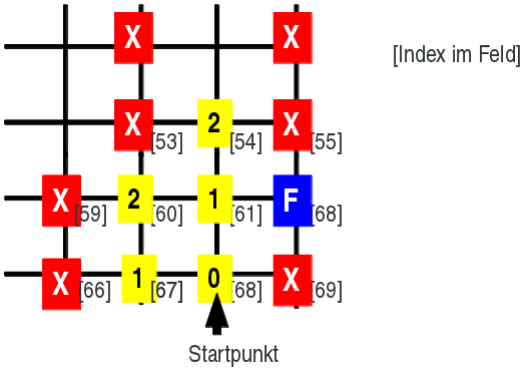


Abbildung 10: Kostenmatrix nach der Expandierung von 2 Knoten

Nach jedem Durchlauf wird der expandierte Punkt aus der Agenda gelöscht. Die while-Schleife bricht ab, wenn sich kein Punkt mehr in der Agenda befindet. Die gefüllte Matrix sieht wie folgt aus:

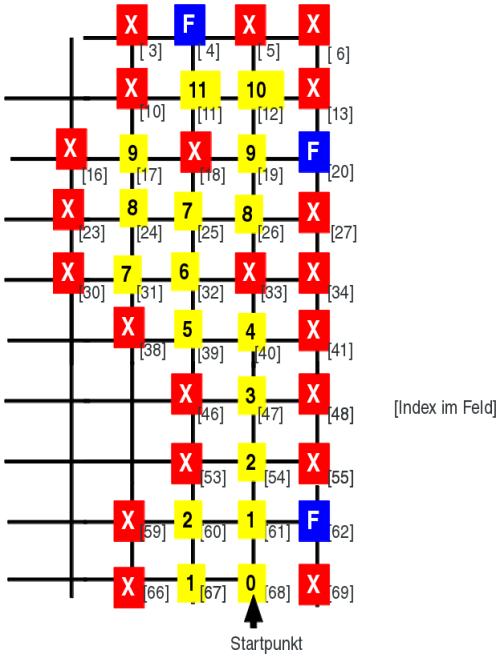


Abbildung 11: Gefüllte Kostenmatrix

Es wird hierbei auf die Darstellung der linken Seite des Feldes verzichtet, da diese nicht gefüllt wird. Das Feld wird mittig durch gesperrte Kreuzungen geteilt, wodurch die Punkte der linken Seite nicht in die Agenda aufgenommen werden.

Finden des Weges
Im nächsten Schritt werden die Zielknoten in einem Array gespeichert. Dafür wird das Feld komplett durchlaufen und es werden diejenigen Knoten in das Array übernommen, die ein 'F' enthalten und erreichbar sind. Erreichbar bedeutet hier, dass der Punkt in mindestens einer Himmelsrichtung eine Zahl

enthält.

Laut Aufgabenstellung soll der Roboter die Abholung von 3 Passagieren beherrschen. Ist das Feld offen gestaltet, werden zunächst alle Zielknoten aufgenommen. Es ist durchaus vorstellbar, dass lediglich ein oder 2 Passagiere gefunden werden. Im nächsten Schritt wird die Anzahl der enthaltenen Zielknoten bestimmt. Überschreitet diese 3, werden die restlichen Knoten ignoriert. Eine Unterschreitung stellt keine Probleme dar.

Es stehen insgesamt 6 Arrays für die "Route" zur Verfügung, davon enthalten jeweils 3 die Hin- und Rückwege. Wir betrachten nun den ersten Zielknoten. Dieser wird an die erste Stelle des ersten Array geschrieben. Von diesem Punkt aus wird nun der Nachbarpunkt ausgewählt, der mit den niedrigsten Kosten gekennzeichnet ist. Hierfür wird zunächst der nördliche Punkt betrachtet. Enthält dieser eine Zahl, wird diese in einem Platzhalter gespeichert und der Index dieses Knotens wird zunächst als Rückgabewert der Methode gemerkt. Nacheinander werden die restlichen Richtungen betrachtet. Sollte es ein Feld geben, dessen Kosten niedriger sind als der in der Variable gespeicherte Wert, wird diese überschrieben und der entsprechende Knoten wird als neuer Rückgabewert gespeichert.

Konnte ein Punkt gefunden werden, wird dieser an die nächst freie Stelle des ersten Arrays geschrieben. Gleichzeitig wird der Index dieses Knotens zurückgegeben. Die Methode wird so oft aufgerufen, bis sie als Rückgabewert einen Feldindex liefert, dessen Kosten Null betragen. Dies bedeutet, dass wir den Startpunkt erreicht haben.

Dieser gesamte Ablauf wird wiederholt, bis das Array mit den Zielknoten leer ist bzw. die Maximalanzahl von 3 Passagieren erreicht ist.

Betrachten wir unser Beispielfeld und nehmen als Startposition den Index 68 an, ergeben sich folgende Belegungen für die Arrays:

```
erstes Array : 4, 11, 12, 19, 26, 25, 32, 39, 40, 47, 54, 61, 68
zweites Array: 20, 19, 26, 25, 32, 39, 40, 47, 54, 61, 68
drittes Array: 62, 61, 68
```

Jedes Array kann 20 Indices speichern, wobei die nicht genutzten letzten Felder mit einem 'a' gefüllt werden.

Vom Index zum Fahrtweg

Im nächsten Schritt werden die 3 obigen Arrays in jeweils eine Route transformiert. Dafür wird vor dem Betrachten eines Arrays die Blickrichtung des Roboters auf Norden festgelegt. Vom letzten Index (dem eigentlichen Startpunkt) ausgehend wird nun die Differenz zum vorherigen Index berechnet. Diese würde für unser erstes Beispielarray 68-61 = 7 betragen. Steht dieser Abstand zum Vorgänger fest, wird die Blickrichtung betrachtet. Eine Differenz von 7 und ein nach Norden ausgerichteter Roboter bedeuten, dass die Kreuzung mit dem Index 68 überfahren werden soll.

Bei einem Blick auf unser Beispielfeld wird dies deutlich:

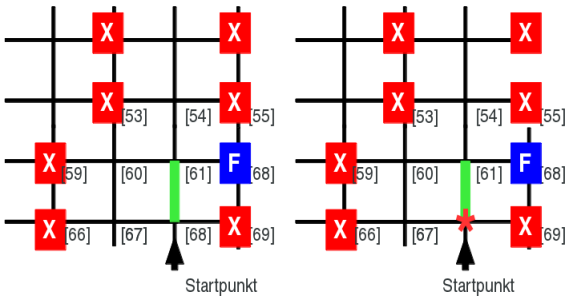


Abbildung 12: Matrixdarstellung mit Indices (links) und mit Fahranweisung (rechts) nach dem ersten Durchlauf

Eine Differenz von 7 entspricht einer Kreuzung, die über dem aktuell betrachteten Punkt gelegen ist. Der Roboter verändert seine Blickrichtung dabei nicht. Daher wird unser Startindex mit einem '*' überschrieben. Der Algorithmus betrachtet nun nacheinander die Differenz von einer Kreuzung mit dem

Index n zu einer Kreuzung mit dem Index n-1. Diese kann entweder 7, -7, 1 oder -1 betragen, je nachdem, ob der Roboter geradeaus nach oben, geradeaus nach unten, rechts abbiegen oder links abbiegen soll. In Abhängigkeit von der Richtung, in die der Roboter gerade schaut, werden neue Blickrichtungen festgelegt. An die erste Stelle der Arrays wird jeweils ein 'Z' eingetragen. Dies interpretiert der Roboter später als 'Ziel erreicht, Passagier einsammeln'.

Dieser Vorgang wird für alle 3 Arrays durchgeführt. Konnten weniger als 3 Zielpunkte gefunden werden, verändert sich die Anzahl der zu betrachtenden Arrays dementsprechend.

Für eine schematische Darstellung dieser Methode siehe „berechneFahrtweg() “auf Seite 40.

Nach der Ausführung dieser Methode ergeben sich folgende Arrays:

```
erstes Array  = Z R L * L R * R L * * * * a a a a a a a
zweites Array = Z R L R * R L * * * * a a a a a a a a a
drittes Array = Z R * a a a a a a a a a a a a a a a a a
```

Auch wenn wir nun die Fahrtwege zu den einzelnen Passagieren haben, so haben wir noch nicht die wirkliche Fahrtsequenz für den Roboter, sondern nur die Information, welche Drehung er an welchem Punkt ausführen soll. Die Methode “ berechneHinweg() “ wandelt nun den die Fahrtwege zu der entsprechenden Sequenz um.// Insgesamt haben wir 6 Arrays für die Wege reserviert. Der Hinweg wird in die 3 noch freien Arrays geschrieben.

```
ArrayIndex = Aktuelles Array+3;
```

Die momentanen Fahrwege führen vom Ziel zum Startpunkt, doch was man braucht sind die Wege vom Start zum Ziel. Um dieses Problem zu lösen, lassen wir zunächst die Länge eines Fahrtweges berechnen und nutzen diese um die Fahrtwege umgekehrt in die freien Arrays zu schreiben.
Sobald dies geschafft ist, prüfen wir die letzte Drehung vor dem Zielknoten “ Z “, denn sobald der Roboter das Ziel erreicht hatte musste er sich wieder in die Ausgangsposition bringen. War die letzte Drehung eine Rechtsdrehung, so muss auch nach dem Zielknoten wieder einer Rechtsdrehung erfolgen, bei einer Linksdrehung dementsprechend eine Linksdrehung und wenn es keine Drehung davor gab, muss der Roboter gerade aus ins Ziel gekommen sein und danach eine 180 ° vollführen. Am Ende des Algorithmus werden die Arrays, welche den umgekehrten Fahrtweg enthielten, bereinigt und alle Elemente werden mit “a“ überschrieben.
Deutlicher wird dies an einem Beispiel. Nehmen wir den kleinsten Fahrtweg im dritten Array.

```
drittes Array: Z, R,*
```

Der erste Schritt des Algorithmus berechnet die Größe des Fahrtweges, also 3. Nun Wird der Weg umgekehrt in das 3. freie Array geschrieben, also in das 6. Array.

```
[6][0] = *;
[6][1] = R;
[6][2] = Z;[]
```

Als Nächstes wird die letzte Drehung vor dem Zielpunkt ermittelt. Zur Ermittlung dieser Drehung gucken wir einfach 2 Elemente zurück und finden “ R “. Bei einer Rechtsdrehung muss der Roboter also wieder eine Rechtsdrehung manchen, um zu seiner Ausgangsposition zurück zu kehren. Da diese Rechtsdrehung eine spezielle in der Robotersteuerung ist, schreiben wir dafür ein kleines “r“. Der gesamte Hinweg sieht also so aus.

```
[6][0] = *;
[6][1] = R;
[6][2] = Z;
[6][3] = r;

sechstes Array = "*RZr";
```

Der Roboter kennt jetzt bereits den Hinweg und kann in seine Ausgangsposition zurückkehren. Was er nun noch braucht ist der Rückweg. Die Funktion dafür haben wir folgendermaßen implementiert.

Die 3 Hinweg Arrays werden der Länge nach durchgegangen und wieder umgekehrt in die ersten 3 Arrays unserer 6 reservierten gespeichert. Wir verwenden also den gleichen Weg zurück wie Hin, nur das der Rückweg spiegelverkehrt ist. Also müssen wir zusätzlich jedes Kommando ebenso umdrehen.

```
L = R
R = L
* = *
Z = nichts passiert
```

Am Ende des Algorithmus wird das letzte Element des Rückwegarrays durch ein “S“ ersetzt. Dieses Kommando sagt der Robotersteuerung, dass der Startknoten wieder erreicht wurde.
Spielen wir auch dies wieder am Beispiel des 3. Weges durch.

```
Hinweg = "*RZr";
```

Wir berechnen die Länge eines Hinwegarrays und ziehen die letzte Drehung, den ’Z’-Character und die darauffolgende Anweisung ab. Die Drehung vor dem ’Z’ wird abgezogen, da wir uns bereits in der Ausgangsposition befinden. Wenden wir diese Rechnung auf unser konkretes Beispiel an, erhalten wir die Länge 1. Wir schreiben das erste Element des Hinweg Arrays also in unser Rückwegarray.

```
Hinweg = "*RZr";
Rückweg= "*";
```

Zu guter Letzt ersetzen wir das letzte Element durch ’S’ und machen damit bekannt, das die nächste Kreuzung der Startpunkt ist. Das Letzte Element ist hier zufälliger Weise auch unser erstes, so steht nun in dem Rückwegarray.

```
Hinweg = "*RZr";
Rückweg= "S";
```

Der letzte Schritt zu einer vollständigen Fahrtsequenz ist es, alle 6 Arrays, also die Hinwege und die Rückwege zu einer großen Route zusammenzufügen. Diese Aufgabe meistert die Funktion “merge“. Sie geht die Arrays in ihrer richtigen Reihenfolge durch schreibt sie nacheinander in ein großes Array namens “route“. Das größte Problem ist dabei die zugegebenermaßen konfuse Reihenfolge der Arrays.

```
[0] = Rückwege a
[1] = Rückweg b
[2] = Rückweg c
[3] = Hinweg a
[4] = Hinweg b
[5] = Hinweg c
```

Die Methode verknüpft nun Array [3] mit Array [0], Array [4] mit Array [1] und Array[5] mit Array [2] zu einer großen Route. Als konkretes Beispiel:

```
[0] = R * R L * L R * * * S a a a a a a a a a
[1] = R L * L R * * * S a a a a a a a a a a a
[2] = S a a a a a a a a a a a a a a a a a a a
[3] = * * * * L R * R L * L R Z r a a a a a a
[4] = * * * * L R * R L R Z r a a a a a a a a
[5] = * R Z r a a a a a a a a a a a a a a a a
route = [3] + [0] + [4] + [1] + [5] + [2]
route = "****LR*RL*LRZrR*RL*LR***S****LR*RLRZrRL*LR***S*RZr*"
```

5.2 Implementierung der Steuerung

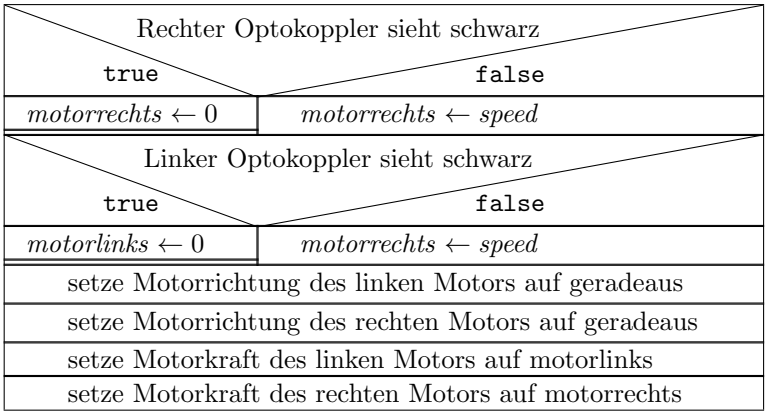
Der Dreh- und Angelpunkt der Steuerung ist die von der Breitensuche erzeugte Route. So wird jedes einzelne Symbol der Routensequenz für sich interpretiert. Die einzelnen Aktionen, die bei den jeweiligen Zeichen ausgeführt werden, sind in separate Funktionen implementiert, die wir genauer erläutert wollen. Zunächst wollen wir jedoch die globalen Variablen für die Steuerung betrachten.

```
char speed = 6;
char motorrechts= 6;
char motorlinks= 6;
int counter = 0;
```

“motorrechts“ und “motorlinks“ geben dabei die Geschwindigkeit der einzelnen Motoren an. Im Normalfall stellen wir diese auf den Wert 6, da wir so eine gute Balance zwischen Kontrollierbarkeit, Energieverbrauch und Geschwindigkeit bekommen. Die Variable “speed’ßtellt eine Hilfsvariable dar, die für die Korrektur benötigt wird. Die Variable “counter“ wird als Kreuzungszähler in der Hauptschleife benötigt. Die Methode “walk“ ist die elementarste Methode. Ohne sie würde die “Crawling Horned“ nicht einmal geradeaus fahren. Zudem ist “walk“ auch für die Linienführung verantwortlich und korrigiert während der Fahrt den Roboter. Verantwortlich dafür sind 2 If-Abfragen, welche die Geschwindigkeit der einzelnen Motoren neu setzt.

```
if(analog(14) > 30){
motorrechts = 0;
}
else{
motorrechts =speed;
}

if(analog(8)>30){
motorlinks = 0;
}else{
motorlinks = speed;
}
```

Nimmt der rechte Optokoppler, welcher an den 14. Analog-Port angeschlossen ist, schwarz wahr, dann wird der rechte Motor angehalten, während der linke Motor weiter auf voller Kraft läuft. So “drückt“ sich der Roboter zurück. Wenn der rechte Optokoppler nun wieder weiß sieht - also sich nicht mehr über der Linie befindet - wird der rechte Motor wieder auf 6 gestellt und kann seinen Weg fortsetzen.



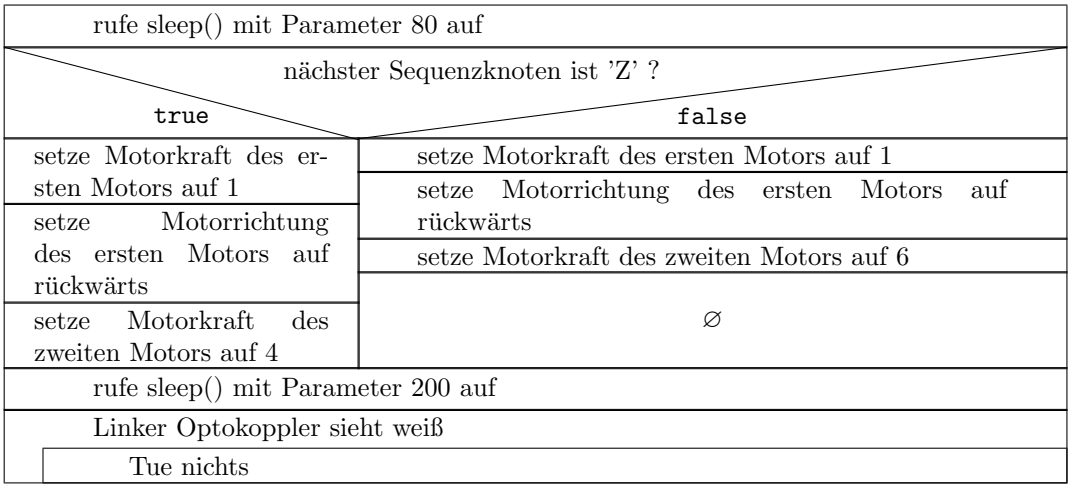
Die nächste Funktion, mit der wir uns beschäftigen wollen, erkennt, ob der Roboter eine Kreuzung passiert hat oder nicht.

```
char isKreuzung(){
if( (analog(14) > 30) && (analog(8)>30))
return 1;
else
return 0;
}
```

Da C keine boolschen Werte verarbeiten, kann greifen wir auf den kleinsten Ganzzahlwert in C zurück, einem Character. Wird eine Kreuzung gefunden, erhalten wir eine 1, wenn nicht eine 0. Um nun genau zu wissen, ob wir eine Kreuzung gefunden haben, untersuchen wir die Werte beider Optokoppler. Sind beide unter 30, befinden sie sich auf schwarz und das bedeutet "Kreuzung gefunden"'.
Da wir nun nachweisen können, ob wir eine Kreuzung erreicht haben, wollen wir uns damit beschäftigen, welche Aktionen die "Crawling Horned" bei einer Rechts- beziehungsweise einer Linksdrehung ausführt. Für diese Drehungen sind die Funktionen "turnLeft" und "turnRight" zuständig. Da beide symmetrisch verlaufen, schauen wir uns nur die "turnRight" genauer an.

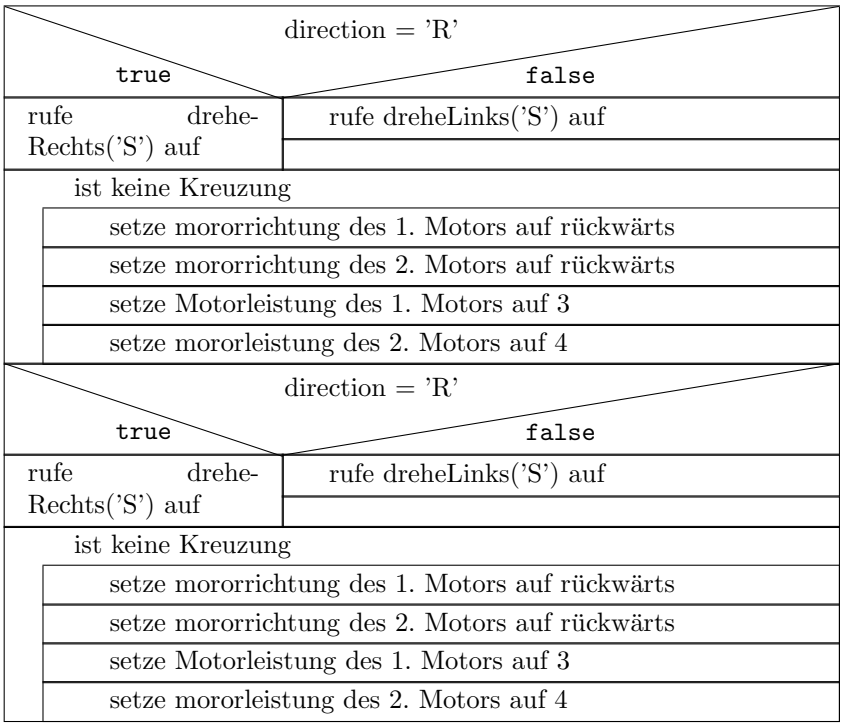
```
void turnRight(){

sleep(80);
if(Sequenz[counter+1] == 'Z')
{
motor_pwm(1,1);Drehrichtung des Rechen Motors auf Rückwärts gestellt

motor_richtung(1,0);
motor_pwm(3,4);
}
else
{
motor_pwm(1,1);
motor_richtung(1,0);
motor_pwm(3,6);
}
sleep(200);
while(analog(14)< 30);

}
```

Nach dem Erkennen einer Kreuzung wird ein 80ms sleep ausgelöst und lässt so den Roboter ein wenig über die Kreuzung hinaus fahren. Danach wird geprüft ob das nächste Sequenzelement ein 'Z' ist. Wenn dem so ist, wird zuerst die Kraft des rechten Motors auf 1 und die Drehrichtung auf rückwärts gestellt. Das Zurückdrehen ist notwendig, da sonst das rechte Rad nachgezogen werden würde. Die Kraft des linken Motors wird derweil auf 4 gestellt, um eine Drehung zu erzeugen. Im Vergleich zu einer normalen Drehung an einer Kreuzung, dessen nächster Punkt kein Zielknoten ist, würde die Kraft etwas verringert werden, um bei der Drehung nicht gleich die Kugel mit der Front des Roboters herunter zu stoßen.



Nachdem die entsprechenden Werte in den Motoren gesetzt wurden, wird durch ein sleep 200ms gewartet, damit die Optokoppler auf die weiße Fläche kommen. Solange sich der rechte Optokoppler nicht wieder auf schwarz befindet, wird die Funktion und somit die Drehung nicht verändert.
Die nächsten Funktionen beschreiben das Verhalten des Roboters beim Erreichen und Verlassen eines Zielknotens. Die Methode "fangeBall" übernimmt die Aufgabe, einen Fahrgast einzusammeln. Wichtig dabei ist das Drosseln der Motoren und die Verringerung der Hilfsvariable "speed". Durch die langsamere Fortbewegung hat die walk-Methode mehr Zeit, den Roboter gerade auszurichten. Dadurch ist eine gerade Zusteuerung auf den Passagier möglich. Gleichzeitig starten wir den Infrarotsender und die Lichtschranke ist aktiviert. Das eigentliche Fangen des Balles übernimmt nun eine Endlosschleife, in der überprüft wird, ob die Lichtschranke durchbrochen wird. Wird sie nicht durchbrochen, wird die walk-Methode weiterhin ausgeführt und der Roboter gerade ausgerichtet. Wenn jetzt aber die Lichtschranke durchbrochen wird, stoppt die "Crawling Horned" und der Servomotor lässt den Fangbügel herunter, um den Ball einzufangen. Zu guter Letzt brechen wir mit einem "break" aus der Endlosschleife aus. Danach werden nur noch die Motorwerte zurückgesetzt, um mit unseren Standardwerten weiter arbeiten zu können.
Nun, wo wir den Ball haben, müssen wir dem Roboter beibringen, in einem Zielknoten zu drehen. Die Funktionen "dreheLinks" und "dreheRechts" erledigen diese Aufgabe für uns. Wir benutzen hier spezielle Methoden zum Links oder Rechtsdrehen, da ein normales "turnRight" uns nicht den ganzen Umfang bereitstellt, den wir für eine solche Drehung benötigen. Zunächst müssen wir den Roboter soweit zurückfahren lassen, bis wir eine Kreuzung finden und halten kurz davor an und drehen dann nach links bzw. rechts. Der Parameter, der übergeben werden muss, wurde eingefügt, um eine Rechts oder Linksdrehung im Startpunkt zu steuern, die später noch erläutert wird. Jede Eingabe, die nicht 'S' ist, löst die Rechts oder Linksdrehung im Zielpunkt aus. Wenn wir nun gerade in einen Zielknoten einfahren, müssen wir auch wieder gerade heraus fahren. Die Links und Rechtsdrehungen können uns nur sehr umständlich in diese Position bringen. Die Methode "dreheUm()" erleichtert uns die Durchführung einer 180° Wende. Auch hier fährt die "Crawling Hornet" bis zur Kreuzung zurück und dreht solange, bis sie zum zweiten Male eine schwarze Linie sieht. Nach zahlreichen Tests hat sich diese einfache Drehung als sehr erfolgreich dargestellt.

Eine letzte Methode brauchen wir nun . Um zu unterscheiden, ob wir auf der rechten oder auf der linken Seite den Startpunkt erreichen, haben wir einen Parameter implementiert. Sollten der Roboter auf der rechten Seite fahren, wird nach dem Erreichen eines Startpunktes zuerst eine Rechtsdrehung ausgeführt und dann zurückgesetzt, bis eine Kreuzung erkannt wird. Das ist wichtig, da unsere Methoden zum Drehen nur eine einwandfreie Drehung garantieren, wenn wir nah genug an einer Kreuzung stehen, so kann sich der Roboter immer wieder gerade ausrichten - ohne ein Zurückfahren würde der Roboter völlig aus der Spur kommen.
Nachdem die Kreuzung erkannt wurde, wird einfach wieder nach rechts gedreht, bis die "Crawling Hornet" wieder gerade aus schaut. Der letzte Schritt zur Vollendung der Startpunktdrehung ist ein weiteres Zurückfahren bis zur Kreuzung. Durch die eigene Trägheit "rutschen" wir über die Kreuzung, stehen am Ausgangspunkt und können eine weitere Fahrt zu einem Passagier unternehmen. Befinden wir uns nicht in der rechten Hälfte des Feldes so müssen wir die Rechtsdrehungen durch Linksdrehungen ersetzen.



Parameter: direction {Gibt an ob sich der Roboter den Rechten oder den Linken Startpunkt ansteuert}

Die Hauptroutine

Die Hauptroutine ist wohl das Wichtigste im Bereich der Robotersteuerung. Die Hauptroutine beginnt mit der Initialisierung der Motorwerte und dem Ausführen der Berechnungsmethoden für die Fahrtroute. Eine Lampe beginnt zu leuchten, sobald diese Rechenarbeit beendet ist.
Danach beginnt der sogenannte Mainloop - eine Endlosschleife, in der die Fahrtsequenz abgearbeitet wird. Sobald die Startlampe auf dem Feld leuchtet, erkennt der Photosensor dies und gibt den Roboter zur Fahrt frei.

```
//Berechnete Fahrtsequenz
route = ****LR*RL*LRZrR*RL*LR***S****LR*RLRZrRL*LR***S*RZr*
```

Es startet eine Schleife, welche die komplette Fahrtsequenz bis zum Erkennen eines a's durchläuft und je nach Zeichen eine Aktion auslöst. Als erstes wird mit der Funktion "isKreuzung" geprüft, ob der

Roboter eine Kreuzung sieht oder nicht. Sollte dies der Fall sein, gibt es verschiedene Möglichkeiten. Es wird geprüft, ob das aktuelle Zeichen entweder ein '*', 'R' oder 'L' ist. Die "Crawling Hornet" fährt also entweder gerade aus, nach Rechts oder nach Links. Zusätzlich wird in den einzelnen Aktionen geprüft, ob die nächste Anweisung ein 'Z', also ein Zielknoten ist, um das Fangszenario vorzubereiten. Um eine spezielle Kreuzung handelt es sich bei der Startkreuzung, welche in der Sequenz mit einem 'S' gekennzeichnet ist. Sie teilt dem Roboter mit, das die nächste Kreuzung der Startpunkt ist.
Befindet sich der Roboter nicht auf einer Kreuzung, so können folgende Fälle eintreffen: Es kann sein, das der Roboter einfach nur gerade aus fahren soll, aber es ist auch möglich, dass er gerade einen Passagier eingesammelt hat und nun eine Drehung ansteht. Am anschaulichsten lässt sich diese Befehlsabarbeitung in einem Flussdiagramm darstellen. Dieses befindet sich im Anhang auf der Seite 41.

Literatur

[1] Wikipedia. Masdar. `http://de.wikipedia.org/wiki/Masdar#Verkehr`, 2012. last checked: 04.01.13.

[2] Wikipedia. Personal Rapid Transit. `http://de.wikipedia.org/wiki/Personal_Rapid_Transit`, 2012. last checked: 04.01.13.

Abbildungsverzeichnis

6 Anhang

Die Dokumentation finden Sie im Anhang dieser E-mail.

6.1 Quellcode

```
#include <stub.h>
/**
 * Definieren des Fahrauftrages und der globalen Variablen.
 * motorrechts und motorlinks : geben die Geschwindigkeit der Motoren an
 * speed :  wird für die Korrektur beim Linienfolgen verwendet
 * counter:  der Zähler, der durch die sequenz iteriert
 * wege[6][20]:  werden genutzt, um 3 Hin- und 3 Rückwege zu speichern
 * Sequenz[120]:  speichert die gesamte Fahrsequenz
 * zielknoten[9]:  speichert die Zielknoten
 * startpos:  legt die Starposition fest
 * costs:  speichert die Kosten für die Kostenmatrix
 * nachbarn[4]:  gibt an, in welcher Richtung sich ein expandierbarer Knoten befindet
 * agenda[10][2]:   speichert die expandierbaren Knoten mit der Anzahl der
 *  in einem Durchgang bereits aufgenommenen Knoten
 */
#define FA1


char speed = 6;
char motorrechts= 6;
char motorlinks= 6;
int counter = 0;
char wege[6][20] = {{"aaaaaaaaaaaaaaaaaaa"},{"aaaaaaaaaaaaaaaaaaa"},{"aaaaaaaaaaaaaaaaaaa"},
{"aaaaaaaaaaaaaaaaaaa"},{"aaaaaaaaaaaaaaaaaaa"},{"aaaaaaaaaaaaaaaaaaa"}};
char Sequenz [120]={"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"};
char zielknoten[9]="aaaaaaaa";
char startpos = 68;
char costs = 0;
char nachbarn[] = "0000";
char agenda[10][2] ={{'a',0}, {'a',0},{'a',0},{'a',0},{'a',0},{'a',0},{'a',0},{'a',0},
{'a',0},{'a',0}};

/** Inkludieren der Felder
 *
 */

#include "fa.h"


/*************************************
 * Implementierung der Breitensuche   *
 *  *
 * ************************************/


/**
 *  Bestimmt die Anzahl der in einem Array gefüllten Felder und gibt diese zurück.
 */
```

```
int bestimmeLaenge(const char* str)
{
int laenge = 0;
while(*str++ != 'a' ) laenge++;
return laenge;
}

/**
 *  Diese Methode bestimmt das erste freie Feld in der Agenda und nimmt in
 * dieses den Knoten mit der Anzahl der bereits in diesem Durchgang
 * aufgenommenen Knoten auf.
 */
void append(char knoten, char counter){
char i = 0;
while(agenda[i][0]!='a'){
i++;
}
agenda[i][0]=knoten;
agenda[i][1]=counter;
}


/**
 *  Expandierung des ersten Elements, indem das n-te Element mit dem n+1-ten Element
 * überschrieben wird.
 *  Anschließend wird die letzte Stelle wieder mit einem 'a' aufgefüllt.
 */
void expand(){
char i = 0;
for(i = 0;i<sizeof(agenda)/2-2;i++){
agenda[i][0] = agenda[i+1][0];
agenda[i][1] = agenda[i+1][1];
}
agenda[sizeof(agenda)/2-3][0]='a';
agenda[sizeof(agenda)/2-3][1]=0;

}

/**
 *  Überprüft einen Punkt mit dem Index "pos" mit Array-Überlauf Schutz
 * durch vorherige Abfrage. Wird im Norden, Westen, Süden oder Osten ein
 * Nachbar gefunden, wird '1' zurückgegeben.
 *  Das Nachbar-Array wird entweder mit 0 oder 1 gefüllt, wobei der Index
 * der Richtung entspricht, in dem ein Nachbarpunkt gefunden wird.
 *  Index 0 = Nachbar im Norden
 *  Index 1 = Nachbar im Westen
 *  Index 2 = Nachbar im Süden
 *  Index 3 = Nachbar im Osten
 *  Wird kein Punkt gefunden, wird '0' zurückgeben.
 *
 *
 */
char checkNachbarPunkte(char pos){
char check ='0';
```

```
if((pos-7)>=0 && _fa[pos-7]=='.'){
nachbarn[0] = '1';
check ='1';
}
else{
nachbarn[0]='0';
}

if((pos+7)<=69 &&(_fa[pos+7]=='.')){
nachbarn[2] ='1';
check = '1';
}
else {
nachbarn[2]='0';
}

if((pos-1)>=0 && _fa[pos-1]=='.'){
nachbarn[1]='1';
check = '1';
}
else{
nachbarn[1]='0';
}

if((pos+1)<=69 && _fa[pos+1]=='.'){
nachbarn[3]='1';
check = '1';
}
else{
nachbarn[3]='0';
}

return check;
}


/**
 *  Diese Methode überprüft die Erreichbarkeit eines Zielknotens.
 *  Ein Zielknoten kann erreicht werden, wenn sich in einer Himmelsrichtung
 * ein mit Kosten gefüllter Punkt befindet. Im Umkehrschluss muss es einen
 * Nachbarpunkt geben, der kein 'F', kein '.' und kein 'x' ist.
 */

char istErreichbar(char pos){
char bool = '0';
if(((pos-7)>=0) && (_fa[pos-7]!='F') && (_fa[pos-7]!='.') &&
(_fa[pos-7]!='x'))  bool = '1';
if(((pos-1)>=0) && (_fa[pos-1]!='F') && (_fa[pos-1]!='.') &&
(_fa[pos-1]!='x'))  bool = '1';
if(((pos+7)<=69) && (_fa[pos+7]!='F') && (_fa[pos+7]!='.') &&
(_fa[pos+7]!='x')) bool = '1';
if(((pos+1)<=69) && (_fa[pos+1]!='F') && (_fa[pos+1]!='.') &&
(_fa[pos+1]!='x')) bool = '1';

return bool;
```

```
}

/**
 *  Fügt die gefundenen und erreichbaren Zielknoten zum Zielknoten-Array hinzu.
 */
void appendZielknoten(){
char i = 0;
char j = 0;
for(i = 0;i<sizeof(_fa)-1;i++){
if(_fa[i]=='F'&& (istErreichbar(i)=='1')){
zielknoten[j]=i;
j++;
}
}

}

/**
 *  Überprüft, ob sich noch expandierbare Knoten im Array befinden.
 */

char isEmpty(){

int i;
for(i = 0;i<sizeof(agenda)/2-1;i++){
if(agenda[i][0]!='a'){
return '0';
}
}
return '1';

}



/**
 *  Diese Methode füllt das Feld mit Kosten für jeden einzelnen Punkt.
 *  Zunächst werden der Startpunkt und das Feld übergeben, der Startpunkt
 * wird dann mit den Kosten 0 zur Agenda hinzugefügt.
 *  Der Algorithmus bricht ab, wenn sich kein expandierbarer Punkt mehr in
 * der Agenda befindet.
 *  1.) Überprüfung, ob der aktuelle Punkt einen Nachbarn hat.
 *  2.) internen Zähler bei jedem neuen Durchgang auf 0 setzen
 *  3.) Überprüfen, ob der aktuelle Punkt als einziger Punkt im letzten Durchgang
 * aufgenommen wurde. Wenn nicht, werden die Kosten heruntergezählt.
 *  4.) Das Nachbarschaftsarray durchgehen, in die jeweiligen Punkte die Kosten
 * schreiben, diese Knoten in die Agenda aufnehmen und den internen
 * Zähler hochzählen.
 *  5.) Kosten hochzählen,Expandieren.
 *  6.) den ersten Punkt in der Agenda als neuen zu expandierenden Punkt festlegen
 *
 *  Wird kein Punkt gefunden, werden die Kosten nicht hochgezählt, sondern es folgt
 * nur eine Expandierung. Dann den neuen erste Punkt als im nächsten Durchgang
 * zu expandierenen Punkt festlegen.
 *
```

```
 */
void berechnenKostenmatrix(char pos, char *f){

char a = 0;
char i = pos;
f[pos]=costs;
costs = costs+1;
append(pos,a);

while(isEmpty()=='0'){

if(checkNachbarPunkte(i)=='1'){
a=0;
if(agenda[0][1]!=0){
costs--;
}

if(nachbarn[0]=='1'){
f[i-7] = costs;
append(i-7,a);
a++;
}
if(nachbarn[1]=='1'){
f[i-1] = costs;
append(i-1,a);
a++;
}
if(nachbarn[2]=='1'){
f[i+7] = costs;
append(i+7,a);
a++;
}
if(nachbarn[3]=='1'){
f[i+1] = costs;
append(i+1,a);
a++;
}

expand();
costs++;
i = agenda[0][0];

}

else{
expand();
i = agenda[0][0];
}


}
}


/**
```

```
 *   Fügt einen Schritt an die letzte freie Stelle im Array "num" ein.
 */
void appendWeg(char schritt,char num){
char i = 0;
while(wege[num][i]!='a'){
i++;
}
wege[num][i] = schritt;

}

/**
 *  Überschreibt den n-ten Punkt mit dem n+1-ten Punkt. Die letzte Stelle wird
 * wieder aufgefüllt.
 */
void expandZiel(){
char i = 0;
for(i = 0;i<sizeof(zielknoten)-2;i++){
zielknoten[i] = zielknoten[i+1];
}
zielknoten[sizeof(zielknoten)-1]='a';
}

/**
 *  Bestimmt den Nachbarpunkt mit den niedrigsten Kosten und fügt den Index
 * dieses Feldpunktes zum wege-Array mit dem Index "num" hinzu.
 *  Gleichzeitig wird der dieser Index zurückgegeben.
 *
 */
char checkWeg(char pos,char num){
unsigned char costs = 100;
char b = 0;

if((pos-7)>=0 && _fa[pos-7]!='F' && _fa[pos-7]!='x' && _fa[pos-7]!='.'){
costs = _fa[pos-7];
b = pos-7;

}
if((pos+7)<=69 && _fa[pos+7]!='F' && _fa[pos+7]!='x' && _fa[pos+7]!='.'){
if(_fa[pos+7]<=costs){
costs = _fa[pos+7];
b = pos+7;
}


}
if((pos-1)>=0 && _fa[pos-1]!='F' && _fa[pos-1]!='x' && _fa[pos+1]!='.'){
if(_fa[pos-1]<=costs){
costs = _fa[pos-1];
b = pos-1;
}

}
if((pos+1)<=69 && _fa[pos+1]!='F' && _fa[pos+1]!='x' && _fa[pos+1]!='.'){
if(_fa[pos+1]<=costs){
```

```
costs = _fa[pos+1];
b = pos+1;
}

}

if((b == pos-7)||(b == pos+7)|| (b == pos-1)||(b == pos+1)) appendWeg(b,num);


return b;

}

/**
 *  Diese Methode bestimmt die Wege in Form der Indices des Feldes.
 *  Zunächst wird die Kostenmatrix gefüllt und die Zielknoten werden bestimmt.
 *  Werden weniger als 3 Zielknoten gefunden, wird die Anzahl der Durchläufe auf
 * die Anzahl der gefundenen Knoten begrenzt.
 *  Werden mehr als 3 Knoten gefunden, wird die Anzahl der Durchläufe auf 3 gesetzt.
 *  Wird ein Startpunkt erreicht ( entspricht _fa[b]==0), bricht die Schleife ab.
 *
 */
void berechneWeg(){
char i,j;
char b=0;
berechnenKostenmatrix(startpos,_fa);
appendZielknoten();

j = bestimmeLaenge(zielknoten);
if(j>3) j=3;

for(i = 0;i<j;i++){
appendWeg(zielknoten[i],i);
b = checkWeg(zielknoten[i],i);
if(b!=0){
while(_fa[b]!=0){
b = checkWeg(b,i);
}

}


}
}


/**
 *  Formt die Indices in Anweisungen um und fügt an die erste Stelle ein 'Z'
 * (für Zielknoten) ein.
 *  Die ersten 3 Wege-Arrays enthalten dann die Rückwege.
 */

void berechnenFahrtweg(){
char status;
char i,j,k;
```

```
char diff,length;
for(i=0;i<3;i++){
length = 0;
status='N';
for(j=0;j<20;j++){
if(wege[i][j] != 97)
length++;
}
for(k=length-1;k>0;k--)
{
diff = (wege[i][k] - wege[i][k-1]);
if (diff == 7){
if(status=='N'){
wege[i][k] = '*';
}
else if(status=='W'){
wege[i][k]='R';
status='N';
}
else if(status=='O'){
wege[i][k]='L';
status='N';
}

}
if(diff == 1){
if(status == 'N'){
wege[i][k]= 'L';
status = 'W';
}
else if (status == 'W'){
wege[i][k] = '*';
}
else if(status == 'S'){
wege[i][k]= 'R';
status = 'W';
}
}
if(diff == -1){
if(status=='N'){
wege[i][k]='R';
status ='O';
}
else if(status=='O'){
wege[i][k]='*';
}
else if(status=='S'){
wege[i][k]='L';
status = 'O';
}
}
if(diff == -7){
if(status=='S'){
wege[i][k]='*';
}
```

```
else if(status=='O'){
wege[i][k]='R';
status='S';
}
else if(status=='W'){
wege[i][k]='L';
status='S';
}
}

if(k==1)wege[i][k-1]='Z';
}

}

}

/**
 *  Nun werden die Hinwege erstellt, indem die ersten 3 Wege-Arrays gespiegelt
 * und in die letzten 3 Wege-Arrays geschrieben. Anschließend wird das vorletzte
 * Element betrachtet und eine entsprechende Drehanweisung für den Roboter hinten
 * angehangen. Zum Schluss werden die ersten 3 Arrays bereinigt.
 */

void berechneHinweg()
{
int i,j,k = 0;

for(i=0;i<3;i++)
{
j = bestimmeLaenge(wege[i])-1;
k=0;

for(;j>=0;j--)
{
wege[3+i][k]  = wege[i][j];
k++;
}

if(wege[3+i][k-2]=='L') appendWeg('l',3+i);
if(wege[3+i][k-2]=='R') appendWeg('r',3+i);
if(wege[3+i][k-2]=='*') appendWeg('d',3+i);

}

for(i=0;i<3;i++)
{
for(j=0;j<20;j++)
{
wege[i][j] = 'a';
}
}

}
```

```
/**
 *  Diese Methode erstellt aus den ersten 3 Wege-Arrays die Rückwege,
 * indem diese gespiegelt und die Anweisungen umgedreht werden.
 *  Links wird zu rechts, rechts zu links, * bleibt *.
 *  Die letzte Stelle wird überschrieben mit einem 'S' für "Startpunkt".
 */
void berechneRueckweg()
{
int i,j,k = 0;

for(i=3;i<6;i++)
{
j = bestimmeLaenge(wege[i])-4;
k=0;

for(;j>=0;j--)
{
switch(wege[i][j])
{
case 'L':
wege[i-3][k] = 'R';
break;

case 'R':
wege[i-3][k] = 'L';
break;

case 'Z':
k--;
break;

default:
wege[i-3][k] = wege[i][j];
break;
}
k++;
}
wege[i-3][k-1] = 'S';
}
}

/**
 *  Diese Funktion fügt alle Wege zu einer Sequenz zusammen.
 */
void merge(){
char indexWeg = 0;
char indexRoute = 0;
while(wege[3][indexWeg]!='a'){
Sequenz[indexRoute] = wege[3][indexWeg];
indexRoute++;
indexWeg++;
}
indexWeg = 0;
while(wege[0][indexWeg]!='a'){
Sequenz[indexRoute] = wege[0][indexWeg];
```

```
indexRoute++;
indexWeg++;
}
indexWeg = 0;
while(wege[4][indexWeg]!='a'){
Sequenz[indexRoute] = wege[4][indexWeg];
indexRoute++;
indexWeg++;
}
indexWeg = 0;
while(wege[1][indexWeg]!='a'){
Sequenz[indexRoute]=wege[1][indexWeg];
indexRoute++;
indexWeg++;
}
indexWeg=0;
while(wege[5][indexWeg]!='a'){
Sequenz[indexRoute]=wege[5][indexWeg];
indexRoute++;
indexWeg++;
}
indexWeg=0;
while(wege[2][indexWeg]!='a'){
Sequenz[indexRoute]=wege[2][indexWeg];
indexRoute++;
indexWeg++;
}
}

/*********************************
 *    *
 * Hauptprogrammroutine   *
 *   *
 * *********************************/

 /**
  *  Methode zum Linienfolgen
  *
  */
void walk(){

if(analog(14) > 30){
motorrechts = 0;
}
else{
motorrechts =speed;
}

if(analog(8)>30){
motorlinks = 0;
}else{
motorlinks = speed;
}
motor_richtung(3,1);
motor_richtung(1,1);
```

```
motor_pwm(3,motorlinks);
motor_pwm(1,motorrechts);

}

/**
 *  Stoppt den Roboter.
 */
void stop(){
motor_pwm(3,0);
motor_pwm(1,0);
}

/**
 *  Methode zum Links-Abbiegen.
 *  Sollte die darauffolgende Anweisung ein Zielknoten sein, wird
 * die Geschwindigkeit heruntergesetzt, um nicht zu schnell abzubiegen.
 */
void turnLeft(){

sleep(80);
if(Sequenz[counter+1] == 'Z')
{
motor_pwm(3,1);
motor_richtung(3,0);
motor_pwm(1,4);
}
else
{
motor_pwm(3,1);
motor_richtung(3,0);
motor_pwm(1,6);
}


sleep(200);
while(analog(8)< 30);

}

/**
 *  Methode zum Rechts-Abbiegen
 *  Sollte die darauffolgende Anweisung ein Zielknoten sein,
 * wird die Geschwindigkeit herunter gesetzt, um nicht zu schnell abzubiegen.
 */
void turnRight(){

sleep(80);
if(Sequenz[counter+1] == 'Z')
{
motor_pwm(1,1);
motor_richtung(1,0);
motor_pwm(3,4);
}
else
```

```
{
motor_pwm(1,1);
motor_richtung(1,0);
motor_pwm(3,6);
}
sleep(200);
while(analog(14)< 30);

}

/**
 *  Diese Methode überprüft, ob eine Kreuzung gesehen wird.
 */
char isKreuzung(){
if( (analog(14) > 30) && (analog(8)>30))
return 1;
else
return 0;
}

/**
 * Methode zum Umdrehen nach einem Zielknoten
 */
void dreheLinks(char status){
motor_richtung(3,0);
motor_richtung(1,0);
motor_pwm(3,4);
motor_pwm(1,4);
while(isKreuzung()!=1);
motor_richtung(3,1);
motor_richtung(1,1);
sleep(250);
if(status!='S'){
servo_arc(0,25);
}
sleep(100);
motor_pwm(3,1);
motor_richtung(3,0);
motor_pwm(1,6);
sleep(100);
while(analog(8)< 30);


}

/**
 *  Methode zum Umdrehen nach einem Zielknoten
 */
void dreheRechts(char status){
motor_richtung(3,0);
motor_richtung(1,0);
motor_pwm(3,4);
motor_pwm(1,4);
while(isKreuzung()!=1);
motor_richtung(3,1);
```

```
motor_richtung(1,1);
sleep(250);
if(status!='S'){
servo_arc(0,25);
}
sleep(100);
motor_pwm(1,1);
motor_richtung(1,0);
motor_pwm(3,6);
sleep(100);
while(analog(14)< 30);


}

/**
 *  Methode zum Fangen des Balls.
 */
void fangeBall(){
motorrechts=2;
motorlinks=2;
speed=3;
walk();
led(3,1);

while(1){
if(digital_in(8) == 1){
stop();
servo_arc(0,30);
break;
}
else
walk();
}
sleep(150);

motorrechts=6;
motorlinks=6;
speed=5;

}

/**
 *  Methode zum Wenden am Startpunkt
 */
void dreheUmStartPunkt(char direction)
{

if(direction == 'R')
dreheRechts('S');
else
dreheLinks('S');

while(isKreuzung()!=1){
motor_richtung(3,0);
```

```
motor_richtung(1,0);
motor_pwm(3,3);
motor_pwm(1,4);
}

if(direction == 'R')
dreheRechts('S');
else
dreheLinks('S');

while(isKreuzung()!=1){
motor_richtung(3,0);
motor_richtung(1,0);
motor_pwm(3,3);
motor_pwm(1,4);
}

}

/**
 *  Methode für eine 180° Drehung nach dem Einsammeln eines Balls
 */

void dreheUm(){
motor_richtung(3,0);
motor_richtung(1,0);
motor_pwm(3,4);
motor_pwm(1,4);
while(isKreuzung()!=1);
motor_richtung(3,1);
motor_richtung(1,1);
sleep(250);
motor_richtung(1,0);
motor_pwm(1,5);
motor_pwm(3,5);
servo_arc(0,25);
sleep(600);
while(analog(14)<30);


}

/**
 *  Hauptprogrammschleife
 *  Die Richtungen werden auf vorwärts gestellt, der Servo-Motor wird nach
 * oben gestellt. Anschließend wird die Berechnung der Breitensuche
 * durchgeführt. Ist diese fertig, wird die LED-Leuchte eingeschaltet.
 *  Der Roboter startet mit der Abarbeitung seiner Route, wenn ein Licht eingeschaltet wird.
 *
 */
int AksenMain(void)
{


motor_richtung(3,1);
```

```
motor_richtung(1,1);
servo_arc(0,90);

berechneWeg();
berechnenFahrtweg();
berechneHinweg();
berechneRueckweg();
merge();
led(2,1);


while(1){

while(analog(0)>50);
led(2,0);
while(Sequenz[counter]!='a'){

if(isKreuzung()!=1){
if(Sequenz[counter]=='l'){
dreheLinks('*');
counter++;
lcd_cls();
lcd_ubyte(counter);
}
else if(Sequenz[counter]=='r'){
dreheRechts('d');
counter++;
lcd_cls();
lcd_ubyte(counter);
}
else if(Sequenz[counter]=='d'){
dreheUm();
counter++;
lcd_cls();
lcd_ubyte(counter);
}
else if(Sequenz[counter] == 'Z'){
counter++;
lcd_cls();
lcd_ubyte(counter);
}
else walk();
}

else{
if(Sequenz[counter] == 'L'){
turnLeft();
if(Sequenz[counter+1] == 'Z'){
fangeBall();
}
counter++;
lcd_cls();
lcd_ubyte(counter);
}
```

```
else if(Sequenz[counter] ==’R’){
turnRight();
if(Sequenz[counter+1] == ’Z’){
fangeBall();
}

counter++;
lcd_cls();
lcd_ubyte(counter);

}

else if(Sequenz[counter] ==’*’){
motor_richtung(3,1);
motor_richtung(1,1);
motor_pwm(3,speed);
motor_pwm(1,speed);
sleep(100);
if(Sequenz[counter+1] == ’Z’){
fangeBall();
}
counter++;
lcd_cls();
lcd_ubyte(counter);

}
else if(Sequenz[counter]==’S’){
if(Sequenz[counter-1] == ’L’)
{
turnRight();
sleep(200);
servo_arc(0,90);
dreheUmStartPunkt(’R’);
sleep(100);



}
else if(Sequenz[counter-1] == ’R’)
{
turnLeft();
sleep(200);
servo_arc(0,90);
dreheUmStartPunkt(’L’);
sleep(100);

}
else
{

sleep(400);
motor_pwm(3,4);
motor_pwm(1,4);
servo_arc(0,90);
dreheUmStartPunkt(’R’);
```

```
sleep(100);
}
sleep(100);
counter++;
lcd_cls();
lcd_ubyte(counter);

}

}
}
stop();



}


}
```

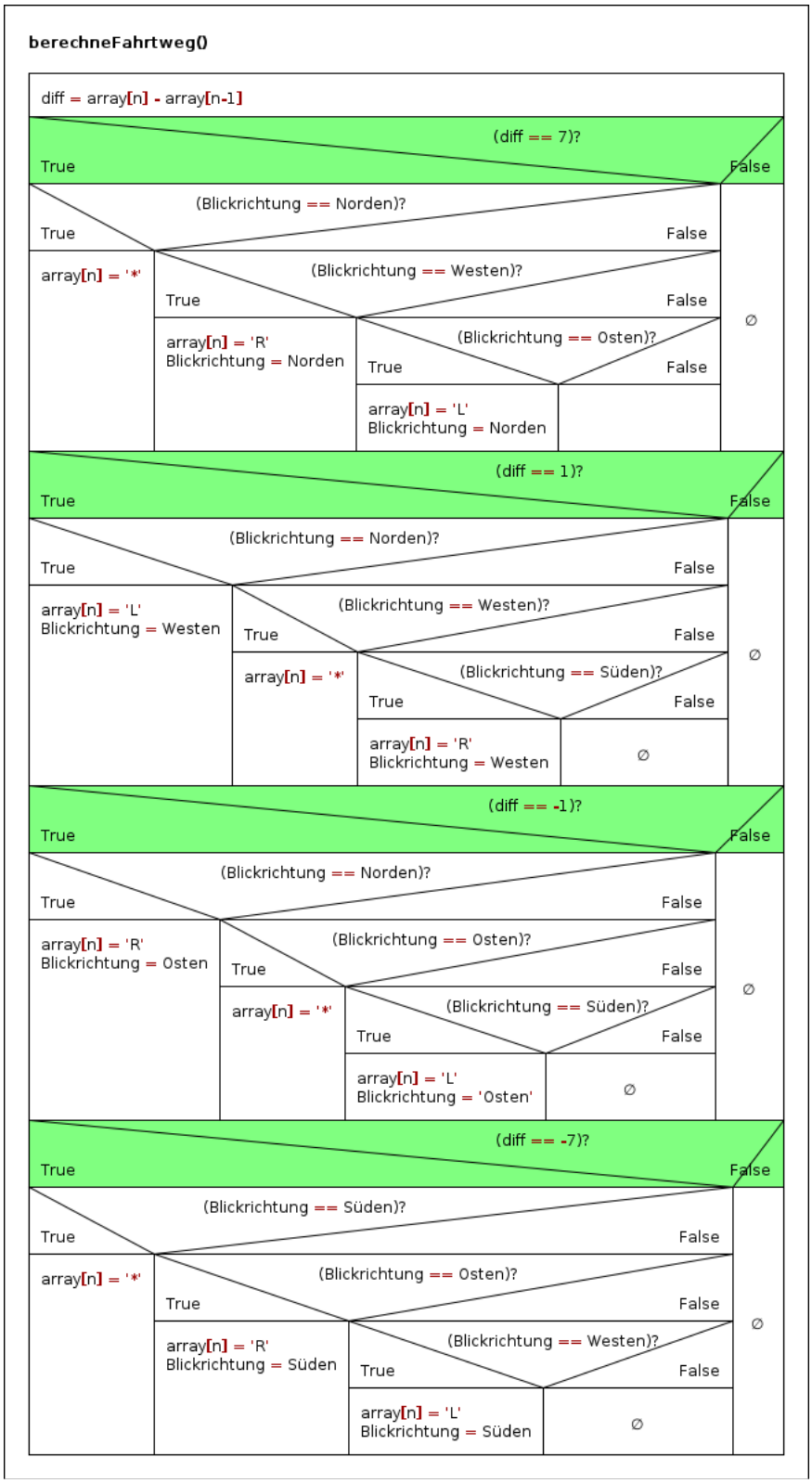


Abbildung 13: berechneFahrtweg()

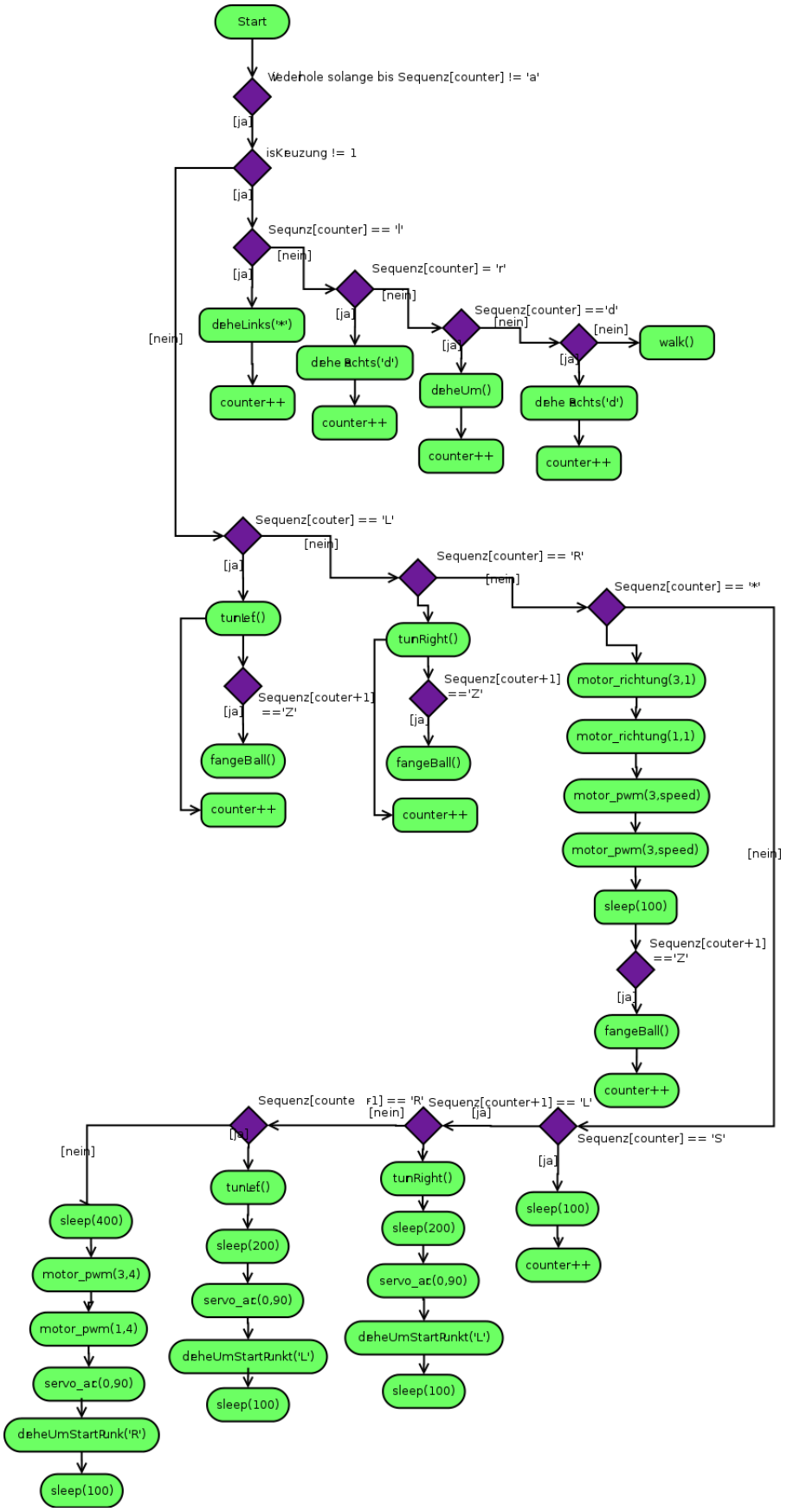


Abbildung 14: Abarbeitung der Fahrsequenz