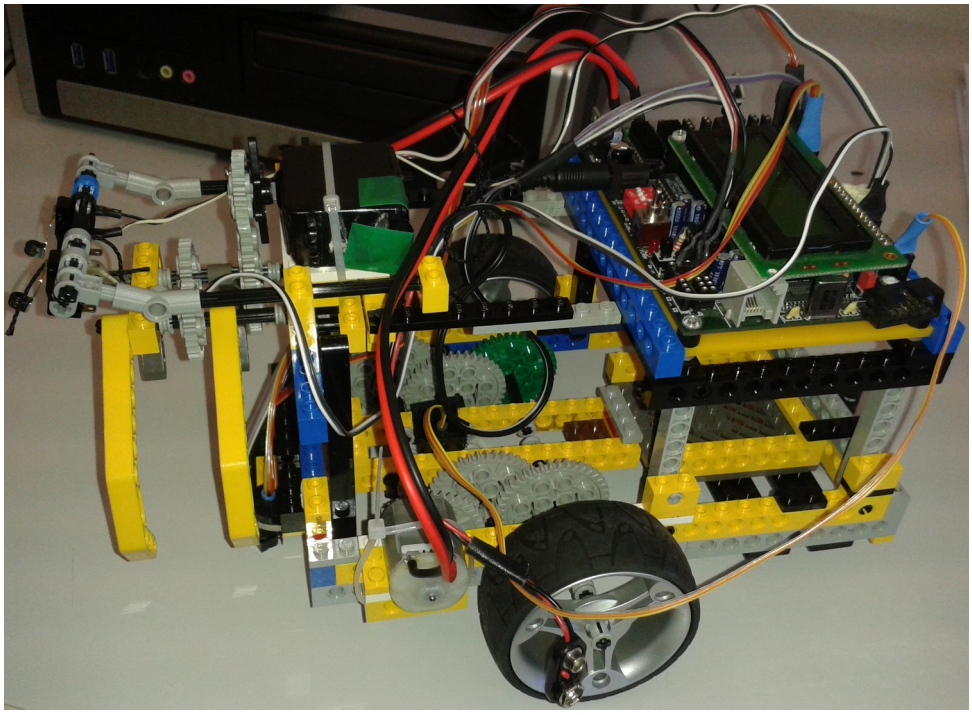
S.A.L.A.T.

Smart Agile Lego Autonomous Transportation



Autoren: Marie-Christin Knoll und Franziska Mieck

Hochschule: Technische Hochschule Brandenburg

Studiengang: Informatik

Semester: Wintersemester 2016/17

Projekt: AMS-Projekt

Inhaltsverzeichnis

1. Vorstellung

1.1 Aufgabenstellung

Eine Simulation des PRT (Personal Rapid Transit) soll stattfinden. Zur Umsetzung gilt es, einen Roboter aus Legobauteilen zusammenzubauen, der autonom Passagiere von ihren jeweiligen Haltestellen abholen und zum Zielort transportieren kann. Die Fahrgäste werden durch blaue Kugeln dargestellt und dürfen nur einzeln befördert werden. Der Roboter kennt die Fahrkarten bereits, doch muss er den besten Weg selbstständig planen und auf vorher bekannte Hindernisse reagieren.

1.2 Gelungene Komponenten

Zu den gelungenen Komponenten zählen besonders die Optoreflexkoppler, welche in Form eines Dreiecks angeordnet sind. Dadurch wurde das Fahren auf der Linie gut gesichert und Kreuzungen wurden gut erkannt.
Um die Sensoren sicher zu befestigen, sind sie mit Klettband befestigt worden. Nach leichter Skepsis war das Ergebnis mehr als zufriedenstellend. Die Position der Komponenten blieb über mehrere Fahrten fast unverändert.
Die Konstruktion des Greifers ist ebenfalls eine der gelungenen Konstruktionen. Wenn er den Ball einmal wirklich sicher gegriffen hat, dann verliert er diesen bis zum Ziel nicht mehr.
Trotz anfänglicher Schwierigkeiten bei der Sicherung des gesamten Konstrukts, sodass keine Teile beim Fahren herunterfallen, wurden mit der Zeit immer mehr Teile fest miteinander verbunden.

1.3 Weniger gelungene Komponenten

Eine der weniger gelungen Komponenten ist die Länge von S.A.L.A.T. und die Wahl des Getriebes, welches eine geringe Wendigkeit auf geraden Strecken zur Folge hat.
Die Halterung der Druckschalter vor dem Greifer ist nicht immer gewährleistet, da durch zu viel Druck diese minimal verschoben werden und dadurch nicht ausgelöst werden können.
Aufgrund der Gewichtsverlagerung von S.A.L.A.T., beginnen die Räder sich nach innen zu biegen. Die Reibungspunkte der Räder sind daher verlagert und haben einen negativen Einfluss auf seine Fahrgeschwindigkeit.

1.4 Wertung der Arbeit

Die Arbeit am Roboter verlief trotz mancher kleiner Probleme, sowohl im Programmieren als auch Bauen, gut und er lief am Ende zuverlässig. Somit vollbrachte S.A.L.A.T. während der Vorstellung und beim Wettbewerb gegen andere gute Leistungen, wenngleich die Konkurrenz sehr gering war.
Jedoch war der Roboter, wie zu erwarten, nicht der Schnellste von allen. Das lag an der Getriebeübersetzung, welche man verbessern könnte. Trotz seiner Langsamkeit war er in der Lage bei allen Karten, außer Fahrplan Nr.7, mindestens zwei Fahrgäste abzuholen und zum Start zurückzubringen, wobei er den letzten Fahrgast immer schon im Greifer hatte.
Als Wertung der Arbeit lässt sich sagen, dass S.A.L.A.T. sehr zuverlässig ist und dem Sieger des Wettbewerbes nur in Sachen Tempo etwas unterlegen war.

2. Lösungsweg

2.1 Strategien und verworfene Ideen

Das Ziel des Projektes war es, einen stabilen und zuverlässigen Roboter zu entwickeln mit einem einzigartigen, leckeren Namen.
Der Roboter sollte nicht der Schnellste sein, sich aber gut auf den Linien halten können, weshalb eine lange Zeit verstrich, bis sich auf eine Getriebeübersetzung geeinigt wurde. Dass die Breitensuche das favorisierte Suchverfahren war, stand schon sehr früh fest, beinahe sofort nach den ersten Programmierversuchen.

Zu den verworfenen Ideen gehört zum Einem der Einsatz von Lichtschranken beim Greifer. Der Greifer sollte sich schließen, sobald der Ball die Schranke unterbricht. Doch dies führte zu zwei großen Problemen: Der Ball wurde selten erkannt, weshalb sich der Greifer nicht aktiviert hat und der Roboter immer weiter gegen die Wand fuhr. Die nächste Schwierigkeit wäre bei einem Wettbewerb ohne Grenzen zwischen den zwei Seiten entstanden, da der Greifer nur zugreift und zurückfährt, wenn er einen Ball gefunden hat. Im Fall, dass das andere Team den Ball vorher wegschnappt hat, würde der Roboter konstant gegen die Wand fahren und sich nicht wenden. Diese Idee wurde später durch die zwei Drucksensoren vor dem Greifer ersetzt.

Eine anfängliche Idee war es, die Motoren zu sichern, damit diese nach längerer Fahrt nicht verrutschen würden. Doch nach der Umsetzung dieser Idee kam ein anderes Problem auf. Die verwendeten Teile bremsten die Rotation des Motors, wodurch dieser langsamer wurde und teils auch stockte. Die Teile wurden anschließend entfernt, wodurch die Motoren, das Getriebe und letztendlich die Räder wieder einwandfrei funktionierten.
Im Hinblick auf die Software gab es zuerst die Idee, dass ein zweidimensionales Array die beste Lösung zum Merken der Kosten wäre. So hätte man jedoch immer zwei Variablen gebraucht, was die Schleifen unnötig lang werden ließ.

2.2 Arbeitsaufteilung und Probleme

Marie	**Franziska**
Anbringen der Drucksensoren	
Bau des Skeletts	Montierung der Getriebe und der Räder
Anbringen der Sensoren	Bau des Greifers
Befestigung von Greifer, Motoren, Akkuhalterung und Stützrad	AksenBoard-Halterung
Zusammenbau des Servomotors	
AksenMain, Fahre_nach_Plan, Fahre_gerade	
Richtung_bestimmen	Indexliste_erstellen
Rueckweg_bestimmen	Kosten_berechnen
Liste_invertieren	naechsten_Knoten_bestimmen
	Wenden + Drehen

Die Aufteilung der Arbeit verlief sehr gut. Es wurde parallel an Bau und Programmierung des Roboters gearbeitet und sich gegenseitig unterstützt, sobald größere Probleme auftraten.

3. Hardware

3.1 Motoren und Getriebe

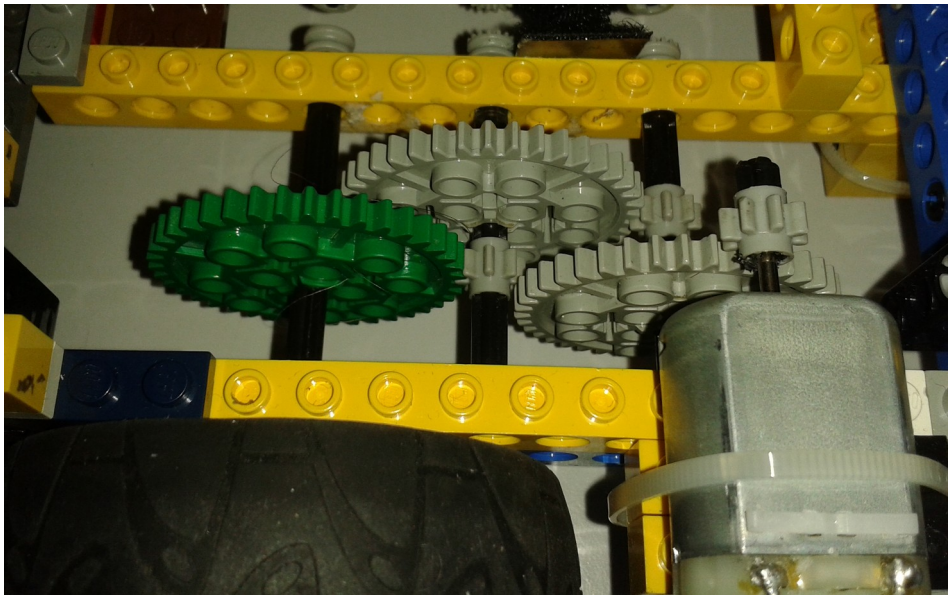


Abbildung 1: Motor mit 1:125 Getriebeuntersetzung

S.A.L.A.T. besitzt zwei Motoren, die für den Antrieb der jeweiligen Räder des Roboters zuständig sind. Somit bilden sie das Grundgerüst für seine Fortbewegung. Das Getriebe eines Rades besteht aus sechs Zahnrädern. Drei davon sind die kleinsten Modelle mit nur acht Zähnen. Die anderen drei gehören zu den größten Modellen, welche 40 Zähne haben. So bildet sich ein guter Ausgleich zwischen kleinen und großen Zahnrädern. Eines der kleinen haben wir an dem Motor befestigt. Dieses ist wiederum mit einem großen Zahnrad verbunden, was eine Untersetzung von 1:5 bedeutet. Genau die gleiche Konstruktion haben wir im Getriebe noch zweimal, also insgesamt dreimal, verwendet. Schlussendlich ergibt sich daraus eine Untersetzung von 1:125. Das heißt, der Motor muss sich 125 mal drehen, damit sich das Rad einmal komplett gedreht hat.
Diese Untersetzung wurde gewählt, damit der Roboter zuverlässig fahren kann und nicht von den Linien abkommt. Bei zu hoher Geschwindigkeit wäre die Wahrscheinlichkeit gegeben, dass der Roboter bei Kurven nicht schnell genug die Linie finden kann und er schon bei der nächsten Kreuzung wäre. Das hat zur Folge, dass er auf gerader Strecke dagegen nur recht langsam voran kommt und er gegenüber anderen Robotern einen

Nachteil hat. Dies ist aber notwendig, weil S.A.L.A.T. sehr lang ist und daher kaum Zeit für ihn bleibt, eine Kurve rechtzeitig zu beenden.

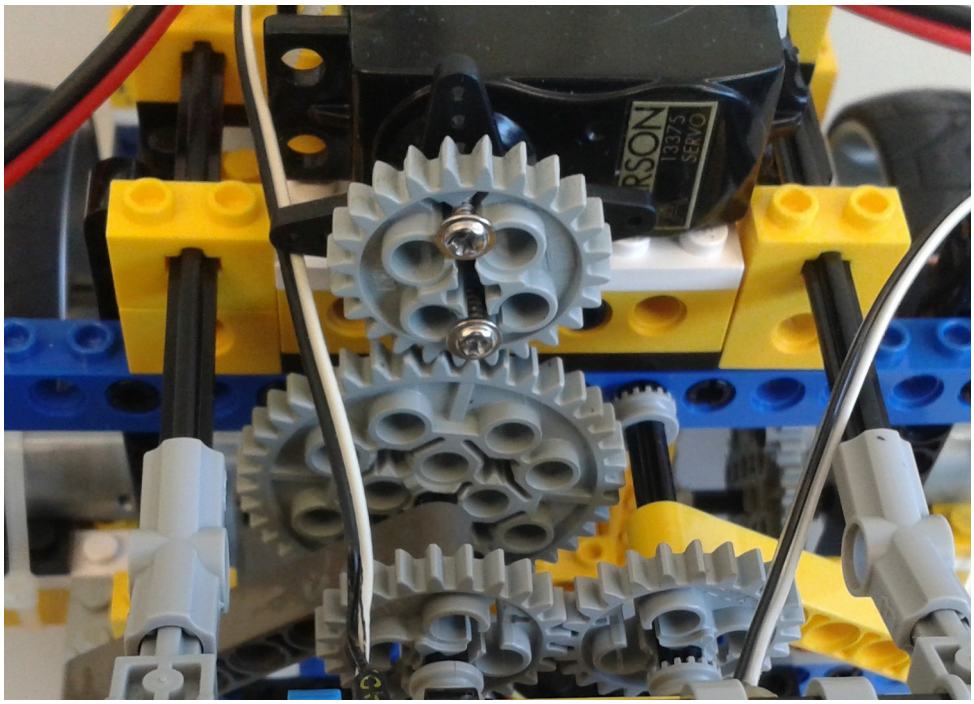


Abbildung 2: Servomotor mit Getriebe

Ein weiterer Motor befindet sich direkt vor dem Greifer. Im Gegensatz zu den anderen beiden handelt es sich hierbei um einen Servomotor. Er dreht sich nicht konstant weiter, sondern nur bis zu einem bestimmten Winkel. Damit der Greifer offen bleibt, setzen wir den Motor zu Beginn auf 45 Grad. Soll er sich schließen, dann dreht er sich bis zur 10 Grad Stellung.

3.2 Sensorik

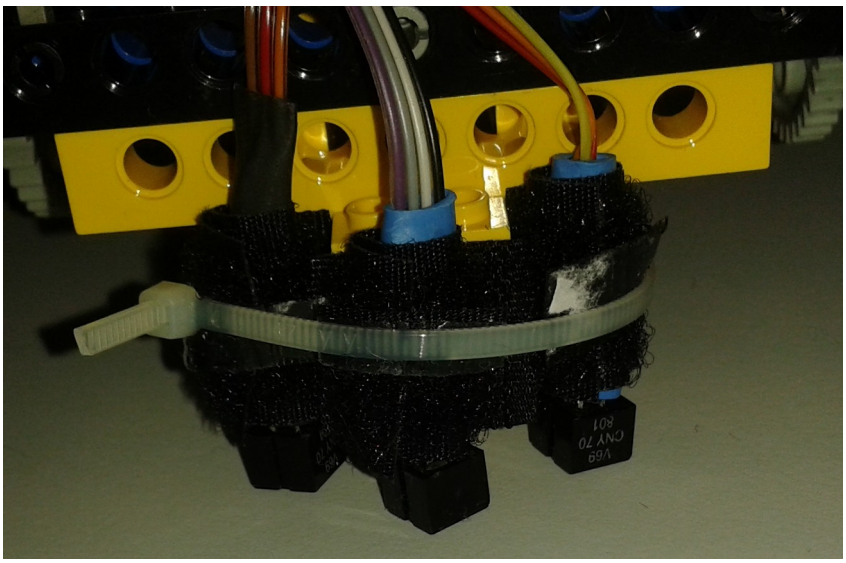


Abbildung 3: Optoreflexkoppler

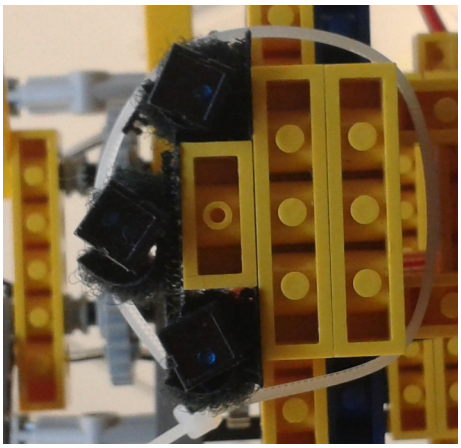


Abbildung 4: Optoreflexkoppler von unten

Die Augen des Roboters sind die drei Optoreflexkoppler unter dem Greifer. Sie lassen sich leicht anhand ihrer stabilen Klettbandbefestigung lokalisieren. Jeder von ihnen beinhaltet eine Infrarot-LED sowie einen passenden Infrarot-Empfänger. Sie hängen nur wenige Millimeter über dem Boden, um gewährleisten zu können, dass die Empfänger das reflektierte Licht einfangen und sich nicht vom Umgebungslicht beeinflussen. Sie dienen als wichtiges Werkzeug, damit der Roboter die Orientierung behält, da sie erkennen, wo helle und wo dunkle Farben sind. Das ist besonders nützlich zum Linienfolgen, denn diese

sind schwarz, während der restliche Boden weiß ist.

Der Optoreflexkoppler in der Mitte ist ein Stück versetzt zu den anderen weiter vorne befestigt. Dadurch bilden sie einen Art Bogen statt einer Linie und die seitlichen Sensoren können früher feststellen, wenn er vom Weg abkommt, was wiederum eine schnelle Richtungskorrektur bedeutet.

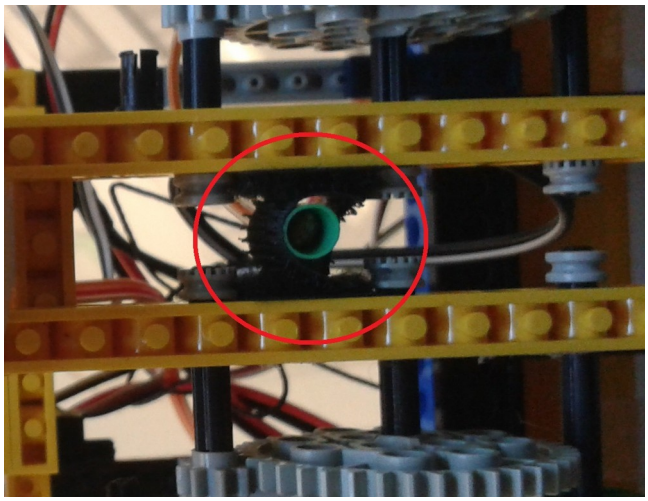


Abbildung 5: Photodetektor

Zudem hat S.A.L.A.T. einen Photodetektor in der Mitte, der das Lichtsignal empfangen soll, das als Startzeichen gilt. Erst wenn er Licht sieht, darf er anfangen zu fahren. Ein Teil des Planens geschieht jedoch schon vorher.

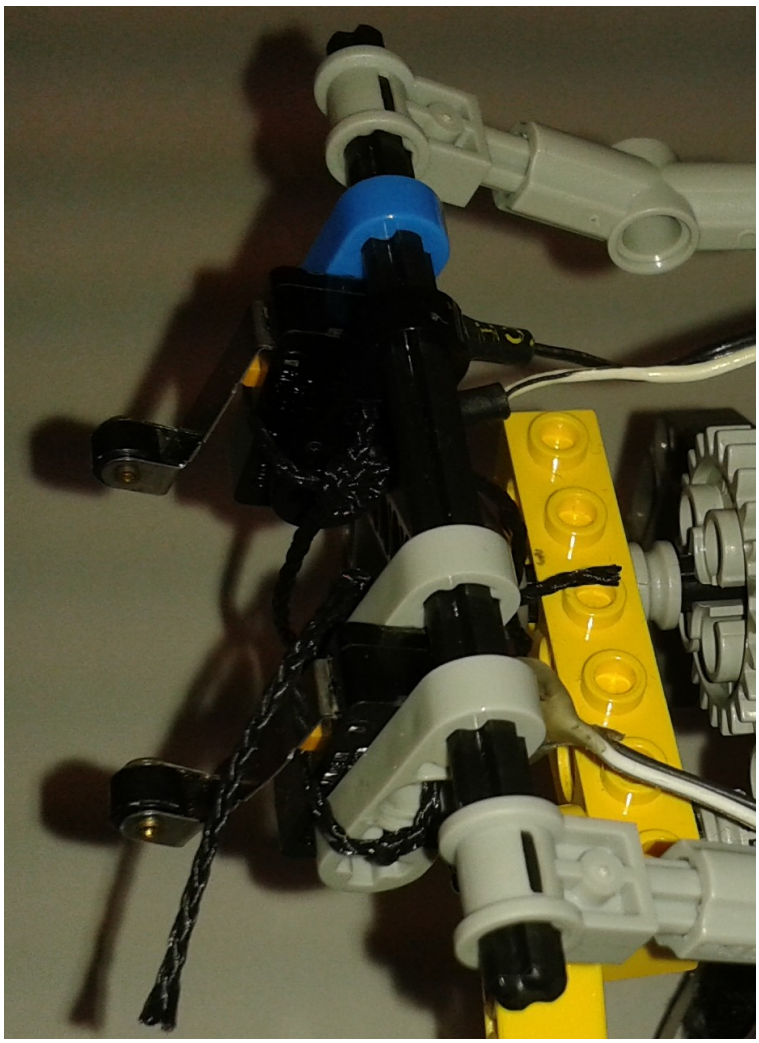


Abbildung 6: Zwei Drucksensoren vor dem Greifer

Als vordersten Punkt des Roboters sind zwei Drucksensoren, bzw. Schalter, angebracht worden. Sie sind die ersten Kontaktpunkte, sollte er gegen eine Wand fahren. Wenn sie ausgelöst werden, geben sie das Zeichen, dass der Greifer sich schließen kann, um den Fahrgast einzusammeln. Die Haltestellen der Fahrgäste liegen alle am Rand, wodurch es eine recht zuverlässige Methode ist, ihre Position zu finden. Sie haben auch den Vorteil gegenüber anderen Sensoren, dass selbst, wenn sich kein Fahrgast mehr dort aufhält, der Roboter problemlos weiterfahren kann.

3.3 Sonstiges

Um den Roboter, trotz Aksenboard und Batterie, möglichst leicht zu halten, wurden zum Aufbau des Skelettes überwiegend Lego Technic Bausteine mit Löchern verwendet. Zur Stabilisierung des gesamten Konstrukts wurden überwiegend 1xn und 2xn Technic- und normale Platten sowie Liftarme, Pins, Achsen und Verteiler verwendet. Somit wurde das

Herausfallen von Teilen und der Zusammenbruch des gesamten Skelettes verhindert. Das Aksenboard und der Akku bekamen eine Art Behälter, um nicht bei der Fahrt herauszurutschen oder dergleichen.

Abbildung 7: Stützrad am Ende des Roboters

Zur Fortbewegung des Roboters wurden die zwei Räder in der relativen Mitte des Konstruktes eingebaut. Um von hinten die nötige Stabilität und Beweglichkeit beim Fahren zu gewährleisten, da der Schwerpunkt durch Batterie und Aksenboard hinten ist, wurde ein Stützrad eingebaut. Dafür hat man zwei Felgen miteinander verbunden und am Ende des Roboters sicher befestigt.

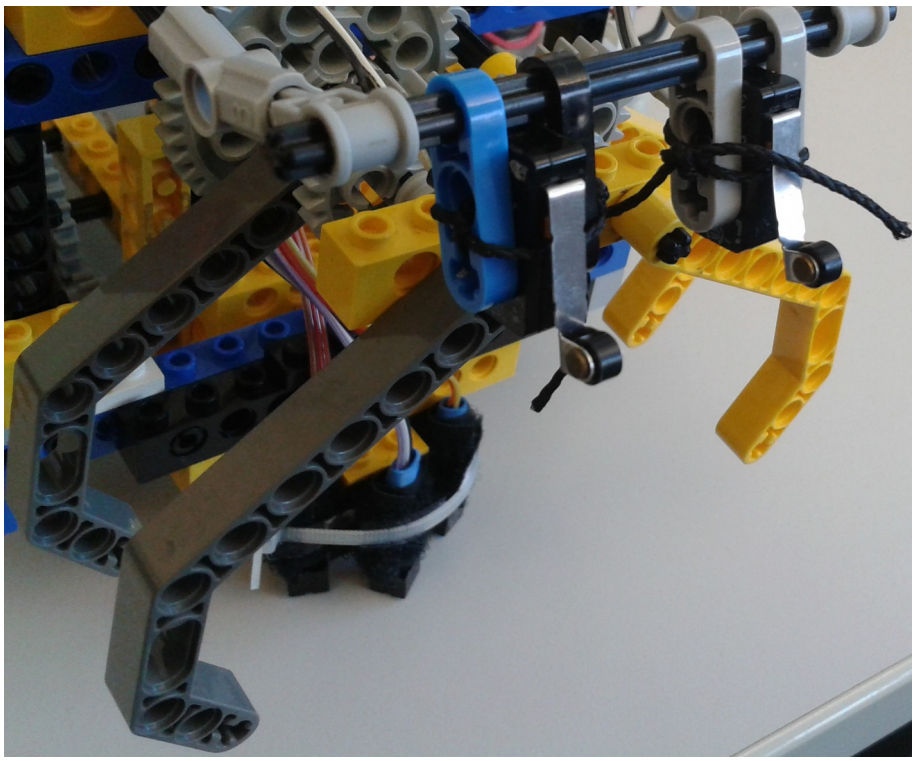


Abbildung 8: Greifer mit Drucksensoren

Um die durch Kugeln auf DUPLO Steinen dargestellten Fahrgäste sicher zu transportieren, wurde ein Greifer vorne am Roboter angebracht. Dieser besteht aus vier Liftarmen, der mit Achsen, Verteilern und Technic Steinen am Roboter angebracht wurde. Um zu Beginn des Wettbewerbes den Start manuell am Aksenboard und nicht direkt im Programmcode einzustellen, wurden die Dipschalter benutzt. Je nachdem, wo der Startpunkt des Roboters ist, wurde Schalter eins auf 0 oder 1 gesetzt.

3.4 Schwierigkeiten

Eine Schwierigkeit im Bereich der Hardware war der Umgang mit einigen defekten Bausätzen. Zum Beispiel waren zwei von drei Optoreflexkopplern defekt, ein Drucksensor reagierte sehr schwer und ein defekter Servomotor war vorhanden. Somit fuhr unser Roboter zuerst nicht auf der Linie trotz richtigen Codes. Er erkannte die Wand durch zu spätes Reagieren des Drucksensors nicht und startete ebenso nicht den Greifer.
Dieses Problem ließ sich teils erst sehr spät lösen, da man nicht von defekten Teilen sondern von Fehlern im Programmiercode ausging. Erst durch die Überprüfung einzelner Teile und das Austauschen mit anderen konnten die Probleme behoben werden.
Manche Komponenten ließen sich sehr schwer anbringen, allen voran die Verbinder, die jedes Mal herausgesprungen sind, sobald man das Getriebe oder die Räder verstellte.

Der Akkustand sowie die Art des Akkus haben einen erheblichen Einfluss auf die Fahreigenschaften des Roboters. Oftmals fuhr er viel langsamer oder konnte die Kurven nicht mehr richtig fahren, sodass er die Linie verfehlte.

4. Software

4.1 Linien folgen

Wie schon bei der Sensorik erklärt, ist das Linienfolgen überaus wichtig, denn darauf basieren im Grunde alle anderen Kommandos. Könnte der Roboter sich nicht an den Linien orientieren, wäre er völlig verloren auf der Karte und käme niemals zu den Fahrgästen. Im schlimmsten Fall behindert er sogar die anderen Teilnehmer.
Die Implementierung als Code erwies sich als relativ einfach, wenngleich es doch einige Anläufe benötigte, die richtigen Bedingungen zu finden. Die Funktion wurde als „Fahre_gerade“ bezeichnet. Sie wird aufgerufen, wenn der Roboter nur geradeaus fährt und wenn er nach dem Abbiegen sicher bis zur nächsten Kreuzung gelangen soll. Beides sind Kommandos, die nur Vorwärtsfahren beinhalten, weshalb die Richtung der Motoren zunächst zurückgesetzt wird.
Dann erfolgt der Hauptteil: eine Schleife, die solange erfüllt ist, bis die nächste Kreuzung erkannt wurde. Ist das der Fall, wird das darauffolgende Kommando ausgeführt.
Die drei Optoreflexkoppler werden bei jedem Schleifendurchlauf abgerufen. Anschließend werden ihre Werte in mehreren Bedingungen überprüft. Fährt der Roboter perfekt auf der Linie, zeigt der mittlere einen Wert über 100 an und er kann mit Höchstgeschwindigkeit weiterfahren. Oftmals ist das jedoch nicht so und er fährt etwas schief. Dafür gibt es links und rechts die Sensoren, die bei hohen Werten einleiten, dass eines der Räder stark verlangsamt wird. Befindet sich der Roboter bspw. links von der Linie, verliert das rechte Rad an Tempo.
Die Funktion wird auch aufgerufen, wenn der Roboter auf den Fahrgast zusteuert. Deshalb gibt es eine weitere Bedingung, in der die Drucksensoren abgefragt werden, denn dann gilt es nicht, die nächste Kreuzung zu erkennen, sondern die Wand. Es werden daraufhin die Kommandos zum Ball aufnehmen und zum Wenden ausgeführt und schließlich springt er aus der Schleife, um den Rückweg zu fahren.

4.2 Fahrkarte erstellen

Kurz vor dem Start erhält S.A.L.A.T. den Fahrauftrag aus der Header-Datei, der in ein neues Array übertragen wird, um es überschreiben zu können. Dieses Array ist so groß wie die Anzahl der Knoten der realen Karte. Daraus ergibt sich eine Länge von 70.
Der erste Schritt zum Planen des Fahrweges besteht aus dem Ersetzen der Punkte. Die Punkte stellen die befahrbaren Knoten dar, doch das reicht nicht aus für eine optimale Wegbestimmung. Um das zu bewerkstelligen, wurde die Breitensuche eingesetzt. Dieses Verfahren ist dafür gedacht, kleine Datenmengen zu durchsuchen und eine optimale Lösung zu finden. Die Welt dieses Projekts eignet sich dafür sehr gut, da der Suchbaum nicht sehr groß werden kann und daher eine hohe Zeit- und Speicherkomplexität verhindert wird. Andernfalls wären Bestensuchen und/oder heuristische Suchverfahren, wie die A*-Suche, zu bevorzugen gewesen. Diese hätten jedoch eine größeren Programmieraufwand verursacht.

```
void Kosten_berechnen(){

    node = start;
    //Startkosten = 0
    auftrag[node] = 48;

    while (auftrag[node + 1] == '.' || auftrag[node - 1] == '.' || auftrag[node + 7] == '.' || auftrag[node - 7] == '.'){

        /*
        Zuteilung der Kosten für die Nachbarn des aktuellen Knotens
        Nachbarskosten: Kosten des aktuellen Knotens + 1
        Überprüfung, ob Nachbarsknoten im Array liegt
        */
        if (auftrag[node + 1] == '.' && node + 1 >= 0 && node + 1 <= 69)
            auftrag[node + 1] = auftrag[node] + 1;//rechts
        if (auftrag[node - 1] == '.' && node - 1 >= 0 && node - 1 <= 69)
            auftrag[node - 1] = auftrag[node] + 1;//links
        if (auftrag[node + 7] == '.' && node + 7 >= 0 && node + 7 <= 69)
            auftrag[node + 7] = auftrag[node] + 1;//unten
        if (auftrag[node - 7] == '.' && node - 7 >= 0 && node - 7 <= 69)
            auftrag[node - 7] = auftrag[node] + 1;//oben

        //freie Nachbarn + geringtse Kosten -> nächster Knoten
        node = naechsten_Knoten_bestimmen();

    }
}
```

Abbildung 9: Funktion zur Kostenberechnung

Begonnen wird mit dem Startknoten, der vorher festgelegt wurde und dem die Kosten „0“ zugeteilt werden. Im gleichen Array wird „.“ des Startindexes also durch „0“ ersetzt. Als nächstes werden die Nachbarn ausfindig gemacht, die ebenfalls mit Kosten ausgestattet werden. Sie entsprechen den Kosten des aktuellen Knotens + 1. Es wurde davon

ausgegangen, dass der Abstand zwischen alle Knoten stets 1 Kostenpunkt beträgt. In weiteren Überlegungen hat sich allerdings ergeben, dass das nicht unbedingt so sein muss, denn mehrfaches Abbiegen nimmt mehr Zeit in Anspruch, als es eine lange Geradeausfahrt würde. Im Endergebnis ist es nur bei der Überlegung geblieben, da in den Testfahrten kein übermäßiges Abbiegen festgestellt werden konnte.

```
unsigned char naechsten_Knoten_bestimmen(){
    unsigned char min = auftrag[0], i, naechster_Knoten = node;

    //Minimum suchen, um geringste Kosten zu finden
    for (i = 1; i<70; i++) {
        if (auftrag[i] < min && auftrag[i] != '.' && auftrag[i] != 'F' && (auftrag[i + 1] == '.'
            || auftrag[i - 1] == '.' || auftrag[i + 7] == '.' || auftrag[i - 7] == '.')) {

            min = auftrag[i];
            naechster_Knoten = i;
        }
    }
    return naechster_Knoten;

}
```

Abbildung 10: Funktion zur Knotenbestimmung

Haben alle Nachbarn ihre Kosten erhalten, wird ein neuer Knoten gesucht, der expandiert werden soll. Das gesamte Array, jetzt mit einigen Kosten gefüllt, wird durchlaufen. Kandidaten zum Expandieren sind diejenigen, die die geringsten Kosten enthalten und außerdem Nachbar besitzen, die noch expandiert werden müssen. Damit wird ausgeschlossen, dass bereits besuchte Knoten ein weiteres Mal expandiert werden. Diese Schritte werden so oft ausgeführt, bis keine Nachbarknoten mehr gefunden werden können, die noch nicht besucht worden sind.

4.3 Routenplanung

```
void Indexliste_erstellen(unsigned char min, unsigned char index){
    unsigned char j, zaehler = 0, i = 1, neues_min = 0, neuer_index = 0;

    //F als ersten Punkt einfügen
    indexliste[0] = index;

    //gesuchtes F ersetzen
    auftrag[index] = 'x';

    //Überprüfen, ob F unerreichbar ist durch Blockaden
    if (auftrag[index + 1] == 'x' && (index + 1) % 7 != 0)
        zaehler++;
    if (auftrag[index + 7] == 'x')
        zaehler++;
    if (auftrag[index - 1] == 'x' && (index - 1) % 7 != 6)
        zaehler++;
    if (auftrag[index - 7] == 'x')
        zaehler++;

    if (zaehler < 3){
        //Überprüfen, ob Wegkosten zu diesem F berechnet werden konnten
        if (auftrag[index + 1] != '.' && auftrag[index + 7] != '.' && auftrag[index - 1] != '.' && auftrag[index - 7] != '.'){

            //bis zum Start durchlaufen, wenn Kosten = 0
            while (auftrag[index] != '0'){

                for (j = 1; j < 70; j++) {
                    if (auftrag[j] < min && (j + 1 == index || j - 1 == index || j + 7 == index || j - 7 == index)){
                        neues_min = auftrag[j];
                        neuer_index = j;
                    }
                }
                indexliste[i] = neuer_index;
                i++;
                index = neuer_index;
                min = neues_min;
            }

            //weitere Schritte zur Bestimmung der Fahrtwege
            Liste_invertieren();
            Richtung_bestimmen();
            Fahre_nach_Plan();
            Rueckweg_bestimmen();
        }
    }
}
```

Abbildung 11: Funktion zur Indexlistenerstellung

Um die korrekte Routenplanung zu gewährleisten, musste zunächst eine Indexliste mit den Positionen der nächsten Kreuzungen als Array erstellt werden. Das erste Element der Indexliste ist dabei das letzte „F“, das Ziel, im gegebenen Array der Fahrkarte. Das gesuchte Ziel wird danach durch ein „x“ ersetzt, damit es bei der nächsten Routenplanung nicht mehr als mögliches Ziel gesehen werden kann. Für den Fall, dass ein Fahrgast durch Blockaden vom Startpunkt aus nicht mehr zu erreichen ist, wird überprüft, wie viele Blockaden um das Ziel liegen. Für jede Blockade wird der Zähler um eins erhöht, solange er unter drei bleibt, kann „F“ erreicht werden, andernfalls wird es als Ziel nicht

wahrgenommen.
Als nächstes werden die vorhanden Wegkosten zum Fahrgast überprüft. Dafür werden die Nachbarn von „F“ kontrolliert, dass sie keine Punkte sind. Andernfalls heißt es, dass das gegebene „F“ nicht vom Startpunkt aus zu erreichen ist.
Die erste Indexliste geht dabei vom Ziel zum Start. Um den eigentlichen Weg zu erhalten, muss diese Liste zuerst invertiert werden. Nach dem Invertieren der Liste kann nun der Routenplan erstellt werden.

```
void Richtung_bestimmen(){
    unsigned char i = 0, j, d;
    unsigned char r = 'N'; //Richtung
    unsigned char ziel = 'k';

    for (j = 1; j < 16; j++){

        //nach oben, rechts, links oder unten fahren aus Sicht der Vogelperspektive und NICHT des Roboters
        if (indexliste[i + 1] - indexliste[i] == -7)
            ziel = 'o';
        if (indexliste[i + 1] - indexliste[i] == 1)
            ziel = 'r';
        if (indexliste[i + 1] - indexliste[i] == -1)
            ziel = 'l';
        if (indexliste[i + 1] - indexliste[i] == 7)
            ziel = 'u';


        //Richtung Norden
        if (ziel == 'o' && r == 'N'){
            d = 'G';
            r = 'N';
            fahrplan[i] = 'G';
        }
        else if (ziel == 'r' && r == 'N'){
            d = 'R';
            r = 'O';
            fahrplan[i] = 'R';
        }
        else if (ziel == 'l'  && r == 'N'){
            d = 'L';
            r = 'W';
            fahrplan[i] = 'L';
        }
```

Abbildung 12: Funktion zur Fahrplanerstellung durch Richtung

Für die Routenplanung wird in der Funktion ein Fahrplan als Array erstellt. Der Plan gibt an, ob der Roboter, abhängig von seiner Richtung und der Postion des Zieles, nach Norden „N“, Osten „O“, Westen „W“ oder Süden „S“ fahren muss. Als Startrichtung für jede Routenberechnung wird Norden festgelegt, d.h. am Anfang schaut der Roboter immer geradeaus. Zur Bestimmung, in welcher Richtung das nächsten Zwischenziel ist, wird die

Differenz zwischen dem Nachfolger und dem aktuellen Element der Indexliste ermittelt. Anhand der Richtung des Zwischenziels und der Richtung, in die der Roboter schaut, wird der Weg des Roboters zum Fahrgast Schritt für Schritt abgearbeitet. In der Abbildung sieht man die Richtungsbestimmung für Norden und Osten. Das gleiche Prinzip wurde auch für Westen und Süden angewandt. Wenn keine Elemente mehr in der Indexliste sind, wird der Rest des Fahrplans mit „.“ gefüllt, das ist auch später zur Rückwegberechnung von Nutzen.

```
void Rueckweg_bestimmen(){
    unsigned char temp, k, l;

    //Liste invertieren
    for (k = 0, l = 15; k < 16 / 2; k++, l--){
        temp = fahrplan[k];
        fahrplan[k] = fahrplan[l];
        fahrplan[l] = temp;
    }

    //Links und Rechts vertauschen
    for (k = 0; k < 16; k++){
        if (fahrplan[k] == 'R'){
            fahrplan[k] = 'L';
        }
        else if (fahrplan[k] == 'L'){
            fahrplan[k] = 'R';
        }
    }

    //Letztes Gerade, Links und Rechts werden zu speziellen Kommandos zum Anhalten und Ball abwerfen
    for (k = 0; k < 16; k++){
        if (fahrplan[k] == 'G' && fahrplan[k+1] == '.'){
            fahrplan[k] = 'N';
        }
        if (fahrplan[k] == 'L' && fahrplan[k+1] == '.'){
            fahrplan[k] = 'W';
        }
        if (fahrplan[k] == 'R' && fahrplan[k+1] == '.'){
            fahrplan[k] = 'O';
        }
    }

    Fahre_nach_Plan();
}
```

Abbildung 13: Funktion zur Rückwegerstellung

Nachdem der Routenplan zum Ziel „F” bestimmt wurde, wird nun die Funktion „Fahre_nach_Plan“ in „Indexliste_erstellen“ aufgerufen. Sobald das Fahren abgeschlossen ist, der Ball aufgenommen wurde und gewendet wurde, ruft man die letzte Funktion „Rückweg_bestimmen“ auf.

Um den Rückweg zu bestimmen und zu fahren, muss zunächst der bereits gefahrene Routenplan invertiert werden und „R“ und „L“ im Plan vertauscht werden. Um zu verhindern, dass der Roboter nach der letzten Kreuzung bei seiner Startposition weiterfährt, was in den Testfahrten einige Male auftrat, muss man das letzte „G“, „R“ oder „L“ bearbeiten. Zum Herausfinden, wann ein Buchstabe der Letzte ist, wird im Fahrplan überprüft, ob das nachfolgende Elemente ein Punkt ist. Nachdem der Rückweg erfolgreich erstellt wurde, wird nun wieder die Funktion „Fahre_nach_Plan“ aufgerufen und ausgeführt.

4.4 Schwierigkeiten

Eine der anfänglichen Schwierigkeiten bei der Programmierung, war das Stoppen am Ende des Rückweges, da einfach den Fahrweg zu invertieren zu Problemen am Ende führte. Als ersten Versuch zur Problembewältigung sollte kontrolliert werden, ob nach der letzten Fahranweisung ein leeres Element folgt. Leider verfügt C über kein Null. Doch durch das Auffüllen des Routenplans mit Punkten konnte man das letzte Gerade, Links oder Rechts des Rückwegs ermitteln und bearbeiten, damit der Roboter stoppt.
Nach dem Stoppen kam es zu einer anderen Problematik: das Wenden am Startpunkt und Auswerfen des Balles. Denn man musste noch abhängig von der Richtung, aus der S.A.L.A.T. kam, anpassen, dass er sich zur Rückwand dreht, den Ball abwirft und dann eine 180° Wendung vollzieht, um wieder in Richtung Norden schauen zu können.

Abbildungsverzeichnis