Gymnasium Damme
Nordhofe 1
49401 Damme

Facharbeit im
Seminarfach
sf102

Betreuende Lehrer:
Herr Rohe
Frau Silwedel

Entwicklung und Implementation eines Algorithmus zur Selbstortung eines autonomen Rettungsroboters

Jan Blumenkamp
Streithorstweg 2b
49163 Hunteburg

Abgabedatum: 21.03.2014

Inhaltsverzeichnis

1 Einleitung

In der vorliegenden Facharbeit beschäftigt sich der Autor mit Möglichkeiten zur Selbstortung von autonomen Rettungsrobotern in autonom erstellten Karten, um Fehler in der Kartenerstellung frühzeitig erkennen zu können.

Im ersten Teil wird zunächst auf die generelle Problematik eingegangen, so wird der Wettbewerb, an dem der Roboter teilnimmt bzw. teilnehmen wird und die Aufgabenstellung, die der Roboter zu bewältigen hat, erläutert und der Entwicklungsstand des Roboters zu Beginn der Facharbeit beschrieben. Daraus resultierend werden die Schwächen des Roboters, was diese Schwächen zu Folge haben und was dagegen unternommen werden kann/muss erörtert und der Übergang zur eigentlichen Fragestellung der Facharbeit geschaffen.

Im Hauptteil der Facharbeit wird der Lösungsansatz SLAM vorgestellt. Dazu wird zu Beginn generell auf den Lösungsansatz und die Relevanz in professionellen Gebieten eingegangen und das gesamte Verfahren so beschrieben, wie es dort zu Einsatz kommt. Dann wird das Verfahren so vereinfacht, dass es auf dem zur Verfügung stehendem System funktionieren kann. Schließlich wird die Implementation dieser Vereinfachung erläutert und hinsichtlich der Laufzeit analysiert.

In der Schlussbetrachtung wertet der Autor den entwickelten Algorithmus hinsichtlich seiner Schwächen aus.

2 Die Problematik

2.1 Der Wettbewerb

Beim RoboCup Junior Rescue B Wettbewerb soll ein autonomer Roboter in einem für Rettungskräfte unzugänglichen Gebäude nach verschütteten Opfern suchen. Dieses Gebäude wird in diesem Wettbewerb für Schüler vereinfacht dargestellt durch ein rechtwinkliges Labyrinth mit Grundfliesen mit einer Größe von 30cm x 30cm; Die Opfer werden durch auf Körpertemperatur aufgeheizte elektrische Heizplatten dargestellt. Das Labyrinth ist dem Roboter unbekannt, im besten Fall erstellt der Roboter daher eine Karte, mit deren Hilfe der Roboter systematisch nach Opfern suchen kann (Absuchen aller Fliesen nach Opfern). Wenn der Roboter das gesamte Labyrinth nach Opfern durchsucht hat, soll er zum Eingang des Labyrinths bzw. des Gebäudes zurückkehren. (vgl. Caldeira [u.a.] 2013, S. 1, http://www.rcj.robocup.org/)

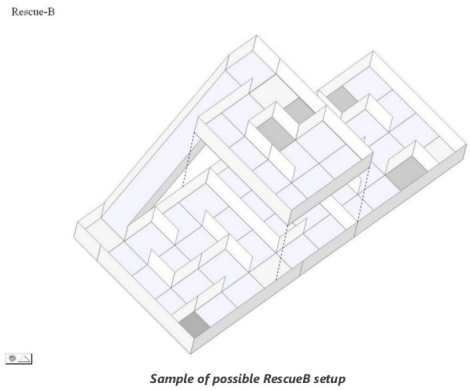

Abbildung 1: Mögliche RoboCup Junior Rescue B Arena

2.2 Aktueller Stand des Roboters

Der Roboter ist dazu in der Lage, zuverlässig eine Karte des Labyrinths zu erstellen und mithilfe dieser Karte systematisch und gezielt zu unbesuchten Fliesen zu navigieren, auf denen sich weitere Opfer befinden könnten. Nachdem das gesamte Labyrinth nach Opfern durchsucht wurde, kehrt der Roboter zum Start zurück.

2.3 Motivation

Obwohl der Roboter sehr zuverlässig und genau arbeitet, kann es vorkommen, dass er z.B. aufgrund von auf dem Boden liegendem Geröll und anderen, möglichen Hindernissen oder einfach nur aufgrund von Fahrungenauigkeiten (z.B. Schlupf der Räder und daraus resultierende Fehler beim Ausrichten an den Wänden/Kollision mit den Wänden) die aktuelle Position in der Karte und somit die Orientierung verliert, woraufhin die Karte nutzlos wird. Der Roboter kann dann nicht

mehr gezielt nach Opfern suchen und in einem Wertungslauf muss ggf. eingegriffen werden, was Punktabzüge zu Folge hat. Idealerweise müsste der Roboter also erkennen können, wenn ein Fehler in der Karte vorliegt und müsste diesen Fehler dann selbstständig korrigieren. In dieser Facharbeit will sich der Autor mit dieser Problematik befassen, eine Lösung finden und diese im Roboter implementieren.

3 Entwicklung und Implementation eines Algorithmus zur Selbstortung eines autonomen Rettungsroboters

3.1 Lösungsansatz: SLAM

> *„Models of the environment are needed for a wide range of robotic applications, from search and rescue to automated vacuum cleaning. Learning maps has therefore been a major research focus in the robotics community over the last decades.“ (Stachniss 2009, S. 3)*

Der RoboCup Wettbewerb existiert nicht nur für Schüler, auch Studenten entwickeln Roboter, die grundsätzlich die gleiche Aufgabenstellung wie die Schüler bewältigen müssen, insgesamt ist dort das dargestellte Gebäude allerdings wesentlich komplexer (größer, nicht nur rechtwinklig etc. (vgl. Abbildung 2)). Aufgrund der Komplexität der Aufgabenstellung ist es für diese Roboter schon lange elementar wichtig, die Orientierung nicht zu verlieren. Eine gängige Methode zur Selbstortung ist SLAM[1]. SLAM ist weniger ein Algorithmus als mehr eine Sammlung von Algorithmen und kann daher als Verfahren bezeichnet werden. (vgl. Stachniss 2009, S. 4ff.)

Abbildung 2: Mögliche RoboCup Rescue Major Arena

Damit ein Roboter autonom ein unbekanntes Gebiet erkunden kann, sind im Wesentlichen drei Schritte notwendig, und zwar die Erstellung einer Karte der Umgebung, die parallele Selbstlokalisierung in dieser Karte und die Planung eines Pfades, um gezielt an bestimmte Orte zu navigieren, im einfachsten Fall um die Karte zu vervollständigen. SLAM kombiniert dabei die Erstellung der Karte und die Loka-

1 „Simultaneous Localization And Mapping“ (Stachniss 2009, S. 3)

lisierung des Roboters in dieser erstellten Karte. Die Planung des Pfades erfolgt unabhängig von SLAM, aber basierend auf die von SLAM erstellte Karte. Dieser Prozess wird „integrated approaches“ oder auch SPLAM („Simultaneous Planning, Localization And Mapping“) (Stachniss 2009, S. 4) genannt (siehe Abbildung 3). Da für eine akkurate Lokalisierung eine gute Karte und zum zuverlässigen Erstellen der Karte eine akkurate Position benötigt wird, wird im Zusammenhang des Begriffs SLAM oft auch auf das Henne-Ei-Problem verwiesen. (ebd.)

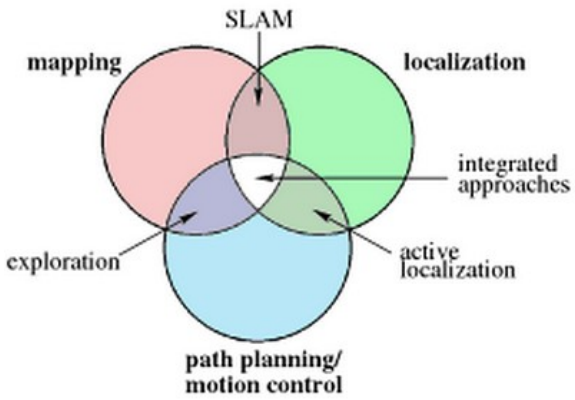

Abbildung 3: Differenzierung der Teilgebiete die nötig sind, um in einem Roboter ein genaues Abbild der Umgebung zu erzeugen

> *„The SLAM process consists of a number of steps. The goal of the process is to use the environment to update the position of the robot. Since the odometry of the robot (which gives the robots position) is often erroneous we cannot rely directly on the odometry.“ (Riisgard [u.a.] 2005, S. 10)*

Das Verfahren SLAM lässt sich wie folgt beschrieben:

Zu Beginn hat der Roboter keine Karte und somit auch keine Informationen über die Umgebung und beginnt in einem Koordinatensystem beim Zeichnen der Karte im Ursprung. Nun müssen Informationen über die Umgebung, in der sich der Roboter befindet, in das Koordinatensystem übertragen werden. Diese Informationen werden i.d.R. bei komplexeren Robotern über ein LIDAR[2] oder über (3D-) Kameras (vgl. Engel [u.a.] 2013, S. 1) gesammelt. Der Roboter hat nun erste Informationen über die Umgebung gespeichert. Als nächstes werden markante sogenannte „Landmarks“ (Riisgard [u.a.] 2005, S. 16) aus dem Laserscan gefiltert, also markante, einfach wiedererkannbare Eigenschaften der Umgebung (z.B. lange, ebene Wände, also Linien, Ecken, oder sonstige hervorstechende Merkmale). Diese Merkmale der Umgebung werden in einer Datenbank gespeichert. Nun fährt der Roboter in eine Richtung, für die Informationen über die Umgebung fehlen, wo also noch weitere Informationen gesammelt werden müssen (nicht Teil von SLAM, hier greifen wieder die „integrated approaches“ (Stachniss 2009, S. 4)). Dabei wird

2 Light Detection and Ranging; Laserscanner, dieser scant meistens mithilfe eines Lasers ein zweidimensionales Bild der Umgebung (vgl. Fujii 2005, S. 1f)

über Odometer, also Sensoren, die die gefahrene Distanz ausschließlich darüber messen, wie weit sich z.B. ein Rad[3] gedreht hat, die neue Position des Roboters in der Karte grob geschätzt und aktualisiert. Das ist recht ungenau, da Räder i.d.R. Schlupf aufweisen. Dieser Fehler würde sich im Verlauf der Kartenerstellung summieren und die Abweichung des Roboters in der Karte von der Ist-Position des Roboters somit immer größer werden. An dieser Stelle werden die gespeicherten Merkmale der Umgebung für den Roboter interessant. Der Roboter führt erneut einen Scan der Umgebung durch, filtert die aktuellen Merkmale heraus und vergleicht diese mit Hilfe eines EKF[4] mit den in der Datenbank in der Nähe des Roboters gespeicherten Merkmalen. Nun wird eine eventuelle Fehlposition des Roboters in der Karte (also eine Abweichung von der Ist-Position des Roboters in der realen Umgebung von der vermuteten Position des Roboters in der Karte) ausgeglichen bzw. korrigiert. Der gesamte Prozess beginnt nun von vorne. Auf diese Art und Weise lässt sich sehr präzise eine Karte der Umgebung erstellen. (vgl. Riisgard [u.a.] 2005, S. 10ff.)

3.2 Entwicklung und Beschreibung des Algorithmus

Da dem im Rahmen dieser Arbeit zu verwendendem Roboter begrenzte Ressourcen zur Verfügung stehen und er insgesamt recht einfach aufgebaut ist, kann keine fertige SLAM Bibliothek genutzt werden[5]. Dem Roboter liegt ein auf 16MHz getakteter Atmel ATmega2560 Mikrocontroller mit 256kB ROM und 8kB RAM (vgl. Atmel 2011, S. 1, http://www.atmel.com/) zu Grunde. Weiterhin stehen kein LIDAR, sondern lediglich zehn einzelne Infrarotentfernungssensoren mit einer Entfernung von 4cm – 30cm (vgl. Sharp 2006, S. 1, http://www.sharpsma.com/) zur Verfügung, die rund um den Roboter angeordnet sind, wobei im-

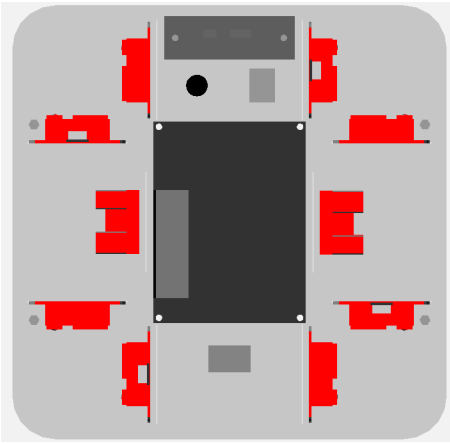

Abbildung 4: Oberer Teil des Chassis des Roboters im Computermodell von unten betrachtet, Entfernungssensoren rot gefärbt

3 Abhängig von der Bauweise des Roboters; auch fliegende Roboter können via SLAM eine Karte der Umgebung anfertigen (vgl. Engel [u.a.] 2013, S. 1)

4 „Extended Kalman Filter“ (Riisgard [u.a.] 2005, S. 28ff)

5 Hier gibt es viele Open-Source Ansätze, z.B. „OpenSLAM“ (Stachniss [u.a.] 2014, http://www.openslam.org/)

mer mindestens zwei Sensoren in eine Richtung zeigen (siehe Abbildung 4). Eine Komplettansicht des Roboters befindet sich unter Abbildung 9 im Anhang. Da auch das Labyrinth im Wettbewerb vereinfacht dargestellt ist (Fliesen mit gleicher, einheitlicher Größe, rechtwinklig), werden nicht zwingend mehr Sensoren benötigt. Das SLAM Verfahren soll stark vereinfacht und auf das Nötigste reduziert implementiert werden (genau wie das im Schülerwettbewerb verwendete Labyrinth stark vereinfacht wurde). Diese Vereinfachung sieht wie folgt aus:

Der Roboter beginnt in einem Koordinatensystem mit einer Auflösung von 30cm[6] in der Mitte einer Fliese. Im Normalfall erkennen immer mindestens zwei Entfernungssensoren (da immer mindestens zwei Sensoren in eine Richtung zeigen) eine Wand, wenn sich eine Wand neben dem Roboter in der entsprechenden Richtung befindet. Da das Labyrinth rechtwinklig ist und jede Wand, wenn eine Wand existiert, eine Länge von 30cm hat, würde eine fertige SLAM Bibliothek hier unnötige Arbeit beim Sortieren bzw. Herausfiltern der markanten Merkmale verrichten, denn eine vorhandene oder nicht vorhandene Wand kann prinzipiell immer als markantes Merkmal gesehen werden bzw. es gibt keine anderen Merkmale als die Wände in unmittelbarer Nähe zum Roboter. Der Roboter hat also pro Fliese immer vier Merkmale (auch keine Wand ist ein Merkmal), an denen er sich orientieren kann. Da die Auflösung des Labyrinths 30cm beträgt, würde es auch keinen Sinn ergeben, weniger als 30cm pro Einheit zu fahren bzw. weniger als 90° zu drehen (Rechtwinkligkeit des Labyrinths). Der Roboter hat also in die Karte ggf. existierende Wände mit einem positiven Wert oder auch nicht existierende Wände mit einem negativen Wert im Speicherplatz eingetragen und fährt nun in eine Richtung, in der sich keine Wand befindet. Nachdem er 30cm gefahren ist, springt der Roboter in eine temporäre, neue Karte (im Folgenden „temporäre Karte“), die unabhängig von der korrekten, bereits gezeichneten Karte (im Folgenden „primären Karte“) ist. Hier werden genau wie in der primären Karte vorhandene und nicht vorhandene Wände eingezeichnet. Nun werden die in der temporären Karte gezeichneten Wände mit den Wänden, die in der primären Karte an der Position, an der der Roboter nun sein müsste, verglichen. Im schlimmsten Fall (zu Beginn des Erstellungsprozesses der Karte, wenn der Roboter erst wenige Fliesen gefahren ist) hat der Roboter nun nur eine Wand, die er mit den neu gesammelten Informa-

6 Kantenlänge der Fliesen, aus denen das Labyrinth besteht (vgl. Caldeira [u.a.] 2013, S. 2, http://www.rcj.robocup.org/)

tionen vergleichen kann (in der Richtung, aus der der Roboter gekommen ist, kann logischerweise keine Wand sein; Wenn sich hier plötzlich eine Wand befindet, muss ein Fehler passiert sein). Beobachtungen des Roboters beim Vergleichen der Wände haben ergeben, dass i.d.R. ein Fehler vorliegt, wenn weniger als drei Wände übereinstimmen. Wenn aber drei oder vier Wände übereinstimmen bzw. keine Informationen über eine mögliche Übereinstimmung zur Verfügung stehen, überträgt der Roboter die gewonnenen Informationen aus der temporären Karte in die primäre Karte, löscht die Informationen in der temporären Karte und beginnt von vorne (fährt also wieder in die Richtung, in der keine Informationen zur Verfügung stehen bzw. in der es weitergeht oder später in die Richtung, in die der Pfad zu einer Zielfliese berechnet wurde). Wenn allerdings nicht genug Übereinstimmung existieren, muss ein Fehler passiert sein (der Roboter ist wahrscheinlich gegen ein Hindernis, eine Wand o.Ä. gefahren und steht nun schief). Damit der Roboter fortfahren kann, muss er zunächst in eine garantiert korrekte Position (d.h. auf die Mitte einer beliebigen Fliese) fahren, er darf also nicht schräg im Labyrinth stehen. Das ist gewährleistet, wenn der Roboter zunächst so lange in eine Richtung fährt, bis sich hinter ihm eine Wand befindet (nun steht der Roboter in einer Achse des Koordinatensystems der Karte wieder im 30cm-Raster), sich dann um 90° dreht und wieder in eine Richtung fährt, bis er auf eine Wand stößt. Der Roboter springt dann nicht wieder in die primäre Karte, sondern löscht die temporäre Karte und arbeitet nun vorerst mit dieser Karte weiter. Die temporäre Karte wird dabei genau so behandelt, wie die primäre Karte, es werden also alle Wände, Sackgassen und Hindernisse wie in der primären Karte eingezeichnet und genau wie in der primären Karte navigiert. Währenddessen werden nach jeder Geradeausfahrt die temporäre Karte und die ursprüngliche Karte abgeglichen, um Übereinstimmungen zu finden. Wenn der Roboter seine aktuelle Position mithilfe der temporären Karte sicher in der primären Karte identifizieren kann, wird die temporäre Karte und, der wichtigste Schritt, die Roboterposition komplett in die primäre Karte übertragen, die temporäre Karte gelöscht und von nun an mit der primären Karte fortgeführt (siehe auch Abbildung 5).

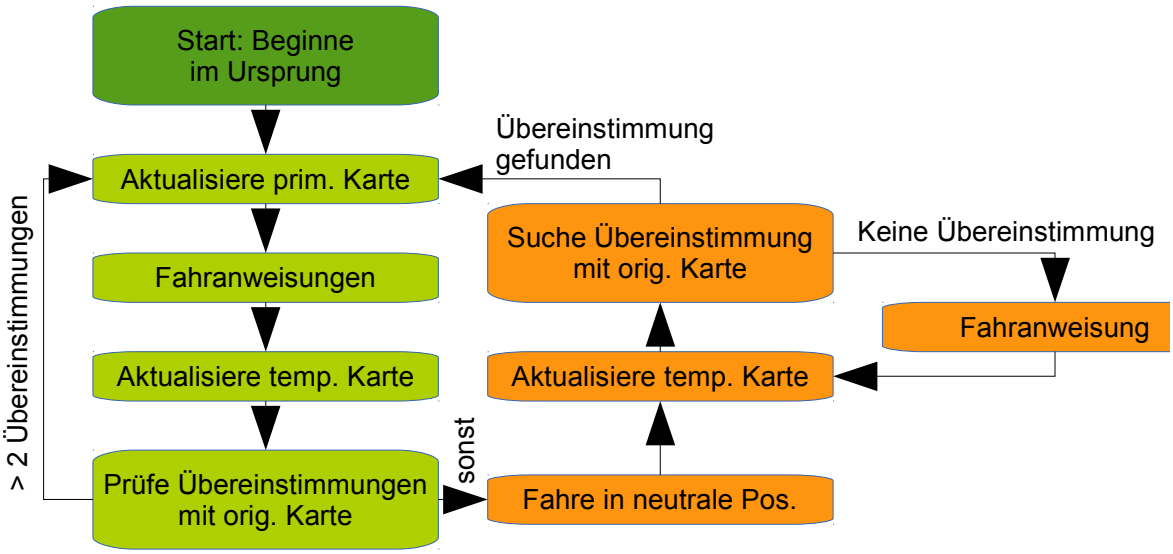

Abbildung 5: Visualisierung des vereinfachten SLAM Algorithmus

Im Folgenden soll sich auf den Algorithmus, der die Übereinstimmungen der primären mit der temporären Karte analysiert, konzentriert werden. Dieser Algorithmus ersetzt den EKF im ursprünglichen SLAM Verfahren.

3.3 Entwicklung und Implementation des Map-Matching-Algorithmus

Zum Finden der Übereinstimmungen soll der Algorithmus grundsätzlich die temporäre Karte für alle möglichen Überlappungen über die bereits (unvollständig) gezeichnete primäre Karte legen und die Übereinstimmung vergleichen. Dieses Verfahren des Durchprobierens aller Möglichkeiten wird auch „Brute-Force-Methode“ (Wolf 2003, S. 557) genannt und wird beispielsweise beim Suchen von bestimmten Textausschnitten in Texten verwendet, dort allerdings logischerweise nur in eindimensionaler Form. Dieses Verfahren soll nun um eine Dimension erweitert werden, sodass auch eine Übereinstimmung zwischen zwei zweidimensionalen Objekten, in diesem Fall Karten, gefunden werden kann.

Es werden alle möglichen Überlappungen der beiden Karten durchgegangen und für alle Überlappungen die Übereinstimmungen der Wände bestimmt. Dazu muss die Karte auch rotiert werden, genutzt wird dazu der Ansatz von Ivan Kadiyski (vgl. Kadiyski 2013, http://algorithmproblems.blogspot.de/). Der Map-Matching-Algorithmus greift auf vorhandene Konstanten, globalen Variablen, Datenstrukturen und

Funktionen der selbst entwickelten Bibliothek zum Erstellen des Labyrinths zurück. Die verwendete Programmiersprache ist C.

Die Implementation und die Erläuterungen der genutzten Konstanten, globalen Variablen, Datenstrukturen und Funktionen befindet sich im Anhang („Erläuterungen”), im Folgenden wird der Algorithmus selbst erläutert:

Der Algorithmus gibt die Fliese in der primären Karte und die Orientierung der temporären Karte zurück, für die die temporäre Karte die höchste Übereinstimmung mit der primären Karte aufweist (siehe Beispiel und Ergebnis des Algorithmus in Abbildung 6).

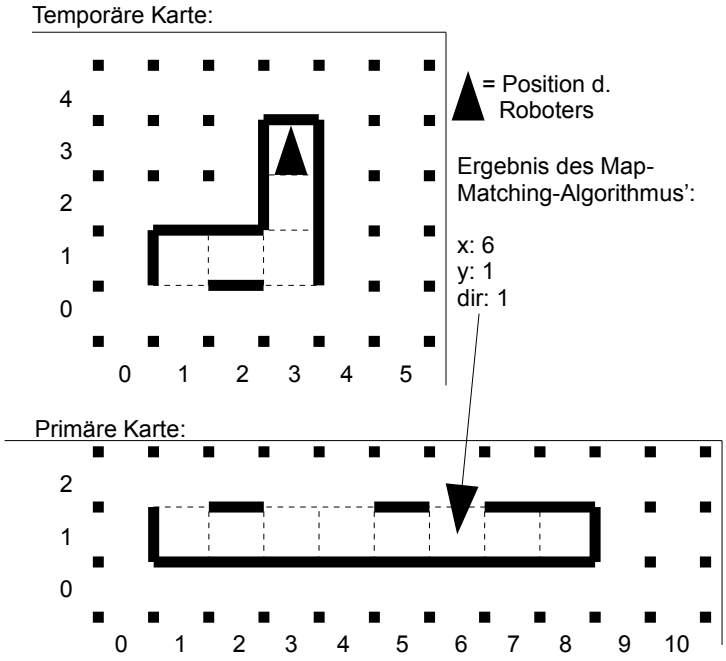

Abbildung 6: Ergebnis des Map-Maptching-Algorithmus

Im Folgenden beziehen sich alle Zeilenangaben auf den Anhang „Quellcode“.

Zunächst wird ein Datentyp `POS` mit dem Namen `maze_orig` deklariert (Zeile 3). Dieser Datentyp stellt die aktuelle Position in der primären Karte dar, für die die Übereinstimmung mit der temporären Karte geprüft wird.

Der Datentyp `COORD` mit dem Namen `maze_save` (Zeile 5) stellt die aktuelle Koordinate der temporären Karte dar, die mit der primären Karte abgeglichen werden soll.

Da die äußeren vier Zählschleifen mit `maze_orig` (Zeile 25 – 43) alle möglichen (bereits besuchten, Zählschleifen zählen bis zur Größe des primären Labyrinths) Ursprungsfliesen der primären Karte durchgehen, muss ein weiterer Datentyp

`COORD` mit dem Namen `maze_orig_save` deklariert werden (Zeile 44), der die fortlaufenden Koordinaten des originalen Labyrinths enthält (Addition von `maze_orig` und `maze_save`, Zeile 50 und 62). In folgender Tabelle wird das am Beispiel von `maze_orig.x` mit einer beliebig breiten und einer Fliese hohen primären Karte und einer drei Fliesen breiten und einer Fliese hohen temporären Karte für die ersten drei Fliesen in der primären Karte dargestellt:

Iteration	`maze_orig.x = ROB_POS_X_MIN`	`maze_save.x = ROB_POS_X_MIN`	`maze_orig_save.x = maze_orig.x + maze_save.x - ROB_POS_X_MIN`
0	1	1	1
1	1	2	2
2	1	3	3
3	2	1	2
4	2	2	3
5	2	3	4
6	3	1	3
7	3	2	4
8	3	3	5

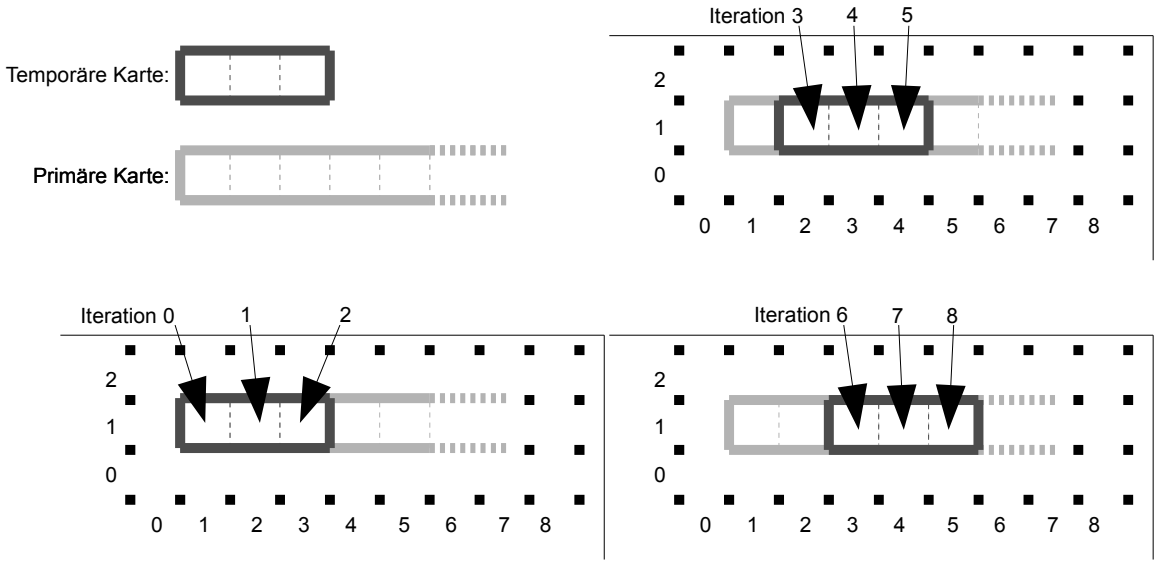

Abbildung 7: Visualisierung des Map-Matching Algorithmus'

Für jede Ursprungsfliese in der primären Karte wird die Übereinstimmung der Fliesen geprüft (d.h. für Iteration 0, 1, 2, Iteration 3, 4, 5 und Iteration 6, 7, 8 wird je

eine, also in der Tabelle insgesamt drei, Übereinstimmungen berechnet). Aktuell werden ausschließlich die Wände geprüft, der Algorithmus kann aber auch einfach auf schwarze Fliesen (No-Go-Areas für den Roboter), Hindernisse oder Rampenanschlüsse (Verbindung von zwei Etagen in der Wettkampfarena durch Rampe) (vgl. Caldeira [u.a.] 2013, S. 2ff., http://www.rcj.robocup.org/) erweitert werden.

Für jede Übereinstimmung (Zeile 78 und 79), d.h. jede passende Wand (Wert an der entsprechenden Stelle sowohl in der primären, als auch in der temporären Karte größer oder kleiner als 0) wird die ganzzahlige Variable `matches` inkrementiert (Zeile 81), wenn die zu vergleichenden Fliesen beide vom Roboter besucht worden sind (Zeile 70 und 71) (wichtig für die Berechnung der Gesamtübereinstimmung später; dafür wird für jede Fliese, für die es Übereinstimmungen geben könnte, die ganzzahlige Variable `maze_save_visitedTiles` inkrementiert (Zeile 73)). Für jede Ursprungsfliese in `maze_orig` wird nun die Übereinstimmung der beiden Karten berechnet (Zeile 88):

```
accordance = (matches * 100) / (maze_save_visitedTiles * 4);
```

Für jede Fliese kann es aktuell maximal vier Übereinstimmungen geben, multipliziert wird das mit der Anzahl der besuchten Fliesen. Dieses Produkt ist nun die Anzahl der maximal möglichen Übereinstimmung. Der Quotient aus der berechneten Übereinstimmung multipliziert mit 100 (da mit Ganzzahlen gearbeitet wird, damit das Ergebnis eine im Bereich von 0 – 100 und als Anteil in % betrachtet werden kann) und den maximal möglichen Übereinstimmung ist somit der Anteil der Übereinstimmungen ausgehend von `maze_orig`.

Für Abbildung 7 ergibt sich:

Ursprung (`maze_orig.x`)	`matches` (von 12 möglichen)	Übereinstimmung (`accordance`)
1 (Iteration 0, 1, 2)	11	91%
2 (Iteration 3, 4, 5)	10	83%
3 (Iteration 6, 7, 8)	10	83%

Auf der Basis dieser Übereinstimmung wird nun eine elementar wichtige Fallunterscheidung getroffen: Wenn die Übereinstimmung größer ist, als die bis jetzt größte berechnete Übereinstimmung `accordance_max` (Zeile 90), die mit 0 initialisiert wurde (Zeile 15), wird `maze_orig` in `match_coord` zwischengespeichert (Zeile 94). An

der Position `match_coord` in der primären Karte gab es also bis jetzt die höchste Übereinstimmung mit der temporären Karte. Wenn die Übereinstimmung gleich groß wie die bis jetzt größte Übereinstimmung ist, ist das Ergebnis nicht eindeutig, das Labyrinth hat also eine gleich Große Übereinstimmung an mindestens zwei verschiedenen Stellen. In dem Fall wird die ganzzahlige Variable `ambiguity` inkrementiert und erst auf 0 gesetzt, wenn es eine neue, höhere Übereinstimmung an einer anderen Stelle gibt.

Schließlich wird die temporären Karte um 90° im Uhrzeigersinn gedreht (Zeile 104) und für die neue Rotation werden erneut für alle mögliche Überlappungen die Übereinstimmung berechnet und ggf. bei einer höheren Übereinstimmung die neue Stelle, an der die Karten überlappen, in `match_coord` gespeichert.

Nachdem alle Rotationen geprüft wurden, werden die berechneten Ergebnisse (Position, Uneindeutigkeiten und Übereinstimmung) in den übergebenen Datentyp (Zeile 1) `MATCHSTAGES` geschrieben (Zeile 107 – 109) und können vom Hauptprogramm ausgewertet werden:

```
1  MATCHSTAGES matchStages;
2  maze_matchStages(&matchStages);
3
4  if((matchStages.ambiguity == 0) &&
5     (matchStages.accordance > 80))
6  {
7    for(uint8_t i = 0; i < matchStages.match.dir; i++)
8      maze_rotateStage(MAZE_SAVESTAGE);
9
10   robot.pos.x += matchStages.match.pos.x - ROB_POS_X_MIN;
11   robot.pos.y += matchStages.match.pos.y - ROB_POS_Y_MIN;
12   robot.pos.z = matchStages.match.pos.z;
13 }
```

Siehe auch CD-ROM: /Quellcode/sys/maze.c, Zeile 680

Im Folgenden beziehen sich alle Zeilenangaben auf den oben aufgeführten Quelltext.

Wenn es Uneindeutigkeiten gibt, wird die temporäre Karte auf jeden Fall weiter gezeichnet (die Bedingung in Zeile 4 und 5 trifft nicht zu). Wenn es es keine Uneindeutigkeiten gibt und die Übereinstimmung über einem bestimmten Schwellwert liegt (80% ist ein recht gut passender Richtwert; Eine 100%ige Übereinstimmung kann selten gewährleistet werden, weil unter Umständen Wände am Ende der Kartierung in der primären Karte und zu Beginn der Kartierung in der temporären Karte falsch eingezeichnet wurden) (Zeile 4), wird die temporäre Karte

`matchStages.match.dir` mal im Uhrzeigersinn gedreht (damit auch die Position des Roboters in Relation zur Übereinstimmung `matchStages.match` liegt) und die Position der Übereinstimmung abzüglich der Mindestposition des Roboters in der Karte zur Roboterposition in der temporären Karte addiert (Zeile 10 – 12). Zuletzt wird die primäre Karte um die in der temporären Karte gewonnenen Informationen ergänzt und die temporäre Karte gelöscht (noch nicht funktionstüchtig implementiert).

3.4 Analyse: Laufzeit

Im Folgenden wird der Map-Matching-Algorithmus hinsichtlich der Laufzeit analysiert.

Die Laufzeit des Algorithmus wurde für zufällig erstellte Karten in Abhängigkeit der Parameter

- Länge der temporären Karte x_t
- Breite der temporären Karte y_t
- Länge der primären Karte x_p
- Breite der primären Karte y_p
- Höhe der primären Karte z_p

für die gilt: $x_t \leqslant x_p$ und $y_t \leqslant y_p$ bestimmt.

Da der Algorithmus hauptsächlich aus fünf ineinander verschachtelten Zählschleifen besteht (für jeden Parameter eine Zählschleife) und in den Körpern der Zählschleifen keine komplexen Berechnungen durchgeführt werden, wurde lineares Verhalten erwartet. Das lineare Verhalten des Algorithmus hat sich für jeden Parameter bestätigt (es wurde nur für eine Etage der primären Karte getestet, der Parameter z_p war immer 0 bzw. 1) (siehe Abbildung 8). Das Liniendiagramm Abbildung 8 zeigt sieben linear ansteigende Kurven. Darüber befinden sich die Informationen darüber, welcher Parameter analysiert bzw. verändert wurde und die dabei ermittelte Laufzeit von `maze_matchStages(MATCHSTAGES *matchStages)` in Millisekunden auf dem verwendetem Mikrocontroller.

Oben sind für die jeweilige Spalte immer die konstanten Parameter angegeben, in

gelb hervorgehobenen Feldern wurde ein Parameter neu initialisiert.

Über der Tabelle mit den Werten steht dabei jeweils der betrachtete Parameter. Der Parameter hat immer den Wert, mit dem er initialisiert wurde, in jedem weiteren Feld nach unten wird er dabei um eins erhöht.

Es wurde fast ausschließlich (bis auf der letzten Datenreihe) der gleiche Parameter zwei mal betrachtet, allerdings mit jeweils unterschiedlichen anderen konstanten Parametern.

Auffallend ist der Knick in der Kurve Xt2 bei $x_t=5$ und Yt2 bei $y_t=4$. Nach näherer Betrachtung des in beiden Kurven konstanten Parameter y_p fällt auf, dass dieser ebenfalls bei 4 bzw. bei 5 liegt und die temporäre Karte in diesen Fällen für zwei Ausrichtungen größer als die primäre Karte ist. Ein Vergleich der Übereinstimmungen ergibt hier keinen Sinn, weshalb der Algorithmus dort abbricht. Bei näherer Betrachtung der betroffenen Laufzeiten für $y_t>4$ und $x_t>5$ fällt auf, dass die benötigte Zeit weiterhin linear ansteigt, allerdings wesentlich flacher.

Unregelmäßigkeiten des linearen Verhaltens der Kurven (kleinere Knicke) sind durch die Art und Weise, wie die Laufzeit berechnet wurde, zu erklären. Die Berechnung der Übereinstimmung erfolgte in der Hauptschleife des Roboters, in der nebensächliche Aufgaben erledigt werden. Ein Scheduler sorgt dafür, dass zeitkritische Aufgaben in regelmäßigen Perioden erledigt werden, dabei wird auch das Prüfen der Übereinstimmung unterbrochen und die Laufzeit verlängert sich minimal.

Der Algorithmus hat also eine Laufzeit von $T(x_t y_t x_p y_p z_p)$, die linear ansteigt, wenn einzelne Parameter verändert werden würden.

Über die Brute-Force-Methode ist kein schnellerer Algorithmus zu realisieren.

Xp = 10	Xp = 10	Xp = 10	Xp = 10	Xp = 3	Xp = 3	Xp = 3
Yp = 4	Yp = 5	Yp = 4	Yp = 4	Yp = 4	Yp = 10	Yp = 3
Zp = 1	Zp = 1	Zp = 1	Zp = 1	Zp = 1	Zp = 1	Zp = 1
Yt = 1	Yt = 4	Yt = 4	Yt = 4	Yt = 3		
Xt = 1	Xt = 1	Xt = 1	Xt = 4	Xt = 3		

Xt1	Xt2	Yt1	Yt2	Xp1	Xp2	Yp
123	192	124	172	143	241	128
153	295	153	250	170	310	143
158	429	156	338	192	376	162
172	528	168	416	216	443	178
184	562	179	435	238	509	194
198	564	192	431	264	576	211
210	564	209	432	287	642	227
222	565	225	441	310	706	241

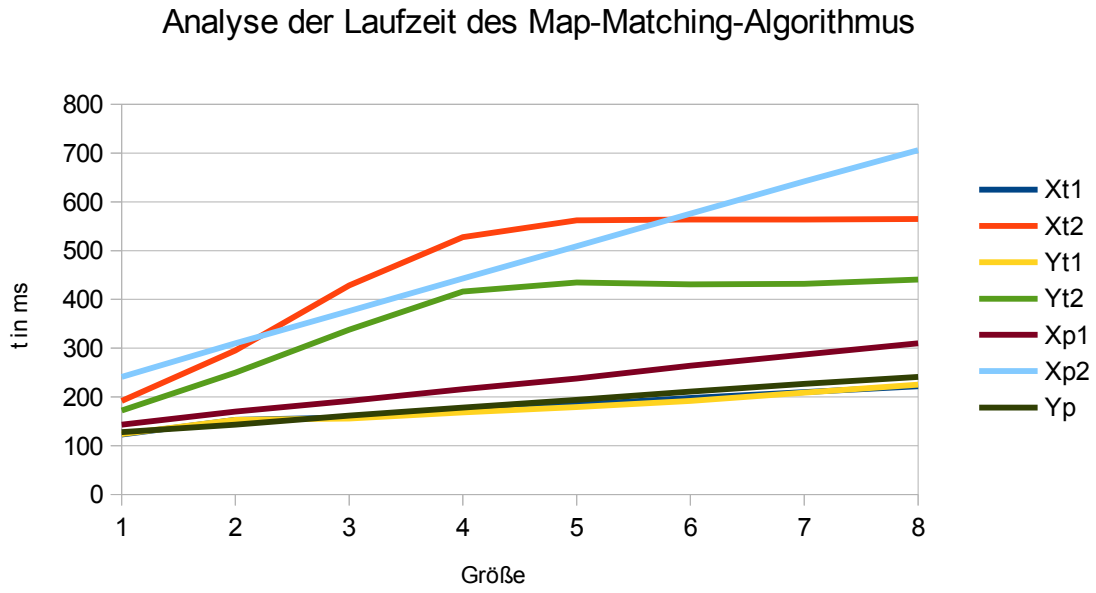

Abbildung 8: Analyse der Laufzeit durch Änderung einzelner Parameter

4 Schlussbetrachtung

Das entwickelte, vereinfachte SLAM Verfahren weist drei größere Schwächen auf.

Die erste Schwäche ist die Trägheit des Verfahrens. Im Gegensatz zu den komplexeren, großen Rettungsrobotern muss erst ein Fehler passieren, bevor der Roboter etwas dagegen unternimmt. Genau genommen ist das bei den komplexeren Rettungsrobotern genau so, allerdings ist dort die Auflösung der Karte und auch der verwendeten Sensoren wesentlich höher, weshalb Abweichungen von der Karte wesentlich schneller erkannt und korrigiert werden können. Dieser Schwachpunkt kann deshalb nicht ohne Weiteres behoben werden.

Eine weitere, elementare Schwäche ist der Fall, dass, während der Roboter sich in

der temporären Karte befindet, ein Fehler passiert. Aktuell ist der Roboter nicht dazu in der Lage, diesen Fehler zu erkennen, lösen lassen könnte sich dieses Problem allerdings dadurch, dass erneut die aktuell vier zur Verfügung stehenden Wände mit den Wänden in der temporären Karte abgeglichen werden. Wenn ein Fehler erkannt wird, richtet sich der Roboter erneut neu aus, löscht die temporäre Karte und beginnt die temporäre Karte von vorne zu zeichnen.

Weiterhin ist aktuell noch nicht der Fall abgedeckt, dass die temporäre Karte größer als die primäre Karte wird. Die berechnete Position ist dann falsch. Das Problem kann behoben werden, indem nicht die temporäre Karte über die primäre Karte, sondern die primäre Karte über die temporäre Karte gelegt wird, wenn die temporäre Karte größer als die primäre Karte wird.

Insgesamt konnte im Rahmen dieser Facharbeit ein recht zuverlässiger Algorithmus zur Selbstortung eines autonomen Rettungsroboters entwickelt werden, der zwar noch ein paar Schwachstellen aufweist, die allerdings einfach behoben werden können.

5 Quellenverzeichnis

5.1 Literaturverzeichnis

Atmel. 8-bit Atmel Microcontroller with 64K/128K/256K Bytes In-System Programmable Flash 2011, San Jose.
http://www.atmel.com/images/doc2549.pdf, abgerufen am 15.03.2014 um 16:23 Uhr.

Caldeira, Tiago [u.a.]. RoboCupJunior Rescue B Rules 2013, Eindhoven. http://rcj.robocup.org/rcj2013/rescueB_2013.pdf, abgerufen am 14.03.2014 um 12:24 Uhr.

Engel, Jakob [u.a.]. Camera-Based Navigation of a Low-Cost Quadrocopter 2013, München: Technische Universität München.

Fujii, Takashi. Laser remote sensing 2005, New York: Springer.

Kadiyski, Ivan. Algorithms and Data Structures – Rotate Matrix 2013. http://algorithmproblems.blogspot.de/2013/01/rotate-matrix.html, abgerufen am 14.02.2014 um 17:17 Uhr.

Riisgaard, Søren [u.a.]. SLAM for Dummies - A Tutorial Approach to Simultaneous Localization and Mapping 2005, Cambridge: Massachusetts Institute of Technology.

Sharp. GP2D120 Optoelectronic Device 2006, Osaka.
http://www.sharpsma.com/webfm_send/1205, abgerufen am 14.03.2014 um 12:22 Uhr.

Stachniss, Cyrill. OpenSLAM – Give your algorithm to the community. http://openslam.org/, abgerufen am 13.03.2014 um 18:04 Uhr.

Stachniss, Cyrill. Robotic Mapping and Exploration 2009, Berlin: Springer

Wolf, Jürgen. C von A bis Z 2003, Bonn: Galileo Computing.

5.2 Abbildungsverzeichnis

6 Anhang

[A] Bilder

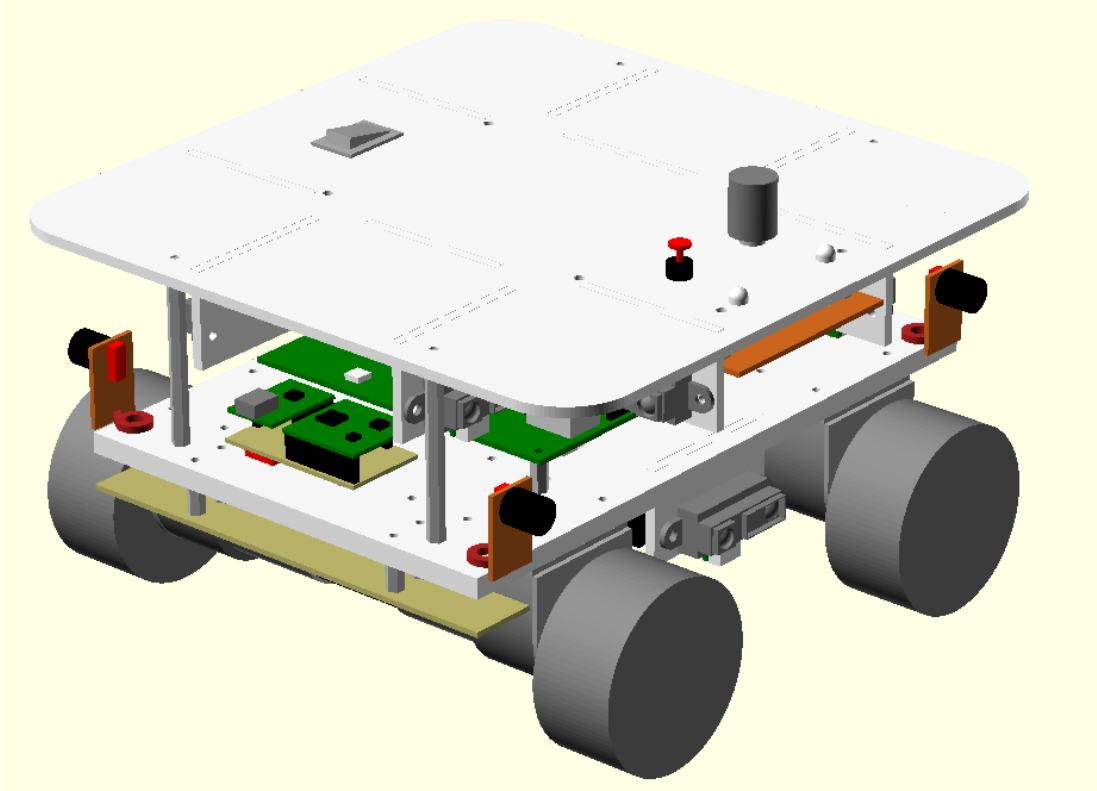

Abbildung 9: Komplettansicht in der Computervisualisierung des Roboters

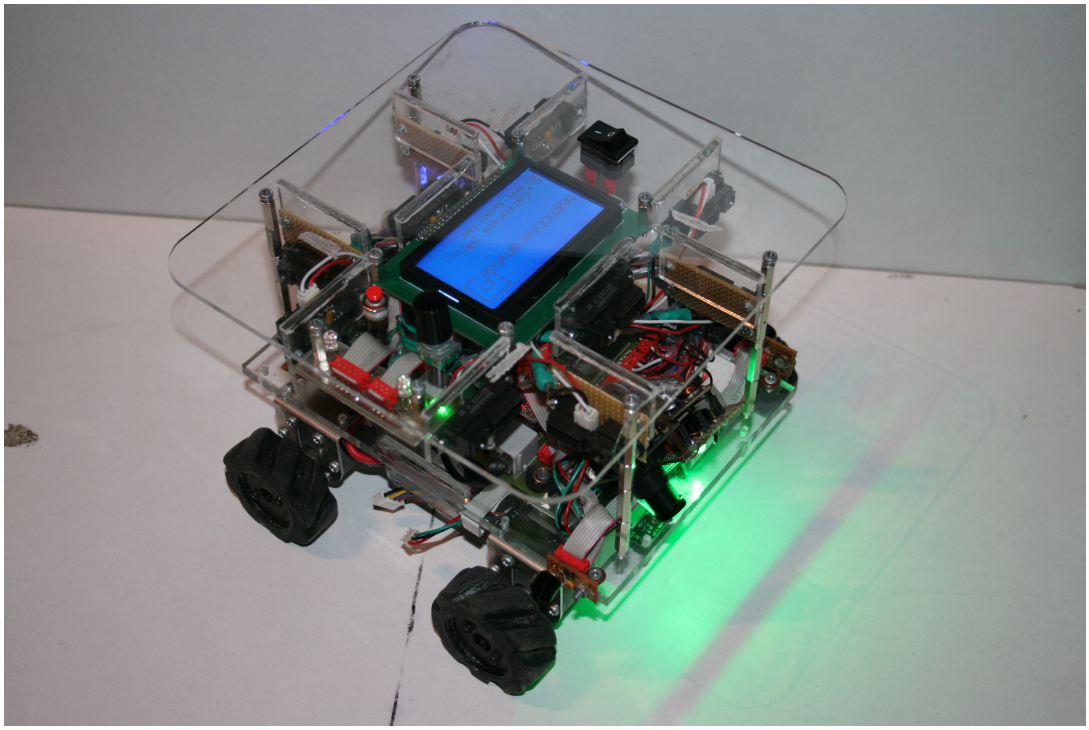

Abbildung 10: Komplettansicht des verwendetem Roboters

[B] Map-Matching Funktion

Erläuterungen

Konstanten:

- `MAZE_SAVESTAGE`

  z-Koordinate der Ebene des Labyrinths im Speicher, in der das temporäre Labyrinth gezeichnet wird.
  Siehe auch CD-ROM: /Quellcode/sys/maze.h, Zeile 24

- `NONE, NORTH, EAST, SOUTH, WEST`

  Himmelsrichtungen (Orientierung im Labyrinth in Relation zur Startrichtung des Roboters: Die Richtung, in die der Roboter startet, ist `NORTH`). Wird normalerweise im Zusammenhang mit einer Koordinate übergeben, bei `NONE` wird die übergebene Koordinate überprüft.
  Siehe auch CD-ROM: /Quellcode/sys/maze.h, Zeile 18

- `ROB_POS_X_MIN, ROB_POS_Y_MIN`

  Mindestposition des Roboters (kleinste `x` und `y` Koordinate). Siehe Abbildung 4: Das Labyrinth beginnt im Punkt P(1|1) und nicht im Ursprung.
  Siehe auch CD-ROM: /Quellcode/sys/maze.h, Zeile 30

- `MAZE_SIZE_X_USABLE, MAZE_SIZE_Y_USABLE`

  Benutzbarer Bereich des Speichers der Karte (die äußeren Fliesen können/dürfen nicht genutzt werden)
  Siehe auch CD-ROM: /Quellcode/sys/maze.h, Zeile 26

Globale Variablen:

- `robot`

  Datenstruktur des Typs `POS`. Speicherung der Position und Ausrichtung des Roboters im Labyrinth.

  Beispiel:
  ```
  robot.pos.x = ROB_POS_X_MIN;
  robot.pos.y = ROB_POS_Y_MIN;
  robot.pos.z = 0;
  robot.dir = NORTH;
  ```
  Siehe auch CD-ROM: /Quellcode/sys/maze.c, Zeile 30

Datenstrukturen:

- `COORD`

  Dreidimensionaler Vektor (`x`-, `y`- und `z`-Koordinate) in Ganzzahlen

  Beispiel:
  ```
  COORD position;
  position.x = 5;
  position.y = 1;
  position.z = 0;
  ```
  Siehe auch CD-ROM: /Quellcode/sys/maze.h, Zeile 63

- `POS`

  `COORD` Vektor (Nested Structure) und Ausrichtung (`dir`) als Ganzzahlen

  Beispiel:
  ```
  POS robot;
  robot.pos.x = 5;
  robot.pos.y = 1;
  robot.pos.z = 0;
  robot.dir = NORTH;
  ```
  Siehe auch CD-ROM: /Quellcode/sys/maze.h, Zeile 69

- `OFF`

  Offset, Informationen über Koordinaten (`x` und `y`) als Ganzzahlen

  Beispiel:
  ```
  OFF offset;
  offset.x = -5;
  offset.y = -3;
  ```
  Siehe auch CD-ROM: /Quellcode/sys/maze.h, Zeile 58

- `MATCHSTAGES`

  Informationen über Position (`POS`) der Übereinstimmung, der Uneindeutigkeiten (`ambiguity`) und dem Anteil der Übereinstimmung (`accordance`)

  Beispiel:
  ```
  MATCHSTAGES matchStages;
  matchStages.match.pos.x = 5;
  matchStages.match.pos.y = 1;
  matchStages.match.pos.z = 0;
  matchStages.match.dir = NORTH;
  matchStages.ambiguity = 0;
  matchStages.accordance = 100;
  ```
  Siehe auch CD-ROM: /Quellcode/sys/mazefunctions.h, Zeile 12

Funktionen:

- `void maze_stageGetSize(COORD *_stage)`

  Erwartet ein Objekt `COORD` mit initialisierter `z`-Koordinate (gewünschte Ebe-

ne). Ändert die x- und y-Koordinate auf Größe der Ebene.

Beispiel:
```
COORD stageSize;
stageSize.z = MAZE_SAVESTAGE;
maze_stageGetSize(&stageSize);
int8_t saveStage_size_x = stageSize.x;
```
Siehe auch CD-ROM: /Quellcode/sys/mazefunctions.c, Zeile 1013

- `int8_t maze_getBeenthere(COORD *_coord, int8_t dir)`

  Gibt wahr/falsch zurück, wenn die übergebene Koordinate im Labyrinth in der umgebenen Richtung.

  Beispiel:
  ```
  int8_t beenthere_north = maze_getBeenthere(&robot.pos, NORTH);
  int8_t beenthere_thisTile = maze_getBeenthere(&robot.pos, NONE);
  ```
  Siehe auch CD-ROM: /Quellcode/sys/mazefunctions.c, Zeile 390

- `int8_t maze_getWall(COORD *_coord, uint8_t dir)`

  Gibt den Wert der Wand auf der übergebenen Koordinate in der übergebenen Richtung zurück.

  Beispiel:
  ```
  int8_t wallValueNorth = maze_getWall(&robot.pos, NORTH);
  ```
  Siehe auch CD-ROM: /Quellcode/sys/mazefunctions.c, Zeile 197

- `void maze_rotateStage(uint8_t stage)`

  Rotiert die übergebene Ebene um 90° im Uhrzeigersinn

  Siehe auch CD-ROM: /Quellcode/sys/mazefunctions.c, Zeile 761

Quellcode

```
1  void maze_matchStages(MATCHSTAGES *matchStages)
2  {
3    POS maze_orig;
4
5    COORD maze_save;
6    maze_save.z = MAZE_SAVESTAGE;
7
8    POS match_coord;
9    match_coord.pos.x = 0;
10   match_coord.pos.y = 0;
11   match_coord.pos.z = 0;
12
13   int16_t matches = 0;
14   int8_t accordance = 0;
15   int8_t accordance_max = 0;
16   uint8_t ambiguity = 0;
17
18   COORD maze_save_size;
19   maze_save_size.z = MAZE_SAVESTAGE;
20   maze_stageGetSize(&maze_save_size);
21   uint8_t maze_save_visitedTiles;
22
23   COORD maze_orig_size;
24
25   for(maze_orig.dir = NONE;
26       maze_orig.dir < WEST;
27       maze_orig.dir ++)
28   {
29     for(maze_orig.pos.z = 0;
30         maze_orig.pos.z < MAZE_SAVESTAGE;
31         maze_orig.pos.z ++)
32     {
33       maze_orig_size.z = maze_orig.pos.z;
34       maze_stageGetSize(&maze_orig_size);
35
36       for(maze_orig.pos.y = ROB_POS_Y_MIN;
37           maze_orig.pos.y <= maze_orig_size.y + ROB_POS_Y_MIN;
38           maze_orig.pos.y ++)
39       {
40         for(maze_orig.pos.x = ROB_POS_X_MIN;
41             maze_orig.pos.x <= maze_orig_size.x + ROB_POS_X_MIN;
42             maze_orig.pos.x ++)
43         {
44           COORD maze_orig_save = maze_orig.pos;
45           maze_save_visitedTiles = 0;

46           for(maze_save.y = ROB_POS_Y_MIN;
47               maze_save.y <= (maze_save_size.y + 1 + ROB_POS_Y_MIN);
48               maze_save.y ++)
49           {
50             maze_orig_save.y = maze_orig.pos.y + maze_save.y
51                                                - ROB_POS_Y_MIN;
52             if(maze_orig_save.y > MAZE_SIZE_Y_USABLE)
53             {
54               matches = 0;
55               break;
56             }
```

```
57
58              for(maze_save.x = ROB_POS_X_MIN;
59                  maze_save.x <= (maze_save_size.x + 1 + ROB_POS_X_MIN);
60                  maze_save.x ++)
61              {
62                maze_orig_save.x = maze_orig.pos.x + maze_save.x
63                                                   - ROB_POS_X_MIN;
64                if(maze_orig_save.x > MAZE_SIZE_X_USABLE)
65                {
66                  matches = 0;
67                  break;
68                }
69
70                if(maze_getBeenthere(&maze_save, NONE) &&
71                   maze_getBeenthere(&maze_orig_save, NONE))
72                {
73                  maze_save_visitedTiles ++;
74                  for(int8_t dir = NORTH; dir <= WEST; dir ++)
75                  {
76                    int8_t wall_save = maze_getWall(&maze_save, dir);
77                    int8_t wall_orig = maze_getWall(&maze_orig_save, dir);
78                    if(((wall_save > 0) && (wall_orig > 0)) ||
79                       ((wall_save < 0) && (wall_orig < 0)))
80                    {
81                      matches ++;
82                    }
83                  }
84                }
85              }
86            }
87
88            accordance = (matches * 100) / (maze_save_visitedTiles * 4);
89
90            if(accordance > accordance_max)
91            {
92              ambiguity = 0;
93              accordance_max = accordance;
94              match_coord = maze_orig;
95            }
96            else if(accordance == accordance_max)
97            {
98              ambiguity ++;
99            }
100           matches = 0;
101         }
102       }
103     }
104     maze_rotateStage(MAZE_SAVESTAGE);
105   }
106
107   matchStages->match = match_coord;
108   matchStages->ambiguity = ambiguity;
109   matchStages->accordance = accordance_max;
110 }
```

Siehe auch CD-ROM: /Quellcode/sys/mazefunctions.c, Zeile 849

[C] CD-ROM

Inhalt:

- Facharbeit in digitaler Form
- Dossier der Facharbeit vom 27.01.2014
- OpenOffice Dateien der selbst erstellten Verdeutlichungen und Diagramme
- Speicherung der Internetseiten
  - http://algorithmproblems.blogspot.de/2013/01/rotate-matrix.html
  - http://openslam.org/
- Kopie der über das Internet öffentlich zugänglichen Literatur (vgl. Literaturverzeichnis)
  - Atmel.pdf
  - Caldeira.pdf
  - Engel.pdf
  - Riisgard.pdf
  - Sharp.pdf
- Kompletter Quellcode des Projekts „RoboCup 2014“ von teamohnename.de (Stand: 20.03.2014)

7 Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die Arbeit selbständig angefertigt, keine anderen als die angegebenen Hilfsmittel benutzt und die Stellen der Facharbeit, die im Wortlaut oder im wesentlichen Inhalt anderen Werken entnommen wurden, mit genauer Quellenangabe kenntlich gemacht habe.
Verwendete Informationen aus dem Internet sind dem(r) Lehrer/in vollständig im Ausdruck zur Verfügung gestellt worden.

Damme, den

Jan Blumenkamp