SIEMENS

Hochschule München

Fakultät für Elektrotechnik und Informationstechnik

Studiengang Elektrotechnik und Informationstechnik

Industrie 4.0 - Pathfinding auf einer SPS

Bachelorarbeit von Niels Garibaldi

Bearbeitungsbeginn:	xx.xx.2016
Abgabetermin:	xx.xx.2016
lfd. Nr.:	xxxxx

HOCHSCHULE
FÜR ANGEWANDTE
WISSENSCHAFTEN
MÜNCHEN

SIEMENS

Hochschule München

Fakultät für Elektrotechnik und Informationstechnik

Studiengang Elektrotechnik und Informationstechnik

Industrie 4.0 - Pathfinding auf einer SPS

Industry 4.0 - Pathfinding on a PLC

Bachelorarbeit von Niels Garibaldi

betreut von Prof. Dr. K. Ressel

Bearbeitungsbeginn: xx.xx.2016
Abgabetermin: xx.xx.2016
lfd. Nr.: xxxxx

Erklärung des Bearbeiters

1. Ich erkläre hiermit, dass ich die vorliegende Bachelorarbeit selbständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe.

   Sämtliche benutzte Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate sind als solche gekennzeichnet.

2. Ich erkläre mein Einverständnis, dass die von mir erstellte Bachelorarbeit in die Bibliothek der Hochschule München eingestellt wird. Ich wurde darauf hingewiesen, dass die Hochschule in keiner Weise für die missbräuchliche Verwendung von Inhalten durch Dritte infolge der Lektüre der Arbeit haftet. Insbesondere ist mir bewusst, dass ich für die Anmeldung von Patenten, Warenzeichen oder Geschmacksmustern selbst verantwortlich bin und daraus resultierende Ansprüche selbst verfolgen muss.

München, den xx.xx.2016, _____

Niels Garibaldi

Zusammenfassung

Zusammenfassungstext hier einfügen

Abstract

Insert abstract text here

Inhaltsverzeichnis

Abkürzungsverzeichnis

AWL Anweisungsliste

CPPS Cyber-physisches Produktionssystem

CT Corporate Technologies

D* Focussed Dynamic A*

DB Datenbaustein

FB Funktionsbaustein

FC Funktion

FTF Fahrerloses Transportfahrzeug

FTS Fahrerloses Transportsystem

FUP Funktionsplan

IoT Internet of Things

KOP Kontaktplan

MA* Memory Bounded A*

OB Organisationsbaustein

PLC Programmable Logic Controller

RFID Radio Frequency Identifikation

RTA* Real-Time A*

SCL Structured Control Language

SPS Speicherprogrammierbare Steuerung

STEP7 STeuerungen Einfach Programmieren Version 7

TIA-Portal Totally Integrated Automation Portal

1

Einführung in das Thema Industrie 4.0

Der Begriff „Industrie 4.0“ ist seit seiner Popularisierung durch die Bundeskanzlerin auf der Hannovermesse 2013 vor allem in den Bereichen Produktion und Fertigung in aller Munde. Da er jedoch je nach Branche unterschiedliche Bedeutungen haben kann, soll als Einführung zunächst einmal erläutert werden, was Industrie 4.0 im Kontext der Automatisierungstechnik bedeutet. Der Begriff beschreibt die vierte industrielle Revolution und die damit einhergehende Verflechtung von informationstechnisch erhobenen Daten in den Produktionsablauf. Ein interessanter Aspekt ist hier beispielsweise der Bereich der prädiktiven Wartung, bei dem Anhand der Auswertung empirischer Daten mögliche Anlagenausfälle frühzeitig erkannt und behoben werden können. Für den weiteren Verlauf dieser Arbeit ist vor allem auch ein sogenanntes Cyber-physisches Produktionssystem (CPPS) von Bedeutung. Diese bestehen aus der Verbindung von einzelnen, dezentralen Objekten wie beispielsweise Produktionsanlagen oder Logistikkomponenten, welche mit eingebetteten Systemen ausgestattet und zudem kommunikationsfähig gemacht werden. Durch eingebaute Sensoren und Aktoren kann die Umwelt erfasst und beeinflusst werden. Mittels der Kommunikationskomponenten können Daten aus der Produktion über ein Netzwerk oder das Internet ausgetauscht, beziehungsweise von entsprechenden Diensten ausgewertet, verarbeitet oder gespeichert werden [1]. Sind mehrere Cyber-physische Produktionssysteme an einem Produktionsprozess beteiligt, unabhängig von ihrem Standort, so spricht man auch von einer Smart Factory. Abschließend ist noch zu erwähnen, dass der Begriff „Industrie 4.0“ vor allem im deutschsprachigen Raum verwendet wird. Im internationalen Kontext werden viele der zentralen Punkte von Industrie 4.0 durch das Konzept des Internet of Things (IoT) abgedeckt.

2

Definition der Anforderungen

2.1 Allgemeine Aufgabenstellung

Diese Arbeit beschäftigt sich mit dem Entwurf und der Realisierung eines CPPS im Modellmaßstab. Die resultierende Anlage soll unter anderem dazu dienen verschiedene Aspekte von Industrie 4.0 vorzuführen und zu veranschaulichen. Kernpunkte, die dargestellt werden sollen, sind vor allem die Dezentralisierung und Skalierbarkeit der Anlage. Den Rahmen für die Bearbeitung dieser Aufgabe bildet die Fachberatung für Automatisierungstechnik der Siemens AG in München. Um die erwähnten Konzepte demonstrieren zu können, soll die Anlage gemäß ihrer realen Vorbilder bestehen aus Bearbeitungsstationen, an welchen der Bearbeitungsprozess simuliert werden kann, und Werkstückträgern, welche Werkstücke durch die Modellanlage zu den Maschinenplätzen transportieren können. Hauptaufgabe des Modells ist es zu zeigen, wie ein Werkstück seine Fertigung selbst organisiert. Zu Beginn des Fertigungsprozesses wird dem Werkstück ein Fertigungsplan übergeben, anhand welchem es sich durch die Anlage bewegt. Im vorliegenden Fall stellen die Werkstückträger gleichzeitig das Werkstück dar, das die Produktionsanlage durchfährt. Ebenso werden die Bearbeitungsstationen nur symbolisch dargestellt durch entsprechende Markierungen in der Anlagentopologie. Anhand verschiedener Use-cases und Szenarien soll es nach Fertigstellung der Anlage möglich sein, verschiedene Aspekte von Industrie 4.0 anschaulich darzustellen, zum Beispiel zu Vorführzwecken bei Kunden.

2.2 Aufteilung der Themenbereiche

Wie soeben beschrieben konzentriert sich die Aufgabe vor allem auf die Entwicklung und Implementierung der Werkstückträgerfahrzeuge. Da dies jedoch

noch immer ein recht allgemein gefasstes Themengebiet ist und der zur Realisierung notwendige Aufwand nur schwer abzuschätzen ist, wurde entschieden, die Hauptaufgabe in zwei Teilaufgaben zu unterteilen. Diese beiden Themenblöcke sind „Autonomes Fahren in industriellen Transportsystemen“ und „Dynamische Wegfindung in einer flexiblen Produktionsanlage“[2].Die Dokumentation der Implementierung der dynamischen Wegfindung ist Hauptbestandteil der vorliegenden Arbeit. Den Teil des autonomen Fahrens mittels eines Fahrerloses Transportsystem (FTS) wird von Herrn Andreas Meier bearbeitet und wird deshalb nur kurz umrissen.

2.2.1 Fahrerloses Transportsystem

Hauptaufgabe der Werkstückträger, die jeweils durch ein Fahrerloses Transportfahrzeug (FTF) realisiert werden sollen, ist es, einen Weg anhand einer vorgegebenen Route ab zu fahren und somit die Werkstücke zwischen den einzelnen Bearbeitungsschritten zur nächsten Station oder Maschine zu transportieren. Herausforderungen hierbei sind die Spurführung und die Navigation innerhalb der Anlagentopologie, da die einzelnen FTF nicht wie sonst üblich schienengebunden sind, sondern sich frei durch die Anlage bewegen können. Da es sich zudem mehrere Fahrzeuge gleichzeitig in der Anlage befinden können, muss durch geeignete Sensorik eine bevorstehende Kollision erkannt und verhindert werden.

2.2.2 Dynamische Wegfindung

Für die Wegfindung ist es wichtig, dass das Werkstück die Ablaufreihenfolge der benötigten Arbeitsschritte kennt. Diese sollen in einer Art Rezept festgehalten werden und als Grundlage für die Berechnung der Teilrouten zu den Bearbeitungsstationen dienen. Für die Bestimmung der Routen wird zudem eine geeignete Beschreibung der Anlagentopologie benötigt. Innerhalb der Anlage können mehrere Bearbeitungsstationen die gleiche Funktionalität liefern. In diesem Fall soll die Wegfindung eine geeignete Station auswählen können Diese sollte universell gehalten sein, damit sie auch von dem FTS genutzt werden kann. Die Wegfindung soll so realisiert sein, dass sie auch auf einer der leistungsschwächeren Steuerungen[1] der Siemens S7-1200er Reihe lauffähig ist, falls dies nicht realisierbar ist, sollen so wenig Komponenten wie möglich auf ein PC-System ausgelagert werden.
Bei der Berechnung von Routen sollen zudem folgende dynamische Ereignisse berücksichtigt werden:

- Persistente Blockaden und länger andauernde nicht permanente Änderungen der Standard-Anlagentopologie.

[1] Speicherprogrammierbare Steuerung (SPS)

- Dynamische Blockaden von Teilstrecken durch andere Fahrzeuge zur Kollisionsvermeidung.

2.3 Erwartete Ziele für die Wegfindung

Da bei der Planung des Arbeitsauftrags noch nicht absehbar war, wie zeitaufwendig die Realisierung der Aufgabe sein würde, wurde die Funktionalität der Wegfindungskomponente der Anlage in mehrere Stufen unterteilt:

Grundstufe: Es können detaillierte Teilrouten generiert werden auf der Basis von Anfangs- und Endpunkten.

Stufe 1: Es können komplette Routen bestehend aus Teilrouten anhand eines einzigen Fertigungsplans generiert werden, es existiert pro Funktionalität nur eine Bearbeitungsstation. Mehrere FTF erhalten die gleiche Route und reihen sich hintereinander ein.

Stufe 2: Es wird werden komplette Routen anhand eines einzigen Fertigungsplans generiert, pro Funktionalität existieren mehrere Bearbeitungsstationen. Die FTF können unterschiedliche Wege nehmen und sich gegenseitig überholen.

Stufe 3: Analog zu Stufe 2, es können jedoch im laufenden Betrieb Störungen in der Form von persistenten Blockaden auftreten, die bei der Wegfindung berücksichtigt werden.

Stufe 4: Analog zu Stufe 3, verschiedene Fahrzeuge können unterschiedliche Fertigungspläne und somit andere Routen verfolgen.

Als zusätzliches Ziel wurde die Visualisierung der geplanten und bereits zurückgelegten Fahrstrecken der verschieden FTF definiert, zur besseren Darstellung der einzelnen Aspekte der definierten Stufen. Die Anlage wird zudem in Kooperation mit der Siemens Entwicklungsabteilung Corporate Technologies (CT) entwickelt wird, welche parallel die gleiche Anlage als Simulation testet und die ermittelten Ergebnisse mit dem Resultat der physikalischen Anlage vergleicht.

3

Theoretische Grundlagen Wegfindung

3.1 Modellierung der Anlagentopologie

Zur Berechnung eines Weges innerhalb der Anlage wird zuallererst die Topologie der besagten Anlage benötigt. Für die Funktionsweise der FTF wurde definiert, das sich alle Fahrzeuge mittels optischer Merkmale auf einer definierten Teilstrecke bewegen. Zudem soll es den Fahrzeugen nur an festgelegten Entscheidungspunkten möglich sein, ihren Fahrzustand zu ändern und eine andere Teilstrecke. Dies bedeutet, das alle Fahrzeuge, sobald sie sich für eine Teilstrecke entschieden haben, dieser bis zum nächsten Entscheidungspunkt folgen. Auf Basis einer solchen logischen Unterteilung der Anlage in Entscheidungspunkte und Teilstrecken als Verbindungen zwischen zwei solcher Punkte, liegt es nahe als Datenstruktur für die Modellierung der Anlagentopologie einen Graphen zu verwenden. Die Entscheidungspunkte entsprechen hierbei den Knoten und die korrespondierenden Teilwegstrecken stellen die Kanten des Graphen dar. Da sich die FTF möglichst frei durch die Produktion bewegen sollen, wird als Grundform der Anlage ein ungerichteter Graph zur Abbildung der Topologie verwendet, jedoch soll es für die spätere Wegberechnung unerheblich sein, ob es sich um einen gerichteten oder ungerichteten Graphen handelt.

insert graphic about graphs here

Da der Graph die Abbildung einer realen Anlage ist, kann zudem ausgeschlossen werden, dass die Gewichtung der Kanten negativ ist, da dies je nach Art der Gewichtung nur wenig Nutzen bringen würde. Es existieren beispielsweise keine negativen Streckenabstände oder Fahrzeiten, die eine spezielle Betrachtung erforderlich machen und somit die Wahl der Wegfindungsalgorithmen einschränken würden.

3.2 Auswahl der Algorithmen

Es existieren mehrere Algorithmen, welche sich mit der Problemstellung der Berechnung eines kürzesten Pfads zwischen zwei Knoten eines gewichteten Graphen befassen. Um einen passende Vorgehensweise auswählen zu können muss zuerst definiert werden, welche Kriterien ein potentieller Wegfindungsalgorithmus erfüllen muss.

3.2.1 Auswahlkriterien

Für ein geeignetes Wegfindungssystem müssen die folgenden Aspekte berücksichtigt werden:

1. Die Berechnung soll auf einer SPS des niedrigen Leistungssegments durchgeführt werden. Somit müssen die daraus resultierenden Beschränkungen bezüglich Komplexität und Speicherbedarf erfüllt sein.

2. Eine SPS ist ein Echtzeitsystem, welches das Programm zyklisch abarbeitet. Gleichzeitig wird auf der Steuerung auch die Fahraufgabe realisiert. Dadurch muss die Rechenzeit kurz genug sein, um eine Reaktionsfähigkeit des Fahrprogramms sicherstellen zu können.

3. Die Wegfindung soll während der Ausführung eines vorausberechneten Fahrauftrags auf andere Fahrzeuge oder Änderungen der Anlagentopologie reagieren können.

4. Es soll ein optimaler Weg gefunden werden. Optimal bedeutet im vorliegenden Fall einen möglichst kurzen Weg zum Zielknoten.

5. Die Anlage soll möglichst gleichmäßig ausgelastet sein in Bezug auf Bearbeitungsstationen mit gleicher Funktionalität.

Eine SPS, wie sie in 1. beschrieben ist, erlaubt nur eine geringe Schachtelungstiefe der Unterprogrammaufrufe. Dies beschränkt den Einsatz von rekursiven Algorithmen auf Anwendungen bei denen die Anzahl der Rekursionen von vornherein bekannt ist. Da die Anlagen aber beliebiger Art sein sollen, können Algorithmen die Rekursion verwenden nicht genutzt werden. Zudem fallen für die Implementierung diejenigen Algorithmen heraus, die nur sehr umständlich in der begrenzten Programmierumgebung für Echtzeitsteuerungen realisiert werden können.
Die Einhaltung der in Punkt 2 beschriebenen Echtzeitbedingung ist keine generelle Frage des Algorithmus sonder eine Sache der Implementierung. Hier müssen Programmteile mit Schleifenkonstrukten möglicherweise aufgebrochen und die Berechnung auf verschiedene Zyklen aufgeteilt werden.

Die in 3. erwähnte Reaktionsfähigkeit erfordert von den Algorithmen, dass sie auch in nur teilweise bekannten Umgebungen sicher einen Weg finden können. Für das Zusammenspiel ist eine Methode für die Vorhersage der Positionen anderer Fahrzeuge notwendig.
Da aus Gründen der Anlageneffizienz die Laufzeit eines Werkstücks durch die Anlage möglichst minimiert werden soll, wird von dem Wegfindungsalgorithmus gemäß Punkt 4 erwartet, dass er nicht nur schnell einen möglichen Weg findet, sondern dass der gefundene Weg auch der kürzeste, mit den begrenzten Anlageninformationen berechenbare, Pfad zum Zielknoten ist.
Da die in 5. erwähnte gleichmäßige Auslastung mehr mit der Auswahl des geeigneten Zielknotens als mit der eigentlichen Wegfindung zu diesem Ziel zu tun hat, ist diese Anforderung nur begrenzt für die Auswahl der Algorithmen von Bedeutung.

3.2.2 Betrachtete Algorithmen

Im Angesicht der in 3.2.1 ermittelten Auswahlkriterien wurden die folgenden Algorithmen näher betrachtet:

Dijkstra : Einer der Grundalgorithmen für das Kürzeste-Pfad-Problem, bei dem auch die Entfernungen aller Knoten zu einem Startknoten ermittelt werden können.

A* : Eine Abwandlung des Dijkstra-Algorithmus, bei dem die Anzahl der betrachteten knoten verringert werden kann, indem Zusatzinformationen in Form einer Heuristik als Entscheidungshilfe für die Betrachtungsreihenfolge verwendet werden.

Memory Bounded A* (MA*)[3]: Dieser Algorithmus beschränkt den Speicherverbrauch indem er den Topologiegraphen in Teilbäume unterteilt und nur erfolgversprechende Knoten und Teilgraphen im Speicher behält.

Real-Time A* (RTA*)[4]: Dieser Algorithmus unterteilt den Hauptalgorithmus in Planungs- und Ausführungsphasen und beschränkt die Anzahl der betrachteten Knoten anhand einer Alpha-Beta-Suche. Durch diese Unterteilung kann ein gewisses Maß an Dynamik gewonnen werden, da der Algorithmus nach jeder Ausführungsphase die Restroute neu evaluiert und gegebenenfalls die aktuelle Route gegen einen neuen optimierten Weg ersetzt.

Focussed Dynamic A* (D*)[5][6]: Dieser Algorithmus wurde entwickelt für Topologien, welche nur teilweise bekannt sind, beziehungsweise die sich dynamisch verändern können. Zeiger die entlang des jeweils kürzesten Weges vom aktuellen Knoten zum Startknoten zeigen werden bei Anlagenänderung durch sogenannte „Modify-Cost-Operationen“ ausgehend von der Anlagenänderung kaskadierend modifiziert bis sich ein neuer statischer Zustand eingestellt hat. Die Wegberechnung nutzt diese Zeiger zur schnelleren Berechnung des kürzesten Pfades.

Der Dijkstra-Algorithmus ist aufgrund seiner langen Berechnungszeit ungeeignet für die Wegfindung im laufenden Betrieb der FTF jedoch kann ausgenutzt werden, das bei der Berechnung des kürzesten Pfades via Dijkstra die Abstände aller berechneten Knoten zum Startknoten ermittelt werden. Diese Abstände bilden die Basis zur Verbesserung der bei einem der Anderen Algorithmen genutzten Heuristiken. Die Algorithmen MA*, RTA* und D* sind alle Weiterentwicklungen des A*-Algorithmus. Da die zu entwickelnde Anlage im vorliegenden Fall nur eine Begrenzte Anzahl von Knoten besitzt, ist der Nachteil des hohen Speicherbedarfs des Grundalgorithmus A* hier vernachlässigbar. Somit kann der zusätzliche Aufwand für die Implementierung von MA* zunächst ignoriert werden. Der D*-Algorithmus wurde wegen seiner höheren Komplexität und der Beschränkungen durch die Programmierumgebung nicht implementiert. Als zyklischen Wegfindungsalgorithmus wurde eine Mischform des A*-Algorithmus mit der Planungs- und Ausführungsphase von RTA* verwendet, bei der in regelmäßigen Abständen der bisher gefundene Teilweg verifiziert und bei Bedarf neu berechnet wird. Die verwendeten Algorithmen sollen nun etwas näher betrachtet werden.

3.3 Dijkstra-Algorithmus

Der Erste der insgesamt zwei für die Wegfindung verwendeten Algorithmen ist der nach seinem Erfinder benannte Dijkstra-Algorithmus, der den kürzesten Pfad zwischen einem Startknoten und einem Endknoten[1] eines Graphen ermittelt. Er kann verwendet werden für beliebige positiv gewichtete, gerichtete oder ungerichtete Graphen. Da in 3.1 negative Kantengewichtungen ausgeschlossen wurden, lässt sich der Algorithmus ohne Abwandlung auf den Graphen der Anlagentopologie anwenden.

3.3.1 Grundprinzip

Das Grundprinzip des Dijkstra-Algorithmus ist Unterteilung aller Graphenknoten in drei Untergruppen[7]:

Gruppe A: Die Menge aller Knoten, zu dem bereits ein kürzester Pfad bekannt ist.

Gruppe B: Die Menge aller Knoten, die mit mindestens einem Knoten aus Gruppe A verbunden sind, jedoch selbst nicht zu A gehören.

Gruppe C: Die Menge der restlichen Knoten, die nicht in den Gruppen A oder B enthalten sind.

Zusätzlich zu den Knoten können auch die Kanten drei unterschiedlichen Teilgruppen zugeordnet werden[7]:

Gruppe I: Die Menge aller Kanten, die in einem kürzesten Pfad zu einem der Knoten aus Gruppe A vorkommen.

Gruppe II: Die Menge aller Kanten, aus denen die nächste Kante für Gruppe I ausgewählt wird, wenn der korrespondierende Knoten aus B zur Gruppe A hinzugefügt wird. Es existiert immer genau eine Kante für jeden Knoten aus Gruppe B.

Gruppe III: Die restlichen Kanten.

Der Algorithmus funktioniert nun wie folgt. Zu Beginn befinden sich alle Knoten und Kanten respektive in den Gruppen C oder III. Als erstes wird nun der Startknoten S zu A hinzugefügt, da ein kürzester Pfad vom Startknoten zu sich selbst mit dem Wert 0 bekannt ist. Ab nun werden folgende Schritte solange wiederholt, bis das Ziel erreicht wurde:

[1] oder allen anderen Knoten.

1. Man betrachte alle Kanten z die den soeben zu A hinzugefügten Knoten mit einem Knoten K aus B oder C verbinden.
   - Gehört K zur Gruppe B, so wird untersucht ob z zu einem kürzeren Pfad zu K führt als der bisherige kürzeste Pfad mit einer Kante aus II zu diesem Knoten. Ist dies der Fall, so ersetzt z die korrespondierende Kante aus II. Die Kante die durch z wird wieder zur Gruppe III hinzugefügt.
   - Gehört K zur Gruppe C, so wird K zu B und z zu II hinzugefügt.
2. Es existiert immer nur jeweils ein Weg vom Startknoten zu einem beliebigen Knoten von B unter Verwendung der Kanten aus I und II. Nun wird derjenige Knoten aus B zur Gruppe A hinzugefügt, dessen Weg der kürzeste ist. Analog wird die zugehörige Kante der Gruppe I zugeordnet.

3.3.2 Darstellung der Funktionsweise

Die **insert graphic here** zeigt die Funktionsweise des Algorithmus an einem einfachen Beispielgraphen mit vier Knoten.

3.3.3 Berechnungsaufwand

Der Berechnungsaufwand des Dijkstra-Algorithmus beträgt in Landau-Notation vereinfacht $\mathcal{O}(|\mathcal{V}|^2)$ beziehungsweise $\mathcal{O}(|\mathcal{V}|\log|\mathcal{V}| + |\mathcal{E}|)$ bei der Implementierung mittels Fibonacci-Heaps[8] wobei $\mathcal{V}$ gleich der Anzahl der Knoten und $\mathcal{E}$ gleich der Anzahl der Kanten des Graphen ist. diese Vereinfachung gilt jedoch nur, wenn die Anzahl der Kanten in der gleichen Größenordnung liegt wie die Anzahl der Knoten.

3.4 A*-Algorithmus

Der zweite verwendete Algorithmus für die Wegfindung ist der sogenannte A*-Algorithmus. Dieser ist eine Weiterentwicklung des in 3.3.1 vorgestellten Dijkstra-Algorithmus. Der grundlegende Unterschied ist jedoch, das bei A* eine Bewertung der Betrachtungsreihenfolge von Knoten bei der Berechnung verwendet werden. Dies bedeutet, dass diejenigen Knoten und Kanten zuerst expandiert werden sollen, bei denen die Wahrscheinlichkeit höher ist, das sie Teil des kürzesten Pfades zum Zielknoten sind. Im Vergleich dazu wurde beim Dijkstra-Algorithmus immer genau der Knoten als nächstes expandiert, der den kürzesten Abstand zu den bereits expandierten Knoten hatte**Insert graphic dijkstra vs A***.

Um eine Abschätzung der Wahrscheinlichkeit eines Knotens bezüglich seiner Zugehörigkeit zu dem kürzesten Pfad vom Start zum Zielknoten treffen zu können, werden somit zusätzliche Informationen über die Beziehung zwischen einzelnen Knoten und dem Zielknoten benötigt. Diese Bewertung von Knoten wird auch als Heuristik bezeichnet. Mittels dieser Bewertung kann nun bereits vor Erreichen des Zielknotens eine Aussage über die Länge der Route gemacht werden.

3.4.1 Grundprinzip

Die Funktionsweise des A*-Algorithmus kann somit wie folgt als Erweiterung des Dijkstra-Algorithmus beschrieben werden[9]:

- *Seien s, n und z beliebige Knoten des Graphen G, wobei s der Startknoten und z der Zielknoten für die Ermittlung des kürzesten Weges von s zu z sind.*

- *Sei $f(n)$ eine Funktion zur Abschätzung der Pfadlänge vom Knoten s zum Knoten z, welcher den Knoten n enthält.*

1. Man markiere s als „offen“ und berechne $f(s)$.
2. Man wähle denjenigen offenen Knoten n aus, der den kleinsten Wert für $f(n)$ besitzt.
3. Falls $n = z$, wird n als „geschlossen“ markiert und beende den Algorithmus.
4. Andernfalls markiere man n als „geschlossen“ und füge alle von n erreichbaren Knoten n' zur Liste der offenen Knoten hinzu, die noch nicht als geschlossen markiert wurden und berechne $f(n')$. Man springe zurück zu Schritt 2.

Man sieht das somit bei dem A*-Algorithmus diejenigen Knoten bevorzugt betrachtet werden, die durch die Abschätzungsfunktion als „wahrscheinlicher zum Ziel führend“ bewertet wurden. Somit ist es nur sinnvoll diesen Bewertungsvorgang näher zu betrachten.

3.4.2 Abschätzungsfunktion f(n)

Die Abschätzungsfunktion, die in 3.4.1 definiert wurde, kann unterteilt werden in zwei Unterfunktionen[9]:

$$f(n) = g(n) + h(n) \tag{3.1}$$

$g(n)$ beschreibt hierbei den Wert des kürzesten Pfads von s zu n und $h(n)$ beschreibt den Wert des Pfades von n zum Ziel z.

Da die genauen Werte für $g(n)$ und $f(n)$ im Verlauf der Berechnung aber möglicherweise noch nicht exakt bestimmbar sind [2], werden Schätzwerte für $g(n)$ und $h(n)$ verwendet. Sei somit $\hat{g}(n)$ der Wert des bisher gefundenen kürzesten Pfads von s zu n und sei $\hat{h}(n)$ der Wert der Heuristik des Pfads von n zum Ziel z so folgt als Abschätzung für $f(n)$

$$\hat{f}(n) = \hat{g}(n) + \hat{h}(n) \tag{3.2}$$

Man kann nun beweisen das für alle Knoten n, die als offen markiert sind, der kürzeste Pfad vom Startknoten aus bereits bekannt ist, und somit gilt:

$$\hat{g}(n) = g(n) \tag{3.3}$$

Da davon ausgegangen werden kann, das alle Pfade $g(k_i)$ für bereits geschlossene Knoten k bekannt und minimal sind und somit auch gilt[3]

$$\hat{g}(n) = g(k_i) + min(v_{k_i,n}) = g(n) \tag{3.4}$$

muss nur noch die Heuristik $\hat{h}(n)$ definiert werden. Es kann bewiesen werden, das der A*-Algorithmus konsistent ist, das heißt, dass jeder bereits geschlossene Knoten nicht erneut geöffnet werden muss, wenn gilt:

$$\hat{h}(n) \leq h(n). \tag{3.5}$$

Für die Berechnung bedeutet dies, dass wir den Wert des Restpfades von n zu dem Zielknoten z durch die Heuristik zwar unterschätzen dürfen, falls wir aber für die Berechnung einen möglichst effizienten Algorithmus benötigen, wir die Heuristik auf keinen Fall überschätzen dürfen.

3.4.3 Vergleich verschiedener Heuristiken h(n)

Die in Kapitel 3.4.2 besprochene Abschätzungsfunktion $f(n)$ hat einen großen Einfluss auf das Verhalten des A*-Algorithmus. Zur Veranschaulichung sollen einige Möglichkeiten für die Heuristiken betrachtet werden[10]:

1. $\hat{h}(n) = 0$**:** Wenn der Wert der Heuristik konstant auf Null gesetzt wird, so ist dies gleich zu setzen damit, als würden keine Zusatzinformationen zur Be-

[2] exakt bestimmbar sind diese erst wirklich beim erreichen des Zielknotens z und sukzessivem Beenden des Algorithmus.

[3] analog zu 3.3.1 Schritt 2 mit $v_{k,n}$ als die Gewichtung der Kante zwischen den Knoten k und n.

rechnung der Route verwendet. Dies bedeutet das immer genau der Knoten aus der Menge von offenen Punkten ausgewählt wird, dessen Abstand zu einem der geschlossenen Knoten am kleinsten ist. Der kürzeste gefundene Pfad durch einen Zwischenknoten n hat zu jedem Zeitpunkt i immer $f(n) = g_i(n)$. Somit entspricht der A*-Algorithmus ohne zusätzliche Informationen genau dem Dijkstra-Algorithmus aus 3.3.1.

2. $\hat{h}(n) < h(n)$**:** Wenn der Abstand eines Knotens n zum Ziel immer unterschätzt wird, also immer kleiner ist als der tatsächliche Abstand, so findet der A*-Algorithmus garantiert den kürzesten Pfad zwischen Start und Ziel. Je kleiner $\hat{h}(n)$ im Verhältnis zu $h(n)$ ist, desto mehr Punkte muss A* expandieren und umso langsamer läuft der Algorithmus.

3. $\hat{h}(n) = h(n)$**:** Wenn der geschätzte Abstand eines Knotens genau dem tatsächlichen Abstand entspricht, expandiert A* immer die minimale Anzahl von Knoten entlang des kürzesten Pfades. Das heißt, dass wenn der kürzeste Pfad sechs Knoten enthält, der A*-Algorithmus nur genau sechs Knoten expandiert bevor der Kürzeste Pfad gefunden wurde. Zudem ist der Wert von $f_k(n)$ für jeden Teilknoten k auf dem kürzesten Pfad immer gleich dem exakten Wert des Abstands zwischen Start und Zielknoten.

4. $\hat{h}(n) > h(n)$**:** Wenn der geschätzte Abstand eines Knotens größer ist als der tatsächliche Abstand zum Zielknoten, so kann nicht garantiert werden, dass nicht kürzere Pfade zu bereits geschlossenen Knoten existieren. Somit müssten bereits geschlossene Knoten erneut geöffnet werden um den tatsächlich kürzesten Pfad zu finden. Wird dies nicht getan, so kann der A*-Algorithmus zwar unter Umständen schneller einen Pfad vom Start zum Ziel finden, es ist aber nicht garantiert, dass es sich um den Kürzesten handelt.

5. $\hat{h}(n) \gg h(n)$ Wenn der Schätzwert der Heuristik viel größer ist als der tatsächliche Abstand zum Zielknoten, so ist die Abschätzungsfunktion $f(n)$ nur noch abhängig von $\hat{h}(n)$ und die anderen Terme können vernachlässigt werden. Somit hängt die Reihenfolge der expandierten Knoten nur noch von den Werten der Heuristik der entsprechenden Knoten ab.

Anhand der vorgestellten Eigenschaften der Heuristik wird erkennbar, das der A*-Algorithmus durch die Modifizierung der Heuristikberechnung an die gegebenen Anforderungen anpassbar ist. Anstatt einer genauen, aber langsamen Pfadberechnung, kann auf Kosten der Genauigkeit auch schnell ein möglicherweise nicht optimaler Weg zum Ziel gefunden werden. Zudem kann durch perfektes Wissen der Anlage in kürzester Zeit der kürzeste Weg ermittelt werden.

3.4.4 Darstellung der Funktionsweise

Anhand eines einfachen Beispiels soll jetzt die Funktionsweise des A*-Algorithmus dargestellt werden. Die Knoten des folgenden Graphen sollen Städte darstellen und die Kanten geben die Verbindungen durch Schnellstraßen oder Autobahnen wieder. Als Heuristik wurde hier die Entfernung in Luftlinie zwischen den jeweiligen Städten und der Zielstadt gewählt, da diese die Bedingung $\hat{h}(n) \leq h(n)$ erfüllt:
insert graphic here

3.4.5 Berechnungsaufwand

Der Berechnungsaufwand für den A*-Algorithmus lässt sich nur schwer festlegen, da wie in 3.4.3 beschrieben, die Berechnung des kürzesten Pfads sehr stark von der Heuristik abhängt. Unter der Annahme, dass auch wirklich der kürzeste Pfad gefunden wird, muss für die Heuristik gelten:

$$0 \leq \hat{h}(n) \leq h(n) \tag{3.6}$$

Somit muss auch der Berechnungsaufwand zwischen dem des Dijkstra-Algorithmus und dem Aufwand im Falle einer exakten Heuristik liegen. Da bei einer exakten Heuristik die Zahl der expandierten Knoten nur von der Anzahl der Knoten auf dem kürzesten Pfad abhängt, ist die Berechnungsaufwand konstant. Daraus folgt für eine beliebige Heuristik, die die Bedingung 3.6 erfüllt:

$$\mathcal{O}(|\mathcal{V}|^2) \geq \mathcal{O}(\hat{h}(n)) \geq \mathcal{O}(1) \tag{3.7}$$

Somit ist der Berechnungsaufwand im schlechtesten Fall quadratisch, im besten Fall aber konstant. Eine bessere Heuristik führt somit dazu, dass weniger Knoten als im schlechtesten Fall expandiert werden müssen und sich somit der Rechenaufwand verringert.

3.5 Zusammenspiel der Algorithmen

Wie bereits in Abschnitt 3.2.2 erwähnt, wird nun ausgenutzt, dass der Dijkstra-Algorithmus die Abstände zu allen betrachteten Knoten im Verlaufe seiner Wegberechnung ermittelt. Der Nachteil, dass er vergleichsweise langsam arbeitet wird umgangen, indem der Algorithmus nicht in der zeitkritischen zyklischen Bearbeitungsphase sondern beim Hochfahren der CPU ausgeführt wird. Hier ist die Fahrzeugsteuerung[4] noch nicht aktiv und somit können auch längere Berech-

[4] und vor allem auch der Watchdog-Timer

nungen ausgeführt werden. Zusätzlich werden nicht alle Knoten als Startknoten für die Abstandsberechnung mittels Dijkstra verwendet, sondern nur solche, die im späteren Anlagenbetrieb als endgültige Zielknoten für die Wegfindung infrage kommen. Transitknoten, die nur im Verlaufe der Abarbeitung einer Route durchfahren werden, aber an denen nie eine Route endet werden hierbei ausgelassen.
Die Ergebnisse der Dijkstra-Berechnungen werden in einer Tabelle gesichert und als Basis für die Heuristik der Wegfindung mittels A* im zyklischen Betrieb verwendet. Wie in 3.4.3 gezeigt wurde ist die Anzahl der expandierten Knoten, und somit auch die Laufzeit, minimal wenn als Heuristik der exakte Abstand zum Ziel genutzt wird. Dieser Fall ist, unter der Bedingung das sich die Anlage seit dem Hochfahren der CPU nur geringfügig geändert hat, durch die Verwendung der Abstände des entsprechenden Ziels aus der Dijkstra-Tabelle gegeben. Auch bei Änderungen wie Sperrungen von Kanten oder der Blockade von Teilstrecken durch andere Fahrzeuge erfüllt diese Heuristik noch die Anforderung $\hat{h}(n) < h(n)$, da die neue exakte Route sicher länger ist, als die vorausberechnete Route ohne Sperrungen. Bei Hinzufügen neuer Kanten zur bestehenden Anlage muss die CPU neu gestartet werden, damit die Bedingung aus Gleichung 3.6 in allen Fällen erfüllt ist.

4

Technische Implementierung der Algorithmen

4.1 Kurze Einführung in die Programmiersprache und Programmierumgebung

Für die Implementierung der Wegfindung wurde im Kapitel 2.2.2 definiert, dass diese auf einer Siemens SPS (englisch: Programmable Logic Controller (PLC)) der S7-1200er Reihe lauffähig ist. Bei der 1200er Reihe handelt es sich um Steuerungen des niedrigen Leistungssegments. Für die Projektierung und Programmierung von Anlagen mit Steuerungen dieser Art wird deine proprietäre Entwicklungsumgebung namens Totally Integrated Automation Portal (TIA-Portal) von Siemens zur Verfügung gestellt.

4.1.1 Siemens TIA-Portal

Das Siemens TIA-Portal vereinigt viele Aspekte der Projektierung von Anlagen in einer einheitlichen Oberfläche. Innerhalb des TIA-Portals können beispielsweise Projekte bestehend aus mehreren Antrieben und SPSen gemeinsam geplant und erstellt werden. Die aktuelle Version des TIA-Portals ist in der Version 13 verfügbar und bietet vor allem eine anwenderfreundliche Oberfläche für komplexe Automatisierungsaufgaben. Die Kernkomponente für die Erstellung von Programmen für Speicherprogrammierbare Steuerungen ist die Programmierumgebung STEP7[1]. Abbildung **INSERT GRAPHIC HERE** zeigt die Projektansicht des TIA-Portals

[1] STeuerungen Einfach Programmieren Version 7 (STEP7)

4.1.2 Programmierumgebung STEP7

Die Grundelemente eines STEP7-Projekts sind die projektierten Steuerungen. Diese sind wiederum unterteilt in Teilelemente wie die Hardware-Konfiguration der Steuerung, das Anwenderprogramm, die verwendeten Variablen, benötigte Datentyp-Definitionen und Komponenten zur Überwachung und Modifizierung der Steuerungsdaten im laufenden Betrieb **Insert grafic here**. Für die Implementierung der Wegfindungsalgorithmen sind vor allem das Steuerungsprogramm und die darin verwendeten Datentypen von Bedeutung. Ein Anwenderprogramm besteht aus bis zu vier Arten von Programmbausteinen[11]:

Organisationsbaustein (OB): Diese Bausteine bilden die Schnittstelle zwischen dem Betriebssystem und dem Anwenderprogramm. Sie haben jeweils vordefinierte Funktionalitäten und bilden somit das Grundgerüst des Anwenderprogramms. Die in dieser Implementierung verwendeten OBs sind zum einen der Systemstart-Baustein und der Baustein zur zyklischen Abarbeitung von Teilschichten des Programms.

Datenbaustein (DB): Datenbausteine dienen zur Speicherung von variablen Daten, die im gesamten Anwenderprogramm benötigt werden. Sie werden unter anderem zur Sicherung der Topologiedaten der Anlage, sowie als Schnittstellen zwischen verschiedenen Programmschichten verwendet.

Funktion (FC): Funktionen sind Bausteine zur elementaren Kapselung von Funktionalitäten. Sie werden im Anwenderprogramm definiert als Unterprogramme, die keinen eigenen Speicher zur Sicherung von Variablenwerten zwischen zwei aufeinanderfolgenden Programmaufrufen benötigen.

Funktionsbaustein (FB): Funktionsbausteine realisieren wie FCs Unterprogramme, stellen aber zusätzlichen Speicherbereich für die permanente Sicherung von Daten internen Variablen zur Verfügung. Bei der Verwendung eines FBs wird bei dessen Initialisierung ein entsprechender Instanz-DB generiert, in dem Daten für die Verwendung in späteren Programmaufrufen gespeichert werden können.

FCs und FBs entsprechen den Funktionsdefinitionen in anderen Programmiersprachen. Es können die Schnittstellen der Bausteine sowie deren Schnittstellentypen definiert werden. IN-Variablen werden beispielsweise nur lesend verwendet, OUT-Variablen werden nur schreibend verwendet und INOUT-Variablen werden sowohl schreibend als auch lesend verwendet. Innerhalb eines Bausteins können sowohl temporäre als auch statische[2] Variablen zur Zwischenspeicherung von Variablenwerten während der Programmabarbeitung genutzt werden. Da statische Variablen einen Instanz-DB benötigen, sind sie nur in FBs verwendbar. Die Bausteine können in einer von vier Programmiersprachen geschrieben werden. Funktionsplan (FUP) und Kontaktplan (KOP) sind Sprachen zur graphischen Programmierung. Anweisungsliste (AWL) ist eine Assembler-ähnliche Sprache für generelle Programmieraufgaben, die unter anderem die byteweise Manipulation von Daten vereinfacht. Structured Control Language (SCL) ist eine Pascal-ähnliche Programmiersprache, die durch ihre einfachen Implementierungsmöglichkeiten von Schleifen geeignet ist für die Programmierung komplexer Aufgabenstellungen **insert comparison graphic here**. Bei der Erstellung des Anwenderprogramms für die Wegfindung wurden die Verwendeten OBs in FUP erstellt und alle anderen Bausteine in SCL.

4.1.3 Arbeitsweise einer SPS

Eine Speicherprogrammierbare Steuerung arbeitet nach dem Prinzip eines Echtzeitsystems. Das projektierte Anwenderprogramm wird in einer Endlosschleife zyklisch abgearbeitet. Zu Beginn eines Bearbeitungszyklus wird ein Prozessabbild aller Eingangsbaugruppen der Steuerung generiert, das für den kompletten Zyklus als Basis für die Werte der Eingänge benutzt wird. Während des Zyklus werden die berechneten Werte für die Ausgänge in ein weiteres Prozessabbild geschrieben, welches erst nach Ende des aktuellen Bearbeitungszyklus an die Ausgangsbaugruppen übertragen wird. Somit müssen Mehrfachzuweisungen innerhalb eines Zyklus vermieden werden, da nur die letzte Zuweisung an die Ausgänge weitergegeben wird **insert PAE PAA graphic here**. Durch OBs können zusätzliche Funktionen außerhalb der zyklischen Bearbeitung realisiert werden. Beispielsweise können im Startup-OB einmalig Anweisungen beim Hochfahren der CPU ausgeführt werden.

4.2 Realisierung der Anlagentopologie

Als Grundlage für die Wegfindung muss zunächst die Anlage definiert werden. Wie bereits in Kapitel 3.1 beschrieben, wird die Anlage durch einen Graphen mit

[2] persistent über Funktionsaufrufe hinaus

positiven Kantengewichtungen dargestellt. Für die technische Realisierung der Anlage ist unter anderem wichtig, dass die Beschreibung der Topologie erweiterbar ist, und sich der physikalische Aufbau leicht umsetzen lässt.

4.2.1 Gewählte Anlagentopologie

Im Hinblick auf die in Kapitel 2.3 definierten Implementierungsstufen und die dazu gehörigen Use-cases wurde für die Topologie der Anlage eine Matrixstruktur gewählt. Bei der Platzierung der Bearbeitungsstationen an die Außenkanten der 2x4 Wabenstruktur, können die Fahrstrecken im Inneren als eine Art Überholspur für Fahrzeuge genutzt werden, die sich schneller durch die Anlage bewegen. Zudem wurden Eingangs- und Ausgangsknoten als Start und Endpunkt für die Wegfindung definiert. Als zusätzliche Rückführstrecke für FTF nach Beendigung ihres Bearbeitungsauftrags wurde eine Rückführstrecke von dem Endzum Startknoten der Anlage hinzugefügt, welche aber in der Wegberechnung nicht berücksichtigt wird. Der Gesamtaufbau der Anlage wird in **INSERT GRAPHIC HERE** schematisch dargestellt.

4.2.2 Physikalische Aufbau

In Kapitel 2.1 wurde dargelegt, dass die Anlage als Vorführmodell für Konzepte von Industrie 4.0 verwendet werden soll. Somit die Portabilität der Anlage ein wichtiger Punkt. In Zusammenarbeit mit Herrn A. Meier wurde als Basis der Anlage ein modulares System aus leichten Kartonplatten definiert. Aufbauend hierauf werden für die optische Fahrsteuerung die Fahrstrecken mit dunklem Isolierband aufgeklebt. Diese lassen sich bei Bedarf auch entfernen oder modifizieren um Änderungen der Anlagentopologie zu simulieren. Die Kreuzungen der Fahrstrecken, die die Entscheidungspunkte für die Fahrzeugsteuerung und die Wegfindung darstellen, entsprechen den Knoten des korrespondierenden Anlagengraphen, die Fahrstrecken selbst, als Verbindungen zwischen einzelnen Entscheidungspunkten, entsprechen somit den Kanten des Graphen.**insert photo Anlage** Für die Wegfindung ist es zudem wichtig, die aktuelle Position eines Fahrzeugs ermitteln zu können, um diese als Basis für den Startknoten der Berechnung zu benutzen. Aus diesem Grund wurden unterschiedliche Methoden zur Orientierung der Fahrzeuge innerhalb der Anlage untersucht. Da eine zentrale Erfassung der Fahrzeugpositionen durch zusammengeschaltete mechanische, analoge oder optische Sensoren nicht in das Konzept der Dezentralisierung von Industrie 4.0 passt, wurde ein Radio Frequency Identifikation (RFID)-basiertes Identifikationssystem für die Positionserfassung ausgewählt. Dieses funktioniert durch die Anbringung von RFID-Transpondern auf der Unterseite der Kartonplatten, jeweils auf Höhe eines Entscheidungsknotens. Jedes Fahrzeug besitzt einen RFID-Sensor,

welcher vor dem Fahrzeug an einem Ausleger befestigt ist, um Anlagenknoten zu identifizieren, an die sich das Fahrzeug annähert. Hier hat sich die Entscheidung für Kartonplatten zur Anlagenmodellierung als Vorteil erwiesen, da die Transpondersignale auch durch den Karton auf der Oberseite der Anlagenplatten detektierbar ist, und die Transponder somit nicht auf der Fahrstrecke selbst befestigt werden müssen, was zu Behinderungen beim Fahren führen könnte**insert photo RFID**.

4.2.3 Beschreibung als Knotenliste

Sowohl für die Wegfindung als auch für die Fahrsteuerung wird eine digitale Repräsentation der Anlage benötigt. Aufgrund der einfachen Erweiterbarkeit wurde hier die Darstellungsform der Knotenliste für Graphen ausgewählt, da bei dieser Datenstruktur einfach Knoten hinzugefügt und entfernt werden können. Für einen Einzelknoten wurde die Anzahl der möglichen Kanten pro Knoten auf vier beschränkt. Dies macht eine Zuordnung der einzelnen Kanten zu den vier Himmelsrichtungen möglich. Dies wird vor allem für die Fahrsteuerung benötigt, da hier eine der Anforderungen die Möglichkeit der Ausführung von 90-Grad-Kurven ist. Da die Fahrsteuerung als getrenntes Programmmodul implementiert wurde besitzt sie eine Kopie der Knotenliste der Anlage mit den zusätzlichen, bei der Wegfindung nicht benötigten Knoten der Rückführstrecke.
Ein Einzelknoten hat folgenden generellen Aufbau:

ID	Nummer des aktuellen Knoten	
Kanten	ID Nord	verbundener Knoten in Richtung Norden
	Dist. Nord	Abstand zum Knoten in Richtung Norden
	ID Ost	verbundener Knoten in Richtung Osten
	Dist. Ost	Abstand zum Knoten in Richtung Osten
	ID Süd	verbundener Knoten in Richtung Süden
	Dist. Süd	Abstand zum Knoten in Richtung Süden
	ID West	verbundener Knoten in Richtung Westen
	Dist. West	Abstand zum Knoten in Richtung Westen

Da in STEP7 Speicher nicht dynamisch alloziert werden kann, hat jeder Knoten immer Speicherplatz für die Daten von allen vier möglichen Kanten, auch wenn er in der realen Anlage mit weniger als vier Knoten verbunden ist. Da die ID 0 für den Startknoten reserviert wurde werden Verbindungsrichtungen durch eine -1 im ID-Feld als nicht verbunden gekennzeichnet. Hierdurch wird auch der zugehörige Abstand ignoriert.
Die Datenstruktur der Knotenliste selbst besteht aus einem einfachen Array mit Einzelknoten als Arrayelemente. Dieses Array wird initialisiert mit der einer

Konstanten, welche die Gesamtanzahl der möglichen Anlagenknoten enthält. Zusätzlich zu dem Knotenarray besteht der Listendatentyp noch aus der Anzahl der wirklich im Array enthaltenen Knoten. Diese wurde aus Gründen der Konsistenz hinzugefügt , da andere selbst definierte Listentypen im Projekt mit variablen Anzahlen von Arrayelementen arbeiten, was bei der Anlagentopologie nur selten der Fall ist. Dies könnte in diesem Falle genutzt werden wenn zu einem späteren Zeitpunkt weitere Knoten zur Anlage hinzugefügt werden. Hier muss aber ein Neustart des FTF da sonst die in Kapitel 3.5 besprochene Heuristik-Berechnung für die neuen Knoten nicht existiert und somit die Knoten nicht für die Wegberechnung verwendet werden können.**Insert Graphic Datatype TIA** Abschließend sei noch erwähnt das die in 4.2.2 erwähnten RFID-Werte der Knoten nicht für die Wegfindung relevant sind. Das Einlesen und die Zuordnung von RFIDs zu den entsprechenden Knoten wird komplett in der Fahrzeugsteuerungsschicht erledigt, welche der Wegfindungsschicht dann nur die ID weitergibt.

4.2.4 Beschreibung als Adjazenzmatrix

Unter dem Aspekt der Erweiterbarkeit ist die Listendarstellung des Graphen ideal, da Knoten unabhängig von ihrer Position innerhalb der Anlage einfach am Ende der Liste angehängt werden können. Solange der neue Knoten gültig ist, kann der erweiterte Anlagengraph verwendet werden. Für die eigentliche Wegfindung ist eine solche Liste jedoch ungünstig, um schnell den Abstand eines Knotens zu einem beliebigen anderen Knoten zu prüfen. Bei einer Liste müsste im schlechtesten Fall bei Überprüfung des letzten Knotens das komplette Array durchlaufen werden. Aus diesem Grund wird beim Hochfahren der CPU, vor der Berechnung der Heuristik mittels Dijkstra, die Knotenliste geparsed und eine Adjazenzmatrix der Anlage generiert. Hier kann in konstanter Zeit[3] der Abstand zwischen zwei Knoten x und y ermittelt werden, indem einfach der in der Matrix unter den Indizes (x, y) hinterlegte Wert abgerufen wird. Dies beschleunigt vor allem die Laufzeit des im zyklischen Betrieb ausgeführten A*-Algorithmus. Für die Berechnung der Heuristiken ist dies weniger von Bedeutung, da hier keine Zeitbeschränkungen vorliegen und nur der weniger kritische Aspekt der kürzeren Hochfahrzeit der SPS beeinflusst wird.**insert graphic here**

4.3 Implementierung der Algorithmen

Das gesamte Programm wurde in einzelne funktionale Schichten aufgebaut, um den Industrie 4.0 Aspekt der Modularisierung darzustellen. Somit macht es keinen funktionalen Unterschied, ob die Wegfindungschicht tatsächlich auf dersel-

[3] Aufwand $\mathcal{O}(1)$

ben Steuerung läuft wie die Fahrzeugsteuerungsschicht. Als Schnittstelle dienen hier sowohl bei der Wegfindungs- als auch bei der Fahrzeugsteuerungsschicht Datenbausteine, in welche die zu übertragenden Ergebnisse geschrieben werden. Diese Daten werden durch die in Kapitel 5.1 näher beschriebene Kommunikationsschicht übertragen. Dies hat den Vorteil das unterschiedliche Arten der Kommunikation einfach ausgetauscht werden können, ohne die anderen Schichten zu beeinflussen.**insert schicht graphic here**.
Die in Kapitel 3 beschriebenen Algorithmen wurden beide auf ähnliche Weise in STEP7 implementiert. Für die A* und Dijkstra wurden jeweils geeignete Datentypen definiert, welche an die Anforderungen der Algorithmen angepasst wurden. Die Implementierung verwendet hier in beiden Fällen eine Priority-Queue auf Basis eines Arrays, um bei der Berechnung den jeweils nächsten besten Knoten zu identifizieren. Das Array als Grundlage der Prioritätsschlange wurde hier gewählt, da sich die Realisierung in STEP7 einfacher gestaltet als beispielsweise eines Heaps. Dafür müssen hier Abstriche bei der Sortiergeschwindigkeit gemacht werden.

4.3.1 Berechnungsdatentypen

Für die Berechnung der Algorithmen wurden Datentypen erstellt, die alle für den Algorithmus benötigten Informationen enthalten. Die Datentypen beschreiben die Knoten einzeln und werden zur Berechnung in eine Priority-Queue eingefügt. Als Ergebnis der Berechnung wird eine geordnete Liste der Einzelknoten ausgegeben. Die Datentypen enthalten folgende Grunddaten:

NodeID	ID des betrachteten Knotens
Dist. to Start	Bisher zurückgelegter Weg
Expected Dist.	Vorausichtl. Gesamtpfadlänge durch diesen Knoten (nur A*)
ParentID	Vorgängerknoten

Der bisher zurückgelegte Weg gibt bei der Dijkstra-Implementierung Aussagen wie weit ein Knoten von einem Startknoten entfernt ist. Wird dieser Startknoten später als Ziel gewählt, so kann der Abstand als Heuristik für A* verwendet werden[4]. Da jeweils der kürzeste Weg gesucht ist, wird für diese Datenkomponente sowie auch für den voraussichtlichen Gesamtweg der Initialwert auf den Maximalwert des verwendeten Datentyps gesetzt, in diesem Fall 65535 für 16-Bit Integer.
Die voraussichtliche Pfadlänge bei dem Datentyp für den A*-Algorithmus setzt sich zusammen aus dem bisher zurückgelegten Weg und dem geschätzten Wert für den restlichen Weg zum Ziel. Bei Verwendung einer exakten Heuristik ist die-

[4]Bei gerichteten Graphen muss zusätzlich die Gegenrichtung betrachtet werden.

ser Wert für alle Knoten auf dem kürzesten Pfad immer konstant, solange sich die Anlage seit Berechnung der Dijkstra-Heuristik nicht verändert hat.
Der Zeiger zum Vorgängerknoten wurde hier als einfache Zahl implementiert, da das Pointer-Konstrukt in STEP7 etwas anders funktioniert als beispielsweise in der Programmiersprache C. Dies hat den Nachteil das bei der Zusammenstellung der Route anhand der Knotenliste etwas mehr Zeit verwendet werden muss, da die Liste jeweils nach dem Vorgängerknoten durchsucht werden muss.

4.3.2 Priority-Queue

Da in STEP7 keine Container-Datentypen existieren, die mit unterschiedlichen Elementen zurechtkommen, wurden für beide Algorithmen getrennte Datentypen für die Priority-Queue definiert. Diese existieren analog zu der in 4.2.3 beschriebenen Anlagenknotenliste aus einem Array der Knotenelemente und der Anzahl der beinhalteten Elemente. Dies wurde aus Gründen der Effizienz auf diese Art realisiert, da der große Vorteil des A*-Algorithmus vor allem darin liegt, dass nur ein kleiner Teil der Knoten der Anlage betrachtet werden muss. Da das Array aber nicht dynamisch mit der benötigten Größe alloziert werden kann, sondern immer den schlechtesten Fall berücksichtigen muss, wurde zur Initialisierung des Arrays die Gesamtanzahl der Anlagenknoten als Konstante verwendet. Damit somit nicht bei den Priority-Queue-Operationen die leeren Restplätze des Arrays mit bearbeitet werden, wurde die Anzahl der Knoten als Datenkomponente mit hinzugefügt. Besteht die Anlage beispielsweise aus 40 Knoten aber es befinden sich nur 6 echte Knoten in der Prioritätsschlange, so laufen alle Schleifen nur über die Elemente null bis fünf.
Zur Verwendung der Priority-Queue wurden für beide Listendatentypen Zugriffs- und Verwaltungsfunktionen implementiert. STEP7 besitzt zwar die Möglichkeit Funktionen mittels eines sogenannten VARIANT-Pointers so zu implementieren, dass sie analog zum Konzept der Überladung mit unterschiedlichen Eingangsdatentypen zurechtkommen[11], dies ist in diesem Fall aber nicht möglich, da nicht effizient auf die Einzelelemente des Datentyps zugegriffen werden kann. Aus diesem Grund wurden die folgenden Operationen für jeden der beiden Datentypen Implementiert:

- **Insert**: Einfügen eines Knotens in die Priority-Queue.
- **Pop**: Ausgabe und Entfernung des nach der Prioritätsbedingung „besten“ Knotens aus der Priority-Queue.
- **Sort**: Wiederherstellung der Prioritätsschlangeneigenschaften durch Sortierung.
- **Contains**: Überprüfung, ob Knoten bereits in der Schlange enthalten ist.

- **UpdateNode**: Modifizierung eines Knotens in der Priority-Queue.

Die Operationen „Insert“, „Pop“ und „UpdateNode“ beinhalten alle eine abschließende Wiederherstellung der Prioritätsbedingung mittels der in **INSERT CODE REFERENCE HERE** dargestelten „Sort“-Funktion.
Um die Zugriffsoperationen effizienter zu machen, wurden die Prioritätsschlangen so implementiert, dass das beste Element der letzte gültige Knoten im Array an dem Index $Anzahl - 1$ ist. Dies Vereinfacht das Entfernen des besten Knotens, da nach der Ausgabe nur die Anzahl um eins dekrementiert wird und nicht jeder Knoten nachgerückt werden muss, wie es der Fall wäre wenn das beste Element am Anfang stehen würde.

4.3.3 Dijkstra-Algorithmus und Heuristik-Tabelle

Die eigentliche Implementierung des Dijkstra-Algorithmus ist recht einfach möglich. Zunächst muss die Prioritätsschlange mit allen Knoten der Anlage gefüllt[5] und die Distanz des Startknotens auf null gesetzt werden. Da hier keine Route gesucht wird sondern nur ein Abstand zum Startknoten, wird der Algorithmus nicht beendet wenn ein bestimmter Zielknoten erreicht wurde, sondern erst wenn alle Knoten betrachtet worden sind. Bereits betrachtete Knoten werden gemäß des in Abschnitt 3.3.1 beschriebenen Prinzips aus der Priority-Queue in eine geordnete Knotenliste übertragen. Dieser Heuristikliste genannte Datentyp enthält die Knotenelemente aufsteigend sortiert nach Abstand zum Startknoten.
Der gesamte Dijkstra-Algorithmus wird in einem FC mit dem Namen „FC_buildHeuristicList“ berechnet **Insert graphic FC here**. Dieser Baustein wird für jeden Knoten der Anlage aufgerufen, der als Endzielknoten[6] markiert ist. Wie bereits erwähnt wird dieser Baustein in einer Schleife im „OB100 Startup“ aufgerufen. Die resultierenden Heuristiklisten werden in einem Datentyp namens Heuristiktabelle in der Arraykomponente gespeichert und im zyklischen Betrieb von dem A*-Algorithmus verwendet.

Notiz: Bausteine näher beschreiben? mit Auflistung und Erklärung aller Schnittstellen?

4.3.4 A*-Algorithmus

Die Implementierung des A*-Algorithmus ähnelt der des Dijkstra-Algorithmus. Auch hier muss vor Beginn der Berechnung zunächst die Priority-Queue vorbereitet werden. Da der A*-Baustein im Gegensatz zu dem Dijkstra-Baustein mehr als einmal aufgerufen wird, bedeutet dies, dass die Prioritätsschlange zunächst

[5] Gruppen II und III aus Abschnitt 3.3.1 werden gemeinsam in der Prioritätsschlange gehalten.
[6] vergleiche Abschnitt 3.5.

in den Grundzustand versetzt werden muss, indem alle Knoten aus vorherigen Berechnungen entfernt werden. Zudem muss die Liste der geschlossenen Knoten, die in Abschnitt 3.4.1 erwähnt wurde, komplett geleert werden, damit kein Knoten bei der Routenberechnung fälschlicherweise als bereits betrachtet angesehen wird.
Die Berechnung selbst startet nun mit dem Hinzufügen des Startknotens zur Priority-Queue, welche die Liste der offenen Knoten[7] darstellt. Es wird nun immer der beste Knoten aus der Priority-Queue herausgenommen und als geschlossen markiert. Im Anschluss wird die Adjazenzmatrix nach neuen erreichbaren Knoten durchsucht. Falls diese noch nicht als geschlossen markiert wurden wird geprüft, ob die Verbindungskante zu dem neuen Knoten in einer speziellen Matrix für permanente Blockaden als blockiert gekennzeichnet wurde. Ist dies nicht der Fall so wird der neue Knoten zu der Liste der offenen Knoten hinzugefügt. Der fertig betrachtete geschlossene Knoten wird dann in die statische Routenliste eingefügt, aus der nach dem Ende der Berechnung die Route generiert wird. Wird der Zielknoten erreicht oder befinden sich keine Knoten mehr in der Priority-Queue, so beendet sich der Algorithmus. Die Routenliste wird in die Ausgangsvariable kopiert und der Baustein meldet das Berechnungsende über einen Bitmerker an die übergeordnete Wegfindungsschicht.
Die Erkennung von permanenten Blockaden wurde durch eine Matrix realisiert, da hier, wie auch bei der Adjazenzmatrix, in konstanter Zeit geprüft werden kann ob auf dem betrachteten Streckenabschnitt Behinderungen vorliegen. Diese permanenten Blockaden liegen über längere Zeitintervalle vor und können deshalb bereits zu Beginn der Routenberechnung berücksichtigt werden. Dies ist bei temporären Blockaden durch andere Fahrzeuge nicht der Fall, weshalb zur Erkennung dieser Art von Behinderungen ein anderer Mechanismus zur Anwendung kommt, der in Abschnitt 5.2 näher beschrieben wird.

4.4 Einhaltung der Echtzeitbedingung

Wenn die Wegfindungsschicht auf der gleichen SPS wie die Fahrzeugsteuerungsschicht implementiert wird, so muss die Wegberechnung vor allem bestimmte zeitliche Anforderungen erfüllen, damit die Fahrzeugsteuerung nicht beeinträchtigt wird. Eine Steuerung arbeitet im zyklischen Betrieb alle OBs nacheinander ab, die als zyklisch definiert wurden. Durch die Modularität des Anwenderprogramms besitzt jede Schicht ihren eigenen zyklischen OB, da sie nur so unabhängig von den anderen Schichten ist. Dies bedeutet, das eine Berechnung eines Weges die CPU nicht so lange in Anspruch nehmen darf, das in dieser Zeit beispielsweise ein überfahrener RFID-Knoten nicht eingelesen wird und somit eine

[7] Open-List

bevorstehende Kreuzung nicht erkannt wird. Dies führt spätestens beim nächsten erreichten Knoten zu der Erkennung eines Fehlers und das Fahrzeug geht in einen permanenten Fehlerzustand. Ebenso kann es bei zu langer CPU-Belegung vorkommen, dass das Fahrzeug die Spur verliert und dies erst bemerkt wenn es komplett von der Fahrstrecke abgekommen ist. Aus diesem Grund ist es sehr wichtig, die Wegfindungsschicht so anzupassen, dass die zwischen zwei Ausführungen der Fahrzeugsteuerung höchsten 10ms liegen. Da auch die Kommunikationsschichten zyklische OBs besitzen, die der Reihenfolge nach abgearbeitet werden müssen, wurde festgelegt, das die Wegberechnung maximal 5ms dauern darf. Dies hat einige Konsequenzen für die Implementierung der Wegfindungsalgorithmen.

4.4.1 Ausführung bei Systemstart

Wie bereits mehrfach erwähnt wird die Heuristiktabelle beim Hochfahren der CPU generiert. Eigentlich ist es nicht notwendig für den A*-Algorithmus, dass bereits ein Wert für die Heuristik existiert. Es könnte auch im zyklischen Betrieb ein Wert für die Abschätzungsfunktion $f(n)$ aus Abschnitt 3.4.3 berechnet werden, beispielsweise mittels einer Funktion für die Manhattan-Distanz, welche den Abstand zweier Knoten anhand der Differenz der Koordinaten ermittelt. Dies würde den Speicherverbrauch erheblich reduzieren, da keine großen Tabellen gespeichert werden müssten, sondern nur die jeweiligen Koordinaten der Knotenpunkte.
Die Berechnung einer Funktion $f(n)$ ist aber zeitintensiver als das Nachschlagen von Werten in einer Tabelle. Da der Speicherbedarf für die Tabelle der realisierten Anlage mit 25 Knoten und 10 Endzielknoten nur insgesamt 2300 Byte beträgt, fällt dies selbst bei der kleinsten Steuerung vom Typ S7-1214C mit 100kB Arbeitsspeicher kaum ins Gewicht. Zugleich kann die Laufzeit des A*-Algorithmus im zyklischen Betrieb deutlich reduziert werden, da die Tabellenheuristik mittels Dijkstra, im Gegensatz zur Manhatten-Distanz, eine exakte Heuristik liefert, die zusätzlich zur eingesparten Rechenzeit noch die Anzahl der betrachteten Knoten minimiert.
Da die Berechnung des Dijkstra-Algorithmus aber unter anderem auch wegen der gewählten Implementierung mit arraybasierten Priority-Queues zeitintensiv ist[12] und einige Sekunden dauern kann, muss die Tabellengenerierung zu einem Zeitpunkt geschehen, an dem die Echtzeitbedingung der Fahrzeugsteuerungsschicht nicht greift. Aus diesem Grund wurde der Hochfahrvorgang der CPU gewählt, um die notwendigen Initialisierungen der Heuristiktabelle durchzuführen, da hier weder die Fahrzeugsteuerung noch der Watchdogtimer, welcher den zyklischen Betrieb standardmäßig auf 50ms beschränkt, aktiv ist. Zusätzlich wird zu diesem Zeitpunkt noch die Adjazenzmatrix aus der Anlagen-

topologie generiert, da auch der Parsevorgang der Anlagenknoten bei größeren Anlagen etwas mehr Zeit in Anspruch nehmen kann.

4.4.2 Zyklische Ausführung

Trotz der Auslagerung der Heuristikberechnung auf die Hochlaufphase der CPU kann bei langen Routen nicht garantiert werden, das die als Ziel gesetzten 5ms für die Berechnungszeit eingehalten werden. Vor allem wenn sich die Anlage seit dem Hochfahren des Fahrzeugs stark verändert hat und somit die Heuristiktabelle keine exakte Heuristikwerte mehr liefert, müssen unter Umständen zusätzliche Knoten betrachtet werden, die nicht auf dem kürzesten Pfad vom Start- zum Zielknoten liegen. Der Aufwand für die Berechnung verschiebt sich also immer mehr nach $\mathcal{O}(|\mathcal{V}|^2)$ wie in Abschnitt 3.4.5 erläutert wurde.
Aus diesem Grund wurde der A*-Algorithmus so implementiert, das der Zeitaufwand pro Zyklus einigermaßen konstant bleibt. Dies wurde erreicht indem die Schleife, welche läuft, solange das Ziel noch nicht erreicht ist, aufgetrennt und durch eine konditionale IF-Abfrage ersetzt wurde. Nach der Schleifenauftrennung wird immer nur ein Knoten untersucht und gegebenenfalls seine Verbindungsknoten zur Open-List hinzugefügt, weitere Knoten werden erst in den darauf folgenden Zyklen expandiert. Dies hat zur Folge, das zwar die Bearbeitungszeit pro Zyklus sehr kurz gehalten wird, im implementierten Beispiel eine Zykluszeit der Wegfindungsschicht kleiner 1ms, die Zeit bis zur Berechnung einer kompletten Route jedoch deutlich länger wird. Dies ist der Fall, da zwischen jeweils zwei Schleifendurchläufen immer die kompletten OBs der anderen Schichten abgearbeitet werden. Dieses Problem kann aber zunächst vernachlässigt werden und wird im Abschnitt 5.1.1 näher behandelt.
Ein zweites Problem, das bei der Aufteilung des Algorithmus auf mehrere Zyklen auftritt ist die Inkonsistenz von Routen. Während der Berechnung des Algorithmus darf die in Abschnitt 4.3.4 erwähnte Routenliste nicht als gültige Route missverstanden werden, da hierdurch das Fahrzeug eine möglicherweise falsche Route nehmen würde, die nicht zum Ziel führt. Somit darf der Baustein nur finale Routen ausgeben, und temporäre Routen nur intern speichern. Der Baustein jedoch in mehreren Zyklen hintereinander aufgerufen wird und zwischen zwei Aufrufen die bereits berechneten Daten nicht verloren gehen dürfen, muss der verwendete Baustein vom Typ FB sein, da nur diese Bausteine einen zugeordneten Instanz-Datenbaustein besitzen, in dem Werte über den Funktionsaufruf hinaus zwischengespeichert werden können.
Durch die Kombination dieser beiden Maßnahmen, kann erreicht werden, dass die CPU-Belegung durch die Wegfindungsschicht im zyklischen Betrieb, quasi konstant[8] auf >1ms gehalten werden kann.

[8] immer noch abhängig von der Kantenanzahl des aktuell betrachteten Knotens.

4.5 Steuerung des Bearbeitungsablaufs

4.5.1 Bearbeitungsreihenfolge

4.5.2 Zuweisung einer Bearbeitungsstation

4.5.3 Simulation der Bearbeitungszeit

4.6 Beispiel für eine einfache Routenberechnung

5

Besonderheiten der Dynamischen Wegfindung

5.1 Kommunikation

5.1.1 Interne Kommunikation

5.1.2 Externe Kommunikation

5.2 Algorithmische Kollisionsvermeidung

[13]
[14]rausnehmen falls originaltext nicht beschaffbar ist bis dahin

5.2.1 Problem des Zeitlichen Indeterminismus

[15]

5.2.2 Verhinderung von Deadlock-Situationen

6

Fazit

6.1 Aktueller Stand der Anlage

6.2 Komplikationen bei der Implementierung

6.3 Ausblick über zukünftige Erweiterungen

7

Anhang

Literaturverzeichnis

[1] T. Bauerhansl, M. ten Hompel, and B. Vogel-Heuser, *Industrie 4.0 in Produktion, Automatisierung und Logistik*. Springer Vieweg, 2014.

[2] B. Pottler, "Industrie 4.0 - modell zusammen mit ct." Präsentation Siemens AG, Oct. 2015. Version 3.

[3] P. Chakrabarti, S. Ghose, A. Acharya, and S. de Sarkar, "Heuristic search in restricted memory," *Artificial Intelligence*, vol. 41, pp. 197–221, dec 1989.

[4] R. E. Korf, "Real-time heuristic search," *Artificial Intelligence*, vol. 42, pp. 189–211, mar 1990.

[5] A. Stentz, "Optimal and efficient path planning for partially-known environments," *Proceedings IEEE International Conference on Robotics and Automation*, Mai 1994.

[6] S. Koenig and M. Likhachev, "Fast replanning for navigation in unknown terrain," *IEEE Transactions on Robotics*, vol. 21, pp. 354–363, June 2005.

[7] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–270, 1959.

[8] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," in *Proc. 25th Annual Symp. Foundations of Computer Science*, pp. 338–346, Oct. 1984.

[9] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, pp. 100–107, July 1968.

[10] A. Patel, "Amit's thoughts on pathfinding: Heuristics." `http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html`, 2016. Abgerufen am 05.08.2016.

[11] Siemens AG, *STEP 7 Professional V13 SP1 Systemhandbuch*, Dec. 2014.

[12] B. Cherkassky, A. V. Goldberg, and T. Radzik, "Shortest paths algorithms: Theory and experimental evaluation," *Mathematical Programming*, vol. 73, pp. 129–174, May 1993.

[13] D. Silver, "Cooperative pathfinding," *Association for the Advancement of Artificial Intelligence*, 2005.

[14] A. Zelinsky, "A mobile robot exploration algorithm," *IEEE Transactions on Robotics and Automation*, vol. 8, pp. 707–717, Dec. 1992.

[15] M. Erdmann and T. Lozano-Perez, "On multiple moving objects," in *Proc. IEEE Int. Conf. Robotics and Automation*, vol. 3, pp. 1419–1424, Apr. 1986.