Freie Universität Berlin
Fachbereich Mathematik und Informatik
Institut für Informatik

Bachelorarbeit

Abhängigkeitserkennung und parallele Ausführung von Code-Modulen im FUmanoid-Framework

Martin Wichner

Gutachter
Prof. Dr. Raúl Rojas
Dr. Daniel Goehring

Betreuer
Daniel Seifert

4. Juni 2015

Eigenständigkeitserklärung

Ich versichere hiermit an Eides statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, den 4. Juni 2015

Zusammenfassung

In dieser Arbeit wird eine Anpassung des Module-Frameworks der BerlinUnited-FUmanoids vorgestellt. Dabei wird als Erstes eine effizientere Berechnung der Abhängigkeiten der Module vorgestellt. Durch das Aufstellen eines Graphens, mit den Modulen als Knoten und den Abhängigkeiten zwischen den Modulen als Kanten, kann darauf eine Tiefensuche ausgeführt werden. Durch die Tiefensuche lassen sich zyklische Abhängigkeiten feststellen, welche dann mit Hilfe des Dijkstra-Algorithmus ausgegeben werden.
Im zweiten Teil der Anpassung, wird das Framework dahingehen geändert, dass es möglich ist mehrere Module parallel auszuführen. Durch eine Topologische Sortierung werden zur Laufzeit die Module ausgeführt, bei denen die Vorbedingungen erfüllt sind. Die Module die ausgeführt werden können, werden dann nach dem FIFO-Prinzip ausgeführt. Eine Erweiterung der parallelen Ausführung ist die Module nach dem Longest-Job-First-Prinzip auszuführen. Dabei werden die Laufzeiten der Module gespeichert und eine Durchschnittslaufzeit berrechnet.
Am Ende werden die Ausführungszeiten der sequentiellen Ausführung, der parallele Ausführung mit First-Come-First-Served und der parallelen Ausführung mit Longest-Job-First miteinander verglichen.

Inhaltsverzeichnis

1 Einleitung

Ein Teilgebiet der Robotik befasst sich mit Robotern, die sich in einer ständig verändernden Umwelt befinden. Der Mensch kann, auf Grund seiner kognitiven Fähigkeiten, auf diese Veränderungen sehr schnell reagieren. Damit ein Roboter dies bewerkstelligen kann, muss auch er auf Veränderungen reagieren können. Um solch hohe Reaktionszeiten zu erreichen, muss der Programmcode, welcher die kognitiven Fähigkeiten modelliert, oft hintereinander ausgeführt werden.

In dieser Arbeit wird eine Anpassung des *BerlinUnited-Frameworks* vorgestellt, welche eine Parallelisierung der Einzelkomponenten bereitstellt. Dazu wird als Erstes eine Abhängigkeitsanalyse der Software-Module aus dem FUmanoid-Projekt durchgeführt, um sicher zu stellen, dass es keine zyklischen Abhängigkeiten zwischen diesen Software-Modulen gibt. Danach werden die Software-Module, soweit es ihre Abhängigkeiten untereinander zulassen, parallel ausgeführt.

Das studentische Projekt *FUmanoids*, der Freien Universität Berlin, forscht an menschenähnlichen (*humanoiden*) fußballspielenden Robotern. Die Roboter der *FUmanoids* besitzen einen Block von Software-Modulen, welcher die kognitiven Fähigkeiten (*Cognition*) modelliert und einen anderen Block, welcher die motorischen Fähigkeiten (*Motion*) modelliert. Die *Cognition*, sowie die *Motion*, bestehen aus mehreren Modulen, welche unterschiedliche Aufgaben erfüllen. Die Module geben an, welche Daten sie benötigen und welche Daten sie bereitstellen. An Hand dieser Informationen wird eine Ausführungsreihenfolge bestimmt.

Durch die ständig wachsenden Anforderungen im RoboCup [3] müssen die Roboter immer mehr leisten. Der im Jahr 2015 eingeführte Ball ist nicht mehr einfarbig orange, sondern nun zu mindestens 50% weiß. Die restlichen Farben sind beliebige Farben. Um diesen, für die Roboter schwer erkennbaren Ball, denoch gut zu erkennen, soll die Auflösung der Kamera von 640x480 auf 1280x720 erhöht werden. Der Rechenaufwand für ein Bild mit dieser Auflösung ist weitaus größer, als mit der geringeren Auflösung. Aus diesem Grund soll das Module-Framework dahin gehend geändert werden, dass Module parallel ausgeführt werden können. Durch die parallele Ausführung soll die Ausführungszeit eines Frames verringert werden. Dardurch können zum einen größere Datenmengen, wie zum Beispiel Bilder mit höherer Auflösung, aber auch Algorithmen und Funktionalitäten, die in ihrer Ausführungszeit teuer sind, verwendet werden.

2 Verwandte Arbeiten

Das *RoboFrame*-Framework[10][9], der technischen Universität Darmstadt, ist ein plattformunabhängiges Framework zur Erstellung von Steuerungsprogrammen für mobile autonome Robotersysteme. Die Kontrollsoftware erzeugt eine Anzahl an Threads und verteilt die vorhandenen Module auf diese. Es ist möglich mehrere Module einem Thread zuzuweisen, diese werden dann immer sequentiell ausgeführt. Es wird zwischen zwei verschiedenen Aufrufen unterschieden. Der eine Aufruf geschieht bei einer eingehenden Nachricht vom Router, der andere nach einem zeitlichen Intervall.

Das Ecto-Framework [1] ist ein Framework für die Organisation von Berechnungen auf einem gerichteten, azyklischen Graphen. Dabei werden *Cells* definiert, welche die Logik enthalten. Die *Cells* werden in einem Graphen (*plasm*) zusammengefasst und miteinander verbunden. Mit einem *Scheduler* werden dann die *Cells* sequentiell ausgeführt.
Ecto bietet zusätzlich einen *MultiPlasm Scheduler*. Dieser kann mehrere *plasm* parallel ausführen. Im FUmanoid-Modul-Framework sollen die Module innerhalb eines Graphen parallel ausgeführt werden. Daher ist das Ecto-Framework nicht geeignet.

3 Plattform

Diese Arbeit entstand bei den *FUmanoids*. Das FUmanoid-Team ist ein studentisches Projekt, an der Freien Universität Berlin, innerhalb der Arbeitsgruppe Intelligente Systeme und Robotik.

3.1 Roboter

Die FUmanoids nehmen mit ihren Robotern an den Tunieren des RoboCups in der Humanoid KidSize Liga mit wiederholtem Erfolg teil [1]. In regelmäßigen Abständen werden die Regeln des RoboCups aktualisiert [3] [2]. Daher werden auch die Roboter immer wieder an die neuen Regeln angepasst. Aktuell sind die Roboter ca. 65cm groß und wiegen ca. 4.5kg. Damit sich die Arme, Beine, Kopf und Hüfte bewegen können, sind 20 Motoren verbaut, die diese Bewegungen ausführen.

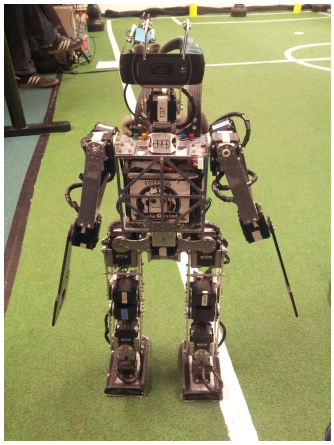


Abbildung 1: FUmanoid Roboter Alan

[1] http://www.fumanoids.de/awards-and-scores/

3.2 BerlinUnited-Framework

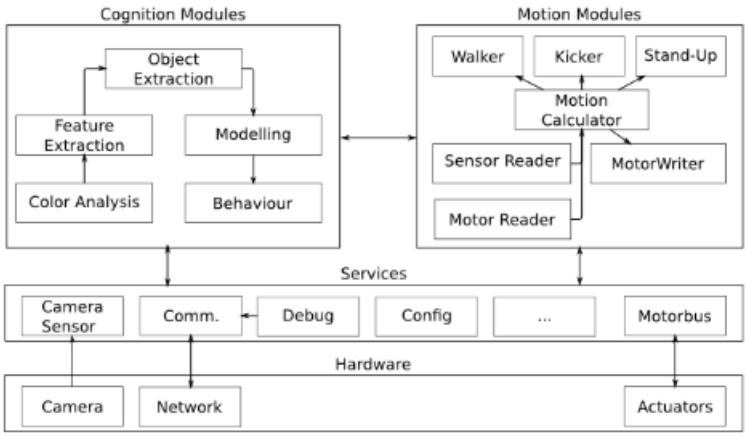


Abbildung 2: Struktur der Kontrollsoftware [12]

Abbildung 2 zeigt ein Blockdiagramm der Software, die auf den Robotern zum Einsatz kommt. Die unterste Ebene bietet Zugang zu verschiedenen Teilen der Roboterhardware. Auf der darüber liegenden Ebene werden zusätzliche Funktionen bereitgestellt, wie zum Beispiel Konfigurationsmanagement, Kommunikationskontrolle und das Behandeln der Debug-Ausgaben. An der Spitze befinden sich zwei Blöcke, welche aus verschiedenen Modulen zusammengesetzt sind. Diese Module werden in einer vordefinierten Reihenfolge ausgeführt. Der *Cognition*-Block wird mit jedem neuen Kamerabild, bis zu 30 mal in der Sekunde, ausgeführt. Die Module des *Motion*-Blocks werden in einem 10ms-Intervall ausgelöst.

3.2.1 Repräsentationen

Repräsentationen stellen im Framework reine Datencontainer dar, welche keine komplexe Funktionalität besitzen.

3.2.2 Module

Module sind die ausführbaren Einheiten im Framework. Sie sind untereinander nicht direkt von einander abhängig, was ein einfaches Hinzufügen von neuen Modulen erlaubt. Module können eine oder mehrere Repräsentationen benötigen (*REQUIRE*). In diesem Fall haben sie nur Lesezugriff auf diese Repräsentationen. Außerdem können Module auch Repräsentationen bereit stellen (*PROVIDE*), dann können sie diese auch verändern.

3.2.3 ModuleManager

Der ModuleManager verwaltet die Module. Beim Starten der Software wird an Hand der *REQUIRE*- und *PROVIDE*-Statements der Module eine Ausführungsreihenfolge bestimmt. Dies geschieht bisher durch eine Art von Bubble-Sort. Die Module werden an Hand ihrer Abhängigkeiten miteinander vertauscht, bis sie in einer korrekten Reihenfolge sind.
Die Module werden dann sequentiell an einen Executor übergeben und ausgeführt. In der FUmanoids-Software gibt es zwei ModuleManager, die nebeneinander laufen. Der eine ModuleManager ist für die Cognition, der andere für die Motion zuständig.

4 Grundlagen

4.1 Graphen

Ein Graph G ist ein geordnetes Paar (V, E), wobei V eine Menge von Knoten (engl. *vertex*) und E eine Menge von Kanten (engl. *edge*) ist [14].

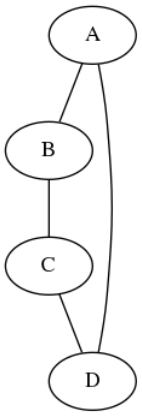


Abbildung 3: Ein ungerichteter zyklischer Graph

Mit Hilfe von Graphen lassen sich Verbindungen zwischen Objekten repräsentieren. Beispiele für einen Graphen sind zum einen Stammbäume, bei denen jeder Knoten ein Familienmitglied und jede Kante eine Verbindung zwischen Elternteil und Kind darstellt. Zum anderen sind auch U-Bahnnetze anschauliche Beispiele für Graphen. Dort repräsentieren die einzelnen Stationen die Knoten. Eine Kante ist dabei eine direkte Zugverbindung zwischen zwei Stationen.
Bei Straßennetzen mit Einbahnstraßen können den Kanten noch eine Richtung zu geordnet werden, hier spricht man von einem gerichteten Graphen.

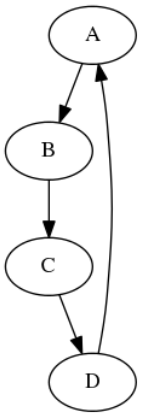


Abbildung 4: Ein gerichteter zyklischer Graph

Eine sich abwechselnde Folge aus Knoten in einem Graphen wird *Weg* genannt.

$$v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_n \quad | \quad v_i \in V$$

Dabei wird v_0 als Startknoten und v_n als Endknoten bezeichnet. Sollten Startknoten und Endknoten in einem Weg identisch sein, so spricht man von einem *Kreis* oder *Zyklus*[5].

Eine Eigenschaft eines Knotens ist der Grad des Knotens. Der Grad $d_G(v)$ ist die Anzahl der Kanten, die den Knoten v mit einem anderen Knoten verbindeen. In einem gerichteten Graphen wird zusätzlich noch zwischen Eingangs- und Ausgangsgrad unterschieden. Der Eingangsgrad ($d_G^+(v)$) beschreibt die Anzahl der direkten Vorgänger eines Knoten v. Der Ausgangsgrad ($d_G^-(v)$) beschreibt die Anzahl der direkten Nachfolger des Knotens v.
Einen Graphen, bei denen die Kanten mit Gewichten ausgestattet sind, nennt man einen kantengewichteten Graphen.

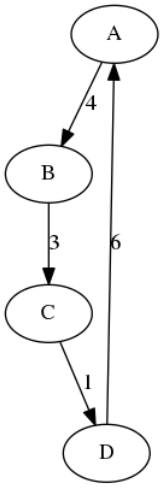


Abbildung 5: Gerichteter, kantengewichteter Graph

4.2 Tiefensuche

Die Tiefensuche ist ein Verfahren, welches durch Expansion des jeweils ersten auftretenden Nachfolgeknoten in einem Graphen nach und nach vom Startknoten aus weiter in die Tiefe sucht [15] [11].

```
DFS(G)
  for alle Knoten  u ∈ V[G]
    do farbe[u] ← WEISS
```

```
      π[u] ← NIL
  zeit ← 0
  for alle Knoten u ∈ V[G]
    do if farbe[u] = WEISS
      then DFS–VISIT(u)

DFS–VISIT(u)
  farbe[u] ← GRAU
  zeit ← zeit + 1
  d[u] ← zeit
  for alle v ∈ Adj[u]
    do if farbe[v] = WEISS
      then π[v] ← u
        DFS–VISIT(v)
  farbe[u] ← SCHWARZ
  f[u] ← zeit ← zeit + 1
```

Listing 1: Pseudocode Tiefensuche [4]

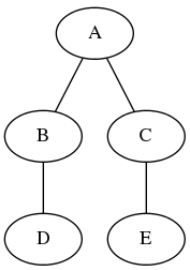


Abbildung 6: Ausgangsgraph für die Tiefensuche

In Abbildung 6 ist ein einfacher Graph dargestellt. In Abbildung 7 ist zu sehen, wie die Tiefensuche vom Startknoten A vonstatten geht.

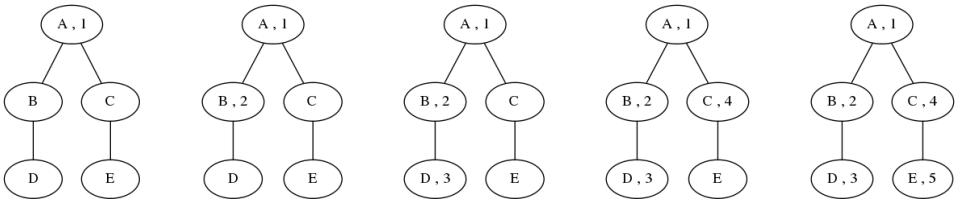
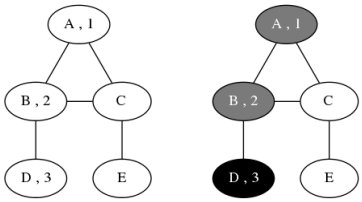


Abbildung 7: Ablauf der Tiefensuche

Mit Hilfe der Tiefensuche kann man einen Graphen auf Kreise testen. Bei der Tiefensuche werden den Knoten verschieden Farben zugeordnet. Dabei

steht die Farbe Weiß für einen unendeckten Knoten, die Farbe Grau für einen entdeckten, aber noch nicht abgeschlossenen Knoten und die Farbe Schwarz für einen abgeschlossenen Knoten.



(a) (b)

Abbildung 8

In Abbildung 8(a) wurde zwischen den Knoten B und C eine Kante eingefügt. Somit enthält dieser Graph einen Kreis. Zu diesem Zeitpunkt wäre der Knoten D bereits abgeschlossen (schwarz, Abbildung 8(b)). Knoten A und B haben noch ein Kindknoten (C), sie wären also noch grau. Nun wird der Knoten C betrachtet und die Nachbarknoten (A,B,E). Die Knoten A und B wurden bereits entdeckt, aber noch nicht fertig bearbeitet, daher sind die Kanten zwischen C und A, sowie C und B Kanten, welche einen Kreis schließen. Diese Kanten heißen Rückwärtskanten (engl. *back-edge*).

4.3 Topologische Sortierung

Eine Topologische Sortierung ist eine Reihenfolge, in der vorgegebene Abhängigkeiten, wie zum Beispiel A muss vor B ausgeführt werden, berücksichtigt werden. Um eine Topologische Sortierung erfolgreich auf einem Graphen ausführen zu können, muss der Graph ein gerichteter, zyklenfreier Graph sein.

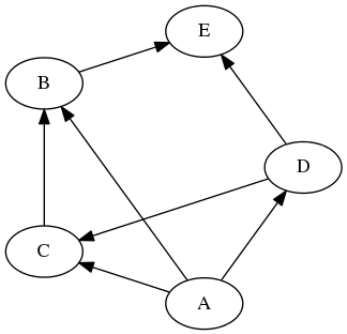


Abbildung 9: Gerichteter, zyklenfreier Graph

```
Funktion TopSort
  while V ist nicht leer do
    Zyklus:=true
    for each v in V do
      if es gibt keine Kante e in E der Form (X,v) then
        // X ist hierbei ein beliebiger anderer Knoten
        lösche v aus V
        lösche alle Kanten der Form (v,X) aus E
        Zyklus:=false
        print v // Ausgabe des Knotens
      endif
    endfor
    if Zyklus=true then
      print "Zyklische Abhängigkeit kann nicht
            aufgelöst werden!"
      break // Abbrechen der while-Schleife
    endif
  endwhile
end
```

Listing 2: Pseudocode Topologische Sortierung [11]

Der Algorithmus sucht sich immer einen Knoten heraus, welcher den Eingangsgrad 0 besitzt (siehe Abbildung 10).

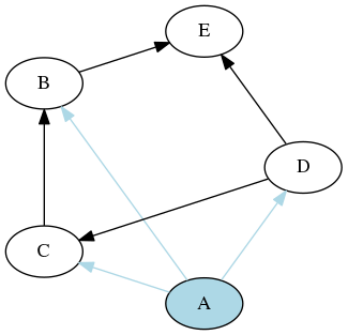
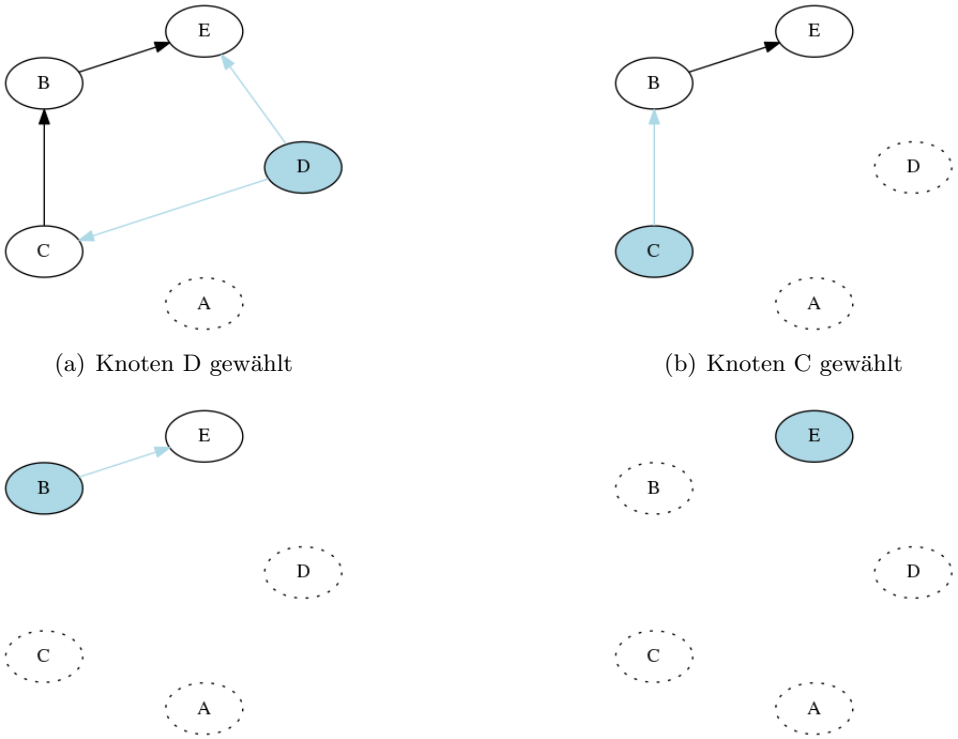


Abbildung 10

In Abbildung 10 sieht man, dass der Knoten A ausgewählt wurde. Die Knoten und die dazugehörigen Kanten wurden blau eingefärbt. Nun werden der Knoten und die dazugehörigen Kanten entfernt und der nächste Knoten mit Eingangsgrad 0 wird ausgewählt.



(a) Knoten D gewählt

(b) Knoten C gewählt

(c) Knoten B gewählt

(d) Knoten E gewählt

Abbildung 11: Ablauf der Topologischen Sortierung

Um nun eine Reihenfolge zu bestimmen, werden die Knoten in der Reihenfolge ihrer Entfernung gespeichert. Die hierbei entstandene Reihenfolge ist am Ende auch die Reihenfolge, in der die Knoten aus dem Graphen entfernt wurden.

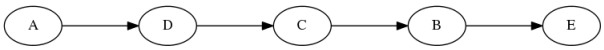


Abbildung 12: Ergebnis der Topologischen Sortierung

4.4 Dijkstra-Algorithmus

Der Dijkstra-Algorithmus löst das Problem der kürzesten Pfade für einen angegebenen Startknoten zu einem, bzw. allen anderen Knoten in einem kantengewichteten Graphen. Die Grundidee des Algorithmus ist es, immer die Kante zu verfolgen, welche den kürzesten Streckenabschnitt verspricht. Allen Knoten werden die Eigenschaften *Distanz* und *Vorgänger* zugewiesen. Hierbei wird dem Startknoten die Distanz 0 und den restlichen Knoten die Distanz mit ∞ initialisiert. Dabei bezieht sich die Distanz immer auf die Distanz zum Startknoten.

```
S = Startknoten
λ(u,v) = Gewicht der Kante (u,v)
d[v] = Distanz von Knoten v zu Startknoten S
vorgänger[v] = Vorgängerknoten von v auf dem kürzesten
               Weg von S nach v

Initialisiere d[S] = 0
Für alle anderen Knoten v d[v] = ∞

WHILE es gibt einen unmarkierten Knoten DO
  wähle einen unmarkierten Knoten v mit kleinstem d[v];
  markiere v;
  FOR alle Kanten (v,w) mit unmarkiertem w DO
    IF d[v] + λ(v,w) < d[w] THEN
      d[w] = d[v] + λ(v,w);
      vorgänger[w] = v;
    END IF
  END FOR
END WHILE
```

Listing 3: Pseudocode Dijkstra-Algorithmus [6]

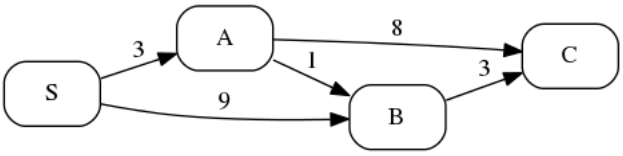


Abbildung 13: Gerichteter, kantengewichteter Graph

In Abbildung 13 ist ein einfacher, gerichteter Graph mit Kantengewichten dargestellt. Im ersten Schritt werden die Distanzen aller Knoten zum Startknoten initialisiert. Die Distanz vom Startknoten S zu sich selbst ist 0. Die restlichen Knoten werden mit ∞ initialisiert. Desweiteren werden die Vorgänger der Knoten mit *NIL* initialisiert.

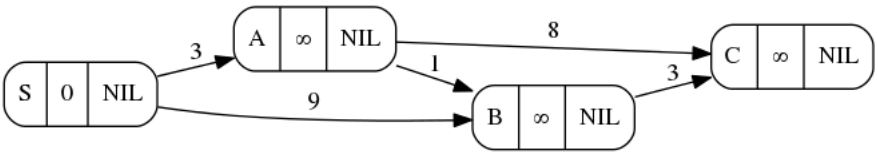


Abbildung 14: Ausgangsgraph für den Dijkstra-Algorithmus mit Initialisierung

In der Hauptschleife werden als Erstes der Startknoten S und die Kanten zu seinen Nachbarknoten ((S,A) , (S,B)) betrachtet.

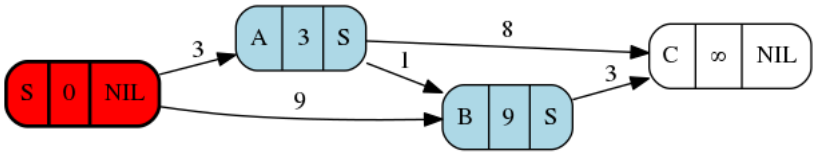


Abbildung 15

Der Startknoten ist somit abgearbeitet und wird makiert (fett umrandet). Da der Knoten A die kürzeste Distanz zum Startknoten besitzt und noch nicht als abgeschlossen markiert wurde, ist dieser der Knoten, der als nächstes betrachtet wird. Es werden also nun alle Kanten von A ausgehend betrachtet ((A,B), (A,C)).

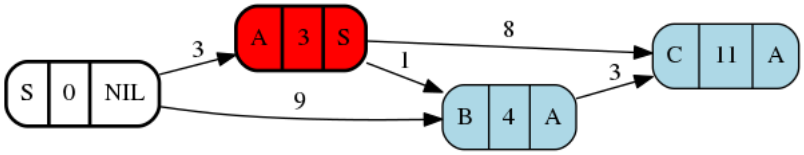
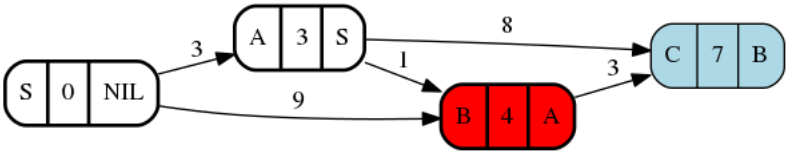
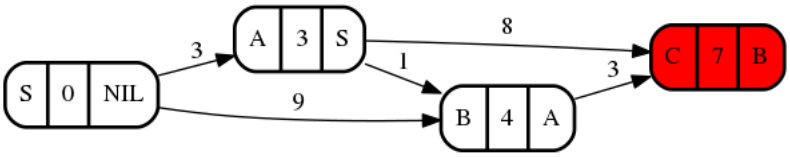


Abbildung 16

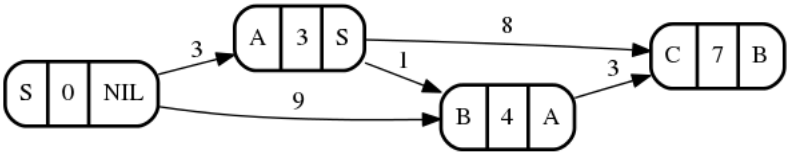
In Abbildung 16 ist zu sehen, dass die Distanz von B zum Startknoten S aktualisiert wurde, da der Weg über A kürzer ist, als auf direktem Weg.
Dies wird nun solange wiederholt, bis es keine unmakierten Knoten mehr gibt.



(a)



(b)



(c)

Jedes Mal wenn eine Distanz geändert wird, muss auch dementsprechend der Vorgänger aktualisiert werden. So erhält man am Ende des Algorithmus für jeden Knoten den Vorgänger auf dem kürzesten Pfad von S. Zum Beispiel wäre der kürzeste Weg von S zu C:
$S \to A \to B \to C$

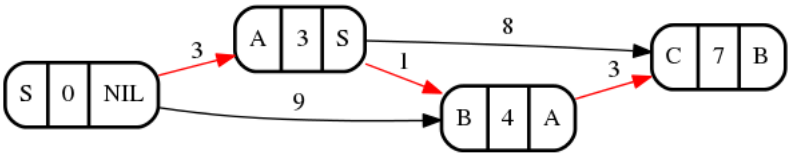


Abbildung 17

4.5 Ausführungsstrategien

First-Come-First-Served[13][8] (*FCFS*) bezeichnet ein Verfahren der Speicherung, bei denen die Elemente, welche zuerst gespeichert werden, auch zuerst wieder aus dem Speicher entfernt werden. Die Datenstruktur der Standard-*Queue*, arbeitet mit diesem Verfahren. Das Element, welches als Erstes der Queue hinzugefügt wird, wird auch als Erstes wieder aus der Queue entfernt.

Bei dem Verfahren Longest-Job-First (*LJF* [7]), werden Elementen eine Laufzeit zugeordnet. Beim LJF wird nun nicht mehr die Standard-*Queue*

verwendet, sondern eine *Priority-Queue*. In einer Priority-Queue werden den Elementen eine Priorität zugeordnet und an Hand dieser sortiert. Dabei können diese Prioritäten aufsteigend oder absteigend sortiert sein. Beim LJF entspricht die Priorität die Laufzeit des Elementes. Wenn mit LJF ein Element in die Priority-Queue eingefügt wird, wird das neue Element in Abhängigkeit seiner Laufzeit einsortiert. So werden die Elemente, welche die höchste Laufzeit besitzen, als Erstes wieder aus der Priority-Queue entfernt.

5 Implementierung

5.1 Abhängigkeitserkennung

Die auf den Robotern laufenden Software-Module (siehe Abschnitt 3.2.2) müssen beim Starten in eine Reihenfolge gebracht werden, die alle Abhängigkeitsangaben erfüllt. So kann ein Modul A, welches Daten aus Repräsentation R benötigt, erst dann ausgeführt werden, wenn ein anderes Modul die Daten in R bereitstellt. An Hand dieser Informationen von *REQUIRE* und *PROVIDE* kann eine Reihenfolge bestimmt werden, die diese Abhängigkeiten berücksichtigt.
Bisher werden die Module mit einer Art Bubble-Sort solange miteinander vertauscht, bis alle Abhängigkeiten erfüllt sind. Sollte es unter den Modulen zyklische Abhängigkeiten geben, werden Module ständig hin und her getauscht. Dies führt zu einer Endlosschleife. Um dem entgegen zu wirken, wurde ein Maximum an Tauschoperationen festgelegt. Wenn das Maximum erreicht wurde, wird eine Meldung ausgegeben. Die berechnete Reihenfolge ist dann fehlerhaft.

Durch die gegebenen Informationen von *REQUIRE* und *PROVIDE* lässt sich ein Graph modellieren, welcher die Abhängigkeiten zwischen den Modulen darstellt. Dabei bilden die Module die Knoten. Es wird eine Kante von einem Modul A zu einem anderen Modul B gezogen, wenn Modul A eine oder mehrere Repräsentationen bereitstellt, welche Modul B benötigt.

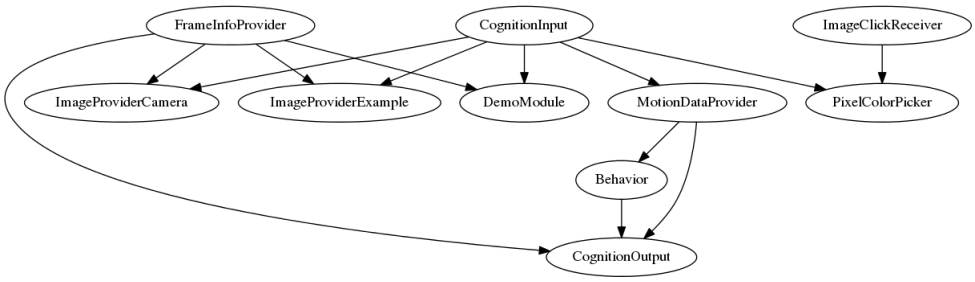


Abbildung 18: Beispielgraph für Cognition (siehe 3.2)

Abbildung 18 stellt einen solchen Graphen dar. Um diesen Graphen aufbauen zu können, werden alle aktivierten Module betrachtet. Für jedes Modul werden die *PROVIDE*-Repräsentationen betrachtet. Für jede gefundene Repräsentation wird in einer *HashMap* ein neuer Eintrag angelegt. Dabei ist der *key* des Eintrages die Repräsentation selbst. Als *value* wird eine Liste angelegt. In dieser Liste werden alle Module gespeichert, die diese Repräsentation bereitstellen.
Im nächsten Schritt geht es darum, die Kanten zwischen den Modulen zu finden. Dazu werden erneut die Repräsentationen aller aktivierten Modu-

le betrachtet. Das Hauptaugenmerk liegt hierbei nun auf den *REQUIRE*-Repräsentationen. Für jede dieser gefundenen Repräsentationen wird in der *HashMap* die Liste der Module gesucht. Zwischen dem aktuell betrachteten Modul und den Modulen aus der HashMap wird eine Kante gezogen. Dabei sind die Module aus der HashMap die Startknoten der Kante und das aktuell betrachtete Modul der Endknoten der Kante. Die Kanten werden in einer Liste gespeichert.
Für eine Berechnung der Ausführungsreihenfolge muss der Graph kreisfrei sein. Mit Hilfe der Tiefensuche (siehe 4.2) lässt sich ein Graph auf Kreisfreiheit überprüfen.

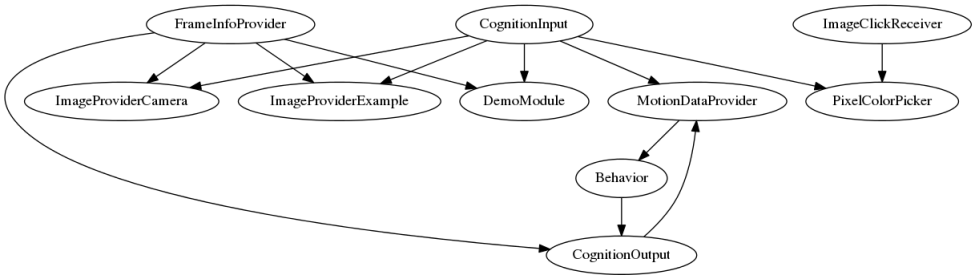


Abbildung 19: Einfacher Graph mit Kreis

In Abbildung 19 wurde in den Graph von Abbildung 18 eine Kante invertiert. Die Kante von *MotionDataProvider* zu *CognitionOutput* wird zu einer Kante von *CognitionOutput* zu *MotionDataProvider*. Nun bilden die Module *MotionenDataProvider*, *Behavior* und *CognitionOutput* einen Kreis. Die Bestimmung einer Ausführungsreihenfolge ist nun nicht mehr möglich. Damit nachvollzogen werden kann, welche Module diesen Kreis bilden, ist es sinnvoll sich den Kreis ausgeben zu lassen.
Die Tiefensuche erkennt die Rückwärtskante (siehe 4.2), die den Kreis schließt. Dies könnte die Kante von *CognitionOutput* zu *MotionDataProvider* sein. Mit Hilfe des Dijkstra-Algorithmus (siehe 4.4) kann der kürzeste Weg zwischen zwei Knoten berechnet werden. Wenn davon ausgegangen wird, dass die Rückwärtskante der Tiefensuche die Kante (*CognitionOutput*, *MotionDataProvider*) ist, dann wird der kürzeste Weg zwischen *MotionDataProvider* und *CognitionOutput* berechnet. Beim Dijkstra-Algorithmus wird immer der Vorgänger eines Knoten gespeichert. Mit Hilfe dieser Informationen ist es möglich vom Zielknoten (*CognitionOutput*) den Weg rückwärts zum Startknoten (*MotionDataProvider*) zu gehen. Die besuchten Knoten (Module) bilden dann den kürzesten Weg und somit den Kreis. Sollte der Graph nicht zyklenfrei sein, dann kann keine Ausführungsreihenfolge bestimmt werden. Durch die Ausgabe der Module, welche diesen Kreis bilden, lässt sich leichter der Einstiegspunkt der Fehlersuche finden.
Daher wird ein Fehler ausgegeben, welcher den Kreis der Module beinhaltet

und das Programm wird beendet.

5.2 Sequentielle Reihenfolge und Ausführung

Sollte der aufgebaute Graph keinen Kreis enthalten, ist es möglich eine Ausführungsreihenfolge zu bestimmen. Mit Hilfe der Topologischen Sortierung (siehe 4.3) ist es möglich eine Reihenfolge zu bestimmen, welche die Abhängigkeiten unter den Modulen berücksichtigt.

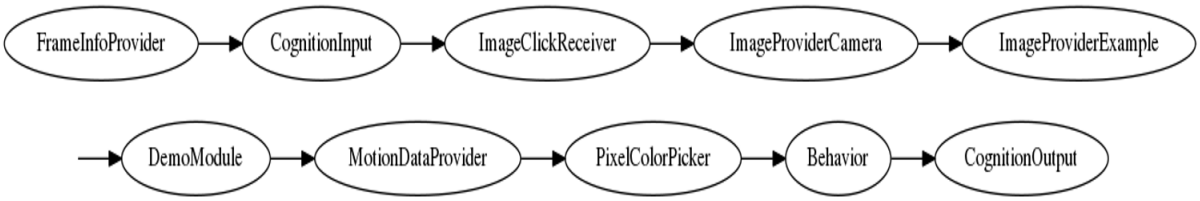


Abbildung 20: Mögliche Ausführungsreihenfolge für den Graph auf Abbildung 18

Sollte es Module in dem Graphen geben, welche voneinander unabhängig sind, dann existieren verschiedene Lösungen. Die Topologische Sortierung liefert eine mögliche Sortierung. Die in Abbildung 20 dargestellte Modulabfolge, ist eine solche Reihenfolge. Der ModuleManger (siehe 3.2.3) besitzt eine Liste von Modulen (*executionList*). Dabei ist die Reihenfolge in der die Module in dieser Liste stehen auch die Reihenfolge in der die Module sequentiell abgearbeitet werden. In der Methode *calculateExecutionList*, des ModuleManagers, wird der Graph aufgebaut und dann mit der Topologischen Sortierung eine Reihenfolge bestimmt. Die *executionList* wird dann mit dem Ergebnis der Topologischen Sortierung überschrieben.
Alle Module, die sich in der *executionList* befinden, werden in jedem Durchlauf ausgeführt. Dazu wird die Liste sequentiell durchgegangen und jedes Modul wird an einen *AsyncModuleExecutor* übergeben. Dieser startet die Ausführung des Moduls. Der ModuleManger wird erst mit dem nächsten Modul beginnen, wenn das aktuelle Modul mit seiner Ausführung abgeschlossen hat.

5.3 Parallele Ausführung

In früheren Robotermodellen wurden nur 1-Kern-Prozessoren verwendet. Daher wurde die sequentielle Ausführung bevorzugt. Eine parallele Ausführung brachte auf einem 1-Kern-Prozessor keinen Vorteil. Mit steigender Komplexität der benutzten Algorithmen, wurden die Roboter dann mit neuen Prozessoren ausgestattet. Diese besitzen einen 4-Kern-Prozessor. Bei der sequentiellen Ausführung werden diese vier Kerne jedoch nicht vollständig

ausgenutzt. Da nur ein Modul zur gleichen Zeit läuft, bleiben andere Kerne unbeschäftigt. Um die Hardware besser ausnutzen zu können, wird die sequentielle Ausführung durch eine parallele Ausführung ersetzt. Dies ist möglich, da nicht alle Module voneinander abhängig sind. So können mehrere Module, dessen *REQUIRE*-Daten zur Verfügung stehen, parallel ausgeführt werden.

Um eine parallele Ausführung zu realisieren, werden weitere Informationen zu einem Modul benötigt. Zum Einen muss das Modul wissen, wann seine *REQUIRE*-Daten zur Verfügung stehen. Dies wird erreicht, indem man den aufgestellten Abhängigkeitsgraphen betrachtet. Der Eingangsgrad des Knotens (Modul) beschreibt, wie viele Module ausgeführt werden müssen, um alle benötigten Daten zur Verfügung zu haben (siehe 4.1). Desweiteren wird gespeichert, für welche anderen Module Daten bereitgestellt werden.
Damit dem eigentlichen Modul die zusätzlichen Informationen zugeordnet werden können, wurde ein Container geschaffen, der diese Zusatzinformationen speichert (*MetaModule*).

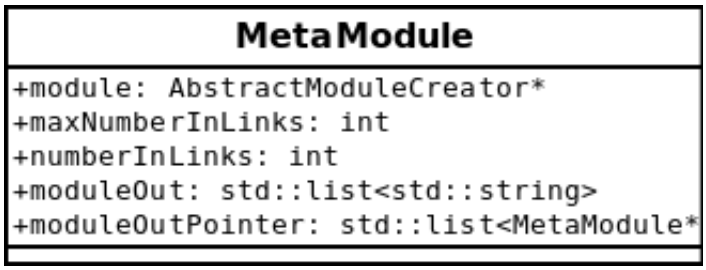


Abbildung 21: URL-Diagramm des MetaModules

Durch die Einführung des *MetaModule* fällt die direkte Berechnung der Ausführungsreihenfolge weg. Je nachdem zu welcher Zeit ein Modul fertig wird, kann sich die Ausführungsreihenfolge bei jedem Durchlauf ändern. Das Aufstellen des Graphen wird denoch benötigt, um sicher zustellen, dass der Graph auch kreisfrei ist. Die *MetaModule* werden während des Erzeugens des Graphen ebenfalls erstellt. Die *modulesOut*-Liste wird an Hand der Kantenliste erstellt. Mit Hilfe der *Repräsenationsmap* (5.1) lässt sich der Eingangsgrad (*maxNumberInLinks*) der Module bestimmen.
Eine weitere Information ist der aktuelle Eingangsgrad des Modules (*numberInLinks*). Dieser wird bei der Ausführung von Modulen nach Beendigung über die *modulesOut* dekrementiert. Um dies schnell zu bewerkstelligen wurde zusätzlich noch eine Pointerliste hinzugefügt (*modulesOutPointer*).
Da nun mehrere Module zur gleichen Zeit ausgeführt werden sollen, muss auch der ModuleManager die Möglichkeit besitzen, mehrere Module ausführen zu lassen. Zu diesem Zweck besitzt er nun nicht mehr nur einen Modul-

Executor sondern mehrere. Da im FUmanoid-Projekt die *Cognition* sehr zeitaufwendige und komplexe Module beinhaltet, wurden ihr drei Executoren zugeteilt. In der *Motion* befinden sich Module, die sich hauptsächlich nicht parallelisieren lassen, daher wurde der *Motion* nur ein Executor zugeordnet. Desweiteren besitzt nun der ModuleManager einen Zähler, welcher die abgeschlossenen Module mitzählt (*countFinishModules*).
Auch der *AsyncModuleExecutor* wurde angepasst. Dieser besitzt nun ein Flag, der angibt, ob der Executor mit der Ausführung seines Moduls fertig ist. Es wurde zusätzlich ein Event hinzugefügt, welches ausgelöst wird, wenn ein Modul beendet wurde. Dieses Event teilen sich alle *AsyncModuleExecutor*, damit der ModulManager darauf warten kann. Sollten alle Executoren ein Modul bearbeiten, wird der ModuleManager auf das Auslösen des Events warten, bevor er überprüft welcher der Executoren fertig geworden ist.

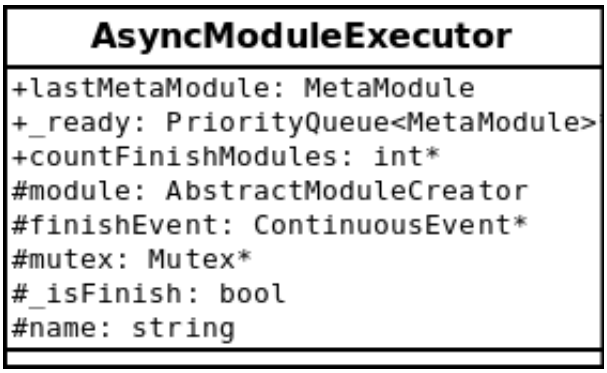


Abbildung 22: URL-Diagramm des AsyncModuleExecutor

In jedem Durchlauf wird geprüft, welche *MetaModule* einen Eingangsgrad von 0 besitzen. Bei diesen Modulen ist sicher gestellt, dass alle Daten, die benötigt werden, vorhanden sind. Diese Module werden in einer Queue (*readyQueue*) gespeichert und sind bereit, ausgeführt zu werden. Zu beachten ist hierbei noch, dass zu Beginn jedes Durchlaufs die *numberInLinks* auf ihren eigentlichen Maximalwert (*maxNumberInLinks*) gesetzt werden. Dies ist auf Grund der Verwendung der *modulesOutPointer* erforderlich, da sonst die *numberInLinks* im nächsten Durchlauf noch auf Null gesetzt sind.
In einer while-Schleife, welche solange läuft, bis die Anzahl der beendeten Module mit der Anzahl der *MetaModule* übereinstimmt, werden nach dem FCFS-Prinzip (siehe 4.5) die zur Ausführung bereitstehenden Module an einen *AsyncModuleExecutor* übergeben. Es werden außerdem die aktuelle *readyQueue* und die aktuellen Anzahl der *countFinishModules* übergeben. Nach dem Beenden des aktuellen Modules, ist das entsprechende *MetaModule* im Executor unter dem *lastMetaModule* gespeichert. Durch die *modulesOutPointer*-Liste innerhalb des *MetaModule*, werden die *numberInLinks* der Nachfolgemodule herunter gezählt. Jedes Modul, welches nach

dem Dekrementieren einen *numberInLinks* von Null aufweist, wird in die *readyQueue* verschoben.

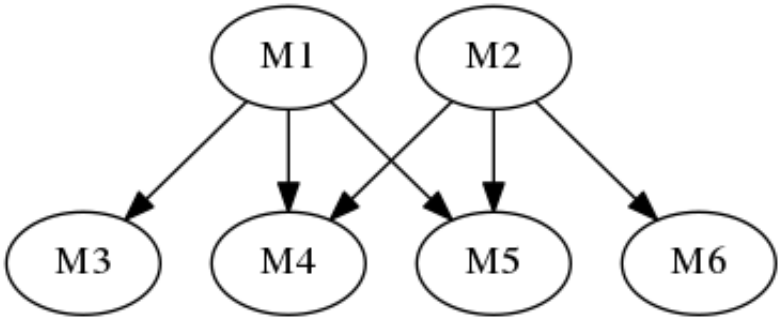


Abbildung 23: Gerichteter Abhängigkeitsgraph

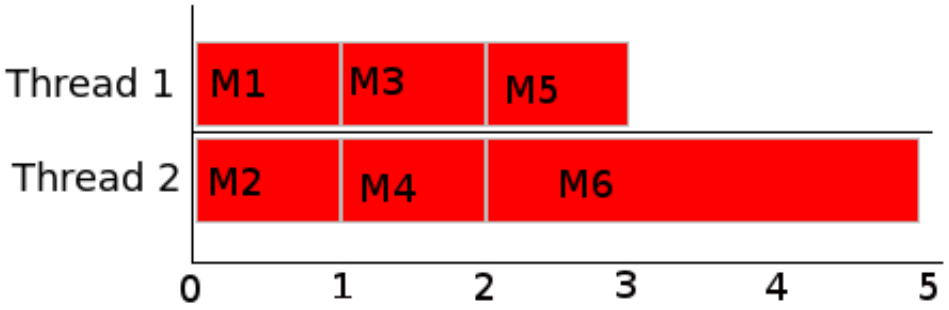


Abbildung 24: Modulausführung mit FCFS

Für den Graphen in Abbildung 23 ergibt sich nach dem FCFS-Prinzip die Modulausführungsreihenfolge, mit zwei Threads, aus Abbildung 24. In der Reihenfolge, wie die Module in die Warteschlange kommen um ausgeführt zu werden, in dieser werden sie auch ausgeführt.

5.3.1 LJF-Modulausführung

Die in Abbildung 24 Modulausführung ist nicht optimal. Wenn das Modul M6 früher ausgeführt wird, können die Module M3, M4 und M5 zur gleichen Zeit ausgeführt werden wie das Modul M6.

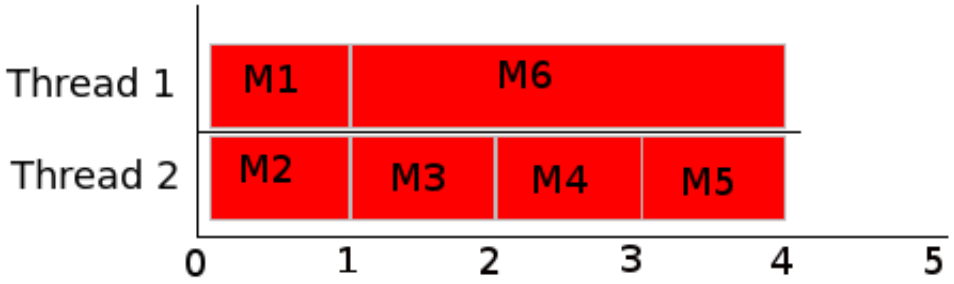


Abbildung 25: Modulausführung mit LJF

Abbildung 25 zeigt eine solche Ausführung. Dabei wurde anstatt des FCFS-Prinzips das Longest-Job-First-Prinzip verwendet (siehe 4.5). Vorraussetzung für dieses Prinzip ist es, dass die Laufzeiten der Module bekannt sind. Durch die Verwendung von LJF kann die Gesamtausführungszeit von 5 Zeiteinheiten auf 4 Zeiteinheiten reduziert werden.

Es ist also besser, zuerst das Modul zu wählen, welches die längste Laufzeit besitzt. Aus diesem Grund wurde das *Longest-Job-First*-Prinzip zur Verringerung der Laufzeit in die parallele Ausführung übernommen.
Die Module werden nun an Hand ihrer Durchschnittslaufzeiten sortiert. Die verwendete Standard-Queue ist nun nicht mehr geeignet, da diese keine Sortierung unterstützt. Deshalb wurde die Standard-Queue durch eine *Priority-Queue* ersetzt. Diese kann die enthaltenen Elemente nach einer Eigenschaft sortieren.
Die verwendete Eigenschaft, um Module in der Priority-Queue zu sortieren, ist die durchschnittliche Laufzeit der Module. Bei jeder Ausführung wird die benötigte Zeit der Ausführung gestoppt. Die Laufzeiten werden in dem *MetaModule* gespeichert. Hierzu wurde das *MetaModule* erweitert, damit diese Zusatzinformationen gespeichert werden können. Es werden die letzten 50 Ausführungszeiten des Modules betrachtet. Abbildung 26 zeigt das veränderte MetaModule.

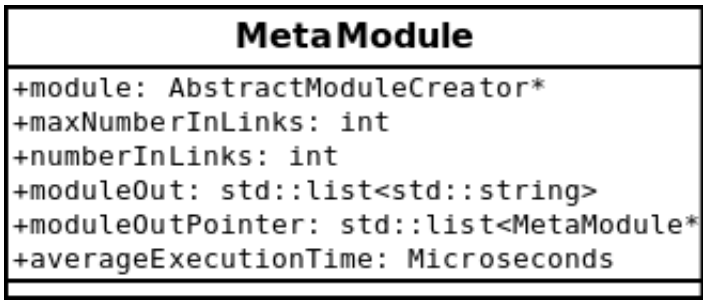


Abbildung 26: URL-Diagramm des MetaModules mit Speicherung der Durchschnittslaufzeit

An Hand dieser Zeiten kann eine Durchschnittslaufzeit berechnet werden. Mit Hilfe dieser Durschnittslaufzeiten werden die Module in die Priority-Queue sortiert.

6 Evaluation

6.1 Ausführungsreihenfolge

Um herauszufinden, wie schnell die neue Berechnung der Ausführungsreihenfolge ist, wurde zuerst die alte Version getestet. Dabei wurde das Berechnen der Reihenfolge 100000 mal wiederholt. Die zu sortierende Liste wurde vor jedem neuen Durchlauf zufällig vertauscht. In Tabelle 1 ist zu sehen, dass es bei der Cognition sowie in der Motion einen erheblichen Zeitgewinn gibt. In der Cognition sind das ca. 83% und in der Motion ca. 59%.

Berechnung ohne Graph		Berechnung mit Graphen	
Cognition	Motion	Cognition	Motion
10.247224	0.908922	1.68265	0.375943

Tabelle 1: Vergleich der Ausführungsdauer

6.2 Parallele Ausführung

Nachfolgend werden die verschiedenen Software-Versionen verglichen. Die Software wird mit einem Log-File abgespielt, damit alle Versionen die gleichen Input-Daten bekommen. Die abgebileten Grafiken zeigen einen einzelnen Frame.

6.2.1 Sequentielle Ausführung

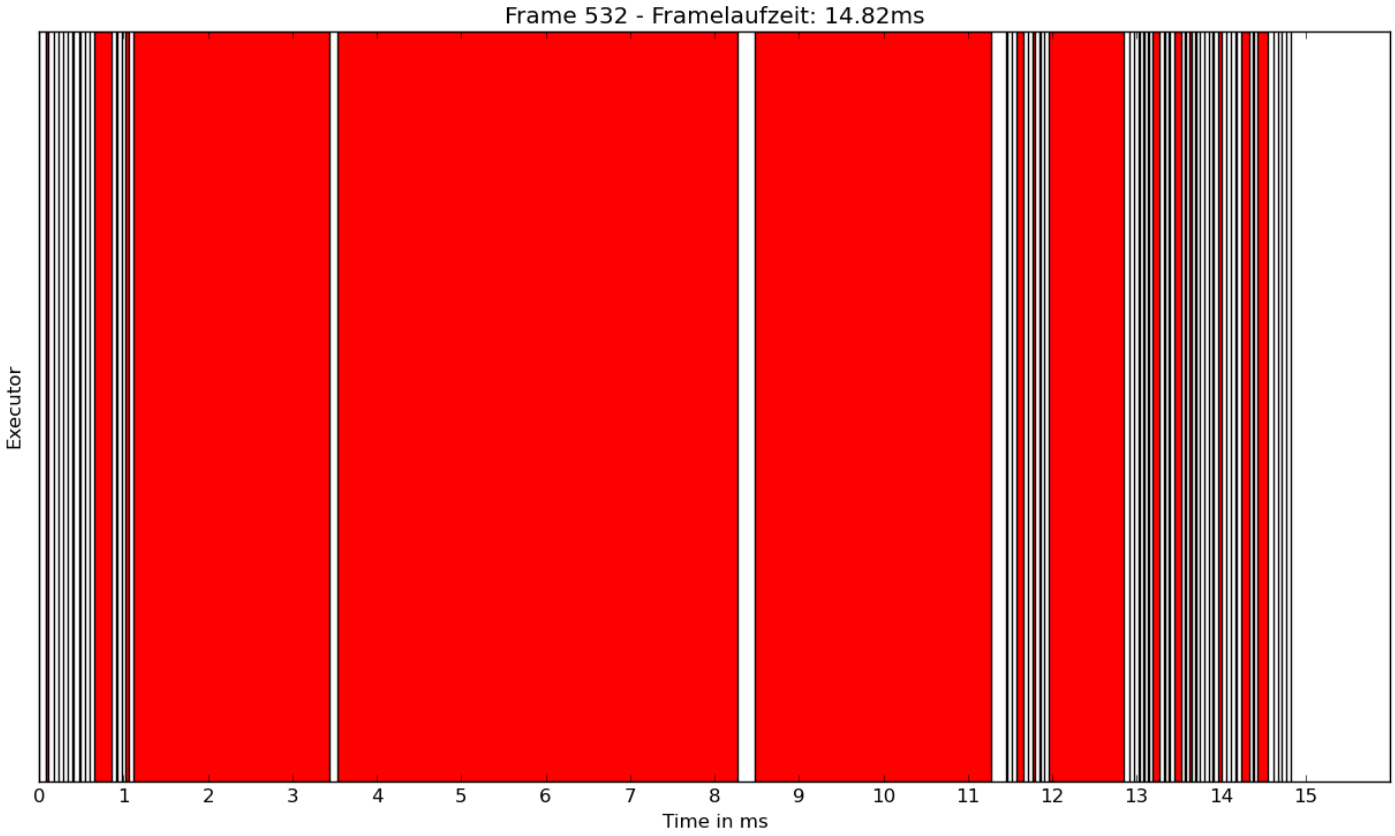


Abbildung 27: Darstellung der Ausführungszeiten der sequentiellen Ausführung

6.2.2 Parallele Ausführung

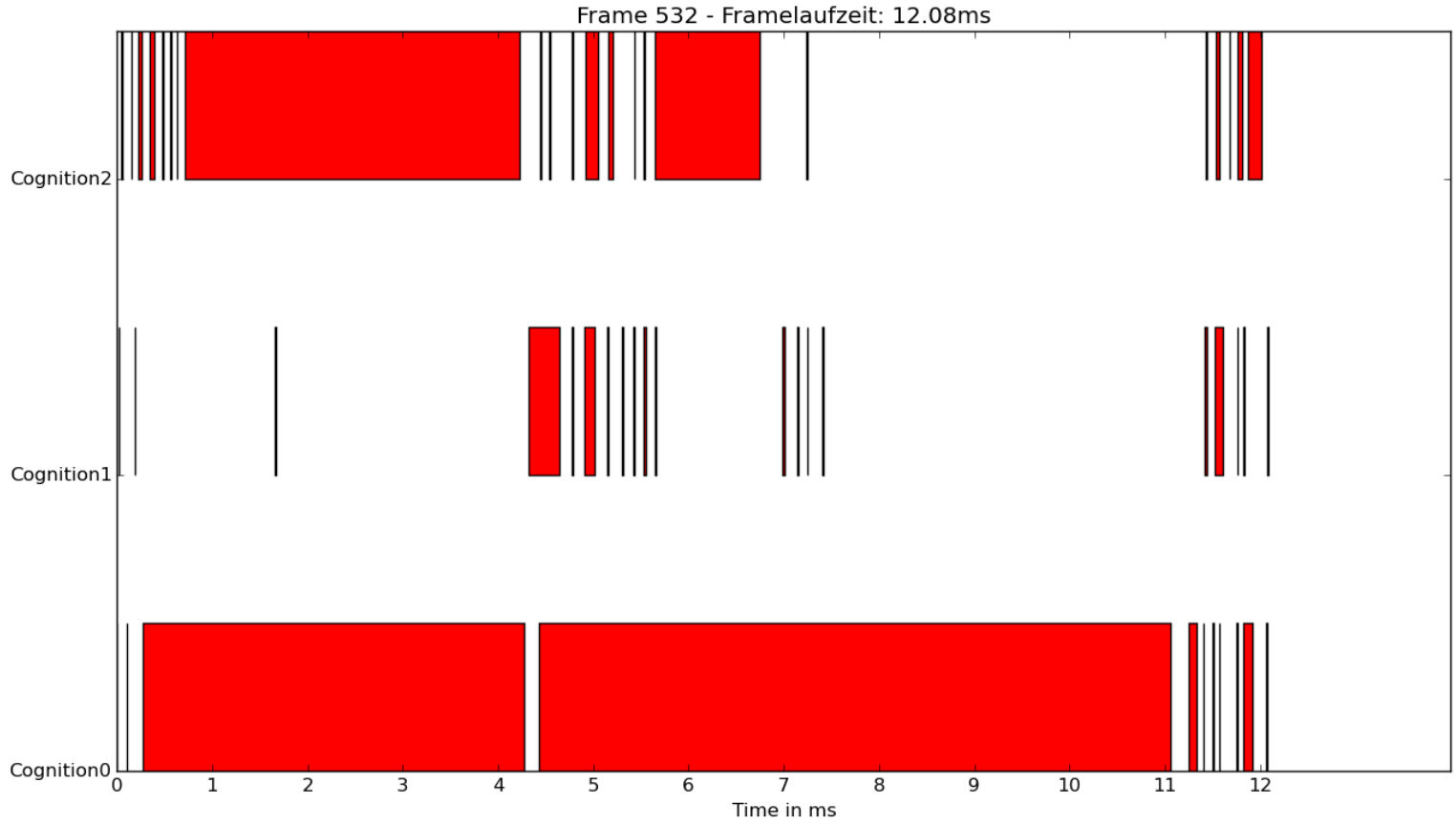


Abbildung 28: Darstellung der Ausführungszeiten der parallele Ausführung mit FCFS

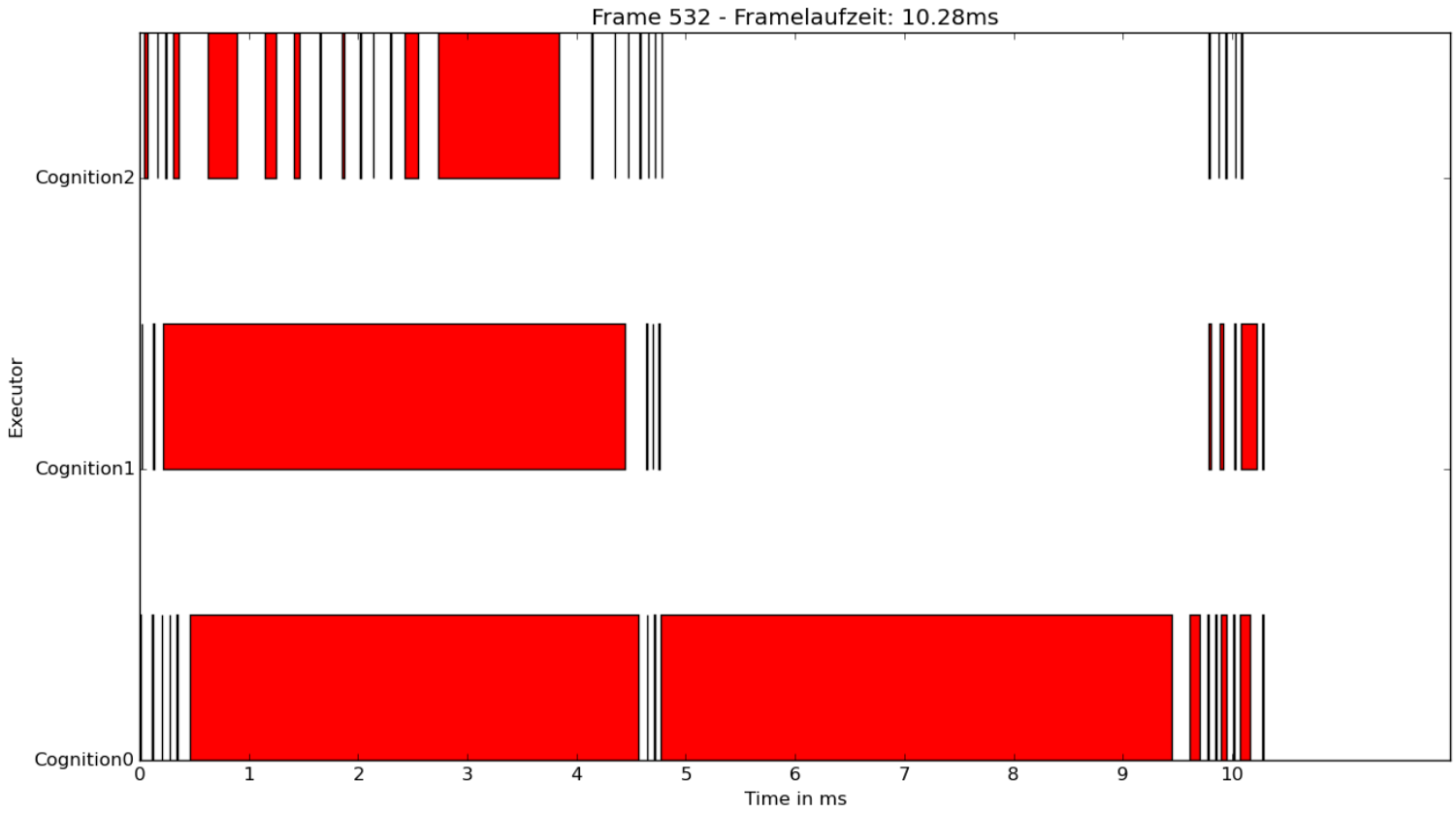


Abbildung 29: Darstellung der Ausführungszeiten der parallele Ausführung mit LJF

Die Abbildung 28, sowie die Abbildung 29, zeigen deutlich die parallele Ausführung von Modulen. Die bei der parallelen Ausführung zu sehenden Lücken in der Ausführung, entstehen durch das Warten auf Module. Um weitere Module ausführen zu können, müssen erst die Daten bereitstehen. Bei langen Modulen, welche Daten für andere Modulen bereitstellen, ist dies sehr gut zu erkennen.

Um alle drei Versionen miteinander vergleichen zu können, wurden vier verschiedene Log-Files verwendet. Alle vier Logs wurden bei den Iran Open 2015 aufgenommen.

- Log-File 1: Nach Ball suchen
- Log-File 2: Nach näherem Ball suchen

- Log-File 3: Ball Richtung Tor bewegen
- Log-File 4: Zum Ball laufen

Die Log-Daten wurden auf dem Roboter abgespielt und die einzelnen Frame-Ausführungszeiten gespeichert. Mit Hilfe der gespeicherten Frame-Ausführungszeiten wurden die Durchschnittslaufzeiten der einzelnen Log-Files berechnet.

Log-Files	Sequentielle Ausführung	Parallele Ausführung mit FCFS	Parallele Ausführung mit LJF
Log-File 1	15.5 ms	13.6 ms ($\approx$ 17% Zeitersparnis)	14.3 ms ($\approx$ 13% Zeitersparnis)
Log-File 2	16.2 ms	13.2 ms ($\approx$ 19% Zeitersparnis)	12.7 ms ($\approx$ 22% Zeitersparnis)
Log-File 3	16.2 ms	13.7 ms ($\approx$ 15% Zeitersparnis)	13.0 ms ($\approx$ 20% Zeitersparnis)
Log-File 4	16.6 ms	13.5 ms ($\approx$ 19% Zeitersparnis)	13.4 ms ($\approx$ 19% Zeitersparnis)

Tabelle 2: Vergleich der Ausführungszeiten

Die Werte in Tabelle 2 zeigen, dass die parallele Ausführung mit FCFS und LJF, in allen Log-Files schneller ist, als die sequentielle Ausführung.
Bei der parallelen Ausführung mit FCFS wurde eine Zeitersparnis zwischen 15% und 19% erreicht. Bei der parallelen Ausführung mit LJF eine Zeitersparnis zwischen 13% und 22%.
Mit Ausnahme vom ersten Log-File, ist auch die parallele Ausführung mit der Verwendung des LJF-Prinzips schneller, als die mit der Verwendung des FCFS-Prinzips.
In Tabelle 2 ist zu sehen, dass die parallele Ausführung im Vergleich mit der sequentiellen Ausführung schneller ist.

7 Zusammenfassung und Ausblick

In dieser Arbeit wurde eine Anpassung des Module-Frameworks der FUmanoids vorgestellt. Als Erstes wurde die Berechnung der Ausführungsreihenfolge mit Hilfe eines Graphen implementiert. Durch die Verwendung des Graphen lässt sich einfach testen, ob unter den Modulen eine zyklische Abhängigkeit besteht. Sollte dies der Fall sein wird eine Meldung ausgegeben, die diesen Kreis darstellt um nachvollziehen zu können, unter welchen Modulen diese zyklische Abhängigkeit besteht.
Es wurden die beiden Implementierungen, zur Berechnung der Ausführungsreihenfolge, miteinander verglichen. Dabei ist zu sehen, dass die Berechnung mit Hilfe des Graphens deutlich schneller ist, als die Berechnung ohne Graphen. Durch die Ausgabe der zyklischen Abhängigkeit in der Implementierung mit Graphen, lassen sich Fehler leichter erkennen.

Im Hauptteil dieser Arbeit wurde eine Änderung der Modulausführung vorgestellt. Die sequentielle Ausführung der Module wurde durch eine parallele Ausführung ausgetauscht. Dabei wurden die zur Ausführung bereitstehenden Module in einer Standard-*Queue* abgelegt. Dort wurden nach dem FCFS-Prinzip die Module zur Ausführung ausgewählt. An Hand eines Beispiels ließ sich erkennen, dass dies nicht die optimalste Strategie war. Man konnte durch Umstellen der Ausführungsreihenfolge eine Einsparung von einer Zeiteinheit erreichen.
Um diese Einsparung zu realisieren wurde die Auswahl nach dem LJF-Prinzip umgestellt. Die Laufzeit der Module wird gestoppt und mit Hilfe dieser Zeit wird eine Durchschnittslaufzeit berechnet. An Hand dieses Durchschnitts werden die Module nun in einer Priority-Queue einsortiert. Somit werden nun Module mit hoher Laufzeit bevorzugt.

Ausblick

Sollte man zu den Modulen die tatsächliche Laufzeit kennen, wäre es möglich weitere Optimierungen zu erreichen. Sollten innerhalb eines Frames bei einem ModuleManager Lücken sein, dann wäre es denkbar, dass Module aus einem anderen ModuleManager, von diesem ausgeführt werden können. So würde die Auslastung der Kerne erhöht und eventuell die Gesamtlaufzeit verringert werden.

Sollte gerade ein sehr langes Modul ausgeführt werden, auf das andere Module warten, können andere Executoren nicht arbeiten. Es wäre auch denkbar, dass diese Executoren in der Zeit, in der sie nicht arbeiten einem anderen ModuleManager zugeteilt werden könnten.
Auch hier ist es notwendig zu wissen wie lange die einzelnen Module für ihre

Arbeit brauchen. Damit lässt sich die Auslastung der Kerne erhöhen und auch die Gesamtaufzeit könnte verringert werden.

Im Hinblick auf die immer komplexer werdenden Anforderungen an die Roboter und die damit einhergehenden höheren Rechenkosten, werden in Zukunft schnellere CPUs benötigt. Eine Möglichkeit wäre der ODROID-XU3. Diese Recheneinheit besitzt zwei verschiedene CPUs. Die eine CPU ist ein Quadcore mit 2.1 GHz und die andere CPU ist ein Quadcore mit 1.5GHz. Mit dieser neuen Recheneinheit könnte man in der Lage sein, bestimmte Module auf bestimmten Kernen laufen zu lassen. So könnten rechenintensive Module auf den schnelleren Kernen laufen. Auf den langsameren Kernen könnten dann die Module ausgeführt werden, die eine längere Laufzeit haben.

Literatur

[1] Ecto - A C++/Python Computation Graph Framework, http://plasmodic.github.io/ecto/, 2015.

[2] Humanoid League Proposed Roadmap, www.robocuphumanoid.org/wp-content/uploads/HumanoidLeagueProposedRoadmap.pdf, 2015.

[3] RoboCup Soccer Humanoid League Rules and Setup For the 2015 Competition in Hefei, http://www.robocuphumanoid.org/wp-content/uploads/HumanoidLeagueRules2015_DRAFT_20141205.pdf, 2015.

[4] T.H. Cormen, C.E. Leiserson, R. Rivest, and C. Stein. *Algorithmen - Eine Einführung*. Oldenbourg Wissenschaftsverlag, 2010.

[5] Sebastian Iwanowski and Rainer Lang. Diskrete mathematik mit grundlagen. In *Diskrete Mathematik mit Grundlagen*, pages 231–294. Springer Fachmedien Wiesbaden, 2014.

[6] Rolf H. Möhring. Verteilte Verbindungssuche im öffentlichen Personenverkehr: Graphentheoretische Modelle und Algorithmen. Preprint 624, Technische Universität Berlin, Institut für Mathematik, 1999. Appeared in Patrick Horster (ed.) Angewandte Mathematik - insbesondere Informatik, Beispiele erfolgreicher Wege zwischen Mathematik und Informatik, Vieweg Verlag, 1999, pages 192-220.

[7] S. Sreekantan Nair and Marcel F. Neuts. A priority rule based on the ranking of the service times for the m/g/1 queue. *Operations Research*, 17(3):pp. 466–477, 1969.

[8] Ronen Perry and Tal Z Zarsky. Queues in law. *Iowa L. Rev.*, 99:1595, 2013.

[9] Sebastian Petters, Dirk Thomas, and Dipl-Math Jutta Kiener. Roboframe-softwareframework für mobile autonome robotersysteme. *TU Darmstadt Diplomarbeit*, 2005.

[10] Sebastian Petters, Dirk Thomas, and Oskar Von Stryk. Roboframe-a modular software framework for lightweight autonomous robots. In *Proc. Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware of the 2007 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2007.

[11] Holger Schlingloff. Zyklensuche in Graphen. In Berthold Vöcking, Helmut Alt, Martin Dietzfelbinger, Rüdiger Reischuk, Christian Scheideler, Heribert Vollmer, and Dorothea Wagner, editors, *Taschenbuch der Algorithmen*, eXamen.press, pages 83–93. Springer Berlin Heidelberg, 2008.

[12] Daniel Seifert, Lutz Freitag, Jan Draegert, Simon Gene Gottlieb, Roman Schulte-Sasse, Gregor Barth, Malte Detlefsen, Nicklas Rughöft, Michael Pluhatsch, Martin Wichner, and Raúl Rojas. Berlin United – FUmanoids Team Description Paper 2015. January 2015.

[13] A.S. Tanenbaum. *Moderne Betriebssysteme*. Pearson Studium - IT. Pearson Deutschland, 2009.

[14] Gerald Teschl and Susanne Teschl. *Mathematik Für Informatiker – Diskrete Mathematik und Lineare Algebra*, volume 1 of *eXamen.press*. Springer-Verlag, Berlin, Heidelberg, New York, 3 edition, October 2010.

[15] V. Turau. *Algorithmische Graphentheorie*. De Gruyter, 2009.