Carsten Hoppe

Fallbasiertes Schließen in strategischen Spielentscheidungen auf der NAO Robotik Plattform

20. August 2015

supervised by:
Prof. Dr. Sibylle Schupp
Dipl.-Ing. Sven Mattsen,
Robert Oehlmann, B.Sc.
Erklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbständig durchgeführt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Hamburg, den 20. August 2015

Carsten Hoppe
Inhaltsverzeichnis

1. Einführung 1

2. RoboCup 3
   2.1. RoboCup 3
   2.2. Standard Platform League 5
      2.2.1. Regelwerk 5
      2.2.2. NAO Robotik Plattform 7

3. Fallbasiertes Schließen (FBS) 9
   3.1. Allgemeine Grundlagen 9
      3.1.1. FBS Zyklus 11
      3.1.2. Anwendungsbereiche des FBS 11
      3.1.3. FBS als Lernprozess 12
   3.2. Verwandte Arbeiten 13
   3.3. FBS im Roboterfußball 16
      3.3.1. Repräsentation eines Falles 16
      3.3.2. Auswahl eines Falles 24

4. Implementierung 33
   4.1. Technische Hilfsmittel 33
   4.2. Technische Umsetzung 33
      4.2.1. XML Serialisierung und Deserialisierung 34
      4.2.2. Objektklassen 36
      4.2.3. Algorithmen-Klassen 38

5. Evaluation 41
   5.1. Laufzeitkomplexität 41
   5.2. Messung der Laufzeit 43
   5.3. Auswertung der Fallbasis 44

6. Zusammenfassung und Ausblick 51

A. Struktur einer XML-Datei eines Falles 53

B. Sortierung der geeigneten Fälle 55

C. Bewegungsbahn der Balles 57
<table>
<thead>
<tr>
<th>Abkürzung</th>
<th>Erklärung</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBS</td>
<td>Fallbasiertes Schließen</td>
</tr>
<tr>
<td>HL</td>
<td>Humanoid League</td>
</tr>
<tr>
<td>HULKs</td>
<td>Hamburg Ultra Legendary Kickers</td>
</tr>
<tr>
<td>ID</td>
<td>Identifikator</td>
</tr>
<tr>
<td>IJCAI</td>
<td>International Joint Conference on Artificial Intelligence</td>
</tr>
<tr>
<td>IR</td>
<td>Infrarot</td>
</tr>
<tr>
<td>IROS</td>
<td>International Conference on Intelligence Robotics and Systems</td>
</tr>
<tr>
<td>KI</td>
<td>Künstliche Intelligenz</td>
</tr>
<tr>
<td>MSL</td>
<td>Middle Size League</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>SL</td>
<td>Simulation League</td>
</tr>
<tr>
<td>SPL</td>
<td>Standard Platform League</td>
</tr>
<tr>
<td>SSL</td>
<td>Small Size League</td>
</tr>
<tr>
<td>TC</td>
<td>Technical Committee</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
</tbody>
</table>
# Abbildungsverzeichnis

<table>
<thead>
<tr>
<th>Kapitel</th>
<th>Bildung</th>
<th>Seite</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Spielfeld Abmessungen</td>
<td>6</td>
</tr>
<tr>
<td>3.1</td>
<td>FBS Zyklus</td>
<td>12</td>
</tr>
<tr>
<td>3.2</td>
<td>Spielfeld als Koordinatensystem</td>
<td>18</td>
</tr>
<tr>
<td>3.3</td>
<td>Beispiel eines Falles</td>
<td>22</td>
</tr>
<tr>
<td>3.4</td>
<td>Symmetrischer Fall</td>
<td>23</td>
</tr>
<tr>
<td>3.5</td>
<td>Beispiel eines Falles</td>
<td>24</td>
</tr>
<tr>
<td>3.6</td>
<td>Trajektorie eines Balles</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>UML-Klassendiagramm</td>
<td>36</td>
</tr>
<tr>
<td>5.1</td>
<td>Messwerte der Laufzeit des Retrieve-Algorithmus</td>
<td>44</td>
</tr>
<tr>
<td>5.2</td>
<td>Häufigkeit geeigneter Fälle</td>
<td>46</td>
</tr>
<tr>
<td>5.3</td>
<td>Häufigkeit nicht geeigneter Fälle (Ähnlichkeitsmaß)</td>
<td>47</td>
</tr>
<tr>
<td>5.4</td>
<td>Häufigkeit nicht geeigneter Fälle (Anpassungskosten)</td>
<td>47</td>
</tr>
<tr>
<td>5.5</td>
<td>Häufigkeit nicht geeigneter Fälle (Anzahl Spieler)</td>
<td>48</td>
</tr>
<tr>
<td>5.6</td>
<td>Häufigkeit nicht geeigneter Fälle (FreePath)</td>
<td>49</td>
</tr>
</tbody>
</table>
Listingverzeichnis

4.1. Struktur einer XML-Datei eines Falles .......................... 34
A.1. Beispiel einer XML-Datei eines Falles .......................... 53
B.1. Sortieralgorithmus .............................................. 55
C.1. Algorithmus zur Berechnung der Ballbahn .................... 57
Algorithmenverzeichnis

1. IsCandidate(problem, case) ................................................. 31
2. Retrieve(problem, caseBase) ................................................ 31
3. FindBestMapping(problem, case) ......................................... 41
4. SelectCandidates(problem, caseBase) ................................. 42
1. Einführung

Die beliebte Sportart Fußball verbindet weltweit zahlreiche Menschen. Ebenso gewinnt der Roboterfußball zunehmend an Bedeutung und Popularität. So bezeichnet J.-S. Gutmann den Roboterfußball als „wissenschaftliches anspruchsvolles Forschungsproblem, das erfordert, Probleme aus den Bereichen Robotik, Künstliche Intelligenz und Multi-Agenten-Systeme zu lösen und die Lösungen in einem System zu integrieren, um ein erfolgreiches Roboterfußballteam zu kreieren [9].“ Der Roboterfußball stellt also ein Problem und eine Herausforderung auf dem Gebiet Künstliche Intelligenz (KI) dar, bei dem die Roboter in realen, agilen Umgebungen agieren und Entscheidungen auf Grundlage unvollständiger Informationen in Echtzeit treffen und umsetzen müssen, wie es J. Diemke weiterhin beschreibt [6].


Während der Entwicklung des Verhaltens müssen die Teams dabei verschiedene Bedingungen und Anforderungen berücksichtigen.

'Soccer, like most other group activities, requires teamwork and some type of division of labor or role assignment, giving each member of the team a job so each knows what to do to help the team as a whole without getting in the way of the others.' [14, S. 238]


Da die Erforschung in Zusammenarbeit mit der studentischen Roboter-AG Hamburg Ultra Legendary Kickers (HULKS) erfolgt, soll eine Anpassung der Implementierung an die vorhandene Codebasis der AG erfolgen. Somit haben die HULKS die Möglichkeit, die Ergebnisse in Zukunft zu verwenden und während der Teilnahme an Wettbewerben einzusetzen.

Die Arbeit basiert auf den Forschungsergebnissen und der ausgearbeiteten Methode des FBS von R. Raquel et al. [23], die die Methode in Hinblick auf den Sony AIBO Roboter entwickelt haben. Da in der SPL aktuell der NAO als einheitliche Plattform eingesetzt wird, erfolgt die Untersuchung der Methode des FBS mit dem Ziel, die Methode auf dieser Plattform einzusetzen.


Die Bachelorarbeit untersucht die Methode FBS die beim Roboterfußball in der SPL eingesetzt werden soll. Im Kapitel 2 werden die Grundlagen erklärt, die das Anwendungsbereich der Methode eingrenzen. Daher wird die Initiative RoboCup vorgestellt, die die Fußballwettbewerbe organisiert. Ebenso werden das aktuelle Regelwerk zusammengefasst und die NAO Robotik Plattform bekannt gegeben. Diese Informationen beschränken das Gebiet, in dem die Methode verwendet werden kann, und stellen Anforderungen, die bei der Entwicklung berücksichtigt werden müssen. Im darauffolgenden Kapitel wird zunächst das FBS im allgemeinen Kontext erklärt, bevor die Entwicklung der Methode im Hinblick auf dem Roboterfußball folgt, die durch R. Ros et al. bereits durchgeführt wurde [23]. Im Anschluss dieses Kapitels wird die Implementierung des Ansatzes auf Grundlage der gegebenen Überlegungen beschrieben. Schließlich erfolgt die Evaluation im nächsten Kapitel 5, in dem unter anderem die Laufzeitkomplexität des Algorithmus diskutiert wird. Die vorliegende Arbeit endet mit der Zusammenfassung der Untersuchungsergebnisse und einem Ausblick wie die Arbeit erweitert werden kann.
2. RoboCup

Die internationale Initiative RoboCup organisiert nicht nur Wettbewerbe, an denen mehrere Teams gegeneinander Roboterfußball spielen. Sie ermöglicht zugleich, dass der Öffentlichkeit Forschungsergebnisse und Technologien bekannt gegeben werden und Ergebnisse aus KI und Robotik verglichen werden. Während der Wettbewerbe können sich die Teams anhand ihrer Ergebnisse in unterschiedlichen Anwendungsgebieten und Ligen messen, zu denen die SPL gehört. Die SPL wird im Abschnitt 2.2 beschrieben. Hierzu wird zum einen das in der SPL vorgegebene Regelwerk in Unterabschnitt 2.2.1 erläutert und zum anderen werden die Eigenschaften und Fähigkeiten des Roboters NAO in 2.2.2 wiedergegeben. In Abschnitt 2.1 wird die Organisation RoboCup zunächst vorgestellt.

2.1. RoboCup


RoboCupSoccer Beim RoboCupSoccer besteht die Herausforderung darin, ein aus autonomen, kooperativen Robotern bestehendes Team zu kreieren, welches konkurrenzfähiges Verhalten und Strategien ausführen kann. Der Schwerpunkt ist dabei das Fußballspiel, in dem ein Roboter-Team mehr Tore als der Gegner erzielen muss.

RoboCupRescue In der Liga RoboCupRescue werden Roboter entworfen, die in gefährlichen Situationen Menschen retten und unterstützen. Die teil-autonomen Roboter
sollen gefährliche und riskante Aufgaben ausführen und dabei komplexe Umgebungen erfassen und Hindernisse überwinden.

**RoboCup@Home** Die Liga *RoboCup@ Home* fördert die Entwicklung interaktiver Roboter, die den Menschen in täglichen Alltagssituationen assistieren und unterstützen.

**RoboCupJunior** Junge Schüler und Schülerinnen werden in der Liga *RoboCupJunior* motiviert, Fähigkeiten und Wissen im Bereich Technik, Naturwissenschaften und Mathematik zu erwerben. Zudem werden ihre sozialen Kompetenzen durch praktische Tätigkeiten während der Entwicklung und Programmierung autonomer Roboter gefördert.

Die Ligen, wie sie auf der Website des *RoboCup* [20] vorgestellt werden, setzen also unterschiedliche Schwerpunkte. Gemeinsam fördern sie jedoch die Entwicklung und den Fortschritt im Bereich der Robotik und der KI. Seit 1997 werden die Wettkämpfe jährlich ausgetragen. Das ursprüngliche Ziel des *RoboCup* formuliert M. G. Lagoudakis wie folgt:

"By the year 2050, to develop a team of fully autonomous humanoid robots that can win against the human world soccer champions!" [12]


2.2. Standard Platform League

Die **SPL** unterscheidet sich von den anderen Ligen darin, dass alle Teams mit einheitlicher Hardware antreten. Im Mittelpunkt stehen dabei die Softwareentwicklung und die Herausforderungen wie zum Beispiel der Objekterkennung und der Lokalisierung, wie P.-M. Ziegler erklärt [26].

Zusammenfassend betrachtet ermöglicht die Initiative RoboCup, Forschungsergebnisse zu vergleichen und diese der Öffentlichkeit zu zeigen. Die Wettkämpfe motivieren die Teams, neue Technologien und Lösungen im Bereich der Robotik und der KI zu entwickeln. Die Organisation von Fachkonferenzen des **RoboCup** bietet zudem die Möglichkeit, die Ergebnisse zu präsentieren.

### 2.2. Standard Platform League


Im Folgenden werden zunächst einige Regeln des Regelwerkes in Abschnitt 2.2.1 erläutert, die in der **SPL** von den Teams befolgt werden müssen. Zudem wird die NAO Robotik Plattform im anschließenden Abschnitt 2.2.2 vorgestellt. Da in der vorliegenden Arbeit die Methode des **FBS** mit Schwerpunkt auf die **SPL** implementiert und untersucht wird, bilden das Regelwerk der Liga und der eingesetzte Roboter NAO den Kontext, in dem die Methode eingesetzt wird.

#### 2.2.1. Regelwerk

Das **RoboCup Technical Committee (TC)** verfasst für die **SPL** ein Regelwerk [21], welches Vorschriften definiert, die während eines Fußballspiels zweier Mannschaften eingehalten werden müssen. Das Regelwerk wird vom **TC** regelmäßig geändert und zu den Events des **RoboCup** veröffentlicht. So ist zum Beispiel im Jahr 2014 die Regel hinzugefügt worden, dass jedes Team einen sogenannten *Coaching Robot* einsetzen darf.

Die erste Regel des Regelwerkes beschreibt das Spielfeld, auf dem ein Fußballspiel stattfindet. Wie in der Abbildung 2.1 zu sehen besteht das Spielfeld aus zwei Hälften, mit jeweils einem Tor und einem Torraum. Die Spielfeldhälften sind durch eine Mittellinie und einen Mittelkreis getrennt. Im aktuellen Regelwerk beträgt die Strecke \( A \) 9000 mm und entspricht der Länge des Spielfeldes. Die Breite des Feldes ist durch die Strecke \( B \) dargestellt und ergibt 6000 mm. Die weiteren Abmessungen, wie zum Beispiel der Durchmesser des Mittelkreises, sind in dem Regelwerk ebenfalls definiert.
Abbildung 2.1.: Abmessungen des Spielfeldes, die im Regelwerk definiert sind [21]

Im zweiten Abschnitt des Regelwerkes wird erklärt, wie ein Roboterteam zusammengesetzt wird. Jedes Team besteht aus nicht mehr als fünf Spielern und einem sogenannten *Coaching Robot*. Es kann höchstens ein Spieler als Torwart ernannt werden, während alle anderen Spieler Feldspieler sind.

Im Gegensatz zu den Feldspielern ist es dem Torwart erlaubt, den Ball innerhalb des Strafraums mit der Hand beziehungsweise mit dem Arm zu berühren. Der Torwart und die Feldspieler tragen Trikots mit Nummern, sodass diese eindeutig unterschieden werden können.


Eine weitere Regel beschreibt, wie die Kommunikation zwischen den Robotern während des Spiels stattfinden darf. Da die Roboten im Spiel nicht durch Menschen ferngesteuert werden dürfen, ist eine Kommunikation nur zwischen den Feldspielern sowie zwischen den Robotern und dem *GameController* zugelassen. Die Kommunikation zwischen den Robotern darf dabei nur über das Netzwerkprotokoll *User Datagram Proto*
2.2. Standard Platform League

col \textit{(UDP)} geschehen, welches das sogenannte \textit{SPL Standard Message Packet} verwendet. Zudem wird vorgegeben, dass nicht mehr als fünf Nachrichten pro Roboter und pro Sekunde geschickt werden dürfen. Der \textit{GameController} nutzt ebenfalls \textit{UDP} um eine Verbindung zu den Robotern aufzubauen.

Als Hardware wird der humanoide Roboter NAO der Firma \textit{Aldebaran Robotics} eingesetzt. Änderungen oder Erweiterung der Roboterhardware sind im Spiel nicht erlaubt. Ebenso ist die Verwendung zusätzlicher Hardware verboten. Diese Regel bedeutet, dass Teams mit der gleichen Hardware gegeneinander antreten und sich somit auf die Entwicklung schneller und effektiver Algorithmen konzentrieren können.

In dem Regelwerk werden schließlich noch weitere Regeln definiert, die unter anderem den Spielablauf und verbotene Aktionen erläutern. Da diese Regeln für die vorliegende Arbeit nicht relevant sind, wird an dieser Stelle auf das Regelwerk verwiesen.

2.2.2. NAO Robotik Plattform

Der \textit{NAO} Roboter wurde im Jahr 2008 zum ersten Mal während der Austragung der \textit{RoboCup} Weltmeisterschaften in der \textit{SPL} eingesetzt. Der Hersteller des Roboters ist das französische Unternehmen \textit{Aldebaran Robotics}, welches von Bruno Maisonnier im Jahr 2005 gegründet wurde. Das Unternehmen ist auf den Handel und die Entwicklung humanoider Roboter spezialisiert und hatte zu Beginn der Gründungsphase das Ziel, einen humanoiden Roboter mit verschiedenen Fähigkeiten zu entwickeln und der Öffentlichkeit zur Verfügung zu stellen \cite{15}.

Wie auf der Homepage des Unternehmens \textit{Aldebaran Robotics} präsentiert wird \cite{2}, ist der NAO Roboter ein 58 cm großer und humanoider Roboter. Er besitzt unterschiedliche Fähigkeiten, um hören, sehen, sprechen und sich bewegen zu können. In mehr als 70 Ländern wird er deshalb im Unterricht an Schulen und in Universitäten eingesetzt. Weiterhin stellt \textit{Aldebaran Robotics} ein \textit{Software Development Kit (SDK)} zur Verfügung, um die Entwicklung und Programmierung des Roboters zu ermöglichen. Dieses \textit{SDK} ist mit verschiedenen Programmiersprachen, wie zum Beispiel C++, Python und Matlab, kompatibel.

Der NAO besitzt mehrere Sensoren und Motoren, die durch das Betriebssystem \textit{NAOqi} gesteuert werden. Insgesamt hat der NAO 25 Freiheitsgrade, die durch elektrische Motoren möglich sind. Zudem besitzt der NAO zwei Kameran sowie neun Berührungssensoren und acht Drucksensoren. Weiterhin hat der Roboter vier Mikrofone und Infrarot (IR) Sensoren und Empfänger. Das Betriebssystem \textit{NAOqi} dient dazu, die von den Sensoren empfangenen Daten zu interpretieren. Im Kopf des Roboters ist eine \textit{Intel ATOM 1,6 GHz CPU} vorhanden \cite{2}.

Des Weiteren wird auf der Website des Herstellers \cite{2} bekannt gegeben, dass der NAO \textit{Wi-Fi} und \textit{Ethernet} unterstützt und mit dem \textit{IEEE 802.11 b/g/n Wi-Fi Standard} kompatibel ist.
3. Fallbasiertes Schließen (FBS)


3.1. Allgemeine Grundlagen


M. M. Richter erwähnt weiterhin, dass es das Ziel des FBS ist, frühere Erfahrungen zur Lösung aktueller Probleme zu verwenden. Solche Erfahrungen werden durch ein geordnetes Paar, welches aus einem Problem und einer Lösung besteht, beschrieben und in einer Datenbank, der sogenannten Fallbasis, aufbewahrt. M. M. Richter erklärt zudem, dass die Erfahrungen beim Lösen eines Problems als Fall bezeichnet werden und die Problemstellung sowie die entsprechende Lösung betreffen [17, 18].

3. Fallbasiertes Schließen (FBS)

Fällen, die einem aktuellen Problem ähnlicher sind, bevorzugt werden, so M. M. Richter [17].

Als Herausforderung des FBS wird zudem die Wartung der Fallbasis betrachtet. Die Fallbasis altert nicht durch ihren Gebrauch, sondern darin, dass die Fälle nicht mehr mit der Realität übereinstimmen. Das Problem bei der Wartung ist es festzustellen, wann ein Fall entfernt, hinzugefügt, beziehungsweise verändert werden soll [17].

Im Folgenden werden die Definitionen der Begriffe gegeben, die beim FBS benutzt werden:

**Problem** Ein *Problem* ist eine ungelöste Aufgaben- oder Fragestellung [8]. Zum Beispiel stellt beim Roboterfußball eine Spielsituation ein Problem dar. Bei der Spielsituation stellt sich die Frage, wie die Situation gelöst werden kann, damit entsprechende Ziele erreicht werden können.

**Lösung** Unter dem Begriff *Lösung* wird die Bewältigung eines Problems [7] und eines Problems verstanden. Beim Roboterfußball ist zum Beispiel die Ausführung von Aktionen, um eine Spielsituation zu lösen und um Ziele zu erreichen, eine Lösung. Es kann zwischen einer *erfolgreichen* und einer *nicht erfolgreichen* Lösung unterschieden werden, was abhängig ist, ob die Ziele der Problemstellung zufriedenstellend gelöst werden.


**Fall** Als *Fall* wird eine Erfahrung bezeichnet, die als ein geordnetes Paar dargestellt wird. Bei diesem Paar wird einer Problemstellung eine Lösung zugeordnet, wobei die Problemstellung und die Lösung entsprechend beschrieben sind.

**Fallbasis** Eine Menge von Fällen bildet eine sogenannte *Fallbasis.*

**Schließen** Beim *Schließen* werden Erfahrungen genutzt und abgeleitet, wie eine aktuelle Problemstellung gelöst werden kann. Beim *Fallbasierten Schließen* liegen die Erfahrungen als Fälle in einer Fallbasis vor. Geeignete Fälle, die der aktuellen Problemstellung ähnlich sind, sind hilfreich bei der Suche einer Lösung des aktuellen Problems. Die geeigneten Fälle müssen zuvor aus der Fallbasis bestimmt werden. Die Lösung, die in einem geeigneten Fall vorliegt, kann somit in der aktuellen Problemstellung direkt angewendet werden. Da dennoch Unterschiede zwischen
3.1. Allgemeine Grundlagen

den Problemstellungen bestehen, ist eine Anpassung der Lösung an das aktuelle Problem notwendig.

Ähnlichkeitsmaß Das Ähnlichkeitsmaß illustriert, wie ähnlich zwei Problemstellungen sind. Beim FBS wird der Fall aus der Fallbasis als geeigneter Fall ausgewählt, der dem aktuellen Problem am ähnlichsten ist.

Adaption Da bei der Auswahl eines geeigneten Falles trotzdem Unterschiede zwischen den Problemstellungen bestehen, ist eine Anpassung beziehungsweise Adaption der Lösung notwendig.

3.1.1. FBS Zyklus

A. Aamodt und E. Plaza visualisieren die Methode des FBS als Zyklus mit vier Schritten [1]. Im ersten Schritt, dem sogenannten Retrieve Schritt, wird der zu einem gegebenen Problem ähnlichste Fall aus einer Fallbasis ausgewählt. Anschließend werden im Reuse Schritt das Wissen und die Informationen, die in dem ausgewählten Fall vorhanden sind, benutzt, um das aktuelle Problem zu lösen. Danach wird die vorgeschlagene Lösung überarbeitet. Während des Revise Schrittes, wird die Lösung eines neu gelösten Problems auf die Korrektheit überprüft. Im Retain Schritt werden die Eigenschaften einer neuen Erfahrung in die vorhandene Fallbasis aufgenommen, damit diese Erfahrungen in neuen Situationen hilfreich sein können.

In der Abbildung 3.1 ist der FBS Zyklus nach A. Aamodt und E. Plaza dargestellt. Wenn eine Beschreibung eines neuen Problems vorliegt, dann wird ein neuer Fall definiert. Dieser Fall wird benutzt, um einen Fall aus der gegebenen Fallbasis auszuwählen, der dem aktuellen Fall am ähnlichsten ist. Der ausgewählte Fall und der neue Fall werden zusammen kombiniert, um eine neue Lösung zu erstellen. Es wird ein gelöster Fall mit der neuen Lösung erstellt. Anschließend wird diese Lösung auf Erfolg überprüft, ob eventuelle Fehler auftreten. Es wird ein getesteter Fall generiert. Danach kann aus dem getesteten Fall ein gelernter Fall erstellt werden, der in die Fallbasis aufgenommen wird. Somit ist der neu gelernte Fall in Zukunft zur Lösung neuer Probleme nützlich [1].

3.1.2. Anwendungsbereiche des FBS


3.1.3. FBS als Lernprozess

FBS ist vorteilhaft, da das wiederholte Lösen eines Problems im Allgemeinen leichter ist. Wie H.-D. Burkhard und R. Berger begründen [4], kann der Anwender beim wiederholten Lösen an alte Lösungen sich erinnern und diese erneut nutzen. Weiterhin kann sich der Anwender an Fehler erinnern, die er bereits begangen hat, um somit das wiederholte Vorkommen dieser Fehler zu verhindern.

3.2. Verwandte Arbeiten

Während der Methode des FBS wird eine neue, aktuelle Situation mit einer bereits bekannten Situation verglichen. Der Prozess, in dem die neue Situation verglichen wird, damit Erfahrungen später aufgerufen werden können, wird als Interpretation bezeichnet [4].

In einer neuen Situation werden die bereits gesammelten Erfahrungen verwendet, um eine neue Lösung zu erzeugen. Da die neue Situation und die alte Situation Unterschiede aufweisen, ist eine Anpassung nötig. Die alte Lösung wird angepasst, damit die neue Situation gelöst werden kann [4].


3.2. Verwandte Arbeiten


H.-D. Burkhard und R. Berger stellen unterschiedliche Ansätze vor [4], wie FBS bereits beim Roboterfußball eingesetzt wurde. So wird FBS unter anderem als Methode zur Selbstonkalisierung verwendet. In dieser Methode stellt ein Problem einen Blick eines


H.-D. Burkhard und R. Berger erklären weiterhin den Prozess der Optimierung einer Fallbasis [4]. In diesem Prozess werden im ersten Schritt alle signifikanten Informationen von jedem Fall aufgefasst, um anschließend Informationen, die weniger wichtig sind, zu löschen. Mit diesem Schritt lassen sich generelle Fälle erstellen. Der weitere Schritt besteht darin, dass Fälle gelöscht werden, die redundant sind. Das heißt, dass Fälle gelöscht werden können, ohne der Verringerung der Kompetenz der Fallbasis. Das Löschen dieser nutzlosen Fälle beschleunigt letztendlich den Retrieval Prozess, in der dem ähnlichsten Fall aus der Fallbasis gesucht wird.

Insbesondere betrachtet, gibt es also verschiedene Ansätze, um die Methode FBS beim Roboterfußball einzusetzen. Bei der Entwicklung eines FBS Ansatzes ist die zentrale Frage, wie man das Wissen und die Erfahrungen strukturieren kann, damit auf die Erfahrungen zurückgegriffen werden kann.

Weiterhin ist die Wartung der Fallbasis und der Fälle notwendig, damit ein Erfolg beim FBS erreichbar ist. Wie H.-D. Burkhard und R. Berger darstellen [4], ist eine große Fallbasis nicht hilfreich, da die Echtzeitanforderungen des Systems erfüllt sein müssen. Die Auswahl eines geeigneten Falles aus der Fallbasis muss unterhalb einer


R. Ros und J. L. Arcos erweitern den Ansatz und untersuchen [22], wie eine robuste Fallbasis für den Roboterfußball erstellt wird. Hierzu wird eine anfängliche Fallbasis manuell erstellt, in der die wesentlichen Fälle mit dem entsprechenden Wissen vorhanden sind. Diese Fälle enthalten übliche Situationen, die in einem Fußballspiel entstehen können. Bei der manuellen Erstellung dieser Fälle können zwar die Beschreibung des Problems und die Beschreibung der Lösung genau definiert werden, der Bereich, in dem die Fälle anwendbar sind, ist jedoch nicht genau bekannt. Es wird erläutert, dass die Bereiche zunächst durch eine Standardgröße definiert werden. Wenn im Spiel nun die Fälle angewendet werden, dann wird überprüft, ob die Anwendung der Lösung erfolgreich oder nicht erfolgreich gewesen ist. Ein Gültigkeitsbereich enthält einen sogenannten inneren Sicherheitsbereich einen äußeren Gefahrenbereich. Wenn die Ausführung erfolgreich gewesen ist, so wird der Gültigkeitsbereich des Falles erweitert und der Fall wird mit dem neuen Gültigkeitsbereich in der Fallbasis angepasst. Ist die Ausführung nicht erfolgreich, so wird der Gültigkeitsbereich verkleinert und der Fall wird in der Fallbasis angepasst. Die Größe des Gültigkeitsbereiches wird nur vergrößert beziehungsweise verringert, wenn sich der Ball innerhalb des äußeren Bereiches befindet. Mit diesem Ansatz ist es möglich, dass sich die Größe des Gültigkeitsbereiches eines Falles während des Spieles automatisch anpasst. R. Ros und J. L. Arcos erklären weiterhin [22], dass die Roboter ein Verhalten ausführen müssen, wenn kein Fall aus der Fallbasis als geeigneter Fall gefunden wird. Wenn kein Fall gefunden wird, so wird ein Verhalten zufällig generiert und ausgeführt. Ist die Ausführung des zufällig generierten Verhaltens erfolgreich gewesen, dann wird dieses Wissen als neuer Fall in der Fallbasis gespeichert. Das neue Wissen ist somit in
der Zukunft anwendbar.


3.3. FBS im Roboterfußball

Im folgenden Abschnitt wird nun erklärt, wie der Ansatz des FBS beim Roboterfußball angewendet werden kann. Zunächst wird in Abschnitt 3.3.1 beschrieben, wie ein Fall dargestellt wird. Ein Fall enthält eine Problembeschreibung, die den Zustand des Spiels beschreibt, eine Lösungsbeschreibung, die die Aktionen der Roboter beschreibt, sowie eine Beschreibung des Bereiches, in dem der Fall anwendbar ist. Dieser Bereich wird als Gültigkeitsbereich bezeichnet. Eine Lösung eines Falles ist nicht nur in einer bestimmten Position anwendbar, sondern in dem Gültigkeitsbereich. Weiterhin wird im nächsten Abschnitt 3.3.2 erklärt, wie ein Fall aus einer Fallbasis ausgewählt wird. Der ausgewählte Fall muss dazu eine gewisse Ähnlichkeit zu dem aktuellen Problem aufweisen, damit die Lösungsbeschreibung an das aktuelle Problem angepasst werden kann. Der Ansatz basiert auf der Studie nach R. Ros et al., die sich auf den AIBO Roboter fokussieren.[23]

3.3.1. Repräsentation eines Falles


Die Problembeschreibung charakterisiert den Zustand des Spiels und die Problembe-
schreibung veranschaulicht die Aktionen, die zur Lösung in diesem Zustand ausgeführt
werden.

Bei einer neuen Situation wird nun der Fall aus der Fallbasis aufgerufen, der der neuen
Spielsituation am ähnlichsten ist. Der aufgerufene Fall wird anschließend an die neue
Spielsituation angepasst. Das heißt, dass die Beschreibung der Lösung an die aktuelle
Situation angepasst wird. Es werden die Aktionen, die in dieser Lösung beschrieben
werden, den Robotern zugeteilt.

Eine Lösung, die in einem Fall beschrieben wird, bildet eine Menge von Aktionsfolgen.
Diese Aktionsfolgen beschreiben, welche Aktionen welcher Roboter ausführen soll.

Der Prozess, in dem der Fall aus der Fallbasis bestimmt wird, der dem aktuellen
Problem am ähnlichsten ist, wird als Retrieve Schritt bezeichnet. Zur Bestimmung des
ähnlichsten Falles werden jeweils zwei Werte berücksichtigt. Zum einen wird die Ähnlich-
keit zwischen der zu lösenden Spielsituation und dem Fall berechnet und zum anderen
werden die Kosten, die entstehen, um das aktuelle Problem dem Fall anzupassen, be-
rechnet. Es wird der ähnlichste Fall gesucht, der eine möglichst hohe Ähnlichkeit mit
dem aktuellen Problem aufweist und möglichst geringe Kosten bei der Anpassung des
Problems an den Fall verursacht.

Wie R. Ros et al. darstellen [23], werden zwischen kontrollierbaren und unkontrollier-
baren Eigenschaften eines Falles unterschieden. Die Positionen der Gegner und die Po-
sition des Balles, die im Fall beschrieben werden, werden als unkontrollierbar betrachtet.
Die Positionen der Teammitglieder werden als kontrollierbar aufgefasst. Das begründet
sich darin, dass ein Team nur das Verhalten der eigenen Roboter implementieren und
somit steuern kann. Damit ein Fall an das aktuelle Problem angepasst werden kann,
können sich die Teammitglieder bewegen und andere Positionen erreichen. Der Ball und
die Gegner Positionen hingegen können während der Anpassung nicht verändert werden.

Während des sogenannten Reuse Schrittes werden die Aktionen, die im Fall beschrie-
ben sind, den Robotern im Team zugeteilt. Jeder Roboter führt eine Folge von mehreren
Aktionen aus. Bevor die Roboter diese Aktionen ausführen, müssen sich die Roboter po-
sitionieren. Das heißt, dass die Roboter die entsprechenden Startpositionen einnehmen,
die im Fall beschrieben werden.

Insgesamt ordnet ein Fall eine Spielsituation einer möglichen Lösung zu. Der Fall
beschreibt einen Zustand des Spiels. In der Beschreibung sind nur Informationen ent-
halten, die für die Lösung relevant sind. Daher werden nur die Positionen der Roboter
beschrieben, die im Fall involviert sind.

Ein Fall besteht somit aus drei Elementen:

Problembeschreibung Die Beschreibung des Problems enthält die wichtigsten Informa-
tionen über den Zustand der Umwelt.

Lösungsbeschreibung Die Beschreibung der Lösung definiert die Folgen von Aktionen,
die die Roboter ausführen, um ein Problem zu lösen.

Gültigkeitsbereich Die Darstellung des Gültigkeitsbereiches beschreibt das Gebiet, in
dem ein Fall angewendet werden kann.
3. Fallbasiertes Schließen (FBS)

Abbildung 3.2.: Das Spielfeld beim Roboterfußball wird durch ein kartesisches Koordinatensystem dargestellt.

Mathematisch wird ein Fall als Tupel dargestellt:

\[ \text{Fall} = (P, A, K) \]

wobei die Beschreibung des Problems als \( P \), die Beschreibung der Lösung als \( A \) und die Darstellung des Gültigkeitsbereiches als \( K \) bezeichnet wird.

Beschreibung des Problems

Die Beschreibung des Problems enthält einige Details über den Zustand der Umgebung des Spieles. R. Ros et al. erläutern [23], dass der Zustand des Spieles wird als Tupel dargestellt wird:

\[ P = (B,Tm,Opp) \]

wobei die Position des Balles mit \( B \), die Positionen der TeamSpieler mit \( Tm \) und die Positionen der Gegner mit \( Opp \) gekennzeichnet werden. In der Beschreibung des Problems werden nur die Positionen der Roboter definiert, die in der Lösung auch relevant sind. Ebenso werden nur die Positionen der Gegner aufgefasst, die für den Fall relevant sind.

- Die Position des Balles wird mit dem Tupel \( B = (x_B, y_B) \) dargestellt, wobei \( x_B \in [-4500, 4500], y_B \in [-3000, 3000] \) und \( y_B \in \mathbb{Z} \) gilt.

- Die Positionen der Teammitglieder werden als Menge \( Tm = \{tm_1 : (x_1, y_1), \ldots, tm_n : (x_n, y_n)\} \) dargestellt, wobei \( x_i \in [-4500, 4500], x_i \in \mathbb{Z}, y_i \in [-3000, 3000] \) und \( y_i \in \mathbb{Z} \) gilt. Wie in Abschnitt 2.2.1 erwähnt wird, besteht ein Team nicht mehr als fünf Spielern. Daher gilt zudem \( n \in [1, 5] \).

- Die Positionen der Gegner werden als Menge \( Opp = \{opp_1 : (x_1, y_1), \ldots, opp_m : (x_m, y_m)\} \) dargestellt, wobei \( x_i \in [-4500, 4500], x_i \in \mathbb{Z}, y_i \in [-3000, 3000] \) und \( y_i \in \mathbb{Z} \) gilt. Wie in Abschnitt 2.2.1 erwähnt wird, besteht ein Team nicht mehr als fünf Spielern. Daher gilt zudem \( m \in [0, 5] \).

Die Beschreibung der Lösung

Die Beschreibung der Lösung besteht aus mehreren Folgen von Aktionen, die jeweils von einem der im Fall beschriebenen Robotern ausgeführt werden. Diese Menge von Aktionsfolgen wird als Gameplay bezeichnet. Ein Gameplay wird als Menge wie folgt dargestellt:

\[ A = \left( \begin{array}{c} tm_1 : [a_{11}, a_{12}, \ldots, a_{1p_1}] \\ \vdots \\ tm_n : [a_{n1}, a_{n2}, \ldots, a_{np_n}] \end{array} \right) \]

Der Bezeichner \( n \in [1, 5] \) identifiziert den Roboter, der die entsprechende Aktionsfolge ausführt. Der Index \( p_i \) entspricht der Anzahl der Aktionen, die der Roboter mit dem Bezeichner \( tm_i \) ausführt.
Wie R. Ros et al. erklären [23], wird im Gameplay zwischen Aktionen unterschieden, die individuell ausgeführt werden, und Aktionen, die gemeinsam ausgeführt werden. Individuell ausführbare Aktionen können von einem Roboter ausgeführt werden, ohne dass dieser auf die Hilfe eines anderen Roboters angewiesen ist. Zu gemeinsam ausführbare Aktionen hingegen gehören Aktionen, die nicht von einem Roboter einzeln ausgeführt werden können. Bei der Ausführung eines Passes müssen zwei Roboter gemeinsam agieren und kommunizieren, damit ein Pass ausgeführt werden kann. In der vorliegenden Arbeit werden folgende Typen von Aktionen betrachtet:

**goalkick** Bei der Aktion `goalkick` wird ein Torschuss durchgeführt. Das heißt, dass ein Schuss des Balles in Richtung des gegnerischen Tores stattfindet. Diese Aktion besitzt keine Parameter.

**kick** Bei der Aktion `kick` wird ein Schuss des Balles durchgeführt. Als Parameter wird das Ziel als Position angegeben, in deren Richtung der Ball geschossen wird.

**goToPosition** Bei der Aktion `goToPosition` bewegt sich der Roboter zu einem Ziel hin. Das Ziel, welches eine Position darstellt, wird als Parameter angegeben.

**getTheBall** Bei der Aktion `getTheBall` nähert sich der Roboter sich dem Ball, damit dieser anschließend eine weitere Aktion, wie zum Beispiel einen Schuss, durchführen kann. Diese Aktion besitzt keine Parameter, die angegeben werden müssen.

**wait** Die Aktion `wait` bedeutet, dass der Roboter auf die Durchführung einer anderen Aktion, die ein anderer Team Mitspieler durchführt, wartet. Nachdem die andere Aktion, auf die gewartet wird, erfolgreich durchgeführt ist, führt der wartende Roboter die nächste Aktion aus, die der `wait` Aktion folgt. Der Index des Mitspielers, der die Aktion, auf die gewartet wird, durchführt, sowie der Index dieser Aktion werden als Parameter angegeben.

**passTheBall** Bei der Aktion `passTheBall` schießt der Roboter den Ball zu einem anderen Mitspieler hin. Der Index der anderen Mitspielers, der den Ball empfangen soll, wird als Parameter angegeben.

**receiveTheBall** Die Aktion `receiveTheBall` bedeutet, dass der Roboter den Ball bei einem Pass erwartet und empfängt. Der Index des Roboters, von dem der Pass empfangen wird, wird als Parameter angegeben.


Die Darstellung des Gültigkeitsbereiches


\[ K = (\text{ball} : (\tau^y_B, \tau^x_B), \text{opp}_1 : (\tau^y_1, \tau^x_1), \ldots, \text{opp}_m : (\tau^y_m, \tau^x_m)) \]

Dabei entspricht $\tau^x_B$ dem x-Radius des elliptischen Gebietes des Balles und $\tau^y_B$ dem y-Radius. Die Werte $\tau^x_i$ und $\tau^y_i$ entsprechen dem x Radius und dem y Radius des Gebietes des Gegners $\text{opp}_i$.

In der Darstellung des Gültigkeitsgebietes werden nur die Positionen der Gegner aufgefasst, die in dem entsprechenden Fall relevant sind. Relevante Gegner sind gegnerische Roboter, die die Ausführung der Lösung stören und somit eine erfolgreiche Ausführung verhindern können.

Im Schritt, in dem entschieden wird, ob ein Fall aus der Fallbasis als mögliche Lösung des aktuellen Problems in Frage kommt, ist es wichtig zu definieren, welche Voraussetzungen erfüllt sein müssen. In diesem Ansatz besitzt die Position des Balles eine wichtige Bedeutung. Es wird angenommen, dass ein Fall als mögliche Lösung nur betrachtet werden kann, wenn die Position des Balles, die in dem aktuellen Problem beschrieben wird, sich innerhalb des Ballgebietes des Falles befindet.

Insgesamt ist der Gültigkeitsbereich darin vorteilhaft, da die Gegner beziehungsweise der Ball keine exakten Positionen aufweisen müssen, damit ein Fall als Lösung angewendet werden kann. Wie R. Ros et al. weiterhin erklären [23], ist es vorteilhafter, über eine Situation zu entscheiden, in der der Verteidiger sich vor dem Ball befindet, als über eine Situation zu entscheiden, in der der Verteidiger sich in der Position $(x, y)$ befindet.

Es ist wichtig zu betrachten, dass im Retrieve Prozess die Positionen der Roboter, die Gebiete der Gegner und die Lösungsbeschreibung in Bezug auf die Ballposition im aktuellen Problem als Referenz Punkt angepasst werden.

Eigenschaften eines Falles

Eine Beschreibung eines Falles definiert also die Positionen des Balles, der Teammitglieder und der Gegner. Diese Positionen weisen symmetrische Eigenschaften bezüglich der x-Achse auf. Das heißt, dass zu einer Position $p = (x, y)$ die zur x-Achse gespiegelte

In Abbildung 3.3 ist der Fall zusehen, der aus dem Fall in Abbildung 3.3 erstellt wird. Die Positionen, die in der Beschreibung des Problems beschrieben sind, werden an der x-Achse gespiegelt. Die Position des Balles lautet daher $B = (2600, 1850)$ und die Position des Spielers $tm_1 = (2000, 2000)$. Der Gültigkeitsbereich ändert sich bei der Spiegelung nicht, sodass dieser Bereich $K = (ball : (1000, 1000))$ lautet. Die Aktionen, die in der Lösung beschrieben sind, werden ebenfalls an der x-Achse gespiegelt. Es werden jedoch nur die Parameter der Aktionen vom Typ kick und goToPosition an der x-Achse gespiegelt. Als Lösungsbeschreibung ergibt sich deshalb $A = (tm_1 : [getTheBall, goalkick])$.

R. Ros et al. speichern im Gegensatz zu der vorliegenden Arbeit bei der Problembe- schreibung die Farbe des eigenen Tores, da die Tore im Spielfeld durch unterschiedliche Farben gekennzeichnet waren [23]. Anhand dieser Information wird der Fall zusätzlich bezüglich der y-Achse gespiegelt, sodass aus einem Fall bis zu drei weitere Fälle abgeleitet werden können. Diese Eigenschaft wird in der vorliegenden Arbeit jedoch nicht berücksichtigt, da keine Unterscheidung der Tore nach Farben erfolgt.
3.3. FBS im Roboterfußball

Abbildung 3.4.: Das Beispiel stellt dar, wie der symmetrische Fall, welcher in 3.3 gegeben ist, aussieht.

Beispiel eines Falles

In Abbildung 3.5 ist ein Beispiel eines Falles gegeben. Die Beschreibung der Problems ist wie folgt gegeben:

\[
P = \begin{cases} 
    B = (-500, 500) \\
    Tm = \{tm_1 : (-1000, 1000), tm_2 : (2000, -1000)\} \\
    Opp = \{opp_1 : (2000, 2000)\}
\end{cases}
\]

Die Lösungsbeschreibung \( A \) des Beispieles ist wie folgt definiert:

\[
A = \left( \begin{array}{c}
    tm_1 : [getTheBall, passTheBall(2)] \\
    tm_2 : [receiveTheBall(1), goalKick]
\end{array} \right)
\]

Der Gültigkeitsbereich \( K \) wird beschrieben durch:

\[
K = (ball : (500, 500), opp_1 : (250, 500))
\]

Das Beispiel zeigt eine Lösungsmethode, die durch einen Pass charakterisiert wird. Der Spieler \( tm_1 \) nähert sich zuerst dem Ball. Anstatt alleine auf das gegnerische Tor mit dem Ball zuzulaufen und einen Torschuss durchzuführen, findet ein Pass des Balles zu dem Spieler \( tm_2 \) statt. Würde der Spieler \( tm_1 \) versuchen, alleine ein Tor zu erzielen, besteht die Gefahr, dass der Gegner \( opp_1 \) die erfolgreiche Ausführung verhindern kann, da er sich in der Nähe des Balles befindet. Findet ein Pass zu Mitspieler \( tm_2 \) statt, so ist der Gegner \( opp_1 \) zu weit von dem Ball entfernt, um einen Torschuss zu verhindern.
3. Fallbasiertes Schließen (FBS)

Abbildung 3.5.: Der Fall zeigt eine Spielsituation und wie diese durch einen Pass gelöst wird.

3.3.2. Auswahl eines Falles

In diesem Abschnitt wird dargestellt, wie ein geeigneter Fall aus einer gegebenen Fallbasis ausgewählt wird, um ein vorliegendes Problem zu lösen. Damit ein Fall als geeigneter Fall ausgewählt werden kann, muss eine gewisse Ähnlichkeit zwischen diesem und dem aktuellen Problem vorhanden sein. Diese Ähnlichkeit begründet, dass der ausgewählte Fall eine hilfreiche Lösungsmethode für das aktuelle Problem besitzt. Da in einer Fallbasis mehrere Fälle als geeignete Fälle gegeben sein können, wird aus der Menge dieser geeigneter Fälle der Fall ausgewählt, der dem aktuellen Problem am ähnlichsten ist. Weiterhin wird bei der Auswahl eines Falles berücksichtigt, dass möglichst wenige Anpassungskosten entstehen, den Fall an das Problem anzupassen. Schließlich wird bei der Auswahl eines Falles dessen Anwendbarkeit der Lösung noch überprüft. Das heißt, dass kontrolliert wird, ob die Bewegungsbahn des Balles in der Lösung frei von Gegner ist. Zum Beispiel kann ein Pass zwischen zwei Spielern nur erfolgen, wenn sich kein Gegenspieler in der Passbahn befindet [23].

Im Folgenden wird die Berechnung des Ähnlichkeitsmaßes erläutert, die die Ähnlichkeit zwischen einem Fall und einem Problem darstellt.

Ähnlichkeitsmaß

Zur Berechnung des Ähnlichkeitsmaßes wird die Position des Balles, die in dem aktuellen Problem beschrieben wird, mit der Position des Balles, die in dem Fall definiert ist, verglichen. Es wird dazu überprüft, ob sich die Ballposition aus dem Problem sich innerhalb des Gültigkeitsbereiches des Falles befindet. Dazu wird eine Gaussian Funktion benutzt, um die Übereinstimmung zu berechnen.
Wie bereits erwähnt wird das Spielfeld durch ein kartesisches Koordinatensystem dargestellt. Die Position eines Balles besteht somit aus einer x-Koordinate und einer y-Koordinate. Die Funktion zur Berechnung der Übereinstimmung lautet wie folgt:

\[
\text{sim}_B(x^P_B, y^P_B, x^C_B, y^C_B) = G(x^P_B - x^C_B, y^P_B - y^C_B) = \exp \left( - \left( \frac{x^P_B - x^C_B}{\tau^x_B} \right)^2 + \left( \frac{y^P_B - y^C_B}{\tau^y_B} \right)^2 \right)
\]

wobei \((x^P_B, y^P_B)\) der Position des Balles im Problem und \((x^C_B, y^C_B)\) der Position des Balles im Fall entspricht. Die Radien des Gültigkeitsbereiches der Ballposition entsprechen den Variablen \(\tau^x_B\) und \(\tau^y_B\).

Die Funktion \(\text{sim}_B(x^P_B, y^P_B, x^C_B, y^C_B)\) gibt die Ähnlichkeit zwischen der Ballposition eines Problems und eines Falles an. Der Wert des Maßes, der mit dieser Funktion berechnet wird, liegt dabei in dem Intervall \([0, 1]\). Es gilt, dass je größer der Abstand der Positionen \((x^P_B, y^P_B)\) und \((x^C_B, y^C_B)\) ist, umso geringer der Wert der Funktion \(\text{sim}_B(x^P_B, y^P_B, x^C_B, y^C_B)\) ist. Entsprechend gilt, dass je kleiner der Abstand der Positionen ist, umso größer der Wert der Funktion \(\text{sim}_B(x^P_B, y^P_B, x^C_B, y^C_B)\) ist. Zwei Positionen werden als ähnlich betrachtet, wenn sich das Ergebnis des Ähnlichkeitsmaßes \(\text{sim}_B(x^P_B, y^P_B, x^C_B, y^C_B)\) einen Grenzwert nicht unterschreitet. Wie in [23] wird als Grenzwert 0.367 verwendet. Das Ähnlichkeitsmaß zwischen der Ballposition \((x^C_B, y^C_B)\) des Falles mit dem Gültigkeitsbereich, der durch die Radien \(\tau^x_B\) und \(\tau^y_B\) gegeben ist, und einer Ballposition \((x^P_B, y^P_B)\) im Problem mit \((x^P_B, y^P_B) = (x^C_B + \tau^x_B, y^C_B + \tau^y_B)\), die genau auf dem Rand des Gültigkeitsbereiches liegt, entspricht dem Grenzwert 0.367.

Das Kostenmaß

Bevor eine Lösung eines ausgewählten Falles ausgeführt werden kann, muss der ausgewählte Fall an das aktuelle Problem angepasst werden. Zum einen wird dazu eine passende Zuordnung der Spieler gesucht. Das heißt, dass jedem Roboter, der in dem Fall beschrieben ist, genau einem Roboter aus dem aktuellen Problem zugeordnet wird. Jeder Roboter des Problems, der einem Roboter im Fall zugeordnet ist, wird bei der Ausführung der Lösung dessen Aktionen ausführen. Zum anderen werden die Anpassungskosten berechnet. Die Anpassungskosten entsprechen der Summe aller Distanzen zwischen den Positionen der Mitspieler im Problem und den zugeordneten Positionen der Mitspieler im Fall. Als Funktion zur Berechnung dieser Kosten wird die folgende Funktion verwendet:

\[
\text{cost}(p, c) = \sum_{i=1}^{n} \text{dist} \left( (x^P_i, y^P_i), \text{adaptPos}_i \right)
\]

wobei \(\text{dist}\) der euklidischen Distanz zwischen der Position \((x^P_i, y^P_i)\) aus dem aktuellen Problem und der zugeordneten Position \(\text{adaptPos}_i\) aus dem Fall ist. Index \(n\) entspricht hierbei der Anzahl der Roboters des eigenen Teams, die in dem Fall beschrieben sind [23].
Zur Bestimmung der Zuordnung der Roboter wird die Zuordnung gesucht, die die geringsten Anpassungskosten verursacht. Da in einem Team nur fünf Roboter als Feldspieler erlaubt sind, werden zur Bestimmung der kostengünstigsten Zuordnung alle möglichen Zuordnungen überprüft. Die Zuordnung, die die geringsten Kosten erzeugt, wird als Zuordnung der Roboter ausgewählt. Die Zuordnung der Roboter ist notwendig, damit eindeutig definiert wird, welche Roboter aus dem aktuellen Problem, welche Aktionen, die in der Lösung beschrieben sind, ausführen soll. Jedem Roboter aus einem Fall kann genau nur einem Roboter aus dem Problem zugeordnet werden und weiterhin kann ein Roboter aus dem Problem höchstens einem Roboter aus dem Fall zugeordnet werden. Daher ist eine Zuordnung der Roboter nur möglich, wenn im Fall nicht mehr Roboter beschrieben sind, als Roboter im aktuellen Problem vorhanden sind [23].

Maß für die Anwendbarkeit


Die Ähnlichkeit der Gegner

Die Ähnlichkeit der Gegner entspricht der Anzahl der Bereiche der Gegner, die in einem Fall beschrieben sind und in dem aktuellen Problem von mindestens einem Gegner besetzt sind. Diese Ähnlichkeit dient dazu, aus der Menge aller möglichen Kandidaten den Fall auszuwählen, in dem die Ähnlichkeit der Gegner möglichst groß ist. Die Ähnlichkeit der Gegner wird mithilfe folgender Funktion [23] berechnet:

\[
\text{sim}_{\text{opp}}(p, c) = \left| \left\{ \text{scp}_j \mid \text{scp}_j \in \text{Sscp}, \exists \text{opp}_i \in \text{Opp} \left( \Omega_{\text{scp}_j}(x^P_i, y^P_i) > \text{thr}_{\text{opp}} \right) \right\} \right|
\]

wobei \(\Omega_{\text{scp}_i}\) gegeben ist durch

\[
\Omega_{\text{scp}_i} = G(x^P_B - x^C_B, y^P_B - y^C_B) = \exp \left( - \left[ \left( \frac{x^P_B - x^C_B}{\tau^x_B} \right)^2 + \left( \frac{y^P_B - y^C_B}{\tau^y_B} \right)^2 \right] \right)
\]

Die Menge der Bereiche der Gegner, die im Fall \(c\) definiert sind, ist mit \(\text{Sscp}\) dargestellt. Die Radien der Bereiche sind durch \(\tau^x_j\) und \(\tau^y_j\) und die Mittelpunkte dieser Bereiche sind durch \(x^C_j\) und \(y^C_j\) gekennzeichnet. Die Variable \(\text{Opp}\) kennzeichnet die Menge der
Positionen der Gegner. Die Position des Gegner \(opp_i\) im Problem ist durch \((x^P_i, y^P_i)\) gegeben. Als Grenzwert \(thr_{opp}\) wird wie in \[23\] der Wert 0.367 benutzt.

Die Formel \(sim_{opp}(p,c)\) gibt also die Anzahl der Bereiche der Gegner zurück, in denen mindestens ein Gegner vorhanden ist. Dazu wird zu jedem Bereich überprüft, ob ein Gegner aus der gegebenen Menge aller Gegner des Problems sich innerhalb des Bereiches befindet. Zur Berechnung des Abstandes wird der euklidische Abstand berechnet. Mit der Gaussian Funktion wird sicher gestellt, dass sich die Position des Gegners sich innerhalb des Bereiches befindet. Es wird angenommen, dass ein Gegner sich innerhalb eines Bereiches befindet, wenn der Wert der Funktion \(\Omega_{scp}\) größer oder gleich dem Grenzwert \(thr_{opp}\) ist.

Insgesamt betrachtet stellt die Ähnlichkeit der Gegner kein kritisches Merkmal dar, welches entscheidet, ob ein Fall als möglicher Kandidat betrachtet werden kann. Statt dessen wird diese Ähnlichkeit zur Sortierung der Kandidaten eingesetzt. Nachdem alle möglichen Kandidaten aus der Fallbasis bestimmt sind, werden diese Kandidaten unter anderem anhand der Ähnlichkeit der Gegner sortiert.


**Bewegungsbahn des Balles**


Da beim Roboterfußball eine gewisse Ungenauigkeit vorhanden ist und somit ein Schuss oder ein Pass nicht genau ausgeführt werden kann, wird eine Trajektorie durch einen Bereich dargestellt. Dieser Bereich ist durch eine Länge und eine Breite gegeben. Es kann überprüft werden, ob ein Gegner sich innerhalb dieses Bereiches befindet. Dazu muss sich der Gegner in der Nähe der Bewegungsbahn befinden. Mit Hilfe folgender Funktion wird überprüft, ob ein Gegner sich innerhalb der Bewegungsbahn befindet:
Abbildung 3.6.: Die Trajektorie einer Ballbewegung wird durch eine Start- und eine Endposition aufgestellt.

\[ \mu_{t_j}(x, y) = \exp \left( - \left( \frac{y}{p(x, r_{min}, r_{max}, l)} \right)^2 \right) \]

Die Trajektorie ist durch einen minimalen Radius \( r_{min} \) und einem maximalen Radius \( r_{max} \) sowie der Länge \( l \) gegeben. Die Funktion \( p \) berechnet die Breite der Trajektorie an der Stelle \( x \). Damit ein Gegner sich innerhalb der Trajektorie befindet, muss der Wert der Funktion \( \mu_{t_j} \) den Grenzwert 0.367 überschreiten [23].

In Abbildung 3.6 ist eine Trajektorie illustriert. Die Fläche der Trajektorie entspricht einem gleichschenkligen Trapez und wird durch die Positionen \( S \) und \( D \) aufgestellt. Die Höhe des Trapez \( l \) entspricht dem Abstand zwischen den Positionen \( S \) und \( D \). Die Länge der kürzeren Grundseite entspricht dem doppelten minimalen Radius \( r_{min} \) und die Länge der längeren Grundseite entspricht dem doppelten maximalen Radius \( r_{max} \).

Die gesamte Bewegungsbahn des Balles kann nun mit Hilfe der Funktion \( \mu_{t_j} \) überprüft werden, ob sich in einer Trajektorie ein Gegner befindet. Dies wird mit folgender Funktion überprüft:

\[ FreePath(p, c) = 1 - \max_{t_j \in T} \left( \phi_{t_j}(Opp) \right) \]

wobei \( \phi_{t_j} \) gegeben ist durch

\[ \phi_{t_j} = \begin{cases} 1, & \exists opp_i \in Opp(\mu_{t_j}(x_i, y_i) > thr_i) \\ 0, & otherwise \end{cases} \]

Die Folge von Trajektorien des Balles ist durch \( T \) und die Menge der Gegner Positionen im Problem \( p \) durch \( Opp \) gegeben. Der betrachtete Fall ist durch \( c \) gegeben. Die Position des Gegners \( opp_i \) aus der Menge \( Opp \) ist durch \((x_i, y_i)\) dargestellt. Die Funktion \( \mu_{t_j} \)
gibt einen Wert aus dem Intervall \([0, 1]\) zurück und stellt die Ähnlichkeit zwischen der Position des Gegners und der Trajektorie des Balles dar. Insgesamt dient die Funktion \(FreePath(p, c)\) dazu, die gesamte Bewegungsbahn des Balles, die durch die Lösungsbeschreibung definiert wird, zu überprüfen und sicherzustellen, dass sich kein Gegner in dieser Bewegungsbahn befindet. Zu jeder einzelnen Trajektorie der Bewegungsbahn wird berechnet, ob ein Gegner die Trajektorie besetzt. Ist mindestens eine Trajektorie durch einen Gegner besetzt, so gibt die Funktion \(FreePath(p, c)\) den Wert 0 zurück. Sonst wird der Wert 1 zurück gegeben [23].

Das Merkmal der Ausführbarkeit eines Falles wird mit der Funktion \(FreePath(p, c)\) berechnet. Dieses Merkmal stellt im Gegensatz zum Merkmal der Ähnlichkeit der Gegner ein kritisches Merkmal dar, da die Bewegungsbahn des Balles frei von Gegner sein muss, damit die entsprechende Lösung ausgeführt werden kann. Die Ähnlichkeit der Gegner dient dazu, die Menge der geeigneten Fälle zu sortieren, um den geeignetsten Fall zu bestimmen [23].

**Auswahl geeigneter Fälle**

In einer neuen Spielsituation, die ein Problem darstellt, wird versucht, mit Hilfe von bereits gesammelten Erfahrungen eine geeignete Lösung zu bestimmen. Dazu werden die Erfahrungen, die als Fälle in der Fallbasis gespeichert sind, durchsucht. Fälle, die eine gewisse Ähnlichkeit zu der aktuellen Spielsituation aufweisen, werden als sogenannte **Kandidaten** bezeichnet. Es ist möglich, dass zu einer Spielsituation kein Fall als Kandidat gefunden wird. Dann wird ein sogenanntes **Standardverhalten** von den Robotern ausgeführt. Aber es ist ebenso erreichbar, dass mindestens ein Fall als Kandidat ausgemacht wird. In dieser Situation wird der Fall der Kandidaten bestimmt, der zum einen eine hohe Ähnlichkeit der gegebenen Spielsituation besitzt und zum anderen auch wenige Anpassungskosten verursacht.


Bei der Filterung der Kandidaten bildet die Fallbasis den Suchraum, in dem die Kandidaten bestimmt werden. Wie R. Ros auffassen [23], wird dieser Suchraum so schnell wie möglich reduziert, aufgrund der Echtzeitanforderungen und der beschränkten Rechenressourcen. Ein Fall wird als Kandidat bezeichnet, wenn folgende Bedingungen erfüllt sind:

- Die Anzahl der Spieler, die in dem Fall beschrieben sind, ist nicht größer als die Anzahl der Spieler, die in dem Problem vorhanden sind. Diese Bedingung ist notwendig, damit eine Zuordnung der Spieler gefunden werden kann.

- Das Ähnlichkeitsmaß ist erfüllt. Das bedeutet, dass sich die Ballposition des aktuellen Problems sich innerhalb der Gültigkeitsbereiches befindet.
• Die Distanzen, die jeder Spieler zu der entsprechenden Startposition zurücklegen muss, überschreiten nicht den gegebenen Grenzwert \( thr_c \). Jeder Spieler wird die ihm angepasste Position einnehmen. Die Distanz, die dazu zurückgelegt werden muss, soll einen Grenzwert nicht überschreiten.

• Die Bedingung \( FreePath \) muss erfüllt sein. Es wird hiermit sichergestellt, dass die Gegner die Ausführung der Lösung nicht stören werden. Befindet sich ein Gegner innerhalb der Bewegungsbahn des Balles, so ist die Ausführung der Lösung nicht möglich.

In Algorithmus \([1]\) ist zusehen, wie die Filterung der Kandidaten die obigen Bedingungen überprüft. Der Algorithmus wurde aus der Arbeit von R. Raquel et al. \([23]\) übernommen und entsprechend angepasst. Der Algorithmus \( IsCandidate(problem, case) \) benötigt als Eingabeparameter eine Problembeschreibung \( problem \) und einen Fall \( case \). Zunächst wird in Zeile 2 überprüft, ob in der Problembeschreibung \( problem \) weniger Spieler vorhanden als in dem Fall \( case \). Wenn in der Problembeschreibung weniger Spieler vorhanden sind, dann ist die Bedingung nicht erfüllt und der Algorithmus terminiert. Ist die Bedingung erfüllt, dann wird in Zeile 4 die Bedingung des Ähnlichkeitsmaßes überprüft. Hierzu wird die Funktion \( sim_B \) mit der Ballposition \( ball_{problem} \) aus der Problembeschreibung und der Ballposition des Falls \( ball_{case} \) aufgerufen. Es wird überprüft, ob die Funktion \( sim_B \) einen Wert größer als der Grenzwert \( thr_b \) zurückliefert. Anschließend wird in Zeile 5 die beste Zuordnung, in der die geringsten Anpassungskosten verursacht werden, gesucht. Anhand dieser Zuordnung wird in Zeile 7 jede einzelne Distanz der Spieler zu den entsprechenden, angepassten Startpositionen bezüglich dem Grenzwert \( thr_c \) untersucht. Schließlich wird in Zeile 11 die Bewegungsbahn des Balles \( ballpath \) anhand der Problembeschreibung \( problem \), des Falles \( case \) und der zuvor berechneten Zuordnung \( mapping \) bestimmt. Diese Bewegungsbahn wird in Zeile 12 kontrolliert, ob diese frei von Gegnern ist und somit die Bedingung \( FreePath \) erfüllt. Sind alle Bedingungen erfüllt, dann wird der betrachtete Fall als Kandidat bezeichnet. Der Algorithmus gibt \( True \) zurück, wenn der Fall \( case \) als Kandidat in Frage kommt. Ansonsten gibt der Fall \( False \) zurück. Aufgrund der beschränkten Ressourcenleistungen eines Roboters wird in dem Algorithmus \([1]\) sofort \( False \) zurückgegeben, sobald eine Bedingung nicht erfüllt ist.

Nachdem mit der Methode der Filterung alle möglichen Kandidaten bestimmt sind, wird diese Liste der Kandidaten nach einem Ranking sortiert. Das erste Element dieser sortierten Liste stellt den geeignetsten Kandidaten dar, der im \( Retrieve \) Prozess zurückgegeben wird. In Algorithmus \([2]\) ist der \( Retrieve \) Algorithmus nach R. Raquel et al. \([23]\) illustriert, welcher als Eingabeparameter eine Problembeschreibung \( problem \) und eine Fallbasis \( case Base \) erwartet. Zu Beginn werden alle Kandidaten aus der Fallbasis bestimmt. Dazu wird zu jedem Fall und dem gegeben Problem der Algorithmus \( IsCandidate \) aufgerufen, um zu prüfen, ob der Fall ein Kandidat darstellt. Alle Kandidaten werden in der Liste \( candidate \) zusammengefasst. In Zeile 2 wird die Liste \( candidate \) sortiert und als Liste mit der Bezeichnung \( ordered\_list \) gespeichert. Wie in Zeile 3 zu sehen ist, stellt das erste Element der Liste \( ordered\_list \) den geeignetsten Fall \( retrieved\_case \) dar. Am Ende des Algorithmus wird der ausgewählte Fall \( retrieved\_case \) zurückgegeben.
Algorithmus 1 IsCandidate(problem, case)

1: if problem.numberOfTeammates < case.numberOfTeammates then
   2: return False
3: else
4:   if sim\_B(ball\_problem, ball\_case) > thr\_B then
5:     mapping = findBestMapping(problem, case)
6:     for all (tm\_problem, tm\_case) ∈ mapping do
7:       if dist(tm\_problem, tm\_case) > thr\_c then
8:         return False
9:     end if
10:   end for
11:   ballpath = determineBallpath(mapping, problem, case)
12:   if FreePath(ballpath, problem) == 1 then
13:     return True
14:   else
15:     return False
16:   end if
17: else
18:   return False
19: end if
20: end if

Algorithmus 2 Retrieve(problem, caseBase)

candidate ← selectAllCandidates(problem, caseBase)
2: ordered\_list ← sort(candidates)
3: retrieved\_case ← first(ordered\_list)
4: return ret\_case
Die Sortierung in dem Algorithmus \(^{2}\) benutzt ein bestimmtes Ranking. Dieses Ranking berücksichtigt folgende Kriterien:

**Anzahl der Spieler** Die Anzahl der Spieler gibt an, wie viele Spieler an der Lösung beteiligt sind. Beteiligte Spieler werden in der Lösungsbeschreibung des Falles aufgeführt. Wie R. Ros et al. erläutern \(^{23}\), ist ein kooperatives Verhalten gegenüber einem individuellen Verhalten vorzuziehen. Daher wird ein Fall bevorzugt, in dem eine höhere Anzahl an Spieler vorhanden ist.

**Anzahl der besetzten Bereiche der Gegner** Diese Anzahl gibt an, wie viele Bereiche der Gegner, die in dem Fall durch die Positionen und den Radien definiert sind, durch jeweils mindestens einem Gegner besetzt sind. Je mehr Bereiche besetzt sind, desto ähnlicher sind der Fall und das aktuelle Problem.

**Verhältnis zwischen den Anpassungskosten und dem Ähnlichkeitsmaß** Ein Fall wird gesucht, der zum einen möglichst geringe Anpassungskosten besitzt und zum anderen eine hohe Ähnlichkeit mit dem gegebenen Problem besitzt. Wie R. Ros erklären \(^{23}\), ist es besser, einen anderen Fall auszuwählen, der eine niedrigere Ähnlichkeit und weniger Anpassungskosten besitzt, wenn die Ähnlichkeit eines Falles zu dem Problem sehr groß ist und die Kosten ebenfalls hoch sind. Es werden dazu folgende Intervalle betrachtet:

\[
H = [0.8, 1.00] \quad h = [0.6, 0.8] \quad l = [0.4, 0.6] \quad L = (0.0, 0.4)
\]

Weiterhin werden die Kandidaten in die Listen \(int_H, int_h, int_l\) und \(int_L\) aufgeteilt. Die Bezeichnungen der Listen \(H, h, l\) und \(L\) entsprechen den obigen Intervallen. In jeder Liste sind die Kandidaten enthalten, deren Ähnlichkeitsmaß sich in dem entsprechenden Intervall befindet. Innerhalb der vier Listen werden die Kandidaten nach den Anpassungskosten sortiert \(^{23}\).

Die Reihenfolge, in der die Kriterien zur Sortierung der Kandidaten angewendet werden, ist wie folgt gegeben:

1. Anzahl der Spieler
2. Anzahl der besetzten Bereiche der Gegner
3. Verhältnis zwischen den Anpassungskosten und dem Ähnlichkeitsmaß

Wie R. Ros et al. erwähnen \(^{23}\), ist es möglich, die Reihenfolge der Anwendung der Kriterien zu ändern. Die am geeignetsten Reihenfolge ist oben gegeben. Dies lässt sich dadurch begründen, dass eine Lösung, an der mehrere Roboter beteiligt sind, vorzuziehen ist. Bei einer Lösung, an der nur wenige Roboter Aktionen ausführen, müsste für die nicht aufgeführten Robotern Aktionen noch berechnet werden.
4. Implementierung

Das Ziel der vorliegenden Bachelorarbeit ist es, eine Methode des FBS zu implementieren, die beim Roboterfußball in der SPL auf der Roboterplattform NAO eingesetzt werden kann. Die Anforderungen, wie ein möglicher Ansatz aussieht, sind in Abschnitt 3.3 bereits gegeben. Im folgenden Kapitel wird erklärt, wie die Operationen zusammenarbeiten und wie die Methode implementiert ist. Zunächst wird in Abschnitt 4.1 erörtert, welche Anforderungen bei der Implementierung berücksichtigt werden müssen. Anschließend folgt in Abschnitt 4.2 wie die einzelnen Klassen und Funktionen implementiert und zusammengesetzt sind.

4.1. Technische Hilfsmittel


4.2. Technische Umsetzung

Der Ansatz der FBS Methode wird durch eine statische Bibliothek implementiert und somit anderen Projekten zur Verfügung gestellt. Die Bibliothek ist als Projekt in Microsoft Visual Studio 2013 erstellt worden und in der Programmiersprache C++ geschrieben. Das Projekt kennzeichnet sich durch die Struktur darin, dass zum einen Klassen implementiert sind, die die Grundelemente beim FBS darstellen. Zu diesen Grundelementen gehören die Problembeschreibung, die Lösungsbeschreibung, die Repräsentation eines

4. Implementierung

Gültigkeitsbereiches und somit der Fall und die Fallbasis. Diese Elemente werden im Abschnitt 3.3.1 beschrieben und nachfolgend im Abschnitt 4.2.2 durch die entsprechende Implementierung gegeben. Neben diesen Grundelementen werden noch weitere Klassen definiert, die benötigte Algorithmen verfügbar machen. Beispielsweise gehören die Operationen zur Bestimmung geeigneter Kandidaten aus einer Fallbasis sowie zur Adaption eines Falles an die Problemstellung. Diese Algorithmen-Klassen werden im Abschnitt 4.2.3 erläutert. Zu Beginn wird im nachfolgenden Abschnitt 4.2.1 die Klasse vorgestellt, mit der Funktionen implementiert sind, um ein Objekt eines Falles in eine XML-Datei zu lesen beziehungsweise aus einer XML-Datei zu lesen. Diese XML Serialisierung und Deserialisierung ist unter anderem für die Evaluation, die in Kapitel 5 folgt, notwendig.

4.2.1. XML Serialisierung und Deserialisierung


Der Aufbau einer XML-Datei, in der ein Fall gespeichert ist, ist in Listing 4.1 zu sehen. Einem Fall entsprechend wird ein Knoten case angelegt, der als Unterknoten die Knoten problemDescription, für die Problembeschreibung, und solutionDescription für die Lösungsbeschreibung des Falles besitzt.


Listing 4.1: Struktur einer XML-Datei eines Falles

```
<?xml version="1.0" encoding="utf-8"?>
<case>
  <problemDescription>
```

34
4.2. Technische Umsetzung

```xml
<ballPosition>
  <position>
    <X>...</X>
    <Y>...</Y>
  </position>
  <scope>
    <radius_x>...</radius_x>
    <radius_y>...</radius_y>
  </scope>
</ballPosition>

<teammates>
  <robot id="...">
    <position>
      <X>...</X>
      <Y>...</Y>
    </position>
  </robot>
  ...
</teammates>

<opponents>
  <robot>
    <position>
      <X>...</X>
      <Y>...</Y>
    </position>
    <scope>
      <radius_x>...</radius_x>
      <radius_y>...</radius_y>
    </scope>
  </robot>
  ...
</opponents>

</problemDescription>

<solutionDescription>
  <robot id="...">
    <action>
      <type>getTheBall</type>
    </action>
    ...
  </robot>
  ...
</solutionDescription>
</case>
```
In der Implementierung ist die Klasse `CaseBaseIO` vorhanden, die die Methoden `WriteXMLFile(...)` und `ReadCaseXMLFile(...)` zur Verfügung stellt. Diese Methoden dienen dazu, zum einen einen Fall als Objekt des Typs `Case` in eine XML-Datei zu schreiben und zum anderen einen Fall aus einer XML-Datei zu lesen.

Eine Fallbasis wird in XML-Dateien geschrieben, indem jeder enthaltende Fall in eine eigene XML-Datei geschrieben wird. Die XML-Dateien werden dazu in einem Ordner auf der Plattform gespeichert. Die Fallbasis wird gelesen, indem die XML-Dateien, die in einem Ordner vorhanden sind, gelesen werden. Im Fußballspiel hat somit der Roboter die Möglichkeit, die Fallbasis, die auf der Plattform vorhanden ist, zu lesen und zu benutzen.


### 4.2.2. Objektklassen


Abbildung 4.1.: UML-Klassendiagramm
4.2. Technische Umsetzung

Klasse einer Position

Die Klasse *Position* enthält zwei Attribute, die die x-Koordinate und die y-Koordinate der Position enthalten. Weiterhin besitzt diese Klasse unter anderem folgende Operationen:


**Adapt** Die Operation *Adapt* ändert die Position. Es werden zwei Parameter erwartet, die zu der x-Koordinate und der y-Koordinate der Position addiert werden. Diese Operation ist notwendig, um Positionen im Prozess der *Adaption* anzupassen.

**DetermineSimilarity** Die Operation *DetermineSimilarity* berechnet das Ähnlichkeitsmaß $\text{sim}_B$. Als Parameter wird eine weitere Position mit dem Gültigkeitsbereich angegeben, um das entsprechende Maß zu berechnen. Als Rückgabe wird ein Wert im Intervall $[0, 1]$ zurückgegeben.

Klasse einer Problembeschreibung


Klasse der Lösungsbeschreibung

Die Klasse *SolutionDescription* enthält als Attribut ein Objekt des Typs *Gameplay*. Das Gameplay enthält einen Container, der jeder Roboter ID eine Folge von Aktionen zuordnet. Es sind unter anderem folgende Operationen in dieser Klasse vorhanden:

**GetRobotWhoGetFirstTheBall** Die Methode *GetRobotWhoGetFirstTheBall* dient dazu, um aus dem Gameplay die ID des Roboters zu bestimmen, der als erstes die Aktion *getTheBall* in seiner Aktionsfolge ausführt. Diese Operation ist notwendig, um die Bewegungsbahn des Balles zu berechnen.

**DetermineBallPath** Die Operation *DetermineBallPath* berechnet die Bewegungsbahn des Balles anhand der gegebenen Lösung. Weiterhin werden als Parameter die Ballposition und die Positionen der Teammitglieder angegeben. Die Bewegungsbahn
wird als Container `std::vector` implementiert und zurückgegeben, die die einzelnen Positionen der Bewegungsbahn als Elemente besitzt.

**Adapt** Die Operation `Adapt` passt die Lösungsbeschreibung an. Diese Methode ist im Schritt der Anpassung des FBS Zyklus notwendig, um einen Fall an das aktuelle Problem anzupassen.

**IsValid** Die Funktion `IsValid` überprüft, ob die Lösungsbeschreibung gültig ist. Dazu wird unter anderem kontrolliert, dass mindestens ein Roboter im Gameplay vorhanden ist, der als erstes die Aktion `getTheBall` ausführt.

### Klasse des Gültigkeitsbereiches eines Falles


### Klasse eines Falles

Die Klasse `Case` stellt einen Fall dar. Diese Basisklasse enthält als Attribute ein Objekt des Typs `SolutionDescription`, ein Objekt des Typs `ProblemDescription` und ein Objekt des Typs `CaseScopeRepresentation`. Diese Attribute stellen die Lösungsbeschreibung, die Problembeschreibung und die Repräsentation des Gültigkeitsbereiches eines Falles dar. Die Klasse besitzt die Operation `DetermineReflectedCaseOnXAxis`, die den entsprechenden, an der x-Achse gespiegelten Fall berechnet. Die Operation `Adapt` passt den Fall an die Position des Balles in dem aktuellen Problem an, indem die Distanzen bezüglich der x-Achse und der y-Achse als Parameter übergeben werden. Die Methode `DetermineBallSimilarity` berechnet die Ähnlichkeit der Ballposition zwischen dem Fall und einem Problem. Die Position des Balles wird als Parameter dieser Funktion übergeben.

### Klasse einer Fallbasis

Die Klasse `CaseBase` entspricht der Fallbasis. Dazu besitzt die Klasse als Attribut den Container `std::vector`, der als Elemente Objekte vom Typ `Case` enthält. Innerhalb der Fallbasis wird keine Indexstruktur bezüglich der Fälle vorgenommen. Neue Fälle, die der Fallbasis hinzugefügt werden, werden am Ende des Containers hinzugefügt.

### 4.2.3. Algorithmen-Klassen

In den nachfolgenden Ausführungen werden die Klassen erläutert, die die Methoden und Operationen zur Verfügung stellen, die zur Implementierung der FBS Methode nötig sind.
4.2. Technische Umsetzung

Hilfsklasse zur Adaption einer Lösung


Hilfsklasse zur Bestimmung der Kandidaten

Die Klasse HelperSelector definiert die Methode `IsCandidate(...)`, welche überprüft, ob ein Fall bei einem gegebenen Problem als möglicher Kandidat zur Lösung in Frage kommt. Mit dem Ziel der Simulation während der Bachelorarbeit, gibt diese Funktion Information zurück, ob ein Fall ein möglicher Kandidat darstellt oder nicht. Wenn der Fall kein Kandidat ist, so wird der Grund, warum der Fall kein Kandidat ist, als Information wiedergegeben. Es werden folgenden Gründe unterschieden, weshalb ein Fall nicht als Kandidat in Frage kommt:

- **NUMBEROFTEAMMATESISNOTVALID** Die Anzahl der Teammitglieder, die im Fall beschrieben sind, ist größer als die Anzahl der Teammitglieder, die in dem aktuellen Problem vorhanden sind. Somit kann keine Zuordnung der Teammitglieder erfolgen und der Fall wird nicht als möglicher Kandidat betrachtet.

- **COSTSARETOOBIG** Die Anpassungskosten, die der Fall entstehen lässt, sind groß. Zur Überprüfung werden alle Distanzen, die die Roboter zu ihren Startpositionen zurücklegen müssen, einzeln kontrolliert. Wenn mindestens eine Distanz zu groß und somit zu teuer ist, dann ist der Fall kein möglicher Kandidat.

- **BALLTRAJECTORYISNOTFREE** Die Bewegungsbahn des Balles ist nicht frei von Gegner. Deshalb kann die Lösung, die im Fall beschrieben wird, nicht in dem aktuellen Problem ausgeführt werden und somit ist der Fall kein möglicher Kandidat.

- **BALLSIMILARITYISNOTVALID** Das Ähnlichkeitsmaß der Ballposition des Falles und des aktuellen Problems ist zu gering. Daher wird der Fall nicht als Kandidat betrachtet.

Die Klasse HelperSelector definiert weiterhin folgende Funktionen:


**DistancesAreValid** Die Operation überprüft, ob die Anpassungskosten gültig sind. Dazu wird überprüft, ob alle Distanzen, die die Roboter zu ihren entsprechenden Startpositionen überwinden müssen, den gegebenen Grenzwert nicht überschreiten.

**DetermineBallPath** Die Funktion *DetermineBallPath* berechnet anhand der Zuordnung der Roboter, der Positionen der Roboter und der Lösungsbeschreibung die Bewegungsbahn des Balles. Die Implementierung dieser Operation ist in Listing C.1 dargestellt.

**Hilfsklasse zur Sortierung der Kandidaten**


**Retrieve-Algorithmus**

5. Evaluation

Das Ziel der vorliegenden Bachelorarbeit ist die Implementierung und Ausarbeitung eines FBS Ansatzes, der beim Roboterteam auf der NAO Robotik Plattform eingesetzt wird. Das folgende Kapitel stellt eine Evaluation dar, ob das Ziel erreicht wurde und ob die im Spiel gestellten Echtzeitanforderungen erfüllt werden können.

Der Retrieve-Schritt beim FBS durchsucht die ganze Fallbasis nach dem geeignetsten Fall. Die Fallbasis stellt somit den Suchraum dar, in dem der Fall vorhanden ist. Bei der Implementierung des Retrieve-Algorithmus wird der Suchraum so schnell wie möglich reduziert, in dem zuerst eine Filterung nach möglichen Kandidaten und anschließend erst die Sortierung der Kandidaten stattfindet. Aufgrund der Filterung wird der Suchraum schnell reduziert, was die Laufzeit bei der Sortierung reduzieren kann.

5.1. Laufzeitkomplexität

Algorithmus 3 FindBestMapping(problem, case)

```
adaptionCosts = ∞
bestMapping = NULL
3: for all mapping ∈ setOfAllPossibleMappings(problem, case) do
    if mapping.getAdaptionCosts < adaptionCosts then
        bestMapping = mapping
    end if
6: end for
return bestMapping
```

In Algorithmus 3 ist der Algorithmus illustriert, der die kostengünstigste Zuordnung zwischen den Roboter aus dem aktuellen Problem und dem Fall berechnet. Der Algorithmus erwartet als Eingabeparameter das Problem `problem` und den Fall `case`. Die Laufzeitkomplexität dieses Algorithmus wird durch die `for`-Schleife von Zeile drei bis Sechs bestimmt. Mit der `for`-Schleife werden zu jeder möglichen Zuordnung die Anpassungskosten mit der Hilfsvariable `adaptionCosts` verglichen, ob die der betrachtete Zuordnung weniger Kosten verursacht. Ist die Bedingung in der vierten Zeile erfüllt, so wird die betrachtete Zuordnung `mapping` in die Variable `bestMapping` geschrieben. Am Ende des Algorithmus wird die Variable `bestMapping` zurückgegeben, die der kosten-günstigsten Zuordnung entspricht. Die Menge `setOfAllPossibleMappings(problem, case)` besitzt die Kardinalität `m`. Der Algorithmus `FindBestMapping(problem, case)` ist daher durch folgende Komplexität gekennzeichnet:

\[ \mathcal{O}(m) \]

In dem aktuellen Regelwerk [21] wird angegeben, dass ein Roboterteam aus nicht mehr als fünf Robotern besteht. Mit der folgenden Formel kann die Kardinalität der Menge
setOfAllPossibleMappings(problem, case) berechnet werden:

\[ m = V_n^{(k)} = n(n-1)(n-2)\ldots(n-k+1) \]

Die Variable \( n \) bezeichnet die Anzahl der Roboter, die in einem aktuellen Problem vorhanden sind, und die Variable \( k \) kennzeichnet die Anzahl der Roboter, die in einem Fall definiert sind. Da in dem Roboterteam nicht mehr als fünf Feldspieler vorhanden sind, gilt weiterhin \( n \leq 5 \) und \( k \leq 5 \). Zudem wird \( n \leq k \) angenommen, weil in dem aktuellen Problem mindestens so viele Roboter vorhanden sein müssen, wie Roboter in dem Fall definiert sind. Ansonsten kann keine Zuordnung zwischen den Robotern gefunden werden. Insgesamt konvergiert die Kardinalität also gegen

\[ m = V_n^{(5)} = 5(5-1)(5-2)\ldots(5-5+1) = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120 \]

Der Algorithmus 4 zeigt den Algorithmus, der aus einer Fallbasis caseBase bei dem gegebenen Problem problem alle möglichen Fälle bestimmt, die das Problem lösen können und als Kandidaten bezeichnet werden.

Algorithmus 4 SelectCandidates(problem, caseBase)

for all case \( \in \) caseBase do
    if isCandidate(problem, case) == TRUE then
        listOfCandidates.push_back(case)
    end if
    if checkReflectedCase then
        reflectedCase = getReflectedCase(case)
        if isCandidate(problem, reflectedCase) == TRUE then
            listOfCandidates.push_back(reflectedCase)
        end if
    end if
end for
12: return listOfCandidates

Die Anzahl der Fälle, die in der Fallbasis caseBase vorhanden sind, werde mit \( cb \) bezeichnet. In dem Algorithmus 4 ist nur eine for-Schleife vorhanden. Zu Beginn der for-Schleife wird zunächst überprüft, ob der betrachtete Fall case aus der Fallbasis ein Kandidat ist. Wenn die Bedingung in der zweiten Zeile erfüllt ist, dann wird der betrachtete Fall in die Liste der Kandidaten hinzugefügt. Von der fünften bis zur zehnten Zeile wird der entsprechende, zur x-Achse gespiegelte Fall auf die Eigenschaft kontrolliert, ob dieser ebenfalls ein Kandidat darstellt. Da in dem Algorithmus 4 nur eine for-Schleife vorhanden ist, die über die Fälle der Fallbasis terminiert, gilt für die Laufzeitkomplexität des Algorithmus folgendes:

\[ O(cb) \]

Zur Sortierung der Liste der Kandidaten wird der Quicksort-Algorithmus verwendet, wie es in Abschnitt 4.2.3 beschrieben wird. T. Ottmann und P. Widmayer erklären,
5.2. Messung der Laufzeit

dass für die Laufzeitkomplexität des Quicksort-Algorithmus allgemein für die mittlere Laufzeit \( O(n\log(n)) \) gilt und für den schlechtesten Fall \( O(n^2) \). Die Anzahl der Fälle der Fallbasis wird wie bereits erwähnt mit \( cb \) gekennzeichnet. Da zur Bestimmung der Kandidaten sowohl die Fälle als auch die symmetrischen Fälle betrachtet werden können, gilt für die maximale Anzahl an Kandidaten \( c = cb + cb \). Weil ebenso der Quicksort-Algorithmus die Liste der Kandidaten sortiert, gilt für die Komplexität im Durchschnitt

\[
O((cb + cb) \cdot \log(cb + cb)) = O(2 \cdot cb \cdot \log(2 \cdot cb))
\]

und im schlechten Fall

\[
O((cb + cb)^2) = O(4 \cdot cb^2)
\]

5.2. Messung der Laufzeit


In der folgenden Abbildung ist die Tabelle dargestellt, die die Messwerte von 100 Durchläufen angibt. Der Index der Messung beziehungsweise des Durchlaufs wird auf der horizontalen Achse dargestellt. Die entsprechende Zeitdauer des Retrieve-Algorithmus wird in der vertikalen Achse in Sekunden dargestellt.

In der Tabelle in Abbildung ist zu erkennen, dass während der vierten Messung die Zeitdauer des Retrieve-Algorithmus mit 62,685 Sekunden das Maximum darstellt. Die Zeitdauer der Messung beträgt hingegen nur 19,289 Sekunden. Das arithmetische Mittel der gegebenen zehn Messwerte ergibt:

\[
x_{arithm} = 26,19
\]

Als Varianz berechnet sich folgender Wert:

\[
s^2 = 42,65
\]

Die Standardabweichung der Messwerte ergibt:

\[
s = \sqrt{s^2} = 6,53
\]
5. Evaluation

Abbildung 5.1.: Messwerte der Laufzeit des Retrieve-Algorithmus

Der Median der Messwerte beträgt 25.09 Sekunden.

Die Auswertung der Messwerte zeigt, dass die Laufzeit des Retrieve-Algorithmus in angemessener Zeit erfolgt. Das arithmetische Mittel liegt bei $x_{arithm} = 26,19$ Sekunden und die Standardabweichung beträgt $s = 6,53$ Sekunden. Die Dauer der Laufzeit ist hiermit für eine Verwendung des Retrieve-Algorithmus im Roboterfußball geeignet.


5.3. Auswertung der Fallbasis

In dieser Auswertung wurde eine Fallbasis bestehend aus 20 Fällen manuell erstellt. Hiermit soll überprüft werden, ob eine Fallbasis bestehend aus 20 Fällen bereits angemessen im Fußballspiel eingesetzt werden kann. Es besteht die Herausforderung darin, die Fallbasis möglichst gering zu halten, damit die Laufzeit, einen Fall aus der Fallbasis auszuwählen, möglichst gering bleibt. Bisher wird keine Indizierung bei der Speicherung der Fälle berücksichtigt. Stattdessen wird bei einem Aufruf des Retrieve-Algorithmus die vollständige Fallbasis nach möglichen Kandidaten durchsucht.

Zur Untersuchung wurde eine Konsolenanwendung erstellt. Weiterhin wurden entsprechend 20 Fälle erstellt, die Standardsituationen, die im Roboterfußball auftreten können, repräsentieren. Zur Untersuchung der Fallbasis wurde wiederholt eine Problemstellung zufällig erstellt, die eine Spielsituation im Roboterfußball darstellt.

Die Problembeschreibung wurde auf folgende Art zufällig erstellt. Die Anzahl der Roboter des eigenen Teams wurde durch einen Wert aus dem Intervall [1,4] zufällig
bestimmt. Anschließend wurden die Roboter zufällig positioniert. Diese Auswertung be-
rücksichtigt nicht das gesamte Feld, da die Fallbasis nur für die Angriffssituationen entwic-
kelt wurden. Die Fallbasis wurde nicht unter Berücksichtigung des gesamten Spielfeldes
entwickelt, da die Situationen, in denen das Team das eigenen Tor verteidigen muss,
anders gelöst wird, als die Situationen, in denen das Team versucht, direkt ein Tor zu
erzielen. Die Roboter wurde entsprechend nur im Bereich der Spielfeldhälfte des Gegners
positioniert. Daher resultieren die zufälligen x-Koordinaten dieser Positionen aus dem
Intervall \([-500, 4500]\) und die y-Koordinaten aus dem Intervall \([-3000, 3000]\). Weiterhin
wurden die Spieler gleichmäßig auf dem Spielfeld verteilt. Die Gegner der Problembe-
schreibung wurden auf die ähnliche Weise generiert. Die Anzahl der Gegner resultiert
aus dem Intervall \([0, 4]\). Anschließend wurden die Gegner zufällig in der Spielfeldhälfte
des Gegners positioniert. Bei der Generierung der Positionen wurde darauf geachtet,
dass sich in einer Position nicht mehr als ein Roboter befindet. Die Position des Balles
wurde ebenfalls zufällig generiert, wobei die x-Koordinate aus dem Bereich \([-500, 4500]\)
und die y-Koordinate aus dem Bereich \([-3000, 3000]\) stammt.

In der Simulation wurde nun wiederholt der \textit{Retrieve}-Algorithmus mit der zufällig
generierten Problemstellung und der zuvor manuell erstellten Fallbasis ausgeführt. Insgesamt
wurde der Algorithmus 100 Mal aufgerufen. Während bei jedem Aufruf die gleiche
Fallbasis verwendet, wurde die Problemstellung zufällig neu generiert.

Mit dieser Untersuchung soll die Fallbasis untersucht und analysiert werden, ob diese
eine geeignete Fallbasis für den Roboterfußball darstellt. Es ist dabei zu achten, wie
häufig ein Fall aus der Fallbasis gefunden wird. Wird zu selten ein Fall aus der Fallbasis
gefunden, so stellt die Fallbasis möglicherweise keine geeignete Fallbasis dar. Es würde
sich ableiten lassen, dass im Roboterfußball Spielsituationen häufiger auftreten, die in der
Fallbasis nicht berücksichtigt sind. Daher müsste die Fallbasis gewartet und angepasst
werden, damit diese häufiger einen Fall als mögliche Lösung liefert.

In dieser Auswertung wird untersucht, wie viele Fälle als Kandidaten aus der Fall-
basis bei dem Aufruf des \textit{Retrieve}-Algorithmus erkannt werden und wie viele nicht als
Kandidaten betrachtet werden. Zur Überprüfung, ob ein Fall ein Kandidat darstellt,
will zusätzlich bei der Untersuchung der entsprechende, zur x-Achse symmetrische Fall
überprüft. Weiterhin wird analysiert, aus welchem Grund ein Fall nicht als Kandidat ge-
funden wird. Entsprechend der Reihenfolge der Eigenschaften, nach der überprüft wird,
ob ein Fall ein Kandidat ist, werden die vier folgende Zustände unterschieden:

- Der Fall wird aufgrund der Ähnlichkeit nicht als Kandidat betrachtet. Die Ballpo-
sition der Problemstellung und des Falles somit nicht überein.

- Die Lösung des Falles ist nicht ausführbar, da die Eigenschaft \textit{FreePath} nicht erfüllt
ist. Die Ballbewegungsbahn des Balles ist durch mindestens einen Gegner besetzt,
sodass der Fall als mögliche Lösung verworfen wird.

- Es ist keine Anpassung des Falles an die Problemstellung möglich, da im Fall
mehr Spieler des eigenen Teams definiert sind, als Spieler in der Problemstellung
vorhanden sind.
• Die Anpassungskosten, die entstehen, sind zu hoch. Daher wird der Fall nicht als möglicher Kandidat beachtet.

Zu jedem Aufruf des Algorithmus wird nun die Häufigkeit der entsprechenden Gründe analysiert, nach denen die Fälle nicht als Kandidaten ausgewählt werden. Die manuell erstellte Fallbasis deckt zur Vollständigkeit die gesamte Fläche des Spielfeldes ab.

Die Abbildung 5.2 zeigt, wie viele Fälle in der Fallbasis bei einem Aufruf des Retrieve-Algorithmus als mögliche Kandidaten gefunden werden. Es ist erkennbar, dass bei 23 der 100 Aufrufe mindestens ein Fall als Kandidat gefunden wird. Das bedeutet, dass die erstellte Fallbasis nicht alle Spielsituationen abdeckt, die beim Roboterfußball entstehen können. Daher müsste diese Fallbasis optimiert werden, indem weitere Standardspielsituationen hinzugefügt werden.

Abbildung 5.2.: Häufigkeit geeigneter Fälle

Die Abbildung 5.3 zeigt, wie viele Fälle aus der Fallbasis bei einem Aufruf des Retrieve-Algorithmus nicht als mögliche Kandidaten bezeichnet werden, da das Ähnlichkeitsmaß nicht erfüllt ist. Das heißt, dass die Ballpositionen, die im Fall und in der Problembeschreibung beschrieben sind, nicht übereinstimmen. Diese Eigenschaft stellt ein kritisches Merkmal dar, damit ein Fall die Problemstellung als Kandidat lösen kann. Als arithmetisches Mittel berechnet sich der Wert \( x_{\text{arithm}} = 26 \). Das bedeutet, dass im Durchschnitt etwa 26 Fälle nicht als Kandidat betrachtet werden können, da die Ballpositionen zu stark abweichen.

5.3. Auswertung der Fallbasis

Abbildung 5.3.: Häufigkeit nicht geeigneter Fälle (Ähnlichkeitsmaß)

Da das arithmetische Mittel sich mit $x_{arithm} = 1,33$ berechnen lässt, ist im Durchschnitt mindestens ein Fall in der Fallbasis vorhanden, der aufgrund der Anpassungskosten, die dieser verursacht, die Problemstellung nicht lösen kann.

Abbildung 5.4.: Häufigkeit nicht geeigneter Fälle (Anpassungskosten)

Die Abbildung 5.5 zeigt, wie viele Fälle aus der Fallbasis bei einem Aufruf des Retrieve-Algorithmus nicht als mögliche Kandidaten berechnet werden, da die Anzahl der Spieler in der Problemstellung zu gering ist, damit eine Zuordnung der Roboter gefunden werden kann. In der Fallbasis sind Fälle vorhanden, in denen jeweils ein, zwei oder drei Roboter vorhanden sind. Hiermit werden mit der Fallbasis die Standardsituationen
beschrieben. Um einen Torschuss zu erzielen, müssen nicht alle Roboter an der Lösung beteiligt sein. Daher wird in dieser Auswertung nur die Standardsituationen betrachtet. Wie die Messwerte zeigen, werden entweder acht Fälle als nicht geeignete Fälle betrachtet oder 28 Fälle. Als arithmetisches Mittel $x_{arithm} = 12; 28$ werden also im Durchschnitt 12 Fälle nicht zur Lösung benutzt, da diese zu viele Roboter definieren.

Die Abbildung 5.5 zeigt, wie viele Fälle aus der Fallbasis bei einem Aufruf des Retrieve-Algorithmus nicht als mögliche Kandidaten berechnet werden, da die Bewegungsbahn des Balles durch Gegner besetzt ist. Damit ist der betrachtete Fall nicht ausführbar, da die Ausführung der Aktionen durch die Gegner verhindert werden kann. Lediglich bei zwei der 100 Aufrufe wird jeweils ein Fall nicht als Kandidat bezeichnet, da die Bewegungsbahn des Balles durch Gegner besetzt ist und da somit die Eigenschaft $FreePath$ nicht erfüllt ist. Bei dem Aufruf 28 und 78 wurde entsprechend ein Fall nicht als Kandidat aufgrund der nicht erfüllten Bedingung $FreePath$ bezeichnet. Das lässt sich darin begründen, dass bei den Aufrufen zuvor die meisten Fälle bereits nicht als Kandidaten erkannt werden, da die übrigen Eigenschaften wie die Anpassungskosten oder das Ähnlichkeitsmaß bereits nicht erfüllt sind. Sind diese Eigenschaften nicht erfüllt, so wird das Merkmal $FreePath$ nicht mehr überprüft.

Insgesamt zeigt die Auswertung der manuell erstellten Fallbasis, dass nicht alle möglichen Spielsituationen abgedeckt werden. Das bedeutet, dass im Durchschnitt 23 der 100 Spielsituationen, die zufällig generiert wurden, durch die Hilfe eines Falles aus der Fallbasis gelöst werden konnten. Zu den restlichen 77 der 100 Spielsituationen wurden hingegen keine Fälle in der Fallbasis erkannt, die eine Hilfe zur Suche einer geeigneten Lösung darstellen. Es lässt sich ableiten, dass die Fallbasis der Größe von 20 Fallen noch nicht hinreichend groß genug ist, um alle möglichen Spielsituationen abzudecken. Trotz der Symmetrie eines Falles werden nicht alle Spielsituationen durch die 20 in den Fällen vorliegenden Erfahrungen gelöst.
5.3. Auswertung der Fallbasis

Abbildung 5.6.: Häufigkeit nicht geeigneter Fälle (FreePath)
6. Zusammenfassung und Ausblick


Die FBS Methode ermöglicht es, Erfahrungen zu benutzen, um neue Spielsituationen zu lösen. Die Erfahrung besteht darin, zu wissen, wie eine bestimmte Situation gelöst werden kann, indem die Roboter entsprechende Aktionen ausführen. Dieses Wissen liegt beim FBS in sogenannten Fällen, die eine Fallbasis bilden, von vor. Beim FBS wird nun der geeignetste Fall bestimmt, der dem aktuellen Problem am ähnlichsten ist und somit bei der Lösung der neuen Situation am besten hilft. Da dennoch Unterschiede zwischen dem geeignetsten Fall und dem aktuellen Problem bestehen, ist eine Adaption des Falles an das Problem notwendig.


Zum Vorgehen in der vorliegenden Bachelorarbeit lässt sich zusammenfassen, dass ein Ansatz des FBS implementiert wurde. In Abschnitt 3.3 wurde das FBS vorgestellt, um eine Implementation umzusetzen. Der entwickelte Ansatz wurde in Kapitel 4 weiterhin aufgeführt und erläutert, wie dieser implementiert wurde. Dabei wurde berücksichtigt, dass der Ansatz an die Codebasis der HULKs angepasst ist und somit dem Team zur Verfügung steht. Zudem wurde die Implementierung dahin erweitert, dass eine Evaluation des Codes stattfinden konnte.


kann. Außerdem wird bei der Berechnung bisher nicht berücksichtigt, dass die direkte Strecke, die der Roboter zurücklegen muss, durch andere Spieler versperrt sein kann. In diesem Fall muss der Roboter, um seine Zielposition zu erreichen, einen möglichen Umweg finden und gehen. Da bei diesem Pfad jedoch eine längere Strecke zurückgelegt werden muss, ist es sinnvoll, die Anpassungskosten entsprechend zu berechnen.

Ein anderer Aspekt, der als Erweiterung angesehen werden kann, ist der, dass derzeit nicht beachtet wird, dass Fälle in der Fallbasis vorhanden sein können, die sich nur in ihrer Lösungsbeschreibung unterscheiden. Es wäre also möglich, dass in einer Spielsituation mehrere Fälle als geeignet angesehen werden, die sich nur in ihrer Lösung unterscheiden. In dieser Situation müsste trotzdem entschieden werden, welcher Fall als geeignetster Fall betrachtet werden kann. Bisher wird bei Sortierung von Fällen, die die gleichen Eigenschaften besitzen, diese sortiert, indem die Reihenfolge, wie diese in der Fallbasis gespeichert sind, beibehalten wird.

Zusätzlich ist es möglich, die Problemstellung eines Falles zu erweitern. Wie R. Ros et al. vorstellen [21], besteht die Möglichkeit, die Problemstellung durch Angabe des aktuellen Spielstandes und der Spielzeit zusätzlich zu definieren. Hiermit könnten die Roboter ihr Verhalten variieren, um zum Beispiel defensiver oder offensiver zu agieren.


A. Struktur einer XML-Datei eines Falles

In Listing A.1 ist ein Beispiel einer XML-Datei zu sehen, in der ein Fall geschrieben ist. Die Baumstruktur der XML-Datei enthält einen Knoten `problemDescription`, der die Informationen über die Problembeschreibung enthält, und einen Knoten `solutionDescription`, der die Lösungsbeschreibung des Falles darstellt.

Listing A.1: Beispiel einer XML-Datei eines Falles

```xml
<?xml version="1.0" encoding="utf-8"?>
<case>
  <problemDescription>
    <ballPosition>
      <position>
        <X>750</X>
        <Y>-2000</Y>
      </position>
      <scope>
        <radius_x>1000</radius_x>
        <radius_y>1250</radius_y>
      </scope>
    </ballPosition>
    <teammates>
      <robot id="1">
        <position>
          <X>-250</X>
          <Y>-1500</Y>
        </position>
      </robot>
      <robot id="2">
        <position>
          <X>2250</X>
          <Y>-750</Y>
        </position>
      </robot>
    </teammates>
    <opponents>
      <robot>
        <position>
          <X>4250</X>
          <Y>0</Y>
        </position>
      </robot>
    </opponents>
  </problemDescription>
</case>
```
A. Struktur einer XML-Datei eines Falles

```xml
<case>
  <scope>
    <radius_x>250</radius_x>
    <radius_y>1250</radius_y>
  </scope>
  <opponents>
    <robot id="1">
      <action type="getTheBall"/>
      <action type="passTheBall" index="2"/>
    </robot>
    <robot id="2">
      <action type="wait" actionIndex="1" robotIndex="0"/>
      <action type="receiveTheBall" index="1"/>
      <action type="goalkick"/>
    </robot>
  </opponents>
  <problemDescription/>
  <solutionDescription>
    <robot id="1">
      <action type="getTheBall"/>
      <action type="passTheBall" index="2"/>
    </robot>
    <robot id="2">
      <action type="wait" actionIndex="1" robotIndex="0"/>
      <action type="receiveTheBall" index="1"/>
      <action type="goalkick"/>
    </robot>
  </solutionDescription>
</case>
```
B. Sortierung der geeigneten Fälle


Listing B.1: Sortieralgorithmus

```cpp
// sort the vector of cases
vector<CaseHelp> HelperSorter::Sort(vector<Case>& pCases_,
                                   const vector<Mapping>& pMapping_,
                                   const ProblemDescription& pProblemDescription_)
{
    int numberOfTeammates, numberOfFulfilledConditions; // help variables
    double adaptionCost, similarity; // help vector
    vector<ScopePosition> scopesVec; // help vector
    vector<CaseHelp> pSortCasesVec; // help vector
    CaseHelp mCaseHelp; // help vector

    for (unsigned int i = 0; i < pCases_.size(); i++) {
        // number of fulfilled conditions
        scopesVec = pCases_[i].getVectorCasesScopes();
        // vector of opponents’ positions and scopes
        numberOfFulfilledConditions = HelperOpponent::OpponentSimilarity(scopesVec,
                                                                         pProblemDescription_.
                                                                         getOpponentsPositionsVector());
        mCaseHelp.mSortCase.SetNumberOfFulfilledConditions(
            numberOfFulfilledConditions);

        // number of teammates —> teamwork is better than individualistic work
        numberOfTeammates = pCases_[i].
            getProblemDescription().GetNumberOfTeammates();
        mCaseHelp.mSortCase.SetNumberOfTeammates(
            numberOfTeammates);
    }
}
```
B. Sortierung der geeigneten Fälle

```c++
// adaption costs
adaptionCost = HelperMapping::Cost(pMapping_[i], pCases_[i], pProblemDescription_);
mCaseHelp.mSortCase.SetAdaptionCosts(adaptionCost);

// ball similarity
similarity = pCases_[i].DetermineBallSimilarity(pProblemDescription_.getBallPosition());
mCaseHelp.mSortCase.SetSimilarity(similarity);

mCaseHelp.mCase = pCases_[i];
mCaseHelp.mMapping = pMapping_[i];
pSortCasesVec.push_back(mCaseHelp);
}

// sort list
SortHelp(pSortCasesVec);

// return sorted list
return pSortCasesVec;
```
C. Bewegungsbahn der Balles


Listing C.1: Algorithmus zur Berechnung der Ballbahn

```cpp
// Calculate from the solution all ball positions
std::vector<Position> SolutionDescription::DetermineBallPath(
    const Mapping& pMapping, const Position& pBallCurrentPosition,
    const std::map<Id, Position>& pTeammatesPositionsMap) const {

    // Create mapping which maps an index in the case to an index in the problem
    std::vector<Position> mBallPath;
    std::map<int, int> mMapping1 = pMapping.getMapping();
    std::map<int, int> mMapping2;
    std::map<int, int>::iterator it1;
    int first, second;
    for (it1 = mMapping1.begin(); it1 != mMapping1.end(); it1++) {
        ...
```
C. Bewegungsbahn der Balles

```plaintext
first = it1->first;
second = it1->second;
mMapping2[second] = first;
}

// Check if gameplay is valid
if (!this->isValid()) {
    throw std::exception("Error: Unvalid gameplay.");
}

// Create a map<Id, int>, which maps the Id of the robot to an integer
std::map<Id, int> actionNrMap;
Gameplay::iterator it3;
Gameplay gameplay_ = mGameplay_;
for (it3 = gameplay_.begin(); it3 != gameplay_.end(); it3++) {
    actionNrMap[it3->first] = 0;
}

// Create map Id ——> int
// An entry stores the actual field position of the correspondence robot
std::map<Id, Position> mapPositionMap = mapPositionMap;

// index of the first robot in the case, which perform the action 'getTheBall'
int indexFirstCaseRobot = GetRobotWhoGetFirstTheBall();
int indexFirstProbRobot = mMapping2[indexFirstCaseRobot];

// the current ball position is the first position of the ball path
mBallPath.push_back(pBallCurrentPosition);

// index of the first robot
int indexRobot = indexFirstProbRobot;

// actual ball position
Position currentBallPosition = pBallCurrentPosition;

// next action
SAction nextAction;
```
ActionSequence actionVec = gameplay_[indexRobot];
while (actionNrMap[indexRobot] < actionVec.size()) {

  // next action, which will be executed by the current robot
  nextAction = gameplay_[indexRobot].operator[](actionNrMap[indexRobot]);

  switch (nextAction.Type) {
    // action wait: check the next action
    case ActionType::Wait:
      actionNrMap[indexRobot]++;
      break;

    case ActionType::ReceiveTheBall:
      actionNrMap[indexRobot]++;
      currentBallPosition.setX(mapPositionMap[indexRobot].getX());
      currentBallPosition.setY(mapPositionMap[indexRobot].getY());
      mBallPath.push_back(currentBallPosition);
      break;

    case ActionType::GoToPosition:
      actionNrMap[indexRobot]++;
      mapPositionMap[indexRobot].setX(nextAction.Params[0]);
      mapPositionMap[indexRobot].setY(nextAction.Params[1]);
      break;

    case ActionType::Kick:
      actionNrMap[indexRobot]++;
      currentBallPosition.setX(nextAction.Params[0]);
      currentBallPosition.setY(nextAction.Params[1]);
      mBallPath.push_back(currentBallPosition);
      break;
  }
}
case ActionType::PassTheBall:
    actionNrMap[indexRobot]++;
    indexRobot = nextAction.Params[0]; //
    index of teamplayer, which receive
    the ball
    break;

case ActionType::GetTheBall:
    actionNrMap[indexRobot]++;
    mapPositionMap[indexRobot].setX(
        currentBallPosition.getX());
    mapPositionMap[indexRobot].setY(
        currentBallPosition.getY());
    break;

case ActionType::Goalkick:
    actionNrMap[indexRobot]++;
    break;
}

return mBallPath;
Literatur


[22] RoboCup: About RoboCup. URL: http://www.robocup.org/about-robocup/ (besucht am 15.08.2015).


