

Gruppe 3: Catchphrase?

$\dots^1, \dots^1, \dots^1$ und \dots^1

FernUniversität in Hagen, Universitätsstraße 47, 58097 Hagen, Deutschland
{...}@studium.fernuni-hagen.de
<https://www.fernuni-hagen.de>

1 Einleitung

Unsere Gruppe besteht aus vier Informatik-Studenten, wovon zwei im Bachelorstudiengang Informatik und zwei im Masterstudiengang Praktische Informatik immatrikuliert sind.

2 Kommunikation und Kommunikationsmittel

Die gruppeninterne Kommunikation erfolgte über den für das Fachpraktikum eingerichteten Discord-Server. Wir trafen uns jeden Mittwoch im Sprachchat der Gruppe, um anfangs die grundsätzliche Softwarearchitektur mittels UML-Diagrammen zu entwickeln und später über aktuelle Aufgaben und Probleme zu sprechen. Der Gruppen-Chat wurde darüber hinaus für kurzfristige Absprachen genutzt. Die Quelltext-Verwaltung erfolgte über das Versionsverwaltungssystem Github, dessen Issue-Management wir ebenfalls verwendeten. Hier wurden alle Aufgaben zusammengetragen und jeweils ein Bearbeiter zugeordnet. Aus dem jeweiligen Issue wurde jeweils ein Feature-Branch erstellt, in dem die Bearbeitung der Aufgabe erfolgte. Abschließend wurden die Änderungen des Feature-Branche mittels eines Pull Requests auf den Master-Branch angewendet. Das Repository wurde so konfiguriert, dass ein Pull Request erst freigegeben wird, wenn er durch mindestens ein anderes Gruppenmitglied überprüft wurde. Dadurch sollte die Qualität des Quelltextes erhöht und das gemeinsame Verständnis des Quelltextes gefördert werden.

3 Technische Rahmenbedingungen und Softwarebasisarchitektur

Für die programmiertechnische Umsetzung entschieden wir uns für die Programmiersprache Java, da alle Gruppenmitglieder mindestens über Grundkenntnisse dieser Programmiersprache verfügten. Außerdem bietet Java als stark typisierte Programmiersprache die Möglichkeit bestimmte Programmierfehler schon vor der Ausführung des Programms zu entdecken. Dadurch wird die Wahrscheinlichkeit von Laufzeitfehlern reduziert. Ein weiteres Argument für die Nutzung von Java ist das von den Entwicklern des Szenarios mitgelieferte Grundsystem

des Agentenprogrammes, in dem die Kommunikation mit dem Server schon implementiert ist. Der durch die Nutzung dieses Grundsystems entfallene Aufwand konnte für die Programmierung der eigentlichen Agentenfunktionalität genutzt werden.

In der Dokumentation des vergangenen Multi-Agent Programming Contests berichtete das Team MLFC, dass das Debugging aufgrund der Verwendung einer Vielzahl von zusätzlichen Frameworks und Tools erschwert wurde [3]. Deshalb entschieden wir uns gegen den Einsatz solcher externen Tools.

Das in dieser Arbeit entwickelte Agentensystem basiert auf der BDI-Architektur. Diese Architektur ist eine Abstraktion des menschlichen Denkens bzw. Schlussfolgerns. Das heißt sie versucht zu beschreiben, wie Menschen sich für ihre nächste Handlung entscheiden, um dadurch übergeordnete Ziele zu erreichen. Die Komponenten der BDI-Architektur sind:

- **Beliefs:** Informationen, die der Agent über seine Umwelt besitzt
- **Desires:** Handlungsoptionen bzw. mögliche Ziele des Agenten
- **Intentions:** Ziele, für die sich der Agent entschieden hat und an deren Realisierung gerade gearbeitet wird

Die besondere Herausforderung dieses Ansatzes besteht in der Frage, wie die Intentions ausgewählt und wie lange sie verfolgt werden. Werden Intentions zu lange verfolgt, sind die Grundbedingungen für deren erfolgreichen Abschluss ggf. nicht mehr gegeben. Werden sie zu schnell gewechselt, wird u. U. nie eine Intention erfolgreich abgeschlossen [2].

Der Vorteil der BDI-Architektur ist, dass eine funktionale Zerlegung des Agenten in die oben genannten Subsysteme vorgegeben ist. Die Komponenten erscheinen dabei intuitiv, da die BDI-Architektur auf dem menschlichen Denkmodell basiert [2]. Aufgrund der Ähnlichkeiten zum menschlichen Denkmodell ist diese Architektur gut verständlich und bietet einen einfachen Einstieg in die Umsetzung, weshalb wir uns für die Verwendung dieses Ansatzes entschieden.

Auf der Grundlage der BDI-Architektur erarbeiteten wir anhand von Klassen-UML-Diagrammen konkrete Systemarchitekturen. Die UML-Diagramme stellen aufgrund der übersichtlichen, visuellen Darstellung des komplexen Systems eine gute Diskussionsgrundlage dar.

Aus der Konzeptionsphase resultierten zwei unterschiedliche Ansätze, die in der Folge auch weitestgehend unabhängig umgesetzt und ausprobiert wurden. Dadurch sollten die Vor- und Nachteile der unterschiedlichen Ansätze untersucht werden.

4 Gruppenbeitrag Heinz Stadler

4.1 Agent V1 - Architektur

Aufbau

4.2 Wissensverwaltung

Belief

4.3 Wegfindung

Pathfinding

4.4 Ziel- und Absichtsfindung

Desires

4.5 Verifikation und Problemfindung

Tests / Debugger

4.6 ...**5 Gruppenbeitrag Melinda Betz****5.1 Agent V2 - Architektur**

Der AgentV2 arbeitet mit der Step-Methode. Er arbeitet auf eigenen Desires (V2desires) und benutzt nicht die Desires des AgentenV1 mit. Außerdem verwendet der Agent nicht das komplette Pathfinding des V1, da dieses OpenGL benutzt, was auf manchen AMD Rechnern nicht, zumindest nicht performant (mehrere Sekunden pro Agent), funktioniert. Meine beiden Rechner sind da keine Ausnahme. Der Agent stellt seine eigenen Berechnungen an.

5.2 Task Bearbeitung und Strategie

Die Desires welche ein Agent ausführen kann, werden unterschieden in Desires die keine Task benötigen, wie z. B. die Bestimmung der Mapgröße und Desires zur Bearbeitung einer Task, wie z. B. Submitten.

Jeder Agent holt sich zuerst eine Rolle in einer Role Zone, da in der Default Rolle keine Blöcke geholt oder submittet werden können. Die Rolle Worker wird für alle Agenten verwendet. Diese Rolle alles kann, manches zwar nicht ganz optimal, aber gut genug.

Alle Worker holen selbständig einen, zu einer aktiven Task passenden, Block. Dabei darf die maximale Anzahl der vorhandenen Blöcke eines bestimmten Blocktyps nicht überschritten werden. Mit diesem Block bewegen sie sich in Richtung Goal Zone. Jeder Agent in einer Goal Zone, der gerade nicht an einer Mehr-Block-Task arbeitet (AgentCooperations), prüft, in jedem Step, für alle aktiven Tasks, ob er den innersten Block dieser Task besitzt. Eine Ein-Block-Task

kann so direkt bearbeitet werden (Block in Position bringen und submitten).

Für eine Mehr-Block-Task benötigt der Agent noch mindestens einen Helfer, um eine neue AgentCooperation mit sich als Master (dieser submittet die zusammengebaute Mehr-Block-Task) bilden zu können. Ein Agent kann sich selbst zum Master machen und die Helfer bestimmen welche ihm die Blöcke beschaffen und in der richtigen Reihenfolge zusammensetzen sollen. Die Kommunikation wann was macht erfolgt über ein Klasse AgentCooperations sowie Statusabfragen zum Stand der Abarbeitung. Es gibt einstellbare Regeln, wie z.B., dass nicht mehr als drei Agenten gleichzeitig Master sein dürfen. Das soll vor Klumpenbildung in der Goal Zone schützen, da so nur eine begrenzte und kontrollierbare Anzahl an Agenten in und um die Goal Zone beschäftigt ist.

5.3 Umgebungsfindung und Synchronisation

Ein Agent des AgentV2 kennt alles (vor allem Dispenser und GoalZones), was schon einmal in seinem Sichtfeld oder dem eines Agenten seiner Gruppe war. Entfernung und Richtung zu diesen Punkten kann er selbst ermitteln.

Bei der Synchronisation der Zusammenarbeit von Agenten geht es neben der beschriebenen Bearbeitung von Mehr-Block-Tasks um die Bestimmung der Mapgröße. Dabei wird für die Ermittlung von Höhe und Breite jeweils aus den ersten beiden Treffen zweier Agenten eine AgentCooperation gebildet. Der Master läuft einmal um die Map herum. Der Helfer wartet bis dieser wieder zurück ist. Aus der Position des Masters relativ zum Helfer vor und nach dem Umlauf, sowie seinem zurückgelegten Weg auf der zu ermittelnden Koordinate, kann nun die genaue Größe der Map berechnet werden.

5.4 Schwierigkeiten und Lösungsstrategien

Einige der Schwierigkeiten waren, dass die Größe des Spielfeldes nicht bekannt war und, dass es schwer ist genau zu wissen wer was attached hat, da diese Information im Percept vom Server nicht detailliert enthalten ist. Außerdem hat man Schwierigkeiten, wenn ein Agent mit einer Geschwindigkeit größer zwei läuft, dessen genau Position festzustellen sollte der Agent in einem Schritt abbrechen. Es ist dann nicht möglich herauszufinden in welchem Schritt der Agent abgebrochen ist oder noch viel wichtiger wie viele Schritte dieser bis zum Abbruch gegangen ist. Was zu Problemen bei der Ermittlung der neuen Position des Agenten führt.

Das Mapgrößen Problem wurde zunächst dadurch entschärft, dass die Größe der Karte einfach bei den Turnieren bekannt war. Später hatten die Agenten das oben beschriebene Desire zum Bestimmen der Mapgröße. Das Attachment-Problem konnte man dadurch lösen, dass man sich in einer Variable merkt, welcher Agent was genau attached hat. Diese muss natürlich ständig aktuell sein

und mit Einführung der Clear – Events war dies nicht mehr 100 Prozent möglich. Des Weiteren läuft kein Agent schneller als zwei um die Position auch bei Partial – Success noch genau zu haben. Ein Grund für die Eignung der Rolle Worker.

Wichtig ist die Größe des Spielfeldes vor allem für die genaue Bestimmung der Position der Agenten. Um einen aus dem Spielfeld laufenden Agenten wieder im Spielfeld zu positionieren braucht man eine Modulo Rechnung und die aktuelle Position mit der Spielfeldgröße. Hier war das Problem, für dessen Erkenntnis ich einige Zeit benötigte, dass Java keine Funktion für Modulo besitzt. Normalerweise nutzt man dafür das % – Zeichen. Aber eigentlich ist das nur der Rest einer ganzzahligen Division, was bei positiven Zahlen Modulo entspricht. Läuft man aber links aus dem Spielfeld in den negativen Bereich, dann braucht man Modulo mit negativen Zahlen. Als Modulo mit negativen Zahlen benutzte ich schließlich: $\text{Modulo } X = (((X \% \text{MapBreite}) + \text{MapBreite}) \% \text{MapBreite})$.

5.5 Tätigkeiten Gruppenlead

In der Leadsgruppe haben wir uns regelmäßig getroffen und die Turnierkonfigurationen zusammengestellt sowie allgemeine Punkte besprochen. Diese Treffen waren auch der Ort, um Fragen unserer Teammitglieder anzusprechen und zu klären.

6 Gruppenbeitrag Phil Heger

6.1 Logging

Anfangs wurde ein Logging-Modul (AgentLogger.java) implementiert, in dem der Logging-Output konfiguriert wird. Dieses Modul ermöglicht es, den Logging-Output, der von einem Agenten in einem bestimmten Desire oder Modul ausgegeben wird, in eine eigene Datei zu schreiben. Dadurch sind die Entscheidungsfindung dieses Agenten und seine Zustandsänderungen besser nachvollziehbar, als wenn der gesamte Logging-Output aller Agenten in der gleichen Konsole bzw. Datei ausgegeben wird und nachträglich gefiltert werden muss.

6.2 Strategien zum Stören gegnerischer Agenten

Als grundlegender Mechanismus für das Stören gegnerischer Agenten ist die clear-Aktion auf die Position des gegnerischen Agenten im Szenario vorgesehen. Ist diese erfolgreich, dann verringert sich das Energielevel des gegnerischen Agenten um einen definierten Betrag. Beträgt sein Energielevel 0, dann wird der Agent für eine ebenfalls in der Server-Konfiguration festgelegte Anzahl von Runden deaktiviert. Wenn der Agent deaktiviert ist, verliert er alle mit ihm verbundenen Blöcke. Wenn die Blöcke nicht (mehr) mit einem Agenten verbunden sind, können sie mit einer einmaligen clear-Aktion entfernt werden.

Beim Stören und Deaktivieren gegnerischer Agenten gibt es zwei Probleme, die die Wirksamkeit einer einfachen Strategie (z. B. Angreifen beliebiger Agenten an beliebigen Stellen der Karte) stark verringern:

- Bei der clear-Aktion muss das Feld angegeben werden, auf dem sich der gegnerische Agent am Ende der Runde befinden wird. Diese Position ist bei gegnerischen Agenten jedoch unbekannt, da er sich in alle Richtungen bewegen oder auch stehenbleiben kann. Die Wahrscheinlichkeit, den Gegner zu treffen, ist dadurch sehr gering.
- Der Schaden einer erfolgreichen clear-Aktion ist (bei den in den Turnieren verwendeten Konfigurationen) klein. So beträgt der max. mögliche Schaden, wenn sich der angegriffene Agent in einem angrenzenden Feld befindet, nur 16 Punkte bei einer Gesamtenergie von 100 Punkten. Der Schaden halbiert sich mit der Distanz zum gegnerischen Agenten. Hinzu kommt, dass die Erfolgswahrscheinlichkeit der clear-Aktion bei allen Rollen (außer der „digger“-Rolle) nur 30 % beträgt.

Für ein wirksames Stören der Gegner ist daher eine komplexere Strategie und das Annehmen der „digger“-Rolle erforderlich.

Dispenser blockieren Die erste Idee besteht darin, das Sammeln von Blöcken für die gegnerischen Agenten zu erschweren, indem ein eigener Agent auf das gleiche Feld wie ein Dispenser geht und dort verbleibt. Dadurch ist es nicht mehr möglich an diesem Dispenser neue Blöcke zu erzeugen. Dabei werden Dispenser ausgewählt, die für die Bearbeitung der aktuellen Aufgaben am wichtigsten sind. Aufgrund der hohen Anzahl an Dispensern in den Turnier-Konfigurationen zeigte sich jedoch, dass dieser Ansatz nicht praktikabel ist, da eine große Anzahl an eigenen Agenten für das Blockieren der vielen Dispenser notwendig wäre. Um die eigenen Agenten nicht auch zu behindern, müssen diese über blockierte Dispenser informiert werden.

Goal Zone verteidigen Bei diesem Ansatz wurde der in der Einleitung dieses Unterkapitels erwähnte grundlegende Mechanismus (clear-Aktion auf Position des gegnerischen Agenten) umgesetzt, wobei der eigene Agent gegnerische Agenten nur in der Zielzone angreift, da die gegnerischen Agenten hier oft auf andere Agenten warten und sich in dieser Zeit nicht bewegen. Der eigene Agent kann sich dadurch nähern und mit dem größtmöglichen Schaden angreifen. Ist der gegnerische Agent deaktiviert, werden alle mit ihm verbundenen Blöcke gelöst. Diese können anschließend vom angreifenden Agenten mit clear-Aktionen entfernt werden. Neben dem Deaktivieren des Agenten für einige Runden war auch der Aufwand des Holens der Blöcke für umsonst. Hinzu kommt die Chance, den gegnerischen Algorithmus des Zusammenbauens von größeren Aufgaben zu stören.

In der Umsetzung wurde jeder der zwei in den Turnier-Konfigurationen definierten Zielzonen ein Agent fest zugewiesen. Für eine ausreichend hohe Effektivität muss der ausgewählte Agent die „digger“-Rolle annehmen, damit die clear-Aktionen mit einer Wahrscheinlichkeit von 100 % ausgeführt werden.

Für die Umsetzung der Strategie wurden folgende Teilaufgaben gelöst:

- Zuordnung eines Agenten zu jeder Zielzone

- Analyse von Dingen, die mit einem Gegner verbunden sind bzw. sein könnten (direkt an ihn angrenzende, zusammenhängende Blöcke)
- Auswahl eines zu attackierenden Gegners basierend auf den mit ihm verbundenen Dingen und der Distanz zu ihm
- Bewegungen in der Zielzone (um Hindernisse herum, auf den Gegner zu bzw. patrouillieren, wenn sich keine geeigneten Gegner im Sichtfeld befinden)
- Energie des angegriffenen Gegners mitzählen
- Gegnerverfolgung (um die Energie eines Agenten mitzählen zu können und für die Verfolgung und Fortsetzung des Angriffs, wenn er sich bewegt)
- Zuletzt angegriffenen Gegner nicht noch einmal angreifen (z. B. wenn Blöcke in seiner Nähe sind, die aber mit einem anderen Agenten verbunden sind, dann können diese nicht durch clear-Aktionen entfernt werden und der gleiche Gegner würde immer wieder angegriffen werden)

Getestet wurde die Strategie, indem die eigenen Agenten auch als Gegner-team eingesetzt wurden. Die Strategie ist nur wirksam, wenn sich gegnerische, Blöcke tragende Agenten ausreichend lange in der Zielzone nicht bewegen. Gegen 1-Block-Aufgaben ist die Strategie dadurch völlig wirkungslos, da die gegnerischen Agenten dabei in die Zielzone laufen und sofort die Aufgabe abgeben.

6.3 Sonstiges

Zu Beginn der Gruppenarbeit entwickelten wir eine mögliche Architektur für den Umgang mit den Desires anhand von UML-Diagrammen. In der Folge realisierte ich prototypisch eine Architektur. Diese stellte sich jedoch als nicht praktikabel heraus, da die Desires sehr kleinteilig aufgeteilt und in jedem Schritt die Desire-Objekte neu erzeugt wurden und somit Zustände nicht gespeichert werden konnten. Das Konzept wurde deshalb überarbeitet.

Zwischenzeitlich entwickelte ich ein Desire, bei dem ein Agent eigenständig alle Blöcke für beliebig komplexe Aufgaben sammelt und zusammenbaut. Dies wurde jedoch zugunsten von kooperierenden Agenten, die sich gegenseitig Blöcke holen, abgebrochen.

Für die Unittests wurde das Mocking Framework „mockito“ eingesetzt, das es ermöglicht das Desire-Objekt, das externe Abhängigkeiten, wie z. B. das Belief-Objekt, besitzt, unabhängig von diesen zu testen, indem die Rückgaben von Methodenaufrufen auf dem externen (nicht zu testenden) Objekt vordefiniert werden.

7 Gruppenbeitrag Björn Wladasch

8 Turniere

Bei den Turnieren sind immer beide Agenten zum Einsatz gekommen. (Ausnahme Turnier 6)

Die Probleme des AgentV2 waren über den Verlauf aller Turniere mehr oder

weniger dieselben. Die Agenten haben sich öfters zu Klumpen zusammengefunden und dabei gegenseitig behindert. Sie haben teilweise ihr Wissen über ihre Umwelt, insbesondere die Lage der Goal Zones, verloren.

Turnier 2

Turnier 3

Turnier 4

Turnier 5

Turnier 6

Bei Turnier 6 ist der AgentV2 nicht zum Einsatz gekommen, da er Probleme mit dem Erkennen und Finden der sich wechselnden Goal Zones hatte und so weniger Tasks wie bisher bearbeiten konnte. Im Nachhinein war dies jedoch nicht so schlimm und der Agent hätte trotzdem zum Einsatz kommen können.

9 Rekapitulation und Ausblick

Vor- und Nachteile der Entscheidung von zwei Architekturen Was sollte noch verbessert werden Wie sind wir zufrieden

Es gibt eigentlich keine nennenswerten Probleme, wenn man zwei Architekturen hat, beide Agenten erfüllen ihre Anforderungen.

Im Großen und Ganz bin ich mit meinem AgentenV2 zufrieden. Wo ich noch Verbesserungspotential sehe, sind, wie oben erwähnt, das Problem der „Verklumpung“ und die Einschränkung der Raumwahrnehmung der Agenten. Leider konnte ich diese Probleme in der vorgegebenen Zeit nicht lösen, da der Agent auch immer weiterentwickelt werden musste und so nicht viel Zeit zur Behebung dieser „grundsätzlichen“ Probleme blieb.

10 FAQ

10.1 Teilnehmer*innen und ihr Hintergrund

Was war die Motivation an dem Praktikum teilzunehmen?

M: Die Motivation waren Interesse am Thema Künstliche Intelligenz und schon vorhandene Java Kenntnisse.

10.2 Statistiken

Wurden die Agents von Grund auf neu implementiert oder auf einer bestehenden Lösung aufgebaut?

AgentV2: Der Agent wurde von Grund auf neu implementiert (er basiert nur auf dem BasicAgent)

Wie viel Zeit wurde in die Entwicklung und Organisation des Praktikums gesteckt?

M: Jeden Tag 3- 4 Stunden, mal mehr mal weniger

Wie war die reingesteckte Zeit im Verlauf des Praktikums verteilt?

M: Die meiste Zeit wurde für die Entwicklung des Agenten (Coden) verwendet, und etwas Zeit davor und währenddessen für die theoretischen Überlegungen was der Agent tun soll. Und Richtung Ende des Praktikums (und währenddessen in Form von Notizen) Zeit für die Doku.

Wie viele Zeilen Code wurden ungefähr geschrieben?

M: ca 6704

Welche Programmiersprache und Entwicklungsumgebung wurde verwendet?

M: Programmiersprache Java und Entwicklungsumgebung Eclipse

Wurden externe Werkzeuge/Bibliotheken verwendet?

AgentV2: keine

10.3 Agenten-System Details

Wie entscheiden die Agenten, was sie machen sollen?

AgentV2: Ein Agent läuft durch alle Desires und merkt sich alle, in seinem momentanen Zustand (Belief) ausführbaren. Es werden Prioritäten vergeben und das Desire mit der höchsten Prio wird zur Intention und diese wird dann vom Agent ausgeführt.

Wie entscheiden die Agenten, wie sie machen sollen?

AgentV2: Die Agenten arbeiten so, wie es vorher programmiert wurde (Actions).

Wie arbeiten die Agenten zusammen und wie dezentralisiert ist der Ansatz?

AgentV2: Sehr dezentralisiert, die Agenten (der Master) suchen sich selbst ihre Hilfen beim Zusammenbauen der Tasks. In der Klasse AgentsCooperations wird festgehalten welcher Agent gerade mit welcher Task beschäftigt ist (und in welchem Status).

Kann ein Agent das generelle Verhalten zur Laufzeit ändern?

AgentV2: Nein, ein Agent erfüllt immer Tasks während der Laufzeit des Programms (außer es existieren gerade keine mit denen er etwas anfangen kann, dann wird gewartet)

Wurden Änderungen (z.B kritische Fehler) während eines Turniers vorgenommen?

AgentV2: Es wurden keine Änderungen während eines Turniers vorgenommen.

Wurde Zeit investiert um die Agenten fehlertoleranter zu machen? Wenn ja, wie genau?

AgentV2: Ja, es wurden viele Testläufe gemacht und immer wenn ein Fehler auftrat wurde dieser so behoben, dass er nicht mehr auftritt (und wenn nur mit einem Skip Befehl).

10.4 Szenario und Strategie

Was ist die Hauptstrategie der Agenten?

AgentV2: Sie holen sich zuerst einmal die Rolle Worker und dann holen sie sich einen Block und laufen mit ihm zur Goal Zone.

Haben die Agenten selbstständig eine Strategie entwickelt oder wurde diese bereits in die Implementierung eingebaut?

AgentV2: Nein, die Strategie wurde in die Implementierung eingebaut.

Wurde eine Strategie implementiert, die Agenten anderer Teams mit einbezieht?

AgentV2: Nein.

Wie entscheiden Agenten, welche Aufgabe sie als nächstes übernehmen?

AgentV2: Die Agenten entscheiden es selbst.

Wie koordinieren die Agenten die Arbeit für eine Aufgabe untereinander?

AgentV2: Die Agenten koordinieren indem sie durch Statusabfragen miteinander kommunizieren, wer wann fertig ist und wann weitergemacht werden kann.

Welche Aspekte des Szenarios waren am Herausforderndsten?

AgentV2: Wenn man nicht weiß wie groß die Karte ist.

10.5 Und die Moral von der Geschichte

Was wurde durch das Praktikum vermittelt?

M: Selbstständig als Gruppe ein Projekt erarbeiten mit nur wenig Vorgaben.

Welchen Ratschlag wäre für zukünftige Gruppen sinnvoll?

M: Nicht mit zu vielen Erwartungen starten, es ist nicht alles so einfach wie man es sich vorstellt

Was waren Stärken und Schwächen der Gruppe?

Was waren Vorteile und Nachteile der gewählten Programmiersprache und weiterer Werkzeuge?

Vorteile: die vorgegebenen Schnittstellen waren alle in Java programmiert
Nachteile: keine

Welche weiteren Probleme und Herausforderungen kamen im Laufe des Praktikums auf?

Was könnte beim nächsten Praktikum verbessert werden?

Welcher Aspekt der Gruppenarbeit hat am meisten Zeit in Anspruch genommen?

M: Koordinieren welches Gruppenmitglied welche Aufgabe übernimmt.

Literatur

1. github.com/agentcontest/massim_2022, [agentcontest/massim_2022](https://github.com/agentcontest/massim_2022/blob/main/docs/eismassim.md),
https://github.com/agentcontest/massim_2022/blob/main/docs/eismassim.md,
EISMASSim Documentation, 21.08.2022
2. Weiss, G.: Multiagent Systems, 2. Auflage, The MIT Press, Cambridge, 2000
3. Ahlbrecht, T., Dix, J., Fiekas, N. und T. Krausburg: The Multi-Agent Programming Contest 2021, Springer, Heidelberg, 2021
4. Hart, P. E., Nilsson, N. J. und Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths, in IEEE Transactions on Systems Science and Cybernetics, 4. Auflage, Nummer 2, Seiten 100-107, Juli 1968
5. Bratman, M.: Intention, plans, and practical reason, Harvard University Press, Cambridge, 1987