

Alchhörnchen: Ein Gamemaster zur Illustration und Evaluierung von KI-Techniken

Bachelorarbeit

von

Robert Vollmann

7. Juni 2006

Eidesstaatliche Erklärung:

Hiermit versichere ich, die vorliegende Arbeit selbständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Saarbrücken, 7. Juni 2006

(Robert Vollmann)

Abstract

In diesem Projekt soll ein Gamemaster implementiert werden, mit dessen Hilfe sich grundlegende Repräsentations- und Suchtechniken der Künstlichen Intelligenz anhand eines Spiels praktisch erproben lassen. Diese Techniken umfassen Algorithmen der uninformierten und informierten Suche oder der Adversarial Search. Die Spieler programmieren, basierend auf diesen Suchtechniken, Agenten, welche ein oder mehrere Eichhörnchen steuern und die durch verschiedene Kommandos mit ihrer Umgebung und untereinander interagieren. Das einfachste Spielszenario ist das Finden einer Nuss. Komplexere Szenarien beschreiben etwa das Aufnehmen und sammeln der Nüsse zur teameigenen Sammelstelle voraus. Der Gamemaster verbindet die Spielagenten, verwaltet den Spielkontext, wertet die Züge der Spieler aus, und sendet ihnen die zum Spielen nötigen Informationen.

Inhaltsverzeichnis

1	Einleitung	9
1.1	Ziele	9
1.2	Aufbau dieser Arbeit	9
2	Spielumgebung	11
2.1	Spielfeld	11
2.2	Objekte	11
2.2.1	Eichhörnchen	11
2.2.2	Nüsse	13
2.2.3	Bäume	13
2.2.4	Sammelstellen	14
3	Gamemaster	15
3.1	Beispiel mit einem Spieler	16
3.2	Beispiel mit zwei Spielern	16
3.3	Grafische Oberfläche	19
4	Spielarten	20
4.1	Parameter	20
4.2	Vordefinierte Spielarten	21
4.2.1	Informierte/Uninformierte Suche	21
4.2.2	Adversarial Search	22
5	Kommunikation	24
5.1	Grammatik	24
5.2	Startphase	26
5.3	Spielphase	29
5.3.1	Repräsentation der Objekte im Gamemaster	30
5.3.2	Überprüfungen	33
5.3.3	Fehler	39
5.3.4	Zugabhängige Updates	40
5.3.5	Zugunabhängige Updates	43
5.4	Endphase	45
6	Evaluation	47
6.1	Testspiel uninformierte Suche	47
6.2	Testspiel informierte Suche	49
6.3	Testspiel Adversarial Search	50
7	Related Work	53
8	Zusammenfassung und Ausblick	53

A	Konfiguration und Start	55
A.1	Start des Gamemasters	55
A.2	Start des Clients	55
A.3	Konfigurationsdateien der Spielarten	56
A.3.1	Konfigurationdateien für zwei Spieler	56
A.3.2	Konfigurationdateien für die un/informierte Suche	59
A.4	Konfigurationsdatei für den Gamemaster	60
A.5	Log-Dateien	61
A.6	Schnittstelle des Gamemasters	61
B	Beispielimplementierung Uniform-Cost-Algorithmus	63

Abbildungsverzeichnis

1	Karte des Spielfeldes als ungerichteter Graph. Die Knoten stehen für die Felder des Spielfeldes, die Kanten stellen die Bewegungsmöglichkeiten von einem Feld zu den angrenzenden Feldern mit den jeweiligen damit verbundenen Kosten dar.	11
2	Darstellung der Eichhörnchen in der graphischen Oberfläche.	12
3	Darstellung einer Nuss.	13
4	Graphische Darstellung eines Baums aus der Spielwelt.	13
5	An Sammelstellen können Nüsse abgelegt werden, wodurch ein Team Punkte erhält. Außerdem sind die Sammelstellen die Startpunkte für die Spieler.	14
6	Ein Beispielspielfeld. Zu sehen ist ein Eichhörnchen auf einem Feld mit einer Nuss, mehrere Bäume und eine Sammelstelle. Die hellen Felder um das Eichhörnchen herum liegen innerhalb der Sichtweite des Eichhörnchens. Die verdunkelten Felder sind schon erkundete, aber zur Zeit nicht sichtbare Felder. Die schwarzen Felder sind noch nicht erkundete Felder.	14
7	Das Sequenzdiagramm zeigt den Kommunikationsfluß für ein Spiel mit einem Spieler. Das Spiel wird in drei Phasen, Start-, Spiel- und Endphase unterteilt. Es ist in jeder Phase zu sehen, welche Arten von Nachrichten der Spieler schickt, wie der Gamemaster diese verarbeitet und welche Arten von Nachrichten der Gamemaster seinerseits verschickt.	17
8	Das Bild zeigt den Kommunikationsfluß für ein Spiel mit zwei Spielern. Wie im Ein-Spieler-Beispiel ist das Spiel in Start-, Spiel- und Endphase eingeteilt. In jeder Phase ist zu sehen, welche Arten von Nachrichten vom Spieler verschickt werden, wie der Gamemaster die Nachrichten verarbeitet und seinerseits an den Spieler verschickt.	18
9	Die Abbildung zeigt die Benutzeroberfläche, bestehend aus zwei Teilen. Im linken Teil wird das Spielfeld repräsentiert. Im rechten Teil werden Informationen zu den Teams und dem Spiel angezeigt.	19

10	Funktionsweise XML-RPC, entnommen von http://www.xmlrpc.com . Die Daten einer beliebigen, XML-RPCs unterstützenden Programmiersprache werden in XML verpackt und per HTTP an den Empfänger gesendet, der sie von XML in die verwendete Programmiersprache übersetzt.	25
11	Formale Grammatik zur Ableitung der zur Kommunikation verwendeten Sprache zwischen Gamemaster und Spieler. Nichtterminale sind im Gegensatz zu Terminalen stets in '<' und '>' eingeschlossen.	27
12	Serverseitiger Ablauf der Startphase. Einem neu verbundenen Spieler wird eine einzigartige ID vergeben und ggf. ein Gegner gesucht. Das Spiel beginnt mit dem Senden der Startinformationen.	28
13	Vorgehensweise bei der Überprüfung einer Zugliste. Jeder einzelne Zug wird auf Korrektheit überprüft. Tritt ein Fehler auf, wird der Fehler je nach Art entsprechend behandelt, bis schließlich der Zug zu Ende ist und der nächste Zug angefordert wird.	34
14	Die Abbildung zeigt die Karte, welche bei dem Testspiel benutzt wird. Das Eichhörnchen startet links oben auf der Sammelstelle (0,0) und muss zur Nuss auf Position (3,1) gelangen.	48
15	Die Abbildung zeigt die Karte, welche bei dem Testspiel benutzt wird. Team 1 startet links oben (0,0). Team 2 startet links unten (6,4). Gewonnen hat die Mannschaft, die als erstes Nüsse im Wert von 12 Punkten zu ihrer Sammelstelle bringt.	51

1 Einleitung

Motivation für dieses Projekt ist die Vorlesung zur Einführung in die Künstliche Intelligenz, in der unter anderem KI-Techniken zur Repräsentation und Lösung von Problemstellungen vorgestellt werden. Die Hörer der Vorlesung werden durch theoretische und praktische Übungsaufgaben an diese Thematik herangeführt. Um dies spielerisch und dadurch für die Hörer so interessant und einfach wie möglich zu gestalten soll ein Computerspiel in praktischen Übungsaufgaben eingesetzt werden. Die Aufgabe der Vorlesungsteilnehmer ist es, die Agenten für dieses Spiel zu programmieren, um die in der Vorlesung erlernten KI-Techniken praktisch umzusetzen. Die Agenten spielen zuerst allein und später gegeneinander. Dies ermöglicht eine Art Wettbewerb, in denen die Agenten aller Vorlesungsteilnehmer im Jeder-gegen-jeden-Prinzip gegeneinander spielen und der beste Agent als Sieger hervorgeht.

Das Spiel besteht aus einem Spielfeld auf dem sich Eichhörnchen befinden, welche sich auf dem Spielfeld bewegen können, mit anderen Eichhörnchen im Team interagieren und vor allem Nüsse sammeln müssen. Dies kann entweder allein oder gegen andere Spieler geschehen.

Ein Programm, der Gamemaster, leitet das Spiel. Er verwaltet das Spielfeld und alle sich darauf befindlichen Objekte. Außerdem gibt er den Spielern die benötigten Informationen und nimmt die Züge der Spieler entgegen, überprüft sie und führt sie aus.

1.1 Ziele

Ziel der Arbeit ist die Implementierung dieses Gamemasters. Dieser beinhaltet einen Webserver, zu dem sich die Clients verbinden können, um das Spiel, allein oder gegeneinander, zu spielen. Die Clients enthalten die Agenten der Vorlesungsteilnehmer und leiten die Kommunikation zwischen dem Gamemaster und den Agenten weiter.

Der Gamemaster stellt für die Spieler verschiedene Spielarten bereit. Für jede Spielart werden unterschiedliche KI-Techniken benötigt um die Aufgabenstellung optimal zu lösen.

1.2 Aufbau dieser Arbeit

In Kapitel 2 wird die Spielumgebung inklusive aller ihrer Objekte und Interaktionen untereinander beschrieben. Danach folgt in Kapitel 3 eine detailliertere Beschreibung des Gamemasters, gefolgt von einer Beschreibung von vordefinierten Spielarten in Kapitel 4. Dies beinhaltet alle Spielparameter, die für jede Spielart festgelegt werden müssen. In Kapitel 5 wird die Kommunikation zwischen dem Gamemaster und den Agenten, welche sich in drei Phasen gliedert, ausführlich beschrieben. Jede Phase beinhaltet bestimmte Überprüfungen, hinzukommen Fehlerbehandlung und Updates. In Kapitel 6 werden für die in Kapitel 4 vorgestellten Spielarten jeweils ein komplettes Testspiel gespielt. Anschließend wird in Kapitel 7 auf ein ähnliches Thema, dem *General Gameplaying Project* der Stanford University, eingegangen. In Kapitel 8 wird dann die Arbeit zusammengefasst und eine Aussicht auf weiterführende Themen gegeben. Abschließend

wird in Teil A des Anhangs erklärt, wie man den implementierten Gamemaster konfiguriert und startet. Außerdem wird der Aufbau der Konfigurationsdateien für Karten und die Spielparameter erläutert, was einem ermöglicht, seine eigenen Karten und Spielarten zu definieren. In Anhang B befindet sich eine Beispielimplementation für einen Agenten, welcher den Pfad mit den wenigsten Kosten von einem Startfeld zu einem Zielfeld finden muss.

2 Spielumgebung

Die Spielumgebung besteht aus einem Spielfeld und sich darauf befindlichen Objekten, welche in den folgenden Abschnitten näher beschrieben werden.

2.1 Spielfeld

Jedes Feld hat eine X- und eine Y-Koordinate, beginnend in der linken oberen Ecke mit (0,0). In bestimmten Spielarten (siehe Kapitel 4) werden Felder allerdings durch symbolische Positionen repräsentiert. Die symbolischen Positionen können aus einem oder mehreren Buchstaben bestehen. Es ist möglich von einem Feld auf ein angrenzendes Feld zu gehen. Je nach Spielart ist ein Zug von einem Feld auf ein anderes mit bestimmten positiven Kosten verbunden. Die Kosten werden jeweils für Paare von aneinander angrenzenden Feldern vor Beginn des Spiels festgelegt. Das Spielfeld kann also als ein ungerichteter Graph (Abb. 1) angesehen werden, bei dem die Knoten des Graphen die Felder darstellen. Die Positionen werden entweder durch die X- und Y-Koordinaten des Feldes oder durch symbolische Positionen repräsentiert, welche aus einzelnen Buchstaben oder Zeichenketten bestehen. Die Kanten des Graphen sind die möglichen Züge zwischen den einzelnen Feldern mit deren Kosten.

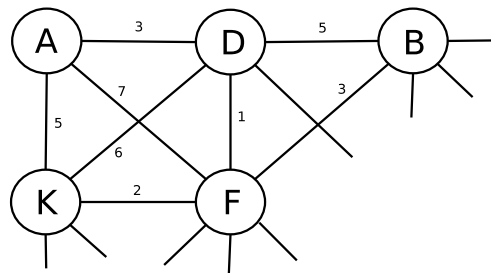


Abbildung 1: Karte des Spielfeldes als ungerichteter Graph. Die Knoten stehen für die Felder des Spielfeldes, die Kanten stellen die Bewegungsmöglichkeiten von einem Feld zu den angrenzenden Feldern mit den jeweiligen damit verbundenen Kosten dar.

2.2 Objekte

Es gibt vier verschiedene Objekttypen: Eichhörnchen, Nüsse, Bäume und Sammelstellen, welche im folgenden näher beschrieben werden. Desweiteren gibt es zu jedem Objekt ein Bild aus der graphischen Oberfläche des Gamemasters. Abbildung 6 zeigt eine grafische Repräsentation einer kompletten Karte, auf der alle Objekte zu sehen sind.

2.2.1 Eichhörnchen

Eichhörnchen sind die Hauptobjekte in der Welt, denn sie werden direkt vom Spieler kontrolliert. Eichhörnchen können sich zwischen Feldern hin und her bewegen, Nüsse aufnehmen, hinlegen

und an andere Eichhörnchen innerhalb des eigenen Teams weitergeben und somit auch Nüsse von anderen Eichhörnchen entgegennehmen. Alle Eichhörnchen eines Teams starten auf ihrer



Abbildung 2: Darstellung der Eichhörnchen in der graphischen Oberfläche.

Sammelstelle. Die Sammelstelle ist somit das einzige Feld, auf dem mehrere Eichhörnchen stehen dürfen.

Jedes Eichhörnchen hat eine spezifische Aufnahmekapazität für Nüsse. Die Restkapazität nach dem Aufnehmen einer Nuss berechnet sich aus der Differenz zwischen der Kapazität vor dem Aufnehmen und dem Gewicht der Nuss. Übersteigt das Gewicht einer Nuss die aktuelle Kapazität, so kann die Nuss nicht aufgenommen werden.

Desweiteren gibt es bestimmte Typen von Eichhörnchen, den schnellen Typ und den Sammlertyp. Der schnelle Typ hat weniger Kapazität als der Sammlertyp, das heißt er kann nicht so viele Nüsse aufnehmen. Sein Vorteil jedoch ist, daß er auch diagonal angrenzende Felder betreten kann.

In speziellen Spielarten, zum Beispiel bei der informierten und uninformierten Suche, können die Eichhörnchen springen. Das bedeutet, sie können auf jedes beliebige erforschte Feld und auf jedes einem erforschten Feld angrenzende Feld springen. Dies macht es einfacher, Suchalgorithmen zu implementieren.

Ein Eichhörnchen kann vier Aktionen, nämlich 'Go', 'Take', 'Drop' und 'Give' ausführen, welche im folgenden Abschnitt genauer erklärt werden.

- **Go**
Ein Eichhörnchen kann von dem Feld auf dem es gerade steht auf ein angrenzendes freies Feld gehen. Ein Feld gilt als frei, wenn kein anderes Eichhörnchen oder kein Baum (siehe Abschnitt 2.2.3) darauf steht.
- **Take**
Ein Eichhörnchen kann von dem Feld, auf dem es sich befindet, eine Nuss (siehe Abschnitt 2.2.2) aufheben, wenn das Gewicht der Nuss die Restkapazität des Eichhörnchens nicht überschreitet.
- **Drop**
Ein Eichhörnchen kann eine Nuss ablegen, wenn sich auf dem Feld nicht schon eine Nuss befindet. Ausnahme sind die Sammelstellen, an denen diese Begrenzung nicht gilt.

- **Give**

Ein Eichhörnchen kann eine Nuss an ein anderes Eichhörnchen weitergeben, falls sich das andere Eichhörnchen auf einem angrenzenden Feld befindet und noch genug Kapazität übrig hat, um die Nuss aufnehmen zu können.

2.2.2 Nüsse

Nüsse sind wertvolle Objekte in der Spielwelt und können von den Eichhörnchen aufgesammelt werden. Nüsse haben einen Wert und ein Gewicht. Je höher der Wert einer Nuss desto wertvoller und möglicherweise auch seltener ist diese Nuss. Je höher das Gewicht einer Nuss desto weniger Nüsse von diesem Typ kann ein Eichhörnchen tragen. Desweiteren können Nüsse nicht auf Fel-

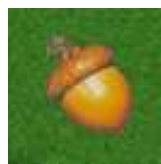


Abbildung 3: Darstellung einer Nuss.

dern liegen oder abgelegt werden, auf denen ein Baum steht. Außerdem können zwei Nüsse nicht auf demselben Feld liegen. Ausnahme ist die Sammelstelle, an der die Nüsse abgelegt werden können.

Ein Beispiel für eine Nuss wäre zum Beispiel eine Nussart mit einem Wert von 6 und einem Gewicht von 1. Dies wäre eine sehr wertvolle Nuss, da sie gleichzeitig einen hohen Wert und ein niedriges Gewicht hat. Im Gegensatz dazu wäre eine Nuss mit Wert 2 und Gewicht 4 nicht sehr wertvoll, da ein Eichhörnchen von dieser Sorte weniger tragen könnte, als von der vorher erwähnten, und gleichzeitig der Wert niedriger ist.

2.2.3 Bäume

Bäume dienen als Hindernisse auf dem Spielfeld. Kein Eichhörnchen kann auf ein Feld gehen, auf dem ein Baum steht. Außerdem sind Bäume unbeweglich. Desweiteren gibt es verschiedene Arten von Bäumen. Die Art des Baumes gibt einen Hinweis darauf, welche Nussart sich in der Nähe befindet.

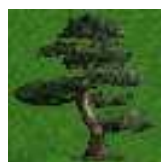


Abbildung 4: Graphische Darstellung eines Baums aus der Spielwelt.

2.2.4 Sammelstellen

Eichhörchen können ihre gesammelten Nüsse auf Sammelstellen ablegen. Wurden die Nüsse dort einmal abgelegt, können sie nicht wieder aufgenommen werden. Die Zielsetzung mancher Spielarten ist das Sammeln und Ablegen von Nüssen auf der teameigenen Sammelstelle. Nur dort abgelegte Nüsse werden gewertet. Außerdem sind Sammelstellen die einzigen Punkte auf dem Spielfeld auf denen mehrere Nüsse abgelegt werden können und auf dem sich gleichzeitig mehrere Eichhörchen befinden können. Die Sammelstelle dient außerdem als Startpunkt für die Spieler.



Abbildung 5: An Sammelstellen können Nüsse abgelegt werden, wodurch ein Team Punkte erhält. Außerdem sind die Sammelstellen die Startpunkte für die Spieler.



Abbildung 6: Ein Beispielspielfeld. Zu sehen ist ein Eichhörchen auf einem Feld mit einer Nuss, mehrere Bäume und eine Sammelstelle. Die hellen Felder um das Eichhörchen herum liegen innerhalb der Sichtweite des Eichhörchens. Die verdunkelten Felder sind schon erkundete, aber zur Zeit nicht sichtbare Felder. Die schwarzen Felder sind noch nicht erkundete Felder.

3 Gamemaster

Eine Hauptaufgabe des Gamemasters ist die Verwaltung der Spieler. Der Gamemaster muss Registrierungen von Spielern annehmen und Gegner für die Spieler suchen. Desweiteren muss er die gespielten Spiele verwalten. Zu jedem Spiel gehört ein Spielzustand. In einem Spielzustand sind das Spielfeld und alle vorhandenen Objekte gespeichert.

Während des Spiels muss der Gameamster den Spielern benötigte Informationen, wie die Startinformationen oder die Veränderungen nach einem Zug des Gegners, mitteilen. Die Startinformationen beinhalten Informationen über die eigenen, die gegnerischen und alle sonstigen Objekte, wie Bäume oder Nüsse.

Er muss die Züge der Spieler annehmen, den richtigen Spielen zuordnen und für jeden Zug eine Reihe von Überprüfungen (siehe Kapitel 5) durchführen. Dies ist einerseits eine generelle Überprüfung, die unabhängig von der Art des Zuges ist und andererseits eine zugspezifische Überprüfung, die in Abhängigkeit der jeweiligen Zugart durchgeführt wird. Wird bei einer dieser Überprüfungen ein Fehler entdeckt, wird zwischen zwei Arten von Fehlern, leichten und schweren, unterschieden. Schwere Fehler sind solche, die ein Spieler mit dem ihm verfügbaren Wissen hätte vermeiden können. Dementsprechend sind leichte Fehler, solche die er nicht durch sein Wissen hätte vermeiden können. Tritt ein leichter Fehler auf, darf der Spieler erneut einen Zug senden. Bei einem schweren Fehler, muss er dies aktuelle Runde aussetzen und darf erst in der nächsten Runde weiterspielen. Ist ein Zug fehlerfrei, muss der Gamemaster den Zug ausführen und den Spielzustand dementsprechend ändern.

Der Gamemaster unterstützt multiple Spielarten, die sich in ihren Spielparametern (siehe Kapitel 4) unterscheiden. Für jede Spielart werden bestimmte KI-Techniken zum Erreichen des Spielzieles benötigt. Der Gamemaster stellt die folgenden Spielarten zur Verfügung:

Uninformierte Suche Der Spieler muss von einem Startfeld zu einer Nuss auf einem Zielfeld gehen. Er hat zu Anfang keine Informationen über seine Umgebung. Er kennt lediglich symbolischen Positionen der angrenzenden Felder.

Informierte Suche Diese Art ist ähnlich der uninformatierten Suche, nur daß der Spieler noch die Koordinaten des Start- und des Zielfeldes bekommt und somit eine heuristische Funktion anwenden kann, um den Weg zum Zielfeld zu finden.

Adversarial Search Zwei Spieler spielen in dieser Spielart gegeneinander, wobei zwischen 'Perfect Information'-Spielen und 'Imperfect Information'-Spielen unterschieden wird. Bei ersterem können die Spieler das ganze Spielfeld sehen. Bei zweiterem können sie lediglich den Teil sehen, der in ihrem Sichtbereich liegt.

Zusätzlich gibt es verschiedene Spielziele. Bei der informierten und der uninformatierten Suche

muss man zum Beispiel eine bestimmte Nuss bzw. den optimalen Weg zu dieser Nuss finden. Bei Zwei-Spieler-Spielen hängt das Spielziel von der eingestellten maximalen Rundenzahl und der zu erreichenden Punktegrenze ab. Wird die Rundenzahl begrenzt, dann gewinnt der Spieler, der nach der maximalen Rundenzahl die höhere Punktzahl erreicht hat. Wird zusätzlich eine Punktegrenze gesetzt, kann ein Spieler das Spiel gewinnen, wenn er als erstes die Punktegrenze erreicht hat.

Die Kommunikation zwischen dem Gamemaster und den Agenten geschieht mittels XML-RPCs[1], welche in Kapitel 5 ausführlicher beschrieben werden.

Im folgenden werden die oben genannten Aufgaben des Gamemasters anhand zweier Sequenzdiagramme (Abb. 7 und 8) für Spiele für einen bzw. zwei Spieler verdeutlicht.

3.1 Beispiel mit einem Spieler

Abbildung 7 zeigt den Kommunikationsfluß für ein Spiel mit einem Spieler. In der Startphase registriert sich der Spieler beim Gamemaster. Der Gamemaster vergibt eine eindeutige ID an den Spieler und initialisiert den Spielzustand. Anschließend beginnt das Spiel und der Gamemaster schickt dem Spieler die benötigten Startinformationen. Nun beginnt die Spielphase und der Spieler macht seinen ersten Zug. Der Gamemaster empfängt den Zug und überprüft diesen auf leichte und schwere Fehler. Begeht der Spieler einen leichten Fehler, darf er seinen Zug nochmals senden. Begeht er jedoch einen schweren Fehler, ist die Runde beendet. Weist ein Zug keine Fehler auf, aktualisiert der Gamemaster den Spielzustand und sendet dem Spieler die Veränderungen und fordert ihn damit gleichzeitig auf seinen nächsten Zug zu senden. Das ganze wiederholt sich nun, bis das Spielziel erreicht ist und die Endphase beginnt. Der Gamemaster beendet das Spiel mit dem Senden der abschließenden Informationen in Abhängigkeit des Spielziels. War das Spielziel zum Beispiel den Pfad mit möglichst wenigen Kosten zu finden, sendet der Gamemaster die Kosten des Pfades, den der Spieler gefunden hat. Mit dem Senden der abschließenden Informationen ist das Spiel beendet.

3.2 Beispiel mit zwei Spielern

Abbildung 8 zeigt den Kommunikationsfluß für ein Spiel mit zwei Spielern. Zuerst müssen sich die zwei Spieler registrieren. Anschließend beginnt das Spiel. Als erstes sendet der Gamemaster dem ersten Spieler die Startinformationen und fordert ihn gleichzeitig auf, seinen ersten Zug zu machen. Der Spieler schickt daraufhin seinen Zug an den Gamemaster, welcher den Zug auf Korrektheit überprüft. Nun können drei Fälle auftreten:

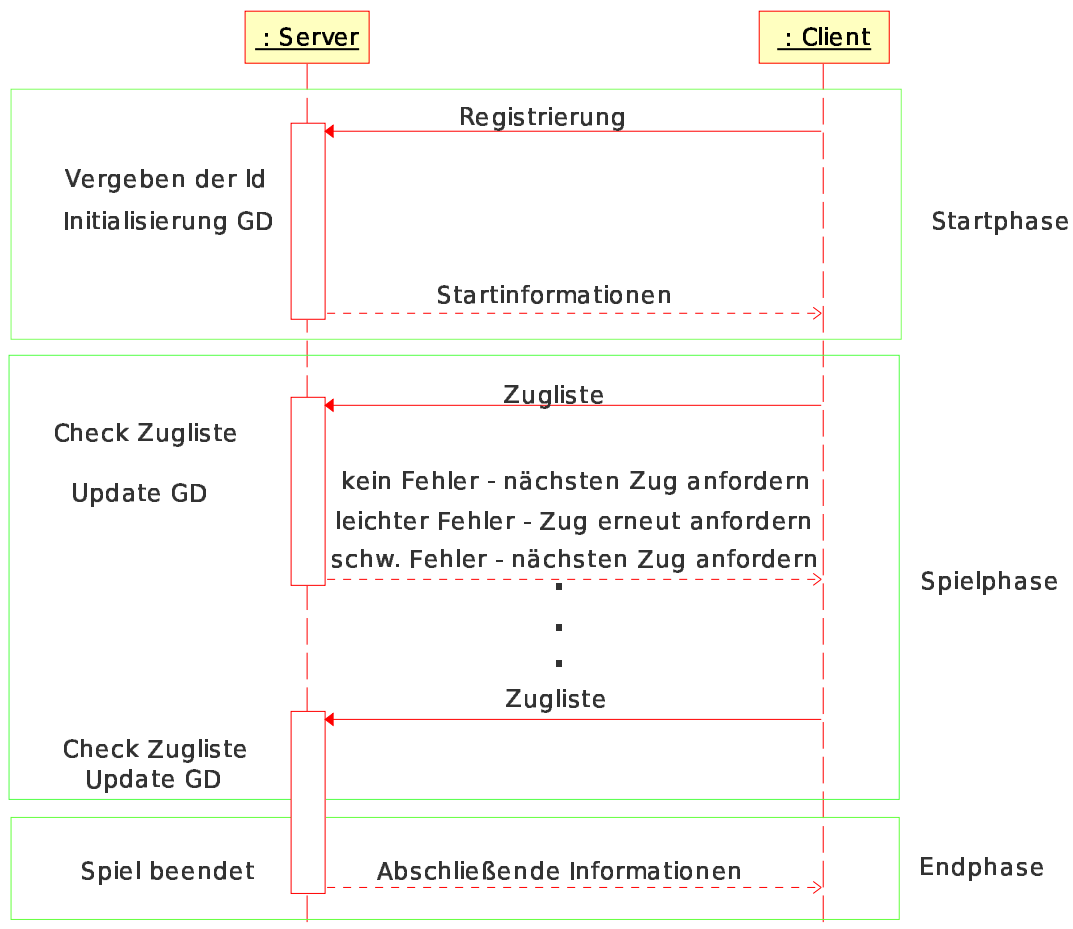


Abbildung 7: Das Sequenzdiagramm zeigt den Kommunikationsfluß für ein Spiel mit einem Spieler. Das Spiel wird in drei Phasen, Start-, Spiel- und Endphase unterteilt. Es ist in jeder Phase zu sehen, welche Arten von Nachrichten der Spieler schickt, wie der Game-master diese verarbeitet und welche Arten von Nachrichten der Gamemaster seinerseits verschickt.

1. Es tritt kein Fehler auf, das heißt, alle Züge aus der Zugliste sind fehlerfrei. Der Spielzustand wird aktualisiert und der nächste Spieler ist am Zug. Dieser bekommt nun selber die Startinformationen gesendet und darf seinen ersten Zug machen.
2. Es tritt ein leichter Fehler auf. Der erste Spieler hat nun die Chance seine Zugliste zu überarbeiten und sie anschließend erneut zu senden. Die Zugliste wird dann erneut überprüft.
3. Es tritt ein schwerer Fehler auf. Die Zugliste des Spielers ist ungültig und der nächste Spieler ist an der Reihe. Der Spieler bekommt den Fehler in der nächsten Runde mitgeteilt und darf erst dann wieder eine Zugliste senden.

Ist anschließend der nächste Spieler an der Reihe, bekommt auch dieser die Startinformationen inklusive der Veränderungen durch den ersten Spieler gesendet. Der Spieler sendet seinen Zug und dieser wird, wie der vorherige, auf Korrektheit überprüft. Diese Abläufe wiederholen sich solange bis das Spielziel erreicht ist. Damit beginnt die Endphase, in welcher der Gamemaster den Spielern mitteilt, wer das Spiel gewonnen hat.

3.3 Grafische Oberfläche

Der Gamemaster verfügt außerdem über eine grafische Oberfläche (Abbildung 9), in der die Spiele, die gespielt werden, visuell repräsentiert werden. Im linken Teil der Oberfläche wird das Spielfeld mit allen seinen Objekten¹ grafisch dargestellt. Die rechte Seite enthält diverse Informationen über die Teams und das Spiel. Für jedes Team wird angezeigt, welche Eichhörnchen sich gerade auf der eigenen Sammelstelle befinden, welche Nussarten zur Sammelstelle gebracht wurden und wieviele Punkte dadurch erzielt wurden. Außerdem werden alle Eichhörnchen mit ihrem Namen, ihrer derzeitigen Position und den Nussarten, die sie zur Zeit bei sich tragen, aufgelistet. Unter den Informationen über die einzelnen Teams befinden sich die Spielinformationen. Es wird angezeigt, wieviele Runden maximal gespielt werden, wieviele Punkte von den Teams erreicht werden müssen und in welcher Runde sich das Spiel gerade befindet. Auf Wunsch können die einzelnen Informationsfenster durch klicken auf die kleinen Pfeile zwischen den Fenstern aus- und auch wieder eingeblendet werden.



Abbildung 9: Die Abbildung zeigt die Benutzeroberfläche, bestehend aus zwei Teilen. Im linken Teil wird das Spielfeld repräsentiert. Im rechten Teil werden Informationen zu den Teams und dem Spiel angezeigt.

¹In Kapitel 2 befindet sich zusätzlich zur Beschreibung der einzelnen Objekte jeweils eine Abbildung mit der grafischen Darstellung der Objekte in der Oberfläche.

4 Spielarten

Der Gamemaster stellt mehrere Spielarten zur Verfügung, bei denen die verschiedenen Parameter vorher eingestellt wurden. In den verschiedenen Spielarten müssen zur optimalen Erfüllung des Spielziels verschiedene KI-Techniken und Suchalgorithmen, wie zum Beispiel uninformierte und informierte Suche, für die Implementierung der Spieleragenten benutzt werden.

Viele KI-Techniken, wie zum Beispiel KI-Planen, können unabhängig von fast allen Spielparametern eingesetzt werden, so daß für viele KI-Techniken keine konkrete Spielart angegeben werden muss.

4.1 Parameter

Im Folgenden werden die Parameter und deren jeweilige Werte, die sie annehmen können, aufgezählt:

1. Spielziel

- **Nuss finden:** Nur bei der informierten/uninformierten Suche. Ziel ist es, eine auf dem Spielfeld liegende Nuss zu finden.
- **Punktzahl erreichen:** Es muss eine bestimmte Punktzahl, die sich aus dem summierten Wert der gesammelten Nüsse zusammensetzt, die an der Sammelstelle abgeliefert wurden, erreicht werden.
- **Sammeln auf Zeit:** Innerhalb einer vorgegebenen Rundenzahl müssen möglichst viele Punkte gesammelt werden.
- **Punktzahl in bestimmter Zeit erreichen:** Es muss eine bestimmte Punktzahl innerhalb einer bestimmten Rundenzahl erreicht werden. Ist in einem Zwei-Spieler Spiel die maximale Rundenzahl erreicht, ohne daß ein Spieler die Punktegrenze erreicht hat, gewinnt der Spieler, der mehr Punkte gesammelt hat.

2. Anzahl der Spieler

- **Ein Spieler:** Ein Spieler spielt allein.
- **Zwei Spieler:** Zwei Spieler kämpfen gegeneinander um den Sieg.

3. Anzahl der Eichhörnchen pro Spieler

- **Ein Eichhörnchen:** Jedes Team hat Kontrolle über ein Eichhörnchen.
- **Mehrere Eichhörnchen:** Jedes Team kontrolliert eine Population von Eichhörnchen.

4. Eichhörnchentypen

- **Ein Typ:** Alle Eichhörnchen haben die gleiche Geschwindigkeit und Kapazität.

- **Mehrere Typen:** Eichhörnchen können verschiedene Geschwindigkeiten und Kapazitäten haben.

5. Springen

- **Springen erlaubt:** Eichhörnchen können auf erforschte oder erforschten Feldern angrenzende Felder springen.
- **Springen nicht erlaubt:** Eichhörnchen können sich nur auf angrenzende Felder bewegen.

6. Sicht

- **Volle Sicht:** Jedes Team sieht das gesamte Spielfeld und hat somit volle Information.
- **Partielle Sicht:** Jedes Team sieht nur das Spielfeld in einem bestimmten Umkreis um die eigenen Spieler.

7. Bäume

- **Ja:** Bäume auf dem Spielfeld sind möglich.
- **Nein:** Es gibt keine Bäume auf dem Spielfeld.

8. Kosten

- **Ja:** Für jede Bewegung von einem Feld zu einem anderen sind spezifische Kosten festgelegt.
- **Nein:** Es sind keine Kosten festgelegt.

4.2 Vordefinierte Spielarten

Der Gamemaster stellt einige vordefinierte Spielarten zur Verfügung, welche im folgenden ausführlicher beschrieben werden.

4.2.1 Informierte/Uninformierte Suche

Ziel dieser Spielart ist es, einen Pfad von einem Startfeld zu einer Nuss auf einem Zielfeld zu finden. Um dieses Ziel zu erreichen, sollen verschiedene Suchalgorithmen angewandt werden. Dabei wird unterschieden zwischen informierter und uninformatierter Suche.

Uninformierte Suche Bei der uninformatierten Suche werden dem Spieler vor jedem Zug alle benachbarten Felder und die dazugehörigen Kosten übermittelt. Der Spieler hat die Möglichkeit auf jedes schon erforschte Feld oder jedes einem erforschten Feld angrenzenden Feld zu springen. Die einzelnen Felder werden dabei durch symbolische Positionen repräsentiert.

Um das Zielfeld zu erreichen, können Suchalgorithmen wie zum Beispiel Breitensuche, Tiefensuche, tiefenlimitierte Suche, iterative Tiefensuche oder Uniform Cost Search benutzt werden.

Informierte Suche Bei der informierten Suche benutzen die Suchalgorithmen eine heuristische Funktion um den nächsten Zug zu berechnen. Aus diesem Grund bekommt der Spieler zusätzlich zu den nächsten Zugmöglichkeiten und deren Kosten noch die Koordinaten des Start- und Zielfeldes gesendet, die für eine heuristische Funktion, wie zum Beispiel die Manhattan Distance, notwendig sind. Die Manhattan Distance beschreibt die Mindestanzahl an Zügen, die man braucht, um zum Ziel zu kommen. Für die informierte Suche einsetzbare Algorithmen sind zum Beispiel A* oder Greedy Search. Im Gegensatz zur uninformierten Suche werden hier die Felder durch ihre Koordinaten repräsentiert.

Folgende Parametereinstellungen werden für diese Spielart benutzt:

1. Spielziel: Nuss finden
2. Anzahl der Spieler: Ein Spieler
3. Anzahl der Eichhörnchen pro Spieler: Ein Eichhörnchen
4. Eichhörnchentypen: Ein Typ
5. Springen: Ja
6. Sicht: Nur das Feld, auf dem das Eichhörnchen steht, ist sichtbar.
7. Bäume: Ja
8. Kosten: Ja

4.2.2 Adversarial Search

Bei der Adversarial Search spielen zwei Spieler gegeneinander. Das Ziel eines jeden Spielers ist es mehr wertvolle Nüsse als der Gegner zu sammeln. Am Ende ist der Spieler mit dem höchsten Gesamtwert an gesammelten Nüssen der Sieger.

Bei dieser Art von Spielen wird unterschieden zwischen 'Perfect Information' und 'Imperfect Information':

Perfect Information Bei 'perfect information' Spielen hat jeder Spieler volle Sicht auf das Spielfeld, weiß somit auch immer wo der Gegner ist und welche Aktionen er ausgeführt hat. Für diese Spielart eignet sich vor allem der Minimax-Algorithmus, wahlweise mit Alpha-Beta-Pruning.

Im Folgenden werden die Parametereinstellungen für diese Spielart aufgelistet:

1. Spielziel: Alle möglich
2. Anzahl der Spieler: Zwei Spieler

3. Anzahl der Eichhörner pro Spieler: Ein/mehrere Eichhörner
4. Eichhörnentypen: Ein Typ
5. Springen: Nein
6. Sicht: Volle Sicht
7. Bäume: Ja/Nein
8. Kosten: Nein

Imperfect Information Bei 'Imperfect Information' Spielen hat der Spieler nur eine partielle Sicht zum Beispiel in einem bestimmten Abstand um die eigenen Eichhörner herum. Der Spieler weiß also nicht, was außerhalb dieses Sichtfeldes auf dem Rest des Spielfeldes passiert.

Für diese Spielart werden den Parametern folgende Werte zugeordnet:

1. Spielziel: Alle möglich
2. Anzahl der Spieler: Zwei Spieler
3. Anzahl der Eichhörner pro Spieler: Ein/mehrere Eichhörner
4. Eichhörnentypen: Ein/mehrere Typen
5. Springen: Nein
6. Sicht: Festgelegter Sichtradius um die eigenen Eichhörner
7. Bäume: Ja/Nein
8. Kosten: Nein

5 Kommunikation

Während des gesamten Spiels herrscht eine ständige Kommunikation zwischen dem Gamemaster und den Spielern (siehe auch Abbildung 7 und 8 aus Kapitel 3). Die Spieler melden sich beim Gamemaster an und übermitteln ihre Spielzüge, der Gamemaster wertet diese Spielzüge aus und sendet den Spielern die Veränderungen. Dabei wird den Spielern überlassen, ob sie einem Spiel beitreten oder ein eigenes Spiel starten wollen. Die bei der Kommunikation verwendete Sprache kann von einer formalen Grammatik (siehe Abschnitt 5.1) abgeleitet werden.

Die in dieser Arbeit verwendete Kommunikation zwischen Gamemaster und Spieler kann, wie in Kapitel 3 vorgestellt, in drei Phasen eingeteilt werden:

1. Start (siehe Abschnitt 5.2)
2. Spiel (siehe Abschnitt 5.3)
3. Ende (siehe Abschnitt 5.4)

Die Kommunikation zwischen dem Gamemaster und den Spielern basiert auf XML-RPCs (Extensible Markup Language - Remote Procedure Calls). Mit Hilfe von XML-RPCs (Abb. 10) kann der Spieler Funktionen beim Gamemaster aufrufen und ihnen außerdem wahlweise Parameter übergeben. Der Aufruf und eventuelle Parameter werden per XML verpackt und dann per HTTP zum Gamemaster geschickt, von dem er wiederum entpackt und ausgeführt wird. Die Antwort des Gamemasters wird dann ebenfalls verpackt und an den Spieler zurückgeschickt. Durch XML als Containersprache ist die Programmiersprache, in der der Gamemaster programmiert wurde, unabhängig von der Programmiersprache, in der die Agenten implementiert werden, solange die jeweilige Programmiersprache XML-RPCs unterstützt. Das macht Agenten und Gamemaster voneinander unabhängig. So wäre es möglich, daß die Teilnehmer der Vorlesung die Agenten in einer beliebigen, XML-RPC unterstützenden, Sprache programmieren.

5.1 Grammatik

Die gesamte bei der Kommunikation verwendete Sprache kann von einer formalen Grammatik (Abb. 11) abgeleitet werden. Nichtterminale sind, im Gegensatz zu Terminalen, von `<` und `>` umgeben. Auf der rechten Seite der Grammatik stehen stets Nichtterminale, welche sich entweder in Nichtterminale, Terminale oder eine Mischung aus beiden ableiten lassen. Im folgenden werden die einzelnen Produktionsregeln der Grammatik näher beschrieben. Wo die einzelnen Produktionsregeln angewendet werden, wird in den folgenden Abschnitten der Start-, Spiel- und Endphase (5.2, 5.3, 5.4) gezeigt.

`<StartParams>/<PlayParams>/<EndParams>` beinhaltet die Startinformationen/Updates im Spiel/abschließenden Informationen, die der Gamemaster an den Spieler sendet.

`<StartPos>/<GoalPos>` ist die Position des Feldes, auf dem das Eichhörnchen startet / welches das Eichhörnchen erreichen soll. Dies wird zum Beispiel bei der informierten Suche benutzt.

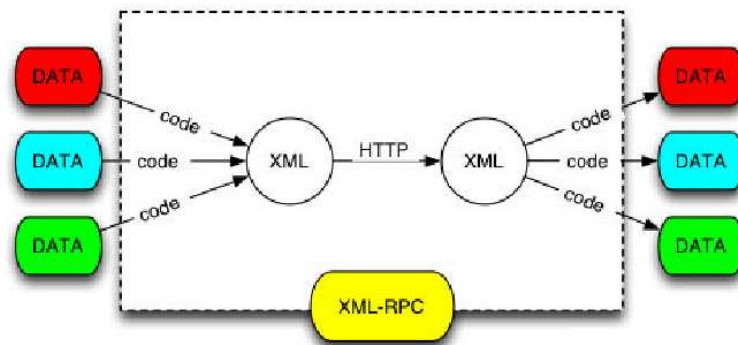


Abbildung 10: Funktionsweise XML-RPC, entnommen von <http://www.xmlrpc.com>. Die Daten einer beliebigen, XML-RPCs unterstützenden Programmiersprache werden in XML verpackt und per HTTP an den Empfänger gesendet, der sie von XML in die verwendete Programmiersprache übersetzt.

<PossibleMoves> gibt die möglichen Züge an, die das Eichhörnchen machen kann (z.B. bei informierter Suche). Die Zugvorschläge werden durch **<MoveCost>** repräsentiert, welches, je nach Spielart, die symbolische Repräsentation (**<SPosition>**) oder die Repräsentation durch die Koordinaten eines Feldes (**<Position>**) und die Kosten (**<Costs>**) enthält.

<ObjectDescription> enthält die Informationen über die eigenen Objekte (**<OwnObjects>**), die gegnerischen Objekte (**<Oppobjects>**) und die sonstigen Objekte (**<MiscObjects>**) auf dem Spielfeld. Eigene Objekte sind Eichhörnchen (**<Squirrels>**) und die Sammelstelle (**<OwnGP>**). Gegnerische Objekte sind Eichhörnchen, die aber nur durch ihren Bezeichner (**<sbez>**) dargestellt werden und die symbolische Position der gegnerischen Sammelstelle (**<OppGP>**). Die sonstigen Objekte sind Bäume (**<Tree>**) und Nüsse (**<Nut>**).

<Squirrels> sind ein oder mehrere Eichhörnchen. Jedes Eichhörnchen hat einen Bezeichner (**<sbez>**), einen Typ (**<stype>**) und steht auf einem Feld (**<Position>**).

<sbez> ist der Bezeichner eines Eichhörnchens, welcher aus dem Wort 'Squirrel', gefolgt von einer natürlichen Zahl zwischen 1 und 1000, besteht.

<stype> ist der Typ eines Eichhörnchens inklusive seiner Kapazität (**<Capacity>**). Der Typ kann entweder 'fast' oder 'slow' sein. Die Kapazität eines Eichhörnchens ist eine natürliche Zahl zwischen 1 und 1000.

<Nuts> kann zu einer oder mehreren Nüssen abgeleitet werden. Eine Nuss **<Nut>** besteht aus ihrem Typ **<nbez>**, welcher wiederum aus dem Wort 'Nut', einer natürlichen Zahl zwischen 1

und 1000, aus ihrem Gewicht <Weight> und aus ihrem Wert <Value>, ebenfalls eine natürliche Zahlen zwischen 1 und 1000.

<Trees> kann zu einem oder mehreren Bäumen abgeleitet werden. Ein Baum <Tree> besteht aus seinem Typ <tbez>, welcher wiederum aus dem Wort 'Tree' und einer natürlichen Zahl zwischen 1 und 1000 zusammengesetzt ist, aus dem Typ Nuss (<nbez>) und dem Feld (<Position>), auf dem er sich befindet.

<Result> gibt den Gewinner des Spiels in einem 2-Spieler-Spiel an.

<Map> ist das Spielfeld, welches aus einzelnen Feldern (<Field>) besteht. Ein Feld besteht aus seiner Position (<Position>) und einer Liste von Objekten (<Objects>). Ein Objekt (<Object>) kann entweder ein Eichhörnchen, eine Nuss oder ein Baum sein.

<Move> ist der Zug, den ein Spieler schickt. Ein Zug besteht aus dem Eichhörnchen, für welches der Zug bestimmt ist, einem Kommando (<Command>) und den Parametern (<MParams>). Bei einem Zug vom Typ 'Go' ist ein Parameter ein Feld, bei einem Zug der Art 'Take' bzw. 'Drop' ist ein Parameter die Nuss, die aufgehoben bzw. hingelegt werden soll. Bei einem Zug der Art 'Give' ist es einerseits die Nuss, die abgegeben werden soll, andererseits das Eichhörnchen, an welches die Nuss gegeben werden soll.

<Error> steht für die Fehler, die ein Zug eines Spielers verursachen kann. Ein Fehler beinhaltet einerseits den Errorcode (<ErrorCode>), der aus einer natürlichen Zahl besteht und der Nummer des Zuges aus der Zugliste, in dem der Fehler aufgetreten ist (<ErrorMoveNumber>). Jede Zahl steht für einen bestimmten Fehler. Desweiteren gehört zu einem Fehler die Fehlerart (<ErrorType>), die vom Typ schwer (hard) oder leicht (soft) sein kann, und die Fehlerbeschreibung (<ErrorDesc>), die in einem Satz den Fehler näher beschreibt.

5.2 Startphase

In der Startphase (Abb. 12) melden sich die Spieler beim Gamemaster an. Für jeden Spieler wird eine eindeutige ID generiert und dem Spieler zugewiesen, mit der er sich bei der späteren Kommunikation authentifizieren kann. Falls die generierte ID schon belegt ist, wird solange eine neue ID generiert, bis diese noch nicht von einem anderen Spieler belegt ist.

Handelt es sich um ein 2-Spieler-Spiel wird ein Gegner gesucht. Ist zur Zeit kein Gegner vorhanden, muss solange gewartet werden, bis sich ein neuer Spieler mit dem Server verbindet und als Gegner bereitsteht.

Sobald ein Gegner vorhanden ist, beginnt das Spiel mit dem Senden der Startinformationen an den ersten Spieler mit dem Kommando (Start <StartParams>). Je nach Spielart variieren die Spielparameter (<StartParams>), die dem Spieler übermittelt werden.

<StartParams>	→	[<StartPos> <GoalPos> <PossibleMoves> <ObjectDescription> <Map>]+
<PlayParams>	→	[<Map> <Error>]+
<EndParams>	→	<Result>
<StartPos>	→	(StartPos <Position>)
<Position>	→	(X X)
<GoalPos>	→	(GoalPos <Position>)
<PossibleMoves>	→	(PosMoves [<MoveCost>]+)
<MoveCost>	→	({<SPosition> <Position>} <Costs>)
<SPosition>	→	[A-Z]+
<ObjectDescription>	→	(ObjDesc <OwnObjects> [<OppObjects>] [<MiscObjects>])
<OwnObjects>	→	(OwnObj <Squirrels> <OwnGP>)
<Squirrels>	→	([<Squirrel>]+)
<Squirrel>	→	(Squirrel <sbez> <Capacity> <stype> <Position>)
<sbez>	→	Squirrel<X>
<X>	→	[1-1000]
<stype>	→	(fast slow <Capacity>)
<Capacity>	→	<X>
<OwnGP>	→	(OwnGP <gpbez>)
<gpbez>	→	gp<X>
<OppObjects>	→	(OppObj [<sbez>]* <OppGP>)
<OppGP>	→	(OppGP <gpbez>)
<MiscObjects>	→	(MiscObj [<Nuts>] [<Trees>])
<Nuts>	→	([<Nut>]+)
<Nut>	→	(Nut <nbez> <Weight> <Value>)
<nbez>	→	Nut<X>
<Weight>	→	<X>
<Value>	→	<X>
<Trees>	→	([<Tree>]+)
<Tree>	→	(Tree <tbez> <nbez>)
<tbez>	→	Tree<X>
<Result>	→	Win Loss Tie
<Map>	→	(Map <Fields>)
<Fields>	→	([<Field>]+)
<Field>	→	(<Position> (<Objects>))
<Objects>	→	[<Object>]*
<Object>	→	(<Squirrel> <Nut> <Tree>)
<Move>	→	(<sbez> <Command> <MParams>) (<SPosition>) (<Position>)
<Command>	→	Go Take Drop Give
<MParams>	→	<Position> <nbez> [<sbez>]
<Costs>	→	<X>
<Error>	→	(Error <ErrorCode> <ErrorMoveNumber> <ErrorType> <ErrorDesc>)
<ErrorCode>	→	<X>
<ErrorMoveNumber>	→	<X> 0
<ErrorType>	→	hard soft
<ErrorDesc>	→	[a-zA-Z]+

Abbildung 11: Formale Grammatik zur Ableitung der zur Kommunikation verwendeten Sprache zwischen Gamemaster und Spieler. Nichtterminale sind im Gegensatz zu Terminalen stets in '<' und '>' eingeschlossen.

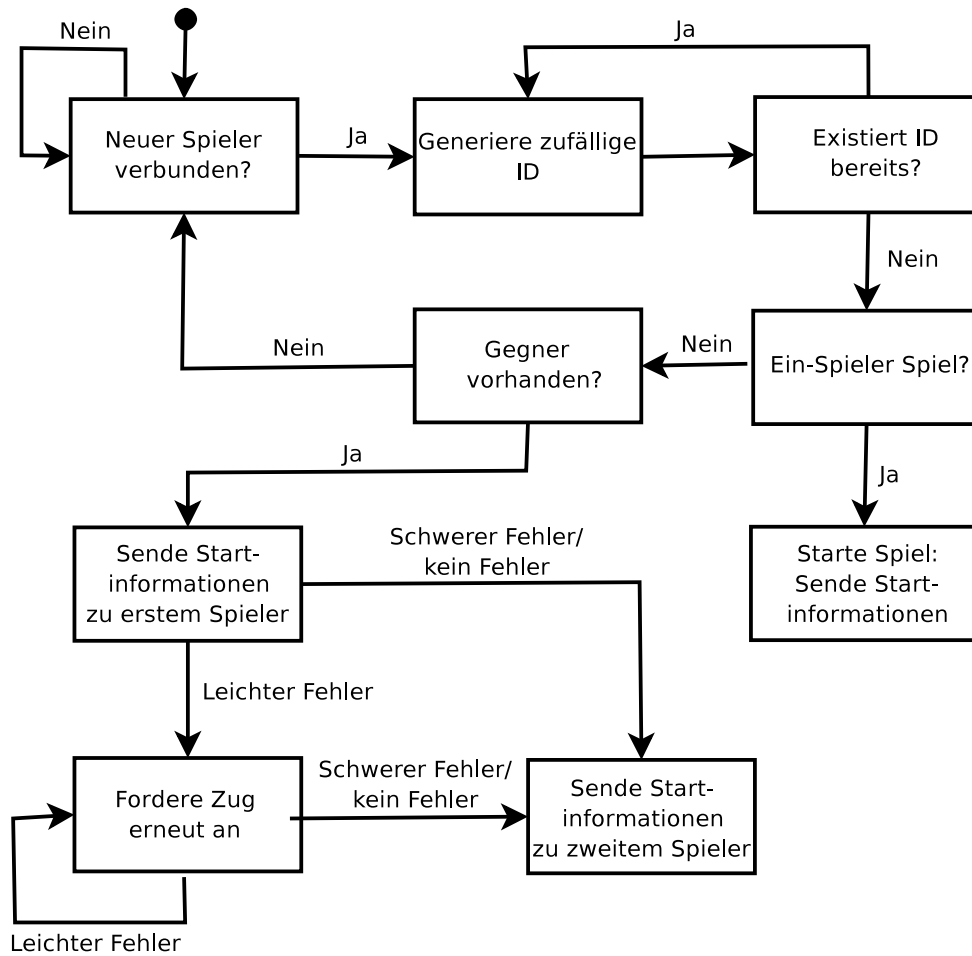


Abbildung 12: Serverseitiger Ablauf der Startphase. Einem neu verbundenen Spieler wird eine einzigartige ID vergeben und ggf. ein Gegner gesucht. Das Spiel beginnt mit dem Senden der Startinformationen.

Im Folgenden wird an zwei Beispielen, nämlich informierte Suche und Adversarial Search, gezeigt, welche Startinformationen vom Gamemaster an den Spieler gesendet werden. Außerdem wird die Bedeutung jeder einzelnen Startinformation erklärt.

- **Beispiel für Startinformationen bei informierter Suche**

Folgende Startinformationen werden dem Spieler vom Gamemaster mitgeteilt:

Gamemaster → **Spieler:** (Start <StartPos> <GoalPos> <PossibleMoves>)

Beschreibung der Startinformationen:

<**StartPos**> und <**GoalPos**> teilen dem Spieler seine Start- und Zielposition mit zur Verwendung für eine Heuristik.

<**PossibleMoves**> teilt dem Spieler seine Möglichkeiten für den nächsten Zug und die damit verbundenen Kosten mit, aus denen dann der Spieler aufgrund seiner Berechnungen eine auswählt.

- **Beispiel der Startinformationen bei Adversarial Search**

Der Gamemaster schickt die folgenden Startinformationen an den Spieler:

Gamemaster → **Spieler:** (Start <ObjectDescription> <Map>)

Beschreibung der Startinformationen:

<**ObjectDescription**> gibt dem Spieler alle nötigen Informationen über die eigenen Eichhörnchen, welche durch Bezeichner, Position, Typ und Kapazität spezifiziert sind, die Bezeichner der gegnerischen Eichhörnchen, Sammelstellen, Bäume und Nüsse.

<**Map**> teilt dem Spieler Informationen über die Karte, das heißt sichtbare Felder und den darauf befindlichen Objekten, mit.

5.3 Spielphase

Nachdem der Spieler die benötigten Startinformationen erhalten hat, muss er dem Gamemaster nun mit einer Zugliste antworten. Eine Zugliste kann aus keinem, einem oder mehreren Zügen bestehen. Sie darf aber höchstens einen Zug für pro Eichhörnchen enthalten.

In zwei Beispielen für die informierte Suche und Adversarial Search wird gezeigt, welche Kommandos der Spieler an den Gamemaster schickt. Anschließend werden die einzelnen Bestandteile der Kommandos näher beschrieben.

- **Beispiel eines Zuges bei informierter Suche**

Spielzug des Spielers:

Spieler → **Gamemaster:** (<Position>)

Beschreibung des Spielzuges:

<**Position**> gibt an, wohin das Eichhörnchen bewegt werden soll. Da bei dieser Spielart lediglich ein Eichhörnchen vorhanden ist und auch nur das 'Go'-Kommando benutzt

wird, wurde der Zug hier vereinfacht. So ist es nicht notwendig, daß der Spieler den Bezeichner des Eichhörnchens und die Art des Zuges übermittelt, sondern es reicht aus, dem Gamemaster mitzuteilen, wohin das Eichhörnchen als nächstes gehen soll.

- **Beispiel eines Zuges bei Adversarial Search**

Der Spieler kann dem Gameserver folgende Kommandos schicken:

Spieler → Gamemaster:

(<Sbez> Go <Position>)
(<Sbez> Take <Nbez>)
(<Sbez> Drop <Nbez>)
(<Sbez> Give <Nbez> <Sbez>)

Beschreibung der einzelnen Bestandteile der Kommandos:

<**Sbez**> ist der Bezeichner für das Eichhörnchen, für welches das Kommando bestimmt ist.

Go, Take, Drop, Give ist das Kommando, welches das Eichhörnchen ausführen soll.

<**Position**> gibt an, wohin sich das Eichhörnchen bewegen soll.

<**NBez**> gibt an, welcher Nusstyp aufgenommen bzw. abgegeben werden soll.

5.3.1 Repräsentation der Objekte im Gamemaster

Die Züge, sowie alle Objekte und der Spielzustand selber werden im Gamemaster objektorientiert repräsentiert. Praktische Anwendung findet diese Repräsentation zum Beispiel bei den weiter unten folgenden Überprüfungen (siehe Abschnitt 5.3.2). Mit Hilfe des Punktoperators ist es möglich, die verschiedenen Eigenschaften eines Objektes abzurufen. Beispielsweise kann man das Kommando ('Go', 'Take', 'Drop' oder 'Give') eines Zuges m per $m.command$ abrufen.

Objekte können aber auch wiederum Objekte enthalten. Angenommen, es gibt ein Objekt $m \in Move$, dann kann man per $m.squirrel$ auf das Eichhörnchen zugreifen, welches den Befehl in Zug m ausführen soll, um dann per $(m.squirrel).type$ den Typ des Eichhörnchens abzufragen.

Der folgende Abschnitt befasst sich mit den verschiedenen Typen der Objekte. Zuerst wird der Typ aufgelistet und es wird beschrieben, um was für einen Typ es sich handelt. Anschließend

werden jeweils die Attribute des Typs, auf die man mit Hilfe des Punktoperators zugreifen kann, ausführlicher beschrieben.

- **Move**

Beschreibung Ein Objekt vom Typ *Move* repräsentiert den Zug, den ein Spieler an den Gamemaster schickt.

Objekteigenschaften Sei $m \in Move$. Auf folgende Bestandteile kann zugegriffen werden:

<i>m.squirrel</i>	Das Eichhörnchen, welches den Befehl ausführen soll
<i>m.command</i>	Der Befehl, der vom Eichhörnchen ausgeführt werden soll
<i>m.params</i>	Die Parameter, die zum Befehl gehören. Da bei einem Zug der Art 'Give' zwei Parameter benötigt werden, kann man in dem Fall per <i>m.params</i> [0] bzw. <i>m.params</i> [1] auf den ersten bzw. den zweiten Parameter zugreifen. Die Parameter sind die Nuss, die übergeben wird, sowie das Eichhörnchen, welches die Nuss in Empfang nimmt.

- **Squirrel**

Beschreibung Objekte vom Typ *Squirrel* repräsentieren die Eichhörnchen, die im Spiel vorkommen.

Objekteigenschaften Sei $s \in Squirrel$. Auf folgende Bestandteile kann zugegriffen werden:

<i>s.type</i>	Der Typ, den das Eichhörnchen hat
<i>s.position</i>	Die Position, auf der sich das Eichhörnchen befindet
<i>s.capacity</i>	Die verbleibende Kapazität des Eichhörnchens
<i>s.objects</i>	Eine Liste mit Objekten, die das Eichhörnchen mit sich trägt

- **Nut**

Beschreibung Objekte vom Typ *Nut* repräsentieren die Nüsse, die im Spiel vorkommen.

Objekteigenschaften Sei $n \in Nut$. Auf folgende Bestandteile kann zugegriffen werden:

$n.weight$ Das Gewicht der Nuss
 $n.value$ Der Wert der Nuss

- **Field**

Beschreibung Objekte vom Typ *Field* repräsentieren die einzelnen Felder des Spielfeldes.

Objekteigenschaften Sei $f \in Field$. Auf folgende Bestandteile kann zugegriffen werden:

$f.coordinates$ Die X- und Y-Koordinaten des Feldes
 $f.objects$ Eine Liste mit Objekten, die sich zur Zeit auf dem Feld befinden

- **Tree**

Beschreibung Objekte vom Typ *Tree* repräsentieren die Bäume, die im Spiel vorkommen.

Objekteigenschaften Sei $t \in Tree$. Auf folgende Bestandteile kann zugegriffen werden:

$t.position$ Das Feld, auf dem sich der Baum befindet

- **Team**

Beschreibung In den Objekten des Typs *Team* sind alle Objekte eines Spielers sowie dessen ID enthalten.

Objekteigenschaften Sei $t \in Team$. Auf folgende Bestandteile kann zugegriffen werden:

$t.id$ Die ID des Spielers
 $t.squirrels$ Die Eichhörnchen, die dem Team angehören
 $t.gp$ Die Sammelstelle des Teams

- **Spielzustand**

Beschreibung Jedes Spiel hat einen Spielzustand, der über Informationen über das Spielfeld und alle sich darauf befindlichen Objekte verfügt.

Objekteigenschaften Sei gs ein Spielzustand. Auf folgende Bestandteile kann zugegriffen werden:

<i>gs.squirrels</i>	Die Menge der Eichhörnchen, welche im Spiel existieren
<i>gs.nuts</i>	Die Menge der Nüsse, welche im Spiel existieren
<i>gs.trees</i>	Die Menge der Bäume, welche im Spiel existieren
<i>gs.fields</i>	Die Menge der Felder, welche im Spiel existieren
<i>gs.teams</i>	Die Menge der Teams, welche im Spiel existieren
<i>gs.canMove</i>	Liste der Eichhörnchen, welche in dieser Runde noch nicht bewegt wurden
<i>gs.teamid</i>	Die ID des Teams, welches gerade am Zug ist
<i>gs.jump</i>	Prädikat, welches den Wert wahr annimmt, wenn Springen erlaubt ist, ansonsten nimmt es den Wert falsch an.

gs.canMove ist eine Liste mit den Eichhörnchen des Teams, welches gerade an der Reihe ist. Ganz zu Beginn des Zuges sind alle Eichhörnchen eines Teams eingetragen. Wurde ein Zug mit einem Eichhörnchen s durchgeführt, wird dieses Eichhörnchen aus *gs.canMove* gestrichen. So weiß der Gamemaster immer, welche Eichhörnchen eines Teams sich noch bewegen dürfen und welche nicht.

5.3.2 Überprüfungen

Der Gamemaster muss nun jeden einzelnen Zug aus der Zugliste auswerten, weswegen er eine Reihe von Überprüfungen durchführt. Abbildung 13 zeigt, wie der Gamemaster dabei vorgeht.

Der Server arbeitet die vom Spieler erhaltene Zugliste Zug für Zug ab. Zuerst wird der ausgewählte Zug einer generellen Überprüfung unterzogen. Diese befasst sich damit, ob der Spieler am Zug ist, ob das Eichhörnchen dem Team des Spielers angehört und ob das Eichhörnchen in dieser Runde schon einmal bewegt wurde.

Als nächstes folgt eine spezifische Überprüfung, da für jede Zugart (Go, Take, Drop, Give) andere Bedingungen überprüft werden müssen. So muss bei einem Go-Zug überprüft werden, ob das Feld, auf das das Eichhörnchen gehen will, überhaupt existiert und ob es begehbar ist. Dagegen muss bei einem Drop-Zug nur das Feld, auf dem das Eichhörnchen steht, darauf untersucht werden, ob eine Nuss dort abgelegt werden kann.

Wird bei beiden Überprüfungen kein Fehler entdeckt, werden die Veränderungen, die der Zug bewirkt, vorläufig in den Spielzustand (gs , Abkürzung für Game State) übertragen. Vorläufig deswegen, weil ein späterer Zug aus der Zugliste Fehler enthalten kann und deswegen alle Änderungen, die eventuelle vorherige Züge aus der Zugliste bewirkt haben, nicht mehr gültig

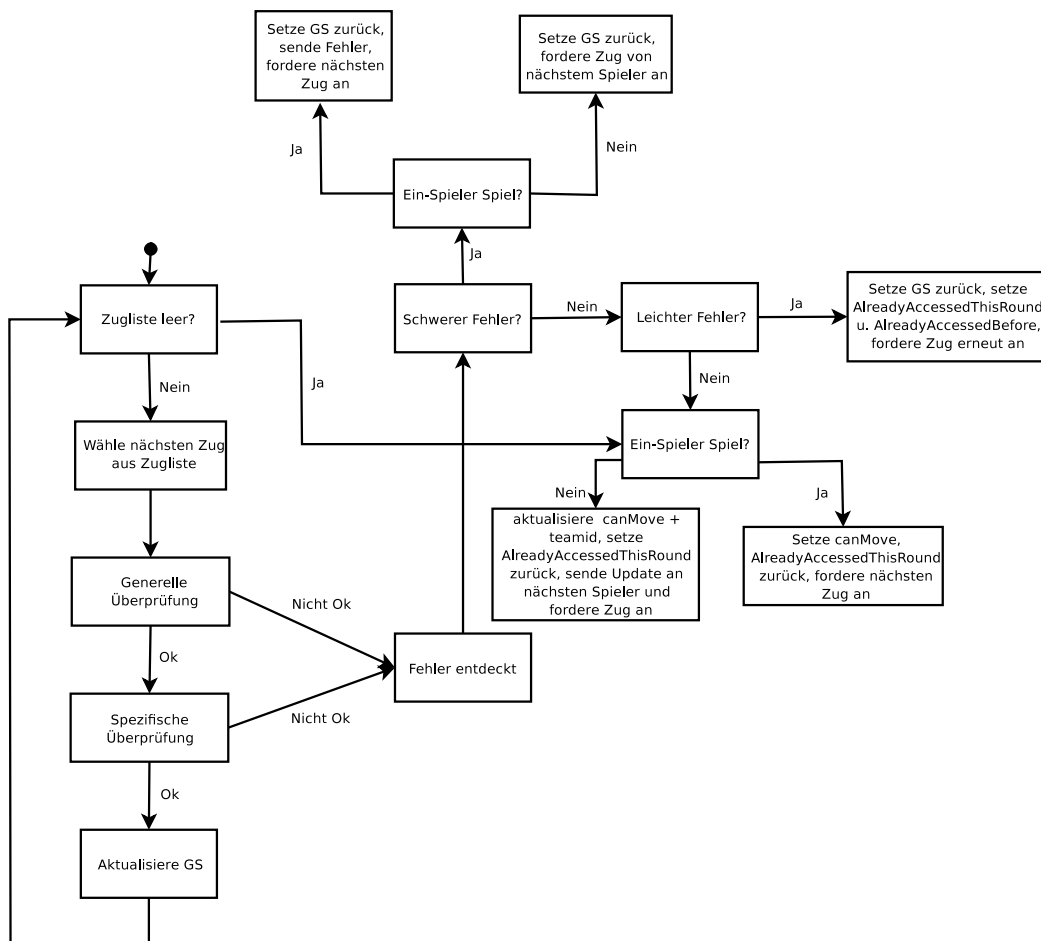


Abbildung 13: Vorgehensweise bei der Überprüfung einer Zugliste. Jeder einzelne Zug wird auf Korrektheit überprüft. Tritt ein Fehler auf, wird der Fehler je nach Art entsprechend behandelt, bis schließlich der Zug zu Ende ist und der nächste Zug angefordert wird.

sind. Nur wenn alle Züge aus der Zugliste fehlerfrei sind, werden die Updates des Spielzustandes endgültig. Wurde also der Spielzustand aktualisiert, wird der nächste Zug aus der Zugliste überprüft. Wird bei einem Zug ein Fehler entdeckt, wird der Spielzustand in den Zustand, den er vor dem Senden der Zugliste hatte, zurückgesetzt.

Bei Fehlern wird zwischen leichten und schweren Fehlern unterschieden. Tritt ein harter Fehler in einem Ein-Spieler-Spiel auf, wird dem Spieler der Fehler mitgeteilt und der Spieler wird aufgefordert, den nächsten Zug zu senden. Da diese Runde nun beendet ist, darf der Spieler wieder alle seine Eichhörnchen bewegen. Deshalb muss nun *gs.canMove* wieder alle Eichhörnchen des Spielers enthalten.

In einem Zwei-Spieler-Spiel ist in diesem Fall die Runde für den Spieler beendet. Er bekommt in der nächsten Runde seinen Fehler gesendet und darf seine Eichhörnchen erst dann wieder be-

fehligen. Da nun der nächste Spieler an der Reihe ist, muss *gs.teamid* auf die ID des Spielers gesetzt werden, der nun an der Reihe ist und *gs.canMove* muss alle Eichhörnchen dieses Spielers enthalten. Der Spieler, welcher nun an der Reihe ist, bekommt nun selber, falls vorhanden, seinen Fehler aus der letzten Runde und die sichtbaren Veränderungen auf dem Spielfeld gesendet und darf seine Zugliste schicken.

Ist ein leichter Fehler aufgetreten, hat der Spieler die Chance, seine Zugliste erneut zu senden. Da der Spieler nun wieder alle Eichhörnchen bewegen darf, muss *gs.canMove* wieder auf den Zustand vor dem Zug gesetzt werden. Außerdem werden *AlreadyAccessedThisRound* und *AlreadyAccessedBefore* (siehe Abschnitt 5.3.3 über Fehler) auf *wahr* gesetzt, da ein Eichhörnchen versucht hat, ein besetztes Feld zu betreten. Am Ende der Runde wird *AlreadyAccessed* allerdings wieder zurückgesetzt, da es nur für die Runde gültig ist, in der es gesetzt wird. *AlreadyAccessedBefore* bleibt, wenn es einmal auf 'wahr' gesetzt wurde, das ganze Spiel auf 'wahr'.

Wurde die komplette Zugliste fehlerfrei abgearbeitet, sind alle Änderungen an dem Spielzustand gültig und er ist nicht mehr vorläufig. Im Falle eines 1-Spieler-Spiels enthält *gs.canMove* nun wieder alle Eichhörnchen des Spielers und der nächste Zug wird vom Spieler angefordert. Handelt es sich um ein 2-Spieler-Spiel, werden alle Eichhörnchen des nächsten Spielers in *gs.canMove* eingetragen. Dann werden dem Spieler, der nun an der Reihe ist, die sichtbaren Änderungen des letzten Zuges mitgeteilt und er darf seinerseits seine Zugliste senden.

Als nächstes folgt eine ausführlichere Beschreibung, welche Überprüfungen bei jedem Zug eines Spielers durchgeführt werden. Die Überprüfungen werden mit Hilfe von Inferenzregeln erklärt. Sind die Prämissen, die oberhalb des Striches stehen wahr, so ist auch die Konklusion unterhalb des Striches wahr. Ist die Prämisse leer, so ist die Konklusion immer wahr. Verändert sich ein Objekt x , so wird dies mit einem Apostroph verdeutlicht. So beschreibt x den Zustand vor der Veränderung und x' den Zustand nach der Veränderung.

Erhält der Gamemaster eine Zugliste von einem Spieler, so wird jeder Zug $m \in Move$ einzeln überprüft. Der Spielzustand wird hier durch *gs* repräsentiert.

$$\frac{\begin{array}{l} CheckMoveGeneral(gs, m) \\ CheckSpecificMove(gs, m) \\ updateGS(gs, m) \rightarrow gs' \end{array}}{gs; m \in Move \rightarrow gs'}$$

Die obige Inferenzregel sagt aus, daß jeder Zug m eine generelle Überprüfung und eine spezifische Überprüfung bestehen muss, damit der Zug Gültigkeit besitzt. Besteht der Zug die Überprüfungen, wird *gs* aktualisiert.

Die nächste Inferenzregel beschreibt rekursiv die Überprüfung einer ganzen Zugliste:

$$\frac{gs; m \rightarrow gs' \quad gs'; ml \rightarrow gs''}{gs; m::ml \rightarrow gs''}$$

Die Zugliste, bestehend aus dem Zug m und dem Rest der Zugliste ml , wird überprüft, indem zunächst der erste Zug der Zugliste überprüft wird, welcher, wenn er fehlerfrei ist, gs in gs' überführt. Anschließend wird die restliche Zugliste überprüft. Ist auch diese fehlerfrei, werden die Änderungen eingetragen und gs' geht somit in gs'' über. Daraus folgt, daß, nur wenn sowohl Zug, als auch Zugliste fehlerfrei sind, die Änderungen in dem Spielzustand eingetragen werden und gs damit nach gs'' übergeht.

Da die obige Regel die Überprüfung rekursiv beschreibt, muss irgendwann die restliche Zugliste leer sein, damit die Rekursion terminiert. Die folgende Regel beschreibt den Vorgang, falls die Zugliste leer ist:

$$\overline{gs; \epsilon \rightarrow gs}$$

Da eine leere Zugliste keine Veränderungen bewirkt, bleibt auch der Spielzustand unverändert. Somit ist die leere Zugliste fehlerfrei.

In der ersten Inferenzregel wird schon deutlich, daß es zwei Überprüfungen, nämlich die generelle und die spezifische Überprüfungen, gibt. Zunächst wird die generelle Überprüfung näher betrachtet und anschließend wird ausführlich auf die spezifische Überprüfung eingegangen.

Generelle Überprüfung Bei der generellen Überprüfung eines Zuges $m \in Move$ wird überprüft, ob das Eichhörchen, welches das Kommando ausführen soll, in dem Spielzustand existiert. Außerdem muss das Eichhörchen dem Team des Spielers angehören, der den Zug m geschickt hat. Darüberhinaus muss überprüft werden, ob das Eichhörchen in diesem Zug schon einmal bewegt wurde. Die generelle Überprüfung gilt als bestanden, wenn der Zug m folgende Bedingung erfüllt:

$$\exists t \in gs.teams : t.id = gs.teamid \wedge m.squirrel \in gs.squirrels \wedge m.squirrel \in t.objects \wedge m.squirrel \in gs.canMove$$

Spezifische Überprüfung Jeder Zug wird nach der generellen Überprüfung einer je nach Zugart (Go, Take, Drop, Give) spezifischen Überprüfung unterzogen. Dafür werden einige Hilfsfunktionen benötigt:

$adjacent(f, g, s)$ überprüft, ob ein Feld f an ein anderes Feld g angrenzt. Welche Felder als angrenzende gelten, hängt vom Typ des Eichhörchens ab. Ist das Eichhörchen schnell, gelten auch diagonal anliegende Felder als angrenzend. Ansonsten gelten nur die Felder über, unter, rechts und links von dem Ausgangsfeld als angrenzend:

$$[f, g \in gs.fields \wedge f.coordinates = (a, b) \wedge g.coordinates = (x, y) \wedge [s.type = slow \wedge ((a = x \wedge (b = y + 1 \vee b = y - 1)) \vee (b = y \wedge (a = x + 1 \vee a = x - 1)))] \vee [s.type = fast \wedge ((a, b) \neq (x, y) \wedge (a \in [x - 1, x + 1] \wedge b \in [y - 1, y + 1])]]$$

$$\Leftrightarrow adjacent(f, g, s)$$

$isAccessibleField(f)$ überprüft, ob ein Feld f von einem Eichhörnchen betreten werden darf, das heißt, ob kein anderes Eichhörnchen oder kein Baum auf dem Feld steht:

$$f.objects \subseteq gs.nuts \Leftrightarrow isAccessibleField(f)$$

$isDropableAtField(f)$ überprüft, ob ein Eichhörnchen eine Nuss auf ein Feld f legen darf, das heißt, ob nicht bereits eine Nuss auf dem Feld liegt. Ausnahme ist die Sammelstelle, auf der beliebig viele Nüsse abgelegt werden können:

$$[\forall n \in Nut : n \notin f.objects \vee \exists t \in Team : t.id = gs.teamid \wedge f = t.gp]$$

$$\Leftrightarrow isDropableAtField(f)$$

$isExploredField(f, gs.teamid)$ überprüft, ob Feld f schon von dem Team mit ID $gs.teamid$ erkundet wurde.

Im Folgenden werden die spezifischen Überprüfungen für jede Zugart (Go, Take, Drop, Give) aufgelistet. Zunächst werden die jeweiligen Überprüfungen formal mit Hilfe der objektorientierten Repräsentation (siehe Abschnitt 5.3.1) aufgeführt und anschließend näher erläutert. Jeder Zug m hat, wie in der Grammatik beschrieben, die Grundform ($\langle sbez \rangle \langle Command \rangle \langle MParams \rangle$).

- **Go**

Folgende Bedingungen werden überprüft:

1. $m.command = "Go"$
2. $m.params \in gs.fields$
3. $(gs.jump \wedge \exists f \in gs.fields : isExploredField(f, gs.teamid) \wedge adjacent(f, m.params, m.squirrel)) \vee (adjacent((m.squirrel).position, m.params, m.squirrel))$
4. $isAccessibleField(m.params)$

Beschreibung Als erstes wird überprüft, ob das Kommando des Zuges korrekt ist. Außerdem muss der Parameter des Zuges ein existierendes Feld sein. Danach wird überprüft, ob das Feld, auf dem das Eichhörnchen zur Zeit steht, und das Feld, auf welches das Eichhörnchen gehen will, aneinander angrenzen und ob das Feld betretbar ist.

- **Take**

1. $m.command = "Take"$
2. $m.params \in gs.nuts$
3. $\exists f \in Field : f \in gs.fields \wedge (m.squirrel).position = f \wedge m.params \in f.objects$
4. $(m.squirrel).capacity \geq (m.params).weight$

Beschreibung Zuerst wird überprüft, ob das Kommando des Zuges korrekt ist und ob der Parameter eine existierende Nuss ist. Als nächstes wird überprüft, ob die Nuss, welche das Eichhörnchen aufnehmen will, auch auf dem Feld liegt, auf dem das Eichhörnchen steht, und ob das Eichhörnchen noch über ausreichend Kapazität verfügt, um die Nuss aufzunehmen.

- **Drop**

1. $m.command = "Drop"$
2. $m.params \in gs.nuts$
3. $m.params \in (m.squirrel).objects$
4. $isDropableAtField((m.squirrel).Position)$

Beschreibung Auch hier wird zunächst überprüft, ob das Kommando des Zuges korrekt ist und ob der Parameter eine existierende Nuss ist. Außerdem muss die Bedingung gegeben sein, daß auf dem Feld, auf dem das Eichhörnchen steht, noch keine Nuss liegt. Damit eine Nuss abgelegt werden kann, muss sich die Nuss, die das Eichhörnchen ablegen will, im Besitz des Eichhörnchens befinden.

- **Give**

1. $m.command = "Give"$
2. $m.params \in (Nut \times Squirrel)$
3. $m.params[0] \in gs.nuts$
4. $m.params[1] \in gs.squirrels$
5. $\exists t \in gs.teams : t.id = teamid \wedge m.params[1] \in t.objects$
6. $adjacent((m.squirrel).position, (m.params[1]).Position, m.squirrel)$
7. $(m.params[1]).capacity \geq (m.params[0]).weight$

Beschreibung Hier wird ebenfalls überprüft, ob das Kommando korrekt ist und ob die Parameter des Zuges eine Nuss und ein Eichhörnchen sind. Danach wird überprüft, ob sich beide Eichhörnchen an angrenzenden Feldern befinden und ob das Eichhörnchen, welches die Nuss annimmt, zum selben Team gehört und noch genug Kapazität frei hat.

5.3.3 Fehler

Bei Fehlern wird zwischen leichten und schweren Fehlern unterschieden. Tritt ein leichter Fehler auf, wird dies dem Spieler direkt mitgeteilt und er hat die Möglichkeit seinen Zug erneut zu senden, ohne daß Nachteile für ihn entstehen. Tritt aber innerhalb eines Zug ein schwerer Fehler auf, wird der Zug ignoriert und die nächste Runde beginnt.

Schwere Fehler sind Fehler, die vom Spieler vermieden werden können, da er eigentlich das dafür benötigte Wissen hätte haben können. Dementsprechend sind leichte Fehler solche Fehler, die nicht vom Spieler hätten vermieden werden können, da er nicht über ausreichend Wissen über das Spielfeld verfügt.

Leichte Fehler können in Spielen, in der der Spieler keine Sicht auf die umliegenden Felder hat, auftreten. Der Spieler muss durch "Trial and Error" versuchen, auf benachbarte Felder zu gehen. Versucht der Spieler zum ersten mal ein nicht begehbares Feld zu betreten, wird dies als leichter Fehler gewertet und der Spieler hat die Chance, seinen Zug erneut zu senden. Versucht er jedoch innerhalb derselben Spielrunde noch einmal auf dieses Feld zu gehen, wird dies als schwerer Fehler gewertet, da ihm bereits bekannt war, daß das Feld nicht begehbar ist.

Es gibt zwei Objekte, welche leichte Fehler auslösen können, nämlich Eichhörnchen und Bäume. Der Unterschied zwischen den beiden Objekten in Hinsicht auf leichte Fehler ist, daß ein Feld, auf dem ein Baum steht, niemals begehbar werden kann, da der Baum während der gesamten Spielzeit auf demselben Feld steht. Ein Feld, welches allerdings in einer Runde nicht begehbar war, weil ein gegnerisches Eichhörnchen darauf stand, könnte in der nächsten Runde wieder begehbar sein, weil sich das Eichhörnchen bewegt hat. Versucht also ein Spieler, nachdem er in einer vorherigen Spielrunde schon einmal versucht hat ein Feld mit einem Baum zu betreten, nun erneut dieses Feld zu betreten, wird das immer als schwerer Fehler gewertet. Angenommen, ein Spieler hat in einer früheren Runde ein Feld betreten wollen, auf dem ein gegnerisches Eichhörnchen stand, und er will in dieser Runde wieder versuchen das Feld zu betreten, dann wird das immer als leichter Fehler gewertet, denn der Spieler hätte nicht wissen können, ob sich das Eichhörnchen noch immer (oder schon wieder) auf dem Feld befindet.

Damit also ein Fehler als leichter Fehler gewertet wird, müssen folgende Bedingungen, wie im letzten Absatz beschrieben, erfüllt sein:

Zug: $(s \text{ Go } f), s \in \text{Squirrel}, f \in \text{Field}$

$\neg \text{AlreadyAccessedThisRound}(s, f) \wedge$

$$[(\exists t \in Tree : t \in f.objects \wedge \neg alreadyAccessedBefore(s, f)) \\ \vee (\exists s \in Squirrel : s \in f.objects)]$$

AlreadyAccessedBefore(s, f) ist wahr, wenn Eichhörnchen *s* schon einmal versucht hat, Feld *f* zu betreten. *AlreadyAccessedThisRound(s, f)* ist wahr, wenn Eichhörnchen *s* in dieser Spielrunde schon einmal versucht hat, Feld *f* zu betreten.

Wenn Eichhörnchen *s* in dieser Runde noch nicht versucht hat Feld *f* zu betreten, ist die Gleichung demnach in zwei Fällen wahr:

1. Auf Feld *f* steht ein Baum und Eichhörnchen *s* hat in diesem Spiel noch nicht versucht, Feld *f* zu betreten.
2. Auf Feld *f* steht ein Eichhörnchen. Eichhörnchen *s* darf in diesem Spiel, jedoch nicht in dieser Runde, schon einmal versucht haben, Feld *f* zu betreten.

5.3.4 Zugabhängige Updates

Nachdem ein Zug auf Korrektheit überprüft wurde, müssen die Veränderungen, die der Zug mit sich bringt, in den Spielzustand eingetragen werden.

$$update(gs, m) \rightarrow gs'$$

Bei den Aktualisierungen muss zwischen Updates, die in Abhängigkeit der verschiedenen Zugarten ausgeführt werden und Updates, welche unabhängig von der Zugart nach jedem Zug bzw. bei jedem Spielerwechsel ausgeführt werden müssen, unterschieden werden.

Im folgenden werden zunächst die zugspezifischen Updates aufgelistet. Zuerst wird jeweils ein Beispielzug inklusive einer Beschreibung des Zuges angegeben. Anschließend werden die notwendigen Updates für diesen Zug, sowie jeweils eine Erklärung der Updates aufgelistet, wobei sowohl die veränderten, als auch die unveränderten Bestandteile eines Objektes dargestellt werden. Dabei wird wieder die objektorientierten Repräsentation der Objekte aus Abschnitt 5.3.1 genutzt. Auch hier gilt, wenn ein Objekt *x* verändert wird, beschreibt *x* den Zustand vor der Änderung und *x'* den Zustand nach selbiger.

- **Kommando 'Go':**

Zug: (*s* Go *f*)

Zugeschreibung: Eichhörnchen *s* geht von Feld *g* nach Feld *f*.

Updates:

$$s'.capacity := s.capacity$$

$$s'.position := g$$

$$s'.objects := s.objects$$

$$f'.objects := f.objects \cup \{s\}$$

$$g'.objects := g.objects / \{s\}$$

$$isExploredField(g, gs.teamid)' := true$$

Updatebeschreibung Die Kapazität und die Objekte, die Eichhörnchen s mit sich trägt, ändern sich nicht. Da sich das Eichhörnchen von Feld g nach Feld f bewegt, muss die Position dementsprechend von g auf f geändert werden.

Da das Eichhörnchen nicht mehr auf Feld g steht, muss es aus der Objektliste von g entfernt werden.

Da sich das Eichhörnchen jetzt auf Feld f befindet, muss es in die Objektliste von Feld f aufgenommen werden.

Außerdem wird das Feld auf welches sich das Eichhörnchen bewegt, zu den erforschten Feldern hinzugefügt.

- **Kommando 'Take':**

Zug: (s Take n)

Zugbeschreibung: Eichhörnchen s nimmt Nuss n von Feld f auf.

Updates:

$$s'.capacity := s.capacity - n.weight$$

$$s'.position := s.position$$

$$s'.objects := s.objects \cup \{n\}$$

$$f'.objects := f.objects / \{n\}$$

Updatebeschreibung Die Kapazität von Eichhörnchen s muss nach dem Aufnehmen der Nuss n um das Gewicht von n verringert werden und die Nuss muss in die Objektliste des Eichhörnchens aufgenommen werden. Die Position von s ändert sich bei dem Vorgang nicht.

Da die Nuss von Feld f aufgenommen wurde, muss sie aus der Objektliste von f entfernt werden.

- **Kommando 'Drop':**

Zug: (s Drop n)

Zugbeschreibung: Eichhörnchen s legt Nuss n auf Feld f ab.

Updates:

$$s'.capacity := s.capacity + n.weight$$

$$s'.position := s.position$$

$$s'.objects := s.objects / \{n\}$$

$$f'.objects := f.objects \cup \{n\}$$

Updatebeschreibung Nach dem Ablegen der Nuss muss die Kapazität von Eichhörnchen s um das Gewicht der abgelegten Nuss n erhöht werden und die Nuss muss aus der Objektliste von s gestrichen werden. Die Position des Eichhörnchens bleibt während des Zuges unverändert.

Die Nuss wurde auf Feld f abgelegt und muss in die Objektliste von f aufgenommen werden.

- **Kommando 'Give':**

Zug: (s Give n t)

Beschreibung: Eichhörnchen s gibt Nuss n an Eichhörnchen t .

Updates:

$$s'.capacity := s.capacity + n.weight$$

$$s'.position := s.position$$

$$s'.objects := s.objects / \{n\}$$

$$t'.capacity := t.capacity - n.weight$$

$$t'.position := t.position$$

$$t'.objects := t.objects \cup \{n\}$$

Updatebeschreibung Eichhörnchen s gibt Nuss n ab, womit sich die Kapazität von s um das Gewicht von n erhöht und n aus der Objektliste von s gestrichen werden muss. Die Position von s bleibt während des Zuges unverändert.

Eichhörnchen t nimmt Nuss n von s entgegen, weswegen sich die Kapazität von t um das Gewicht von n verringert. Außerdem muss n in die Objektliste von t aufgenommen werden. Die Position von t verändert sich während des Zuges ebenfalls nicht.

5.3.5 Zugunabhängige Updates

Zusätzlich zu den zugspezifischen Updates gibt es noch zugunabhängige Updates, die nach jedem Zug ausgeführt werden. Dabei wird unterschieden zwischen Zügen mit und ohne Fehler und ob der Zug der letzte in der Zugliste ist oder nicht.

Im folgenden werden jeweils die Bedingungen, die der gerade überprüfte Zug erfüllen muss, aufgelistet. Anschließend wird ein Beispielzug inklusive Beschreibung angegeben und anschließend, wie schon bei den zugspezifischen Updates, aufgezählt, welche Updates jeweils durchgeführt werden. Der Spielzustand wird dabei durch gs repräsentiert. Schlußendlich folgt jeweils eine Beschreibung der einzelnen Updates.

- **Kein Fehler, nicht der letzte Zug in Zugliste:**

Zug: beliebiger Zug mit Eichhörnchen s , ohne Fehler

Update:

$$\begin{aligned} gs.canMove' &:= gs.canMove/\{s\} \\ gs.teamid' &:= gs.teamid \end{aligned}$$

Updatebeschreibung Da Eichhörnchen s in diesem Zug bewegt wurde, darf es für den Rest der Runde nicht mehr bewegt werden. Deswegen wird Eichhörnchen s aus $gs.canMove$ entfernt. $gs.teamid$ verändert sich nicht, solange der gleiche Spieler am Zug ist.

- **Kein Fehler, letzter Zug in Zugliste:**

Zug: beliebiger Zug mit Eichhörnchen s , ohne Fehler

Update bei Spielen mit einem Spieler:

Sei $t \in gs.teams \wedge t.id = gs.teamid$:

$$\begin{aligned} gs.canMove' &:= t.squirrels \\ gs.teamid' &:= gs.teamid \\ AlreadyAccessedThisRound' &:= false \end{aligned}$$

Updatebeschreibung Da dies der letzte Zug in der Zugliste ist, muss $gs.canMove$ wieder mit den Eichhörnchen des Spielers gefüllt werden. Da es sich um ein Ein-Spieler Spiel handelt, bleibt $gs.teamid$ gleich. Da $AlreadyAccessedThisRound$ nur für diese Runde *wahr* ist, wird es für die nächste Runde wieder auf *false* gesetzt.

Update bei Spielen mit zwei Spielern:

Sei $t \in gs.teams \wedge t.id \neq gs.teamid$:

$$\begin{aligned} gs.canMove' &:= t.squirrels \\ gs.teamid' &:= t.id \\ AlreadyAccessedThisRound' &:= false \end{aligned}$$

Updatebeschreibung Da dies der letzte Zug in der Zugliste ist, muss $gs.canMove$ mit den Eichhörnchen des nächsten Spielers gesetzt werden. Außerdem muss $gs.teamid$ auf die ID des nächsten Spielers gesetzt werden.

Tritt ein leichter Fehler auf, müssen ebenfalls Updates durchgeführt werden:

- **Leichte Fehler:**

Zug: $(s \text{ Go } f), s \in Squirrel, f \in Field$

Update:

$$\begin{aligned} gs' &:= resetGS \\ AlreadyAccessedBefore(s, f) &:= true \\ AlreadyAccessedThisRound(s, f) &:= true \end{aligned}$$

Updatebeschreibung Da ein leichter Fehler aufgetreten ist, hat der Spieler die Chance, erneut eine verbesserte Zugliste zu schicken. Zu diesem Zweck wird der Spielzustand per $resetGS$ wieder auf den Zustand vor dem ersten Zug des Spielers in dieser Runde gesetzt.

AlreadyAccessedBefore und *AlreadyAccessedThisRound* werden auf *wahr* gesetzt, da der Spieler in dieser Runde und somit in diesem Spiel versucht hat, Feld f zu betreten.

Außerdem hat der Spieler nun das Wissen darüber erlangt, daß Feld f belegt ist und darf es in dieser Runde nicht mehr versuchen, zu betreten. Je nachdem, ob das Feld von einem Eichhörnchen oder einem Baum belegt war, darf er in einer späteren Runde wieder versuchen, es zu betreten.

Im Falle eines schweren Fehlers müssen folgende Updates vorgenommen werden:

- **Schwere Fehler:**

Zug: Jeder Zug, der einen schweren Fehler verursachen kann

Update bei Spielen mit einem Spieler:

$$gs' := resetGS$$

Updatebeschreibung Da ein schwerer Fehler aufgetreten ist, wird die Runde beendet und eine neue Runde beginnt, weswegen der Spielzustand per *resetGS* wieder auf den Stand vor dieser Runde gesetzt wird.

Update bei Spielen mit zwei Spielern:

Sei $t \in gs.teams \wedge t.id \neq gs.teamid$:

$$\begin{aligned} gs' &:= resetGS \\ gs.teamid' &:= t.id \\ canMove' &:= t.objects \end{aligned}$$

Updatebeschreibung Da ein schwerer Fehler aufgetreten ist, ist nun der andere Spieler an der Reihe. Deswegen muss der Spielzustand auf den Stand vor dem Zug zurückgesetzt werden. Außerdem ändert sich *gs.teamid* auf die ID des anderen Spielers, welcher nun an die Reihe kommt und *gs.canMove* wird mit den Eichhörnchen dieses Spielers gefüllt.

5.4 Endphase

Ist das Spiel in einem Endzustand, das heißt, ist das Spielziel erreicht, müssen dem Spieler das Ende des Spiels und abschließende Informationen mitgeteilt werden. Die Art der abschließenden Informationen hängt von der Spielart ab. So kann dem Spieler in einem Spiel mit dem Ziel, den Weg mit den niedrigsten Kosten zu finden, mitgeteilt werden, wieviel Kosten sein gefundener

Weg verursacht hat. Desweiteren kann den Spielern in einem Zwei-Spieler-Spiel der Gewinner des Spiels mitgeteilt werden. Die abschließenden Informationen werden dem Spieler per (Stop <EndParams>) mitgeteilt. Im Folgenden werden zwei Beispiele für die Versendung der abschließenden Informationen, nämlich zur informierten Suche und zur Adversarial Search, präsentiert und beschrieben.

- **Beispiel zur informierten Suche**

Folgende abschließende Informationen werden dem Spieler vom Gamemaster mitgeteilt:

Gamemaster → **Spieler:** (Stop <Cost>)

<Cost> teilt dem Spieler mit, wieviele Kosten der vom ihm gefundene Pfad hat.

- **Beispiel zu Adversarial Search**

Folgende abschließende Informationen werden dem Spieler vom Gamemaster mitgeteilt:

Gamemaster → **Spieler:** (Stop <Result>)

<Result> teilt dem Spieler mit, wer das Spiel gewonnen hat.

6 Evaluation

Bei der Evaluation werden die in Kapitel 4 vorgestellten Spielarten, nämlich uninformierte Suche, informierte Suche und Adversarial Search an der Implementation des Gamemasters getestet. Dadurch wird gezeigt, daß die theoretischen Überlegungen, die in den vorherigen Kapiteln präsentiert wurden, sich auch praktisch umgesetzt wurden. Sämtliche Beispiele lassen sich mit dem implementierten Gamemaster und einem Test-Client, bei dem die Eingaben von Hand gemacht werden können, nachspielen. Alle zur Konfiguration und zum Start des Gamemasters und des Clients benötigten Informationen befinden sich in Anhang A. Anhang B zeigt eine Beispielimplementation eines Agenten, welcher die Uniform-Cost-Suche verwendet.

Nachrichten vom Gamemaster zum Spieler sind durch \Rightarrow gekennzeichnet. Dementsprechend sind Nachrichten vom Spieler zum Gamemaster mit \Leftarrow markiert. Alle Zeilen, die nicht mit \Leftarrow oder \Rightarrow beginnen, sind lediglich nachträglich hinzugefügte Zusätze, um die Züge des Spielers besser nachvollziehen zu können. Die Registrierung der Spieler wurde jeweils weggelassen. Das Spiel beginnt immer mit dem Senden der Startinformationen an den Spieler.

6.1 Testspiel uninformierte Suche

Bei der uninformierten Suche muss der Spieler versuchen, die einzige, auf dem Spielfeld vorhandene Nuss zu finden. Er bekommt jeweils vom Gamemaster die symbolischen Positionen der angrenzenden Felder und die Kosten, die mit dem Gehen auf das jeweilige Feld verbunden sind, geschickt. Der Spieler verwendet zur Lösung des Problems den Uniform-Cost-Algorithmus[6]. Zusätzlich zu den Nachrichten zwischen Gamemaster und Spieler wird in jeder Runde eine nach Kosten sortierte Liste *nextMove*, die aus Paaren von Pfaden und Kosten besteht, aktualisiert. Diese Liste dient ausschließlich zum besseren Verständnis des Beispiels und wurde nachträglich hinzugefügt. Das letzte Feld eines jeden Pfades ist ein noch nicht erforschtes, aber einem schon erforschten Feld angrenzendes Feld, auf welches das Eichhörnchen springen kann. Die Kosten in *nextMove* stehen für die Gesamtkosten, die für den Pfad anfallen. Nach jeder Runde wird *nextMove* aktualisiert und danach das erste Element für den nächsten Zug verwendet. Wurde das Zielfeld mit der Nuss erreicht, teilt der Gamemaster dies dem Spieler mit und stoppt das Spiel. Eine Beispielimplementation eines Agenten, welcher den Uniform-Cost-Algorithmus verwendet, befindet sich in Anhang B.

Dieses Spiel lässt sich nachspielen, indem man die Beispielkonfigurationsdatei, welche auch in Anhang A.3.2 zu finden ist, verwendet. Abbildung 15 zeigt eine grafische Darstellung des Spielfeldes.

```

 $\Rightarrow$  (( StartPos g) (PosMoves (u 1) (r 3)))
  nextMove: [(gu 1) (gr 3)]
 $\Leftarrow$  (u)

```



Abbildung 14: Die Abbildung zeigt die Karte, welche bei dem Testspiel benutzt wird. Das Eichhörnchen startet links oben auf der Sammelstelle (0,0) und muss zur Nuss auf Position (3,1) gelangen.

⇒ (PosMoves (g 1) (z 2) (j 4))
nextMove: [(gr 3) (guz 3) (guj 5)]
 ⇐ (r)

⇒ (PosMoves (g 3) (j 2) (v 3))
nextMove: [(guz 3) (guj 5) (grj 5) (grv 6)]
 ⇐ (z)

⇒ (PosMoves (u 2) (h 7) (c 5))
nextMove: [(guj 5) (grj 5) (grv 6) (guzc 8) (guzh 10)]
 ⇐ (j)

⇒ (Error 21 1 SOFT "Field is occupied by a Tree but you could not know that. Move again")
nextMove: [(grv 6) (guzc 8) (guzh 10)]
 ⇐ (v)

⇒ (PosMoves (r 3) (s 5) (f 10))
nextMove: [(guzc 8) (guzh 10) (grvs 11) (grvf 16)]
 ⇐ (c)

⇒ (PosMoves (z 5) (j 3) (t 3) (l 6))
nextMove: [(guzh 10) (grvs 11) (guzct 11) (grvf 16)]
 ⇐ (h)

⇒ (PosMoves (z 7) (t 2))
nextMove: [(grvs 11) (guzct 11) (guzht 12) (guzcl 14) (grvf 16)]

⇐ (s)

⇒ (PosMoves (j 2) (v 5) (l 7) (q 2))

nextMove: [(guzct 11) (guzht 12) (grvsq 13) (guzcl 14) (grvf 16) (grvsl 18)]

⇐ (t)

⇒ (PosMoves (h 2) (c 3) (o 8))

nextMove: [(grvsq 13) (guzcth 13) (guzcl 14) (grvf 16) (grvsl 18) (guzcto 19)]

⇐ (q)

⇒ (Stop "Win: Nut found")

Das Ergebnis lautet nach 10 Schritten: Pfad qrvsq mit Kosten 13.

6.2 Testspiel informierte Suche

Die Spielart der informierten Suche ist ähnlich der der uninformierten Suche. Die Unterschiede sind, daß alle Felder durch ihre X- und Y-Koordinaten repräsentiert werden und daß der Spieler am Anfang die Koordinaten des Zielfeldes mitgeteilt bekommt. Dies ermöglicht ihm, in jedem Schritt eine heuristische Funktion zu benutzen. Ein Beispiel ist die Manhattan Distance, welche die Mindestanzahl der Züge berechnet, die benötigt werden, um das Ziel von der aktuellen Position aus zu erreichen. Wie in der uninformierten Suche wird auch hier wieder die Liste *nextMove* benutzt, nur daß die Pfade nun in Koordinaten angegeben werden. In diesem Spiel wurde der A*-Algorithmus[6] angewendet. Zu diesem Zweck wird die Liste nach dem Ergebnis der Addition der Gesamtkosten eines Pfades und der heuristischen Funktion für das letzte Feld eines Pfades sortiert.

Auch dieses Spiel lässt sich nachspielen, indem man die Beispielfunktionsdatei, welche auch in Anhang A.3.2 zu finden ist, verwendet. Die Karte ist dieselbe, die auch bei der uninformierten Suche benutzt wurde.

⇒ ((StartPos (0 0) (GoalPos (3 1)) (PosMoves ((0 1) 1) ((1 0) 3)))

nextMove: [((0 0) (0 1) 1+3=4) ((0 0) (1 0) 3+3=6)]

⇐ (0 1)

⇒ (PosMoves ((0 0) 1) ((0 2) 2) ((1 1) 4))

nextMove: [((0 0) (1 0) 3+3=6) ((0 0) (01) (02) 3+4=7) ((0 0) (01) (11) 5+2=7)]

⇐ (1 0)

⇒ (PosMoves ((0 0) 3) ((1 1) 2) ((2 0) 3))

nextMove: [((0 0) (0 1) (0 2) 3+4=7) ((0 0) (0 1) (1 1) 5+2=7) ((0 0) (1 0) (1 1) 5+2=7) ((0 0) (1 0) (2 0) 6+2=8)]

⇐ (0 2)

\Rightarrow (PosMoves ((0 1) 2) ((0 3) 7) ((1 2) 5))
nextMove: [((0 0) (0 1) (1 1) 5+2=7) ((0 0) (1 0) (1 1) 5+2=7) ((0 0) (1 0) (2 0) 6+2=8) ((0 0) (0 1) (0 2) (1 2) 8+3=11) ((0 0) (0 1) (0 2) (0 3) 10+5=15)]
 \Leftarrow (1 1)

\Rightarrow (Error 21 1 SOFT "Field is occupied by a Tree but you could not know that. Move again")
nextMove: [((0 0) (1 0) (2 0) 6+2=8) ((0 0) (0 1) (0 2) (1 2) 8+3=11) ((0 0) (0 1) (0 2) (0 3) 10+5=15)]
 \Leftarrow (2 0)

\Rightarrow (PosMoves ((1 0) 3) ((2 1) 5) ((3 0) 10))
nextMove: [((0 0) (0 1) (0 2) (1 2) 8+3=11) ((0 0) (1 0) (2 0) (2 1) 11+1=12) ((0 0) (0 1) (0 2) (0 3) 10+5=15) ((0 0) (1 0) (2 0) (3 0) 16+1=17)]
 \Leftarrow (1 2)

\Rightarrow (PosMoves ((0 2) 5) ((1 1) 3) ((1 3) 3) ((2 2) 6))
nextMove: [((0 0) (1 0) (2 0) (2 1) 11+1=12) ((0 0) (0 1) (0 2) (0 3) 10+5=15) ((0 0) (0 1) (0 2) (1 2) (1 3) 11+4=15) ((0 0) (0 1) (0 2) (1 2) (2 2) 14+2=16) ((0 0) (1 0) (2 0) (3 0) 16+1=17)]
 \Leftarrow (2 1)

\Rightarrow (PosMoves ((1 1) 2) ((2 0) 5) ((2 2) 7) ((3 1) 2))
nextMove: [((0 0) (1 0) (2 0) (2 1) (3 1) 13+0=13) ((0 0) (0 1) (0 2) (0 3) 10+5=15) ((0 0) (0 1) (0 2) (1 2) (1 3) 11+4=15) ((0 0) (0 1) (0 2) (1 2) (2 2) 14+2=16) ((0 0) (1 0) (2 0) (3 0) 16+1=17) ((0 0) (1 0) (2 0) (2 1) (2 2) 18+2=20)]
 \Leftarrow (3 1)

\Rightarrow (Stop "Win: Nut found")

Das Ergebnis lautet nach 8 Schritten: Pfad (0 0) (1 0) (2 0) (2 1) (3 1) mit Kosten 13

6.3 Testspiel Adversarial Search

Bei dieser Spielart spielen zwei Spieler gegeneinander. Ziel des Spiels ist es, entweder nach einer bestimmten Rundenzahl mehr Punkte als der Gegner gesammelt zu haben oder eine bestimmte Punktegrenze zu erreichen. Auch lassen sich fast alle Parameter bis auf die Anzahl der Teams mit Hilfe der Konfigurationsdatei für diese Spielart (siehe Anhang A) festlegen. Die Beispielkonfigurationsdatei in Anhang A.3.1 entspricht der Konfigurationsdatei für dieses Spiel.

In diesem Testspiel spielen zwei Teams mit jeweils zwei Eichhörnchen gegeneinander. Die Eichhörnchen s1 und s2 gehören zu Team 1 und s3 und s4 gehören zu Team 2. Jedes Team hat ein Eichhörnchen der langsamen Art (s1, s3) mit Kapazität 12 und ein Eichhörnchen der schnellen Art (s2, s4) mit Kapazität 5. Die Anzahl der Runden ist auf 100 beschränkt und die Punktegrenze liegt bei 12 Punkten. Das bedeutet, daß das Team, welches als erstes Nüsse im



Abbildung 15: Die Abbildung zeigt die Karte, welche bei dem Testspiel benutzt wird. Team 1 startet links oben (0,0). Team 2 startet links unten (6,4). Gewonnen hat die Mannschaft, die als erstes Nüsse im Wert von 12 Punkten zu ihrer Sammelstelle bringt.

Wert von 12 Punkten auf seiner Sammelstelle abgelegt hat, das Spiel gewonnen hat. Springen ist nicht erlaubt und der Sichtradius der Eichhörnchen liegt bei einem Feld. Das Testspiel findet aus Sicht von Team 2 statt und beginnt mit den Startinformationen für dieses Team.

- ⇒ ((ObjDesc (OwnObj (Squirrel s3 12 SLOW (4 4))(Squirrel s4 5 FAST (4 4))(OwnGP gp2 (4 4))) (OppObj (Squirrel s1 SLOW)(Squirrel s2 FAST)(OppGP gp1)) (MiscObj (Nut n1 6 8)(Nut n2 5 6)(Tree t1 n1)(Tree t2 n2))) (Map ((3 3) nil)((3 4) nil)((4 3) nil)((4 4) (gp2 s3 s4))))
- ⇐ ((s3 go 4 3)(s4 go 3 4))
- ⇒ (Map ((2 3) nil)((2 4) nil)((3 2)(t1))((3 3) nil)((3 4)(s4))((4 2) nil)((4 3)(s3))((4 4)(gp2)))
- ⇐ ((s3 go 4 2)(s4 go 2 3))
- ⇒ (Map ((1 2)(t1))((1 3)(s2))((1 4)(t2))((2 2)(n1))((2 3)(s4))((2 4) nil)((3 1) nil)((3 2)(t1))((3 3) nil)((3 4) nil)((4 1)(t2))((4 2)(s3))((4 3) nil))
- ⇐ ((s3 go 4 3)(s4 go 2 2))
- ⇒ (Map ((1 1) nil)((1 2)(t1))((1 3) nil)((2 1) nil)((2 2)(n1 s4))((2 3) nil)((3 1)(s1))((3 2)(t1))((3 3) nil)((3 4) nil)((4 2) nil)((4 3)(s3))((4 4)(gp2)))
- ⇐ ((s3 go 3 3))

- \Rightarrow (Map ((1 1) nil)((1 2)(t1))((1 3) nil)((2 1)(s1))((2 2)(n1 s4))((2 3) nil)((2 4) nil)((3 1) nil)((3 2)(t1))((3 3)(s3))((3 4) nil)((4 2) nil)((4 3) nil)((4 4)(gp2)))
 \Leftarrow ((s3 go 2 3))
- \Rightarrow (Map ((1 1)(s1))((1 2)(t1))((1 3)(s2))((1 4)(t2))((2 1) nil)((2 2)(n1 s4))((2 3)(s3))((2 4) nil)((3 1) nil)((3 2)(t1))((3 3) nil)((3 4) nil))
 \Leftarrow ((s4 go 3 1)(s3 go 2 2))
- \Rightarrow (Map ((1 1) nil)((1 2)(t1))((1 3) nil)((2 0) nil)((2 1) nil)((2 2)(n1 s3))((2 3) nil)((3 0)(t2))((3 1)(s4))((3 2)(t1))((3 3) nil)((4 0)(n2))((4 1)(t2))((4 2) nil))
 \Leftarrow ((s4 go 4 0)(s3 take n1))
- \Rightarrow (Map ((1 1) nil)((1 2)(t1))((1 3) nil)((2 1) nil)((2 2)(s3))((2 3) nil)((3 0)(t2))((3 1) nil)((3 2)(t1))((3 3) nil)((4 0)(n2 s4))((4 1)(t2)))
 \Leftarrow ((s4 take n2)(s3 go 2 3))
- \Rightarrow (Map ((1 2)(t1))((1 3) nil)((1 4)(t2))((2 2) nil)((2 3)(s3))((2 4) nil)((3 0)(t2))((3 1) nil)((3 2)(t1))((3 3) nil)((3 4) nil)((4 0)(s4))((4 1)(t2)))
 \Leftarrow ((s4 go 3 1)(s3 go 2 4))
- \Rightarrow (Map ((1 3) nil)((1 4)(t2))((2 0) nil)((2 1) nil)((2 2) nil)((2 3) nil)((2 4)(s3))((3 0)(t2))((3 1)(s4))((3 2)(t1))((3 3) nil)((3 4) nil)((4 0) nil)((4 1)(t2))((4 2) nil))
 \Leftarrow ((s4 go 4 2)(s3 go 3 4))
- \Rightarrow (Map ((2 3) nil)((2 4) nil)((3 1) nil)((3 2)(t1))((3 3) nil)((3 4)(s3))((4 1)(t2))((4 2)(s4))((4 3) nil)((4 4)(gp2)))
 \Leftarrow ((s4 go 4 3)(s3 go 4 4))
- \Rightarrow (Map ((3 2)(t1))((3 3) nil)((3 4) nil)((4 2) nil)((4 3)(s4))((4 4)(gp2 s3)))
 \Leftarrow ((s4 go 4 4)(s3 drop n1))
- \Rightarrow (Map ((3 3) nil)((3 4) nil)((4 3) nil)((4 4)(gp2 s3 s4)))
 \Leftarrow ((s4 drop n2))
- \Rightarrow (Stop "Win")

7 Related Work

Ein diesem Thema ähnliches Thema ist das *General Gameplaying Project* der Stanford University[2]. Auch dort ist das Kernstück des Forschungsprojektes ein Gamemaster. Die Besonderheit daran ist, daß der Spieler im voraus nicht weiß, welches Spiel er spielen wird. Der Gamemaster hat eine erweiterbare Auswahl von Spielen und entscheidet zufällig, welches Spiel gespielt wird. Nachdem der Spieler sich mit dem Gamemaster verbunden hat, schickt der Gamemaster dem Spieler die Spielbeschreibung in der eigens dafür entwickelten Game Description Language[3] (GDL). Der Spieler muss nun in der Lage sein, die Spielbeschreibung zu interpretieren und das Spiel zu spielen, wobei es möglich ist, daß er das Spiel noch nie zuvor gespielt hat oder er es überhaupt nicht kennt, da der Gamemaster nach Belieben neue Spiele in sein Repertoire aufnehmen kann.

Der Unterschied zu dieser Arbeit ist, daß die Spieler hier vorher wissen, welche Spielart gespielt wird. Ihnen kann lediglich das Aussehen des Spielfeldes, also Größe der Karte, Anzahl und Standorte der Objekte wie Nüssen oder Bäumen, unbekannt sein, denn erst wenn das Spiel startet, werden ihnen Informationen über das ihnen sichtbare Spielfeld gesendet. Der Grund, warum dieses Prinzip nicht benutzt wird, ist, daß diese Arbeit zunächst für eine Einführung in die Künstliche Intelligenz gedacht ist und dafür die Komplexität viel zu hoch wäre. Es wäre jedoch auch denkbar, daß dem Spieler in zukünftigen Implementationen Definitionen neuer Befehle oder neuer Objekte beim Spielstart mitgeteilt werden. Die Sprache wäre dann insofern zu erweitern, als daß der Spieler alle wichtigen Informationen, zum Beispiel, was ein neuer Befehl bewirkt oder was für Eigenschaften ein neues Objekt hat, erhalten kann.

8 Zusammenfassung und Ausblick

In dieser Arbeit wurden die theoretischen Überlegungen, die vor und während der Implementation des Gamemasters gemacht wurden, beschrieben. Die Eigenschaften des Spiels, welches gespielt wird, und die Spielarten, die der Gamemaster anbietet, wurden definiert. Desweiteren wurden die Aufgaben des Gamemasters, wie die Evaluation der Züge der Spieler und die damit verbundenen Aktualisierungen, beschrieben. Außerdem wurde bei der Evaluation durch mehrere Testspiele gezeigt, daß sich diese theoretischen Überlegungen auch praktisch umsetzen lassen.

Ein weiterführendes Thema ist die Kommunikation der Eichhörnchen mit neutralen Agenten, um das Spielziel zu erreichen. Denkbar wäre es, daß man Fragen beantwortet oder Rätsel löst, um sich bestimmte Informationen, die zum Erreichen des Spielziels nützlich sind, zu beschaffen.

Ein mögliches Anwendungsgebiet im Rahmen des Gamemasters wäre das Gebiet der verteilten Künstlichen Intelligenz, zum Beispiel in einer Fortgeschrittenen- oder Spezialvorlesung. Angenommen jeder Agent steuere jeweils ein Eichhörnchen eines Teams. Alle Agenten eines Teams hätten ein gemeinsames Spielziel, würden sich jedoch keine globalen Informationen, zum Beispiel über die ihnen sichtbaren Spielfelder und darauf befindlichen Objekte, teilen, sondern jeder

Agent bekäme nur die Informationen, die das von ihm gesteuerte Eichhörnchen empfängt. Das heißt, bei einer Spielart mit beschränkter Sicht bekäme ein Agent nur Informationen über die seinen Eichhörnchen sichtbaren Spielfelder. Die Agenten eines Teams müssten somit miteinander kommunizieren, um Wissen, zum Beispiel über ihnen unbekannte Teile des Spielfeldes, auszutauschen. Sie müssten ihre Pläne und Ziele miteinander koordinieren, um ihr Spielziel zu erreichen und ihre unterschiedlichen Fähigkeiten (siehe Kapitel 2.2.1 über die verschiedenen Eichhörnchentypen) gezielt nutzen, um eine möglichst effiziente Lösung zu finden. Wissensaustausch könnte dann mit bestimmten Kosten verbunden sein, damit die Agenten gezielt nur bestimmte Teile ihres Wissens austauschen, um mit möglichst wenig Kostenaufwand möglichst viel nützliches Wissen zu erhalten.

Noch einen Schritt weiter ginge die Anwendung im Bereich der Sozionik, einer Mischung aus Soziologie und Künstlicher Intelligenz, in der man sich damit beschäftigt, Programme zu entwickeln, die sich an der menschlichen Gesellschaft orientieren. So könnte man versuchen, daß die Agenten sich auf der Basis von sozialen Abläufen und Strukturen organisieren, um so miteinander zu kooperieren und das Spielziel zu erreichen.

Ein weiteres Thema ist der Grad der Informiertheit bzw. der Uninformiertheit des Spielers. Ein Beispiel sind die in Kapitel 5.3.3 besprochenen leichten und schweren Fehler. Dort wird erkannt, wenn der Spieler einen ungültigen Zug vollführt, den er hätte vermeiden können, da er die zur Vermeidung benötigten Informationen eigentlich zur Verfügung stehen hat. Im Moment wird nur überprüft, wenn ein Spieler auf ein nicht betretbares Feld geht, ob er dieses Feld schon einmal betreten hat und somit eigentlich Kenntnis über die Begehbarkeit des Feldes hat. Zukünftige Implementierungen könnten kompliziertere Fehler ausfindig machen. Ein Beispiel wären unnötige Schleifen, zum Beispiel gleiche Pfade, die mehrmals gegangen werden ohne einen Informationsgewinn zu erbringen. Die Aufgabe des Gamemasters wäre dann herauszufinden, ob dieser Fehler vermeidbar gewesen wäre. Daran ließe sich wiederum eventuell auch feststellen, wie effizient ein Algorithmus arbeitet. Produziert er viele vermeidbare Fehler, arbeitet er nicht sehr effizient, da sein Weg zur Erfüllung des Spielzieles umständlicher ist, als der optimale Weg.

A Konfiguration und Start

Der Gamemaster und ein Client, mit dem das Spielen erprobt werden können, liegen als Jar-Archive vor. In diesen Jar-Archiven befinden sich alle Klassendateien, die zum Ausführen des Gamemasters und des Clients benötigt werden. Jar-Archive haben den Vorteil, daß sie einfach mit Hilfe des Befehls `java`² ausgeführt werden können. Um den Gamemaster zu starten, sollte man sich in dem gleichen Verzeichnis befinden, in dem sich auch das Jar-Archiv befindet. Sämtliche, zum Start und zur Konfiguration von Gamemaster und Client benötigten, in einem Tar-Archiv zusammengefasste Dateien lassen sich unter [5] herunterladen.

A.1 Start des Gamemasters

Der Startbefehl für den Gamemaster lautet:

```
java -jar gamemaster.jar
```

Will man den Gamemaster mit der graphischen Oberfläche starten, muss man zusätzlich `gui` an den Startbefehl hängen:

```
java -jar gamemaster.jar gui
```

Wurde der Gamemaster gestartet, können sich Clients mit diesem verbinden, um ein Spiel zu spielen.

A.2 Start des Clients

Mit dem Client kann man alle Spiele testen, indem man die Nachrichten, die man an den Gamemaster schicken möchte, von Hand eingibt. Um den Client zu starten, muss man folgenden Befehl ausführen:

```
java -jar client.jar GAMETYPE [OPTION]
```

GAMETYPE steht dabei für die Spielart, welche man spielen möchte. Es werden zur Zeit folgende Spielarten vom Gamemaster angeboten:

uis steht für die uninformierte Suche. Diese Spielart wird in Kapitel 4 beschrieben.

is steht für die informierte Suche, die ebenfalls in Kapitel 4 beschrieben wird.

²Um den Befehl benutzen zu können, wird eine installierte Java Laufzeitumgebung[4] in der Version 1.5 vorausgesetzt

1P steht für ein Ein-Spieler-Spiel, bei dem sämtliche Spielparameter, außer die Anzahl der Teams, nach Belieben eingestellt werden können.

2P steht für ein Zwei-Spieler-Spiel, bei dem, wie bei 1P, sämtliche Spielparameter nach Bedürfnis eingestellt werden dürfen. Zusätzlich muss beim Starten des Clients aber noch angegeben werden, ob man ein Spiel starten möchte (`create`) und somit den ersten Zug machen darf oder einem Spiel beitreten will (`join`) und der Gegner den ersten Zug machen darf. Beispielsweise tritt man einem Spiel mit dem Kommando

```
java -jar client 2P join
```

bei. Will man ein Spiel eröffnen, ersetzt man lediglich das `join` durch `create`.

Wurde der Client gestartet, muss man nun noch die URL des Gamemasters eingeben. Wurde der Gamemaster auf dem gleichen Computer wie der Client gestartet, reicht es die Enter-Taste zu drücken, da diese URL dem Client schon bekannt ist. Als nächstes wird man gefragt, ob man die Züge für die Eichhörnchen per Hand eingeben möchte oder ob ein programmierter Agent existiert. Existiert ein programmierter Agent, muss dieser in einer Klasse `Agent` implementiert sein und es muss entweder die Quelldatei `Agent.java` oder die kompilierte Datei `Agent.class` existieren.

A.3 Konfigurationsdateien der Spielarten

Für jede Spielart kann man selbst Konfigurationsdateien erstellen, welche einerseits die einstellbaren Parameter beinhalten und andererseits die Definition der Karte. Im Ordner `maps` stehen vier Beispielformatierungsdateien (`testUIS.map`, `testIS.map`, `test1P.map` und `test2P.map`) für die im vorherigen Abschnitt beschriebenen Spielarten zur Verfügung.

A.3.1 Konfigurationdateien für zwei Spieler

Eine Beispielformatierungsdatei für ein Zwei-Spieler-Spiel sieht folgendermaßen aus:

```
numberTeams          2
numberSquirrels      2
numberRounds         100
numberPoints         12
jump                 false
view                 1
```

```
[ Nuts ]
```

```
n1 8 6
```

```
n2 6 5
```

```
[ Trees ]
```

```
t1 n1
t2 n2

[ Squirrels ]
s1 12 SLOW 1
s2 5 FAST 1

s3 12 SLOW 2
s4 5 FAST 2

[ Gathering Points ]
gp1 1
gp2 2

[Map Params ]
lengthX 5
lengthY 5

[ Fields ]
field 0 0 gp1

field 4 4 gp2

field 2 2 n1

field 1 2 t1
field 3 2 t1

field 0 4 n2
field 4 0 n2

field 3 0 t2
field 4 1 t2
field 0 3 t2
field 1 4 t2
```

Eine Konfigurationsdatei für ein Ein-Spieler-Spiel ist genauso strukturiert. Lediglich die Zahl der Teams muss 1 sein und es dürfen nur Objekte für Team 1 definiert werden.

Der erste Abschnitt der Konfigurationsdatei beschäftigt sich mit den Spielparametern. Im Folgenden werden die einzelnen Spielparameter und ihre möglichen Werte aufgelistet. Außerdem wird die Bedeutung eines jeden Parameters noch einmal erläutert.

- **numberTeams:** 1-2
Mit **numberTeams** wird die Anzahl der Teams eingestellt.
- **numberSquirrels:** >0
Mit diesem Parameter wird die Anzahl der Eichhörnchen pro Team festgelegt.
- **numberRounds:** ≥ 0
Mit **numberRounds** wird die Anzahl der Runden festgelegt, die gespielt werden. Wenn **numberRounds** 0 ist, wird die Rundenbeschränkung außer Kraft gesetzt.
- **numberPoints:** ≥ 0
NumberPoints legt fest, wieviele Punkte ein Team erreichen muss, damit es als Sieger aus der Partie feststeht. Ein Team bekommt dann Punkte, wenn eine Nuss auf der teameigenen Sammelstelle abgelegt wurde. Die Anzahl der Punkte richtet sich dabei nach dem Wert der Nuss, die abgelegt wurde. Ist **numberPoint** 0, gibt es keine Punktegrenze, die erreicht werden muss.
- **jump:** true | false
Mit **jump** kann festgelegt werden, ob springen erlaubt ist. Falls es erlaubt ist, darf ein Eichhörnchen auf alle vorher erkundeten oder an erkundete Felder angrenzende Felder springen. Ist springen nicht erlaubt, darf ein Eichhörnchen lediglich auf angrenzende Felder gehen.
- **view:** ≥ -1
View legt die Sichtweite der Eichhörnchen fest. Beträgt die Sichtweite -1, sieht ein Eichhörnchen immer die ganze Karte. Bei Sichtweite x sieht ein Eichhörnchen alle Felder innerhalb eines Radiuses von x Feldern.

Der nächste Abschnitt [**Nuts**] beschäftigt sich mit der Definition der im Spiel vorkommenden Nussarten. Eine Nussart wird definiert, indem man in einer Zeile ihren Bezeichner, ihren Wert und ihr Gewicht angibt. Die allgemeine Schreibweise einer Nussdefinition lautet:

nx *Gewicht Wert* (Beispiel: n1 5 8)

nx ist der Bezeichner der Nussart, wobei $x \in [1, 9]$. *Gewicht* und *Wert* müssen positive, ganze Zahlen sein. Das Beispiel bedeutet, daß eine Nussart der Art "n1" mit Gewicht 5 und Wert 8 definiert wurde.

Danach werden die Baumarten in ähnlicher Form definiert:

tx ny (Beispiel: t1 n1)

tx steht für den Bezeichner der Baumart, wobei auch hier $x \in [1, 9]$. $y, z \in N^+$. ny ist die Nussart, die in der Nähe dieser Baumart zu finden ist und muss eine existierende Nussart sein. Dem Beispiel nach wurde also eine Baumart der Art t1 definiert, in dessen Umgebung sich die

Nussart $n1$ befindet.

Im darauffolgenden Abschnitt werden die verschiedenen Eichhörnchen, die in dieser Spielart auf dem Spielfeld vorkommen, definiert. Die allgemeine Schreibweise lautet:

sx *Kapazitaet* *Art* *Team* (Beispiel: $s1$ 15 slow 1)

sx ist der Bezeichner des Eichhörnchens. Die *Kapazitaet* ist eine positive, ganze Zahl. Die *Art* des Eichhörnchens kann entweder *slow* oder *fast* sein. Ein schnelles Eichhörnchen kann auch diagonal anliegende Felder betreten. Die Nummer des *Teams* ist entweder 1 oder 2. Im Beispiel wird also ein Eichhörnchen mit dem Bezeichner $s1$ der langsamen Art mit Kapazität 15 definiert, welches Team 1 angehört.

Nun müssen noch die Sammelstellen der Teams definiert werden. Dies geschieht folgendermaßen:

gpx *Team* (Beispiel: $gp1$ 1)

Auch hier steht gpx für den Bezeichner der Sammelstelle. *Team* gibt an, zu welchem Team die Sammelstelle gehört. Die Sammelstelle $gp1$ aus dem Beispiel gehört also zu Team 1.

Als letztes folgt die Einstellung der Kartengröße und die Platzierung der Objekte auf der Karte. Mit *lengthX* bzw. *lengthY* lassen sich die Anzahl der Felder in der X- bzw. in der Y-Richtung festlegen.

Nun müssen die Felder angegeben werden, auf denen ein Objekt platziert werden soll. Die allgemeine Schreibweise lautet:

field X Y *Objektbezeichner* (Beispiel $field$ 1 3 $n1$)

Voraussetzung für die Platzierung eines Objekts ist, daß es vorher definiert wurde. Da alle Eichhörnchen von ihrer Sammelstelle starten, müssen keine Eichhörnchen platziert werden. Es genügt die Sammelstelle des Teams zu platzieren. Im Beispiel wird auf dem Feld mit der X-Koordinate 1 und der Y-Koordinate 3 eine Nuss des Nusstyps $n1$ platziert.

A.3.2 Konfigurationsdateien für die un/informierte Suche

Die Konfigurationsdateien für die informierte bzw. uninformierte Suche sind eine abgespeckte Version der Konfigurationsdateien für die im vorigen Abschnitt beschriebenen Spielarten, da die meisten Parameter schon vorher festgelegt wurden. Eine Konfigurationsdatei könnte folgendermaßen aussehen:

numberRounds 0

```
seed 0
```

```
[Map Params]
```

```
lengthX 7
```

```
lengthY 5
```

```
[Fields]
```

```
field 0 0 gp1
```

```
field 0 1 n1
```

```
field 0 3 t1
```

```
field 5 0 t1
```

```
field 6 1 t1
```

Der Parameter *numberRounds* ist schon aus dem vorherigen Abschnitt bekannt. Neu ist der Parameter *seed* mit welchem die zufällige Generierung der Kosten für das Gehen von einem Feld zu einem anderen und die symbolischen Positionen der Felder beeinflusst werden können. Ist *seed* 0, dann werden bei jedem Start eines Spieles der informierten bzw uninformierten Suche, die Kosten, sowie die symbolischen Positionen zufällig neu generiert. Ist *seed* aber größer als 0, dann werden zwar die Kosten und die symbolischen Positionen mit Hilfe einer Zufallsfunktion generiert, jedoch bei gleichem *seed* haben zwei gestartete Spiele die gleichen Kosten und symbolischen Positionen. Voraussetzung ist allerdings, daß die Kartengröße bei beiden Spielen die gleiche ist.

Nach dem Setzen der Parameter muss nun noch die Karte definiert werden. Dazu wird ebenfalls die Kartengröße mit *lengthX* und *lengthY* bestimmt. Danach müssen die Objekte auf der Karte verteilt werden. Das Spielziel in dieser Spielart lautet mit Hilfe eines einzelnen Eichhörnchens eine Nuss zu finden. Da es nur ein Eichhörnchen und eine Nuss auf dem Spielfeld gibt, werden beide intern definiert. Man muss nur noch die Nuss (Zielfeld) mit Bezeichner *n1* und die Sammelstelle *gp1*, von der das Eichhörnchen *s1* startet, setzen. Darüberhinaus existiert noch eine Baumart *t1*, die beliebig auf dem Spielfeld verteilt werden kann. Wie bei den anderen Spielarten wird ein Objekt folgendermaßen auf das Feld mit den Koordinaten (x y) gesetzt:

field x y Objektbezeichner

A.4 Konfigurationsdatei für den Gamemaster

In der Konfigurationsdatei *server.conf* können die vier in Abschnitt A.3 erwähnten Spielarten aktiviert oder deaktiviert werden. Ist eine Spielart aktiviert, dann muss der Pfad zur Konfigurationsdatei der Spielart angegeben werden. Eine Beispielkonfiguration sieht folgendermaßen aus:

```
[Enabled Game Types]
UninformedSearch      true
InformedSearch        true
OnePlayer              false
TwoPlayer              true

[Files]
UninformedSearch      maps/testUIS.map
InformedSearch        maps/testIS.map
OnePlayer              maps/test1P.map
TwoPlayer              maps/test2P.map
```

In dieser Konfigurationsdatei sind alle Spielarten bis auf das Ein-Spieler-Spiel aktiviert und für jede Spielart wird im Abschnitt [Files] eine Konfigurationsdatei angegeben.

A.5 Log-Dateien

Alle Log-Dateien befinden sich im Ordner "logs". In der Datei server.conf wird gespeichert, wenn sich ein Spieler für ein Spiel registriert, ein Spiel gestartet und ein Spiel beendet wird. Darüber hinaus gibt es für jedes gespielte Spiel eine eigene Log-Datei. Der Name der Log-Datei wird dabei in Abhängigkeit der IDs der Spieler, der Uhrzeit und des Datums gewählt. Ein Beispiel für einen Namen einer Log-Datei ist:

```
792 - 532 | 2006.05.05. - 21:33:59.log
```

Dieser Name sagt aus, daß am 05.05.2006 um 21:33:59 Uhr ein Spiel mit zwei Spielern, welche die IDs 792 und 532 zugewiesen bekommen haben, gestartet wurde. In diesen Logs werden alle von einem Spieler empfangenen Spielzüge und gegebenenfalls daraus resultierende Fehler gespeichert.

A.6 Schnittstelle des Gamemasters

Auf folgende Methoden des Gamemasters kann per XML-RPC zugegriffen werden um einen eigenen Clienten zu implementieren.

```
public Hashtable regSearch (String gametype)
```

Mit `regSearch` ist die Registrierung eines neuen Spielers für uninformierte bzw. informierte Suche möglich. Damit der Server weiß, welches Spiel man spielen will, muss der String `gametype` entweder den Wert 'uis' für uninformierte Suche oder 'is' für informierte Suche haben. Rückgabewert der Methode ist ein Hashtable-Objekt, welches die beiden Schlüssel 'id' und 'output' beinhaltet. 'id' hat als Wert die neu generierte ID des Spielers und 'output' hat als

Wert die Startinformationen für den Spieler.

```
public Hashtable reg1P ()
```

Mit `reg1P` ist die Registrierung eines neuen Spielers für ein Ein-Spieler-Spiel möglich. Wie bei `regsearch` ist der Rückgabewert ein Hashtable Objekt, welches die ID und die Startinformationen für den Spieler beinhaltet.

```
public Hashtable reg2P (boolean createGame)
```

Mit `reg2P` ist die Registrierung eines neuen Spielers für ein Zwei-Spieler-Spiel möglich. Wie bei `regsearch` und `reg1P` ist der Rückgabewert ein Hashtable Objekt, welches die ID und die Startinformationen für den Spieler beinhaltet. Damit der Server weiß, ob der Spieler ein Spiel starten will oder ob er einem Spiel beitreten will, muss das Argument `createGame`, welches entweder 'true' oder 'false' sein darf, mitgeliefert werden.

```
public String submitMoveSearch (int id, String move)
```

```
public String submitMove1P (int id, String move)
```

```
public String submitMove2P (int id, String move)
```

Mit `submitMoveSearch`, `submitMove1P` und `submitMove2P` kann ein Spieler seinen Zug für die jeweilige Spielart an den Server übermitteln. Der Spieler muss dafür seine ID als `int` und den Zug als `String` an den Server übergeben. Der Rückgabewert der Methode ist ein `String` mit der Antwort des Gamemasters.

B Beispielimplementierung Uniform-Cost-Algorithmus

Die nachfolgende Beispielimplementation eines Agenten ist in JAVA programmiert und implementiert die Uniform-Cost-Suche. Die grobe Funktionsweise ist in dem Testspiel der uninformierten Suchen in Sektion 6.1 zu sehen. Hauptbestandteil ist eine Liste mit Pfaden und deren Kosten. In jeder Runde wird das letzte Feld des Pfades mit den aktuell wenigsten Kosten ausgewählt, bis das Ziel erreicht ist.

```

1 import java.util.Vector;
2 import java.util.TreeSet;
3
4 public class Agent {
5
6     private int round = 0;
7     private String startPos;
8     private String goalPos;
9     private Vector<String> pathlist = new Vector<String>();
10    private Vector<Integer> cost = new Vector<Integer>();
11    private TreeSet<String> alreadyVisited = new TreeSet<String>();
12    private String last = "";
13
14    public String getMove(String infos)
15    {
16        String[] splittedInfos = infos.split("\\\\|posmoves");
17        String[][] parsedInfos = new String[splittedInfos.length][];
18        int infoIndex = 1;
19
20        infos = infos.toLowerCase();
21
22        for (int i = 0; i < splittedInfos.length; i++)
23            parsedInfos[i] =
24                splittedInfos[i].replaceFirst("[(\\_)]+", "").split("[(\\_)]+");
25
26        if (round==0)
27        {
28            this.startPos = parsedInfos[0][1];
29            this.goalPos = parsedInfos[1][1];
30            infoIndex += 2;
31            this.pathlist.add(this.startPos);
32            this.cost.add(new Integer(0));
33            this.alreadyVisited.add(this.startPos);
34        }
35
36        int indexOfSmallest = indexOfSmallestElement();
37
38        for (; infoIndex < parsedInfos.length; infoIndex++)

```

```

39     if (!alreadyVisited.contains(parsedInfos[infoIndex][0]))
40     {
41         pathlist.add(pathlist.elementAt(indexOfSmallest)
42             + parsedInfos[infoIndex][0]);
43         cost.add(new Integer(parsedInfos[infoIndex][1])
44             + cost.elementAt(indexOfSmallest));
45     }
46
47     pathlist.removeElementAt(indexOfSmallest);
48     cost.removeElementAt(indexOfSmallest);
49
50     while(true)
51     {
52         indexOfSmallest = indexOfSmallestElement();
53         String chosenMove = pathlist.elementAt(indexOfSmallest);
54         last = chosenMove.substring(chosenMove.length()-1,chosenMove.length());
55
56         if (alreadyVisited.contains(last))
57         {
58             pathlist.removeElementAt(indexOfSmallest);
59             cost.removeElementAt(indexOfSmallest);
60         }
61         else
62             break;
63     }
64
65     alreadyVisited.add(last);
66     round++;
67     System.out.println("Current path with smallest cost: "
68         +pathlist.elementAt(indexOfSmallest)+" "
69         +cost.elementAt(indexOfSmallest));
70     return last;
71 }
72
73 private int indexOfSmallestElement()
74 {
75     int indexOfSmallest = 0;
76     int smallestElement = this.cost.elementAt(0);
77
78     for (int i = 0; i < cost.size(); i++)
79         if (cost.elementAt(i) < smallestElement)
80         {
81             indexOfSmallest = i;
82             smallestElement = this.cost.elementAt(indexOfSmallest);
83         }

```

```
84
85     return indexOfSmallest;
86 }
87 }
```

In den Zeilen 16-24 wird die vom Server kommende Nachricht in einzelne Teile aufgeteilt und in Arrays gespeichert. Anschließend wird in den Zeilen 26-34 die vom Server übermittelte Start- und Zielposition gespeichert, sowie die Startposition als erstes Feld in der Pfadliste mit Kosten von 0 gespeichert und der Menge der schon besuchten Felder hinzugefügt. Die Zeilen 28-33 werden allerdings nur in der ersten Runde ausgeführt. Danach werden in den Zeilen 36-48 die vom Server übermittelten angrenzenden Felder in die Pfadliste aufgenommen, solange sie nicht schon bekannt sind. Da in dieser Implementation die Pfadliste nicht nach Kosten sortiert ist, wird eine Hilfsmethode `indexOfSmallest` verwendet. `indexOfSmallest` gibt den Index für die Pfadliste zurück, an dem der Pfad mit den aktuell niedrigsten Kosten gespeichert ist. Dies ist gleichzeitig der Pfad, auf dem das Eichhörnchen zu diesem Zeitpunkt steht, da es sich immer auf das Feld bewegt, welches, in Addition mit den Kosten des schon zurückgelegten Weges, die niedrigsten Kosten hat. Jedes noch nicht erkundete Feld wird also an den zur Zeit kürzesten zurückgelegten Pfad angehängen. Ist dies geschehen, wird der Pfad mit dem Feld, auf dem das Eichhörnchen steht, aus der Pfadliste gelöscht. In den Zeilen 50-62 wird anschließend das letzte Feld des neuen Pfades mit den derzeit niedrigsten Kosten ausgewählt. Da das Eichhörnchen dieses Feld als nächstes betreten will, wird der Bezeichner dieses Feldes anschließend dem Gamedeveloper geschickt. Dies wiederholt sich nun solange bis die Nuss gefunden wurde und der Agent gibt den kürzesten Pfad zu vom Startfeld zum Zielfeld aus.

Literatur

- [1] UserLand Software, Inc., *XML-RPC Home Page*, <http://www.xmlrpc.com>
- [2] Stanford University, *General Game Playing Project*, <http://games.stanford.edu>
- [3] Nathaniel Love, Timothy Hinrichs, Michael Genesereth, *General Game Playing: Game Description Language Specification*, http://games.stanford.edu/gdl_spec.pdf
- [4] Sun Developer Network, *Sun Java Home Page*, <http://java.sun.com>
- [5] Robert Vollmann, *AIchhörchen Home Page*, <http://w5.cs.uni-sb.de/~dudel>
- [6] Stuart J. Russel, Peter Norvig, *Artificial Intelligence: A Modern Approach (Second Edition)*, Prentice Hall Series in Artificial Intelligence. Eaglewood Cliffs, New Jersey