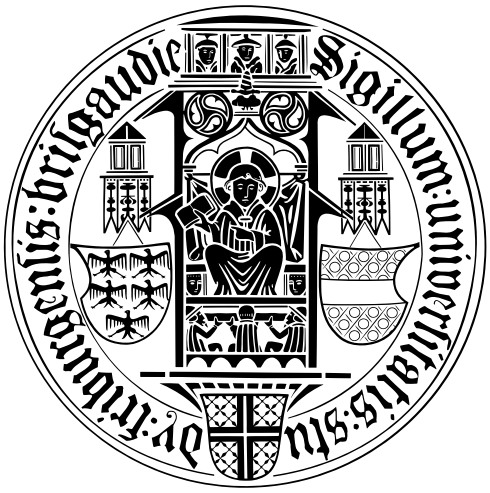
Arbeitsgruppe Grundlagen der künstlichen Intelligenz
Institut für Informatik
Albert-Ludwigs Universität



Erweiterung eines Planungssystems zum Lösen von Ein-Personen-Spielen

Unter der Aufsicht von:
Prof. Dr. Bernhard Nebel
Thomas Keller
Robert Mattmüller

Eingereicht von: Silvan Sievers

Am: 21.10.2009

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbstständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe und alle Stellen, die aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht bereits für eine andere Prüfung angefertigt wurde.

Freiburg, den 21. Oktober 2009

Silvan Sievers

Inhaltsverzeichnis

1 Einführung

Die künstliche Intelligenz hat sich schon früh mit Spielen beschäftigt. Zu Beginn wurde zum Beispiel das Schachspiel intensiv untersucht. Es wurden Schachcomputer und Schachprogramme entwickelt, die auch schon früh gegeneinander antraten (Erste World Computer Chess Championship 1974, organisiert von der International Computer Games Association[1]). Schließlich wurden die Programme aber immer wettbewerbsfähiger, auch gegenüber dem Menschen, und konnten letztendlich sogar Schachweltmeister besiegen. So gelang es zum Beispiel Deep Blue von IBM gegen Schachweltmeister Kasparow in den Jahren 1996 und 1997 zu gewinnen [2]. Auch andere Spiele wurden detailliert untersucht und entsprechende Programme können zumeist auf einem ordentlichen Amateurniveau mithalten, und teilweise sogar professionelle Spieler schlagen. Allerdings kommen Programme bei Spielen wie Go aufgrund der nahezu unbegrenzten Möglichkeiten an Spielzügen und Spielsituationen und der komplexen taktischen Vielfalt an ihre Grenzen.

Ein neuerer Bereich der künstlichen Intelligenz versucht sich an der Lösung allgemeiner Spiele, die in einer bestimmten Kodierung an Computerprogramme übermittelt werden. Diese können dann also keinerlei spielspezifisches Wissen ausnutzen, sie müssen nur an Hand der Spielregeln versuchen, optimale Spielzüge herauszufinden. Dieses Gebiet der künstlichen Intelligenz wird General Game Playing (GGP) [6] genannt. Mit einem Spezialfall davon, nämlich mit der Lösung allgemeiner Ein-Personen-Spiele, beschäftigt sich diese Bachelorarbeit. Diese sind in ihrer Art der Handlungsplanung recht ähnlich, da keinerlei Interaktion mit anderen Agenten berücksichtigt werden muss und man davon ausgehen kann, dass der Spielzug, für den man sich entschieden hat, auch tatsächlich gespielt wird und die erwartete Wirkung hat. Auch in der Handlungsplanung werden allgemeine Probleme in einer bekannten Kodierung übermittelt. Planungssysteme lösen dann auf unterschiedliche Arten und Weisen diese Probleme. Das Ziel dieser Bachelorarbeit ist es, ein erfolgreiches Planungssystem, den FastDownward-Planer [8], dazu zu verwenden, um Ein-Personen-Spiele zu spielen. Dazu bedarf es zunächst einer Transformation von der Spielekodierung in das Format der Planungsprobleme und weiterhin einiger Modifikationen am Planungssystem selbst.

Die ersten beiden Kapiteln dieser Arbeit geben zunächst eine Einleitung in die beiden Gebiete GGP und Handlungsplanung und grenzen sie voneinander ab. Im darauf folgenden Kapitel, dem ersten von zwei Hauptteilen der Arbeit, wird zunächst formal erläutert, was ein Spiel und was eine Planungsaufgabe ist. Im Anschluss daran wird eine Transformation einer Spielkodierung zu einer Planungskodierung definiert und ihre Korrektheit bewiesen. Im darauf folgenden zweiten Hauptteil der Arbeit werden die vorgenommenen Modifizierungen am FastDownward-Planer sowie der grobe Ablauf des Suchalgorithmus beschrieben. In einer abschließenden Zusammenfassung wird ein Fazit gezogen und ein Ausblick gegeben.

[1] http://www.icga.org/

2 General Game Playing

General Game Playing ist das Gebiet der Künstlichen Intelligenz, welches sich mit extensiven Spielen beschäftigt. Von Bedeutung ist hierbei vor allem die Tatsache, dass Computerprogramme allgemeine Spiele lösen sollen, die sie zuvor nicht kennen. Definiert ist nur das Eingabeformat, über welches die Spielregeln sowie alle weiterhin relevanten Informationen übermittelt werden. Neben dem Forschungsaspekt und ihrem Unterhaltungswert sind allgemeine Spiele und Spielmodellierung auch für die Wirtschaft von Bedeutung: Nicht zuletzt die Spieltheorie, ursprünglich ein Gebiet der Wirtschaftswissenschaften, kann bei der Lösung extensiver Spiele mit einbezogen werden. Umgekehrt können dann auch Lösungen von Computerprogrammen für als „Spiel“ modellierte Probleme und Situationen aus der Wirtschaft von Interesse sein.

2.1 Wettbewerb

Einen guten Rahmen für GGP liefert der jährlich Jahre abgehaltene Wettbewerb [6] der Association for the Advancement of Artificial Intelligence (AAAI), in welchem Teilnehmer in einem Turnierformat ihre Computerprogramme gegeneinander antreteten lassen können. Dabei stellt die AAAI einen Game Master (GM) zur Verfügung, der mit den anderen Spielern online über das Hyper Text Transfer Protocol (HTTP) kommuniziert. Die Spiele werden in der Game Description Language (GDL) [9] kodiert übermittelt. Nach Ablauf einer vorher festgelegten Vorbereitungszeit beginnt das Spiel und der GM verlangt dann regelmäßig von allen Teilnehmern die Übermittlung eines Spielzuges, um diesen sowie die Spielzüge der anderen Spieler an alle Teilnehmer zu übermitteln. Weiterhin kann der GM zufällige Züge auswählen, falls ein illegaler oder gar kein Zug übermittelt wurde, so dass der entsprechende Spieler im Spiel verbleiben kann. Die Teilnehmer können außerdem über ein grafisches Interface die Spiele auf eine anschauliche Art mitverfolgen, haben aber natürlich keinerlei Einfluss mehr auf den Spielverlauf.

2.2 Spielmodell

Ein Spiel, welches in GDL kodiert ist, ist ein endliches, synchrones Spiel. Endlich bedeutet hierbei, dass sowohl die Anzahl der Spieler als auch die der möglichen Spielzüge (und damit der Spielsituationen) begrenzt ist. Synchron heißt, dass alle Spieler gleichzeitig einen Spielzug machen (siehe Wettbewerbsmodell). Dies stellt allerdings keine Einschränkung für extensive Spiele mit abwechselnden Zügen dar, da in den meisten Spielen pro Runde immer nur ein Spieler alle legalen Spielzüge zur Wahl hat, während alle anderen nur eine leere „nichts-tun“ Aktion (noop) spielen dürfen. Auf diese Weise werden auch Spiele mit abwechselnden Zügen von GDL abgedeckt. Wie schon oben erwähnt stellt GDL eine allgemeine Kodierung für Spiele dar, was die eigentliche Herausforderung für die Computerprogramme darstellt, da sie sowohl einfache Spiele lösen müssen (eventuell sogar solche, in denen alle möglichen Spielsituationen berechnet werden können und man deshalb immer den optimalen Spielzug kennt) als auch extrem komplexe, bei denen man stark auf effiziente Suchalgorithmen angewiesen ist.

Ein in GDL kodiertes Spiel lässt sich wie folgt beschrieben: In der Spielwelt gibt es verschiedene Objekte, die mit Hilfe der Spielzüge, die durch die Spielregeln definiert werden, verändert werden können. Das Spiel beginnt mit einer bestimmten Konfiguration der Spielwelt und endet, sobald ein bestimmtes Kriterium erfüllt wurde. Anschaulich lässt sich dies an Hand eines allgemeinhin bekannten Beispiels darstellen: Im Zwei-Personen-Spiel *TicTacToe* entspricht das quadratische Spielfeld, welches aus 3×3 Feldern besteht und zu Beginn leer ist, der Spielwelt. Die Zellen des Feldes können dabei als Objekte aufgefasst werden, welche die Eigenschaft „leer“, „Kreis“ oder „Kreuz“ haben können. Die Spieler markieren nun abwechselnd mit Hilfe sogenannter Aktionen, den Spielzügen, ein Kästchen des Spielfelds mit ihrem Symbol. Das Ziel ist es, 3 Felder in einer Reihe mit seinem eigenen Symbol zu markieren. Die Symbole müssen dabei entweder eine horizontale, vertikale oder diagonale Linie bilden. Die formalen Definitionen solcher Spiele werden in Kapitel 4 gegeben.

Computerprogramme, welche allgemeine Spiele lösen und dies auf wettbewerbsfähige Art und Weise tun, implementieren meistens Suchalgorithmen, mit denen die Spielbäume durchsucht werden. Der Schlüsselpunkt ist es, für die aktuelle Runde immer einen möglichst guten Spielzug zu berechnen, der auch die möglichen Spielzüge der anderen Spieler mit einbezieht. Man kann nicht weit vorausplanen, da die neue Spielsituation entscheidend von den Zügen der Mitspieler abhängt, was die Anzahl an zu betrachtenden Möglichkeiten enorm erhöht.

3 Handlungsplanung

Handlungsplanung ist ein weiterer Bereich der Künstlichen Intelligenz, in welchem man für allgemein definierte Probleme in unterschiedlichen Varianten Lösungen sucht. Mit der Planning Domain Definition Language (PDDL) [4] gibt es auch hier eine bewährte Kodierung, die auch stets aktualisiert und optimiert wird. Die klassische Handlungsplanung, welche sich am ehesten mit dem Spezialfall von Ein-Personen-Spielen, vergleichen lässt, befasst sich mit deterministischen Planungsaufgaben wie zum Beispiel Transportdomänen, Logistikdomänen oder Ein-Personen-Spiel-Domänen wie Blocksworld oder Freecell. In der Handlungsplanung gibt es bereits zahlreiche Möglichkeiten, den Problemstellungen zu begegnen; nicht zuletzt an der Uni Freiburg wurden sehr erfolgreiche Planungssysteme entwickelt, die solche allgemeinen Planungsprobleme lösen können.

3.1 Wettbewerb

Auch hier gibt es einen Wettbewerb, welcher alle zwei Jahre im Rahmen der International Conference on Automated Planning and Scheduling (ICAPS)[2] stattfindet. Den Teilnehmern werden dabei unterschiedlichste Planungsdomänen samt zugehörigen Problemsets in PDDL kodiert übermittelt. Es gibt mehrere sogenannte „Tracks“ im Planungswettbewerb, bei denen unterschiedliche Teilgebiete des Planens untersucht werden (optimales oder satisficing Planen, deterministisches oder nicht-deterministisches Planen, temporales Planen uvm.). Die Teilnehmer müssen bis zu einer gewissen Deadline ihre Planungssysteme übermitteln. Danach werden diese in einer vorher festgelegten Testumgebung auf zahlreichen Problemsets getestet und evaluiert. Nach bestimmten Kriterien der unterschiedlichen Tracks werden dabei Punkte vergeben und so der Gewinner ermittelt. Bereits drei erfolgreiche Planungssysteme wurden an der Universität Freiburg entwickelt: FastForward [10], FastDownward [8] und Lama [12].

3.2 Planungsaufgaben

Die Probleme der Handlungsplanung werden mit Hilfe von Planungsaufgaben formuliert. Diese werden dann den Planungssystemen zum Lösen übermittelt. Auch hier gilt, dass die in den teilweise sehr großen Zustandsräumen begründete Schwierigkeit darin liegt, dass man keinerlei problemspezifische Tatsachen ausnutzen kann, da die Planer allgemein kodierte Probleme unterschiedlichster Art lösen sollen.

In PDDL kodierte Planungsprobleme werden ähnlich wie Ein-Personen-Spiele in GDL durch Objekte und Aktionen beschrieben. Die Planungswelt besteht aus Objekten mit Eigenschaften, welche durch Aktionen verändert werden können. Zu Beginn ist eine bestimmte Situation der Welt gegeben, welche in eine Zielkonfiguration transformiert werden soll. Anschaulich lässt sich dies wiederum durch ein Beispiel erläutern. Im Planungsproblem *Blocksworld* gibt es verschieden benannte Blöcke (Objekte), die zum Beispiel auf einem Tisch liegen. Zu Beginn liegen sie in einer mehr oder weniger willkürlichen

[2] http://icaps-conference.org/

Anordnung neben- und aufeinander. Es kann stets nur ein Block über einem anderen liegen, niemals jedoch einer auf zweien oder zwei auf einem. Die Eigenschaften der Blöcke ist dann zum Beispiel ihre Position, also „auf Block X“ oder „auf dem Tisch“. Als Ziel ist dann eine bestimmte Konfiguration der Blöcke vorgegeben, die erreicht werden muss. Man kann mit Hilfe von Aktionen immer nur einen Block bewegen, entweder von einem Block auf einen anderen oder auf den Tisch bzw. umgekehrt vom Tisch auf einen Block. Das Ziel ist es also eine bestimmte Abfolge von Aktionen zu finden, welche die Blöcke in ihre Zielkonfiguration bewegt. Eine formale Definition wird im Abschnitt 4 gegeben.

In klassischen Planungsaufgaben sind Aktionen deterministisch definiert und haben eine eindeutige Wirkung auf die Planungswelt. Außerdem ist ihre Anzahl sowie die Anzahl der möglichen Zustände der Welt beschränkt. Typischerweise implementieren Computerprogramme, die Planungsprobleme lösen, Suchalgorithmen unter Verwendung einer oder mehrerer kombinierter Heuristiken. Im Unterschied zu allgemeinen Spielen müssen Programme hier keinerlei Fremdinteraktionen (wie von anderen Spielern) berücksichtigen.

4 Von GDL nach PDDL

In diesem ersten großen Teil der Bachelorarbeit wird die Transformation von in GDL kodierten Spielen nach PDDL beschrieben, damit diese dann von dem modifizierten FastDownward-Planer (siehe Kapitel 5) zur Lösung des Spiels verwendet werden können. Da GDL auf Datalog basiert [9], einer Datenbanksprache, die wiederum eng an der logischen Programmiersprache Prolog angelehnt ist, besteht diese Transformation hauptsächlich aus logischen Operationen und Vereinfachungen und nur wenigen GDL- bzw. PDDL spezifischen Änderungen.

4.1 GDL

Um GGP-Spiele einheitlich zu kodieren und Teilnehmern am Wettbewerb in einem klar definierten Format zu übermitteln, wurde GDL entwickelt. Im Unterschied zu natürlicher Sprache, welche viel zu komplex und teilweise nicht eindeutig genug für solche Zwecke ist, bietet GDL klare Strukturen, die es den Programmierern der Systeme erlaubt, sich ganz auf die eigentliche Aufgabe, nämlich dem Lösen der Spielen, zu konzentrieren. Im Folgenden soll die Syntax von GDL stark zusammengefasst wiedergegeben und erläutert werden.

4.1.1 Syntax

Im Wettbewerb und auch allgemein wird eine für Computer geeignetere Version als die Reinform von GDL verwendet, und zwar das „Knowledge Interchange Format“ (KIF), welches bestimmte (für Computer „angenehme“) Schreibweisen für Variablen, Relationsbezeichner und Objekte festlegt, so dass auch die Portabilität auf unterschiedlichen Betriebssysteme gewährleistet bleibt. An dieser Stelle soll darauf nicht weiter eingegangen werden, da nur die abstrakte Syntax von Bedeutung ist, nicht jedoch die konkrete. Weiterhin werden einige Annahmen gemacht, die für diese Arbeit genügen und die Allgemeinheit nicht verletzen: Die GDL-Beschreibung, die nach PDDL transformiert werden soll, ist schon normalisiert, das heißt, es gibt insbesondere keine Disjunktionen mehr in den Rümpfen von Regeln. Außerdem wird angenommen, dass jedes Prädikat statisch ist oder direkt von einer Aktion beeinflusst wird.

Definition 1 (GDL-*Signatur*)**.** Eine GDL-*Signatur* ist ein 4-Tupel (V, F, P, α), wobei V eine abzählbare Menge von *Variablen*, F eine endliche Menge von *Funktionssymbolen*, P eine zu F disjunkte endliche Menge von *Prädikatensymbolen* (auch *Relationssymbole*) und α: $F \cup P \to \mathbb{N}_0$ eine Funktion ist, welche jedem Funktions- und Prädikatsysmbol eine *Stelligkeit* zuordnet.

Bei GDL gibt es weiterhin eine Menge von vordefinierten Prädikatensymbolen, die später noch näher im Detail erläutert werden.

Definition 2 (Vordefinierte Prädikate)**.** $PreDef := \{NEXT, INIT, TRUE, DOES, LEGAL, GOAL, TERMINAL, DISTINCT, ROLE\}$ mit $PreDef \subseteq P$.

Sei eine solche Signatur (V, F, P, α) gegeben.

Definition 3 (Term, Atom, Literal)**.** Ein *Term* t ist entweder eine Variable $v \in V$ oder ein Ausdruck $f(t_1, \ldots, t_n)$ für ein $f \in F$ mit $\alpha(f) = n$ und Terme $t_1, \ldots, t_n$, wobei hier nullstellige Funktionen, also (Objekt-)Konstanten, mit eingeschlossen sind (für $n = 0$). Um unendlich große Terme zu verhindern, dürfen die Terme $t_1, \ldots, t_n$ im zweiten Fall nur Variablen oder Konstanten sein. Die Menge aller Terme sei T. Ein *Atom* ist ein Ausdruck $p(t_1, \ldots, t_n)$ für $p \in P$ mit $\alpha(p) = n$ und Terme $t_1, \ldots, t_n$. Die Menge aller Atome sei AF. Ein *Literal* ist entweder ein Atom $a \in A$ oder der Ausdruck $\neg a$ für ein Atom $a \in A$. Die Menge aller Literale sei L.

Definition 4 (Datalogregel)**.** Sei $fr(literal)$ die Menge der freien Variablen eines Literals $literal \in L$. Eine *Datalogregel* ist eine Formel

$$\forall \vec{x} : \left(h(\vec{x}) \leftarrow \exists \vec{y} : \underbrace{\bigwedge_i b_i(\vec{x}, \vec{y})}_{=: b(\vec{x}, \vec{y})} \right), \; h(\vec{x}) \in AF, \; b_i(\vec{x}, \vec{y}) \in L$$

wobei $fr(h(\vec{x})) = \{x_1, \ldots, x_k\} =: \vec{x}$ und $fr(b(\vec{x}, \vec{y})) \setminus fr(h(\vec{x})) = \{x_{k+1}, \ldots, x_{k+m}\} =: \vec{y}$. Hierbei ist $h(\vec{x})$ der Kopf der Regel (Head), welcher gerade die Folge der Implikation darstellt, und $\bigwedge_i b_i(\vec{x}, \vec{y})$ der Rumpf der Regel (Body), die Bedingung.
Die Menge aller Datalogregeln sei $\mathcal{R}$.

Datalogregeln sind stets allquantifiziert über den freien Variablen des Kopfes und existenzquantifiziert über den verbleibenden freien Variablen des Rumpfes. In GDL gibt es ein sogenanntes *Sicherheitskriterium*, welches stets gelten muss: Wenn eine Variable im Kopf einer Regel oder in einem negativen Literal vorkommt, muss sie auch in einem positiven Literal im Rumpf der Regel stehen.

Definition 5 (Grundausdruck)**.** Enthält ein Ausdruck (Term, Atom, Literal oder Regel) keine Variablen, so bezeichnet man ihn als *Grundausdruck* (Grundterm, Grundatom, Grundliteral oder Grundregel).

Um Problemen mit Negation in Datalog entgegen zu wirken, müssen Mengen von Datalogregeln *stratifiziert* sein. Eine Menge von Datalogregeln ist stratifiziert, wenn man von einem Relationssymbol p ausgehend niemals $\neg p$ erreichen kann, indem man den Regeln rückwärts folgt. Formal ist Stratifizierbarkeit basierend auf dem *Abhängigkeitsgraph* definiert.

Definition 6 (Abhängigkeitsgraph)**.** Die Knoten des Abhängigkeitsgraphen G für eine Menge von *Datalogregeln* $R \subseteq \mathcal{R}$ werden durch die Relationssymbole der Signatur gebildet. Es gibt eine Kante von $r_2 \in R$ nach $r_1 \in R$, wenn es eine Regel gibt, die r_1 im Kopf und r_2 im Rumpf hat. Diese Kante wird mit $\neg$ beschriftet, falls r_2 in einem negativen Literal vorkommt.

Definition 7 (Stratifizierte Datalogregeln)**.** Eine Menge von Datalogregeln $R \subseteq \mathcal{R}$ ist genau dann *stratifiziert*, wenn kein Zykel im Abhängigkeitsgraphen dieser Datalogregelmenge mit $\neg$ beschriftete Kanten enthält.

Definition 8 (GDL-*Spezifikation*)**.** Eine GDL-*Spezifikation* (oder GDL Spielbeschreibung) ist ein Tupel $\Gamma = (AF, R, s_0, \mathcal{T})$, wobei AF die Menge der atomaren Formeln ist und $R \subseteq \mathcal{R}$ eine Menge von Datalogregeln. Weiterhin seien $x_1, x_2, \ldots$ eine abzählbare Menge von Variablen.
Sei GAF die Menge der Grundinstanzen der atomaren Formeln aus AF. Der Startzustand ist durch $s_0 \subseteq GAF$ gegeben, als eine Menge grundatomarer Formeln. Selbiges gilt für die Menge der Terminalzustände $\mathcal{T} \subseteq 2^{GAF}$.

Hierbei ist anzumerken, dass sich s_0 und $\mathcal{T}$ prinzipiell aus den Regeln R ablesen lassen und diesen auch entsprechen müssen. Diese Redundanz der Definition erleichtert die technische Transformation in 4.3.
Weiterhin gelten einige Einschränkungen, insbesondere gibt es die oben genannte Menge vorbestimmter Prädikatensymbole *PreDef*, wobei ihre Elemente folgende Bedeutungen haben:

- $NEXT(t)$ besagt, dass ein Term t im nächsten Zustand gültig sein wird.
- $INIT(t)$ beschreibt die im Startzustand gültigen Terme t.
- $TRUE(t)$ beschreibt analog zu $INIT(t)$ die im aktuellen Zustand gültigen Terme.
- $DOES(a)$ beschreibt die Ausführung einer Aktion a.
- $LEGAL(a)$ kennzeichnet die Anwendbarkeit einer Aktion a.
- $GOAL(x)$ ordnet dem Spieler einen Wert x im entsprechenden Zielzustand zu.
- $TERMINAL$ besagt, dass das Spiel zu Ende ist.
- $DISTINCT(t_1, t_2)$ testet auf Ungleichheit zwischen t_1 und t_2.
  (Auf einen Test auf Gleichheit kann wegen der Unique Name Assumption verzichtet werden.)
- $ROLE(r)$ legt ein Konstantensymbol r als Spieler fest.

Für diese vordefinierten Prädikatensymbole gelten folgende Restriktionen:

- Die *ROLE*-Relation taucht nur in atomaren Sätzen auf, und zwar als Kopf einer Datalogregel mit leerem Rumpf.
- Die *INIT*-Relation kommt nur in Köpfen von Datalogregeln vor und in G darf *INIT* nicht in derselben Zusammenhangskomponente wie *TRUE*, *DOES*, *NEXT*, *LEGAL*, *GOAL* oder *TERMINAL* stehen.
- Die *TRUE*-Relation steht nur im Rumpf von Datalogregeln.

- Die *NEXT*-Relation kommt nur in Köpfen von Datalogregeln vor.
- Die *DOES*-Relation kommt nur in Rümpfen von Datalogregeln vor und in G gibt es keine Pfade zwischen dem *DOES*-Knoten und anderen Knoten, die mit *LEGAL*, *GOAL* oder *TERMINAL* beschriftet sind.

Eine weitere Restriktion, die hier kurz angeführt werden soll, ist die *Rekursionsrestriktion*. Diese soll in erster Linie verhindern, dass unendlich große Funktionsterme auftreten können und dafür sorgen, dass Dataloganfragen stets entscheidbar sind.

Definition 9 (Rekursionsrestriktion)**.** Sei G der Abhängigkeitsgraph für eine Menge von Datalogregeln R. R enthalte eine Regel der Form

$$p(t_1, \ldots, t_n) \Leftarrow b_1 \wedge \ldots \wedge q(v_1, \ldots, v_k) \wedge \ldots \wedge b_m,$$

wobei q in einem Zykel mit p in G auftritt. Dann ist $\forall j \in \{1, \ldots, k\}$ entweder v_j grund, $v_j \in \{t_1, \ldots, t_n\}$, oder $\exists i = 1, \ldots, m : b_i = r(\ldots, v_j, \ldots)$, wobei r nicht in einem Zykel mit p vorkommt.

Im Zusammenhang mit Datalogregeln ist der Begriff der *Successor State Axioms* zu nennen. Diese sind syntaktisch im Prinzip den Datalogregeln mit *NEXT* im Kopf gleichzusetzen. Die Bedeutung, die ihnen zukommt, ist folgende: Solche Regeln, hier Axiome genannt, berechnen ausgehend von einem aktuellen Zustand den Folgezustand derart, dass nur die Prädikate gelten, die explizit durch die Implikationen der Regeln wahr gemacht werden. Alte Prädikate werden dabei nicht übernommen. Aufgrund dieser Tatsache haben Successor State Axioms den großen Nachteil, dass man sehr viele Regeln benötigt, die keinerlei Effekte haben außer die im Vorgängerzustand bereits gültigen Prädikate auch im Folgezustand gültig zu machen, wenn sie nicht gerade durch andere Regeln falsch gemacht werden. Solche Axiome werden auch *Rahmenaxiome* oder *Frameaxiome* genannt. Während diese bereits das *representational problem* gut lösen, welches darin besteht, Nicht-Effekte von Aktionen zu spezifizieren, so lösen sie nicht das *inferential problem*, welches darin besteht, diese Nicht-Effekte tatsächlich zu berechnen. Möchte man zum Beispiel ein Prädikat in Situationen beweisen, in denen es nicht gerade trivialerweise durch eine Regel wahr gemacht wurde, so muss man über verschiedene Instanzen von Rahmenaxiomen diese Eigenschaft bis dorthin zurückverfolgen, wo sie erzeugt wurde oder gegeben war. Die Idee der Transformation zu PDDL, welches *Successor Update Axioms* beinhaltet und damit das inferential problem löst, liegt dieser Arbeit zu Grunde und wird ausführlich von Michael Thielscher beschrieben [13].

4.1.2 Semantik

Die Semantik basiert auf dem logischen Modell, welches die Menge der Datalogregeln erfüllt. Sei $\mathcal{L}_{GDL}$ die Sprache, die auf dem *Vokabular* der GDL-Signatur (Variablen, Funktionssymblole, Relationssymbole und Objektkonstanten) definiert ist und eine Menge grundatomarer Sätze ist.

Definition 10 (Modell)**.** Ein Modell für $\mathcal{L}_{GDL}$ ist eine Menge von grundatomaren Formeln in $\mathcal{L}_{GDL}$.

Definition 11 (Erfüllbarkeit)**.** Sei M ein Modell und der gegebene Satz eine Datalogregel.

- $\models_M t_1 = t_2$ genau dann wenn t_1 und t_2 der syntaktisch gleiche Term sind.
- $\models_M p(t_1, \ldots, t_n)$ genau dann wenn $p(t_1, \ldots, t_n) \in M$
- $\models_M \neg\varphi$ genau dann wenn $\not\models_M \varphi$
- $\models_M \varphi_1 \wedge \ldots \wedge \varphi_n$ genau dann wenn $\models_M \varphi_i$ für alle i
- $\models_M h \Leftarrow b_1 \wedge \ldots \wedge b_n$ genau dann wenn $\not\models_M b_1 \wedge \ldots \wedge b_n$ oder $\models_M h$ oder beides.
- $\models_M \forall x : \varphi(x)$ genau dann wenn $\models_M \varphi(t)$ für alle Grundterme t

Zur Beschreibung der Semantik wird das minimale Modell gesucht. Dies bereitet einige Probleme, wenn man Datalog mit Negation verwendet, wie es in GDL der Fall ist. Deshalb müssen weitere Einschränkungen gelten.

Definition 12 (Stratum)**.** Sei R eine Menge stratifizierter Datalogregeln und G der zugehörige Abhängigkeitsgraph. Ein Relationssymbol r ist genau dann in *Stratum* i, $i \in \mathbb{N}_0$, wenn die maximale Anzahl mit $\neg$ beschrifteter Kanten aller in r endender Pfade genau i ist. Eine Regel, deren Kopf ein Realtionssymbol in Stratum i enthält, wird selbst als *Regel in Stratum* i bezeichnet.

Jede Regel gehört zu einem endlichen Stratum und es gibt mindestens eine Regel in Stratum 0. Für diese Regeln gibt es ein eindeutiges minimales Modell M_0, da sie Horn sind. Davon ausgehend kann das Modell M_k iterativ wie folgt konstruiert werden:

Definition 13 (Stratifizierte Datalogsemantik)**.** Sei R eine Menge von Datalogregeln und M_0 das minimale Modell, welches die Regeln in Stratum 0 erfüllt. Um M_i für $i > 0$ zu berechnen, initialisiere M_i mit M_{i-1}. Wiederhole folgenden Prozess, bis keine Änderungen mehr auftreten:
Füge $h\sigma$ zu M_i hinzu, wobei h der Kopf einer Regel $h \leftarrow b_1 \wedge \ldots \wedge b_n$ in Stratum i ist und σ eine Substitution dieser Regel, so dass $M_i \models (b_1 \wedge \ldots \wedge b_n)\sigma$ gilt.
Sei k das größte vorkommende Stratum der Regeln in R. Die *stratifizierte Datalogsemantik* M von R ist M_k.
Abkürzend wird $SD := M$ als stratifizierte Datalogsemantik einer Regelmenge R geschrieben.

Hierbei kommt das Sicherheitskriterium von Datalogregeln zum Tragen: Ohne dieses könnten unendliche Modelle entstehen. Außerdem stellt die Rekursionsrestriktion sicher, dass die beschriebene Prozedur zur Erzeugung des Modells terminiert, da die neu hinzugefügten Köpfe jeder Runde keine unendlichen Modelle erzeugen können.
Es kann weiterhin eine Aussage getroffen werden, ob ein Literal φ bereits aus einer Regelmenge folgt.

Definition 14 (Logische Schlussfolgerung)**.** Sei R eine Menge von Datalogregeln und M die stratifizierte Datalogsemantik für R. Dann gilt: $R \models \varphi$ genau dann wenn $\models_M \varphi$.

Das *Transitionssystem* als Semantik einer GDL-Spezifikation sei der induzierte Spielgraph, dargestellt in der Form eines (endlichen deterministischen) Automaten. Um das Transitionssystem definieren zu können, bedarf es zu nächst einiger weiterer Definitionen.

Definition 15 (Zustand)**.** Die Menge aller Zustände ist $S := 2^{GAF}$. Ein Zustand s ist ein Element dieser Menge S.

Definition 16 (GDL-Aktion)**.** Sei $R = \{r_1, \ldots, r_n\}$ die Menge aller Datalogregeln einer GDL-Spezifikation Γ. Die Menge der Atome über den Datalogregeln ist

$$\begin{aligned} Atoms_\Gamma(R) \quad := \quad & \Bigg\{ \varphi(\vec{x}) \mid \exists r_i \in R : r_i = \forall \vec{x} : \Bigg(h(\vec{x}) \leftarrow \exists \vec{y} : \bigwedge_i b_i(\vec{x}, \vec{y}) \Bigg) \\ & \text{und } \varphi(\vec{x}) = h(\vec{x}) \text{ oder } \varphi(\vec{x}) = b_i(\vec{x}, \vec{y}) \text{ für ein } j = 1, \ldots, n \Bigg\} \end{aligned}$$

Die Menge der Aktionen von Γ ist dann

$$\begin{aligned} Actions_\Gamma \quad := \quad & \{ a(\vec{x}) \mid \exists \varphi : \varphi(\vec{x}) = (DOES\ a(\vec{x})) \text{ für ein Relationssymbol } a \\ & \text{und } \varphi(\vec{x}) \in Atoms_\Gamma \} \end{aligned}$$

Eine Aktion $a(\vec{t})$ ist dann ein Element dieser Menge $Actions_\Gamma$.

Definition 17 (Anwendbare Aktionen, Anwendung einer Aktion)**.** Sei $s \in S$ ein Modell für $\mathcal{L}_{GDL}$. Die Menge $A(s)$ der in diesem Zustand s *anwendbaren Aktionen* ist definiert als:

$$A(s) := \Big\{ a(\vec{t}) \in Actions_\Gamma \mid (LEGAL\ a(\vec{t})) \subseteq s \Big\}$$

wobei $\vec{t}$ ein Vektor von Grundtermen ist.
Die *Anwendung einer Aktion* $a(\vec{t}) \in A(s)$ ist als $applyGDL(a(\vec{t}), s)$ definiert und berechnet einen Folgezustand s'.

Um den Folgezustand s' zu definieren, wird zunächst ein Zwischenschritt s_1 eingeführt, der gerade die stratifizierte Datalogsemantik *SD* des alten Modells s vereinigt der auszuführenden Aktion $a(\vec{t})$ beschreibt. Danach müssen in einem weiteren Zwischenschritt s_2 alle *NEXT*-Prädikate durch *TRUE*-Prädikate ersetzt werden, da man sich nun im Folgezustand befindet und gerade die *NEXT*-Prädikate gelten. Anschließend wird noch einmal die stratifizierte Datalogsemantik davon berechnet und damit ist die Definition des Modells s' des Folgezustandes vollständig.

Definition 18 (Folgezustand im Transitionssystem, Transitionslabel)**.** Sei s ein Zustand und $a(\vec{t})$ eine in diesem Zustand anwendbare Aktion. Die Anwendung dieser Aktion $applyGDL(a(\vec{t}), s)$ führt zum *Folgezustand* s', der folgendermaßen definiert ist:

$$s_1 := SD\Big(s \cup \{(DOES\ a(\vec{t}))\}\Big)$$

$$s_2 := \{\varphi \in AF \mid \varphi \in s_1 \text{ und } \varphi \neq (NEXT\ t) \text{ für alle } t \in T \text{ oder ex. } \psi \in s_1 \text{ mit } \psi = (NEXT\ t) \text{ für ein } t \in T \text{ und } \varphi = (TRUE\ t)\}$$

$$s' := SD(s_2)$$

Die Kante von s nach s' wird dann mit dem Namen der Aktion, a, beschriftet.

Ein Nachfolgezustand wird also komplett neu berechnet, die im alten Zustand geltenden Prädikate werden nicht mitgenommen. Indem die stratifizierte Datalogsemantik des alten Zustands erweitert um die auszuführende Aktion $(DOES\ a(\vec{x}))$ berechnet wird, ergibt sich eine neue Menge von geltenen Atomen, $(TRUE\ \varphi(\vec{y}))$. Diese beschreibt dann gerade den neuen Zustand.

Das Transitionssystem lässt sich dann insgesamt durch folgende Komponenten beschreiben:

- S: Menge von Zuständen, wie oben definiert
- $Actions_\Gamma$: Aktionsmenge des Spielers, wie oben definiert
- $A(s) \subseteq Actions_\Gamma$: legale Züge in einem Zustand
- $applyGDL : A \times S \to S$: Partielle Transitionsfunktion, die aus der gewählten Aktion des Spieles in einem Zustand den Nachfolgezustand wie oben bestimmt.
- $s_0 \in S$: Initialzustand
- $\mathcal{T} \subseteq S$: Terminalzustände
- $g : \mathcal{T} \to \{0, \ldots, 100\}$: Rewardfunktion, die in jedem Terminalzustand dem Spieler einen Nutzenwert (reward, je höher desto besser) zuordnet.

Ein allgemeines Spiel beginnt also in einem Initialzustand s_0 und wird gespielt, indem der Spieler pro Runde eine seiner anwendbaren Aktionen $a \in A(s)$ auswählt. Der Folgezustand berechnet sich dann über die Transitionsfunktion. Es wird solange gespielt, bis ein beliebiger Terminalzustand $t \in \mathcal{T}$ erreicht ist. Für diesen berechnet dann die Rewardfunktion $g(t)$ den Nutzen.
In allgemeinen Mehrpersonenspielen gäbe es zusätzlich eine Menge R von *Rollen*, die für die Spieler stünden, und es müssten die Mengen S und $Actions_\Gamma$ sowie die Funktion g für alle Spieler definiert werden.

4.2 PDDL

Analog zu GDL für GGP-Spiele und den GGP-Wettbewerb ist PDDL für den Bereich der Handlungsplanung ein festgelegtes Format für Planungsaufgaben, welches weithin akzeptiert ist und auch im Planungswettbewerb verwendet wird. PDDL gibt es in zahlreichen Variationen, im Folgenden wird beschrieben, wie das PDDL, das in dieser Arbeit verwendet wird, definiert ist.

4.2.1 Syntax

Die Syntax von PDDL beschreibt Planungsaufgaben. Da PDDL analog zu KIF in GDL gesehen werden kann, spielt die konkrete Syntax (für Computerprogramme) hier keine Rolle für die Arbeit, weshalb sie nicht beschrieben wird. Es wird stattdessen die abstrakte Syntax einer formalen Planungsaufgabe erläutert.

Definition 19 (PDDL-Signatur)**.** Eine PDDL-*Signatur* ist ein 4-Tupel (V, F, P, α), wobei V eine abzählbare Menge von *Variablen*, F eine endliche Menge von *Funktionssymbolen*, P eine zu F disjunkte endliche Menge von *Prädikatensymbolen* bzw. *Relationssymbolen* und α: $P \to \mathbb{N}_0$ eine Funktion ist, welche jedem Prädikatsysmbol eine *Stelligkeit* zuordnet.

Sei eine solche Signatur (V, F, P, α) gegeben.

Definition 20 (Terme)**.** Ein *Term* t ist entweder eine Variable $v \in V$ oder eine (Objekt-)Konstante. Terme haben also Schachtelungstiefe 0. Sei T die Menge aller Terme.

Weiterhin sind *Atome* und *Literale* analog zu denen im Abschnitt über GDL-Syntax definiert, siehe Definition 3.

Definition 21 (PDDL-Aktion, Grundaktion)**.** Eine *Aktion* $a(\vec{x})$ ist ein Tupel, welches ihren *Namen*, die *Vorbedingung* und eine Menge von *Effekten* enthält:

$$a(\vec{x}) := \langle name,\ pre(\vec{x}),\ effects(\vec{x}) \rangle$$

wobei *name* ein String ist, $pre(\vec{x}) = \bigwedge_i p_i(\vec{x})$, $p_i(\vec{x}) \in L$ eine Formel, und $effects(\vec{x})$ eine Menge *bedingter Effekte*. Diese sind dabei allquantifiziert über die freien Variablen der Effektliterale; die Effektvorbedingungen sind eine Konjunktion von Literalen. Sei $fr(literal)$ die Menge der freien Variablen eines Literals $literal \in L$.

$$effects(\vec{x}) = \left\{ ef_i(\vec{x}) \mid ef_i(\vec{x}) = \forall \vec{x}_i : \left\langle \exists \vec{y}_i : \underbrace{\bigwedge_j p_{ij}(\vec{x}, \vec{y})}_{=:p(\vec{x},\vec{y})} \triangleright l_i(\vec{x}) \right\rangle,\ i = 1, \ldots, k \right\},$$

mit $p_{ij}(\vec{y}), l_i(\vec{x}) \in L$, $fr(l_i(\vec{x})) = \{x_1, \ldots, x_k\} := \vec{x}$ und
$fr(p_{ij}(\vec{x}, \vec{y})) \setminus fr(l_i(\vec{x})) = \{x_{k+1}, \ldots, x_{k+m}\} =: \vec{y}$.
Sei A die Menge aller Aktionen. Dann ist die Menge aller Grundinstanzen von Aktionen definiert als $GrundInst(A) := \{a \in A \mid a \text{ grund}\}$. Ein Element $a \in GrundInst(A)$ wird dann als *Grundaktion* bezeichnet.

Definition 22 (PDDL-Spezifikation, Planungsaufgabe)**.** Eine PDDL-*Spezifikation* ist die *Planungsaufgabe* $\Pi = (AF, A, s_0, G)$, wobei AF die Menge der atomaren Formeln ist und A eine Menge von *Aktionen*. Weiterhin seien $x_1, x_2, \ldots$ eine abzählbare Menge von Variablen.
Sei GAF die Menge der Grundinstanzen der atomaren Formeln aus AF. Der *Startzustand* ist durch $s_0 \subseteq GAF$ gegeben, als eine Menge grundatomarer Formeln. Selbiges gilt für die Menge der Zielzustände $G \subseteq 2^{GAF}$.

Oben wurden die Successor State Axioms erwähnt; im Zusammenhang mit PDDL müssen nun die *State Update Axioms* erläutert werden. Diese sind eine ähnliche Form von Axiomen, welche auch das „inferential problem“ lösen. In PDDL entsprechen sie gerade den allquantifizierten bedingten Effekten von Aktionen. Folgezustände werden in PDDL dadurch bestimmt, dass *zusätzlich* zu den im vorherigen Zustand geltenden Prädikaten die neu wahr gemachten hinzukommen sowie diejenigen, die falsch gemacht wurden, entfernt werden. Der Zustand wird also nur aktualisiert, daher „update“ Axiome, während hingegen bei den vorher beschriebenen Successor State Axioms jeder Folgezustand komplett neu berechnet werden muss. Auf die Weise wird das von Thielscher in [13] beschriebene Frameproblem von Successor Update Axioms komplett gelöst, allerdings ist die formale Darstellung wesentlich komplexer. Die erhebliche Reduzierung der Anzahl von Axiomen ist einer der Hauptgründe, diese Transformation umzusetzen und das Ergebnis mit Hilfe des FastDownward-Planers zu testen.

4.2.2 Semantik

Analog zur Semantik von GDL gilt auch hier, dass $\mathcal{L}_{PDDL}$ die Sprache über dem Vokabular der PDDL-Spezifikation ist, und Modelle dieser Sprache eine Menge grundatomarer Sätze sind. Die Modelle und Erfüllbarkeitsbedingungen sind hierbei wie allgemein in der Logik der ersten Stufe definiert, als Referenz siehe zum Beispiel [3]. Einzig die Modellbeziehung für Exisitenzquantoren muss zusätzlich definiert werden:

Definition 23 (Erfüllbarkeit existenzquantifizierter Formel)**.** Sei $s \in S$ ein Zustand. Dann gilt: $s \models \exists \vec{x} : \varphi(\vec{x})$ genau dann, wenn Grundterme $\vec{t} \subseteq T$ existieren, so dass $s \models \varphi[\vec{t}/\vec{x}]$.

Das *Transitionssystem* als Semantik einer solchen Planungsaufgabe wird ebenfalls als induzierter Graph bzw. Automat dargestellt, der visuell den Zustandsraum verdeutlicht. Ebenso bedarf es wieder einiger Definitionen, bevor das Transitionssystem in einer kompletten Übersicht dargestellt werden kann.

Definition 24 (Zustand)**.** Die *Zustandsmenge* S ist analog zu der in GDL definiert, $S := 2^{GAF}$.

Definition 25 (Anwendbare Aktionen, Anwendung einer Aktion)**.** Sei $s \in S$ ein Modell für $\mathcal{L}_{PDDL}$. Die Menge $A(s)$ der im Zustand s *anwendbaren Aktionen* ist definiert als:

$$A(s) := \{\langle name,\ pre,\ effects\rangle \in GrundInst(A) \mid s \models pre\}$$

Die *Anwendung einer Aktion* $a(\vec{t}) = \left\langle name,\ pre(\vec{t}),\ effects(\vec{t})\right\rangle$, $\vec{t}$ Terme, im Zustand s sei $applyPDDL(a(\vec{t}), s)$; ihre Ausführung ergibt einen Folgezustand s'.

Die Berechnung eines Folgezustandes geschieht folgendermaßen: Es müssen sowohl die sogenannten Add- als auch die Delete-Effekte der Aktion berechnet werden. Dabei werden den bereits geltenden Prädikaten des alten Zustands neue hinzugefügt (Add) oder welche entfernt bzw. falsch gemacht (Delete).

Definition 26 (Folgezustand im Transitionssytem, Transitionslabel)**.** Sei s ein Zustand und $a(\vec{t}) = \left\langle name,\ pre(\vec{t}),\ effects(\vec{t}) \right\rangle \in A(s)$ eine in diesem Zustand anwendbare Aktion. Dann ist der Folgezustand s' wie folgt definiert:

$$s' := applyPDDL(a(\vec{t}), s) = \Big(s \setminus DEL(a(\vec{t}), s)\Big) \cup ADD(a(\vec{t}), s), \text{ wobei}$$

$$DEL(a(\vec{t}), s) := \bigcup_{ef(\vec{t}) \in effects} DEL(ef(\vec{t}), s) \text{ und } ADD(a(\vec{t}), s) := \bigcup_{ef(\vec{t}) \in effects} ADD(ef(\vec{t}), s)$$

$$DEL(ef(\vec{t}), s) := \Big\{ atom[\vec{t'}/\vec{x}] \mid ef(\vec{t}) = \forall \vec{x} : (\exists \vec{y} : \bigwedge_i p_i(\vec{x}, \vec{y}) \triangleright \neg atom(\vec{x})) \text{ und } s \models \exists \vec{y} \bigwedge_j p_j[\vec{t'}/\vec{x}, \vec{y}],\ \vec{t'} \text{ grund} \Big\}$$

$$ADD(ef(\vec{t}), s) := \Big\{ atom[\vec{t'}/\vec{x}] \mid ef(\vec{t}) = \forall \vec{x} : (\exists \vec{y} : \bigwedge_i p_i(\vec{x}, \vec{y}) \triangleright atom(\vec{x})) \text{ und } s \models \exists \vec{y} \bigwedge_j p_j[\vec{t'}/\vec{x}, \vec{y}],\ \vec{t'} \text{ grund} \Big\}$$

Die jeweilige Kante im Transitionssystem wird dann gerade mit dem Namen *name* der Aktion beschriftet.

Das Transitionssystem lässt sich insgesamt also durch Beschreibung folgender Komponenten zusammenfassen:

- S: Menge von Zuständen, wie oben definiert
- A: Menge von Aktionen, wie oben definiert
- $A(s) \subseteq A$: Menge der anwendbaren Aktionen in einem Zustand
- $applyPDDL : A \times S \to S$: Transitionsfunktion, die mit einer gewählten Aktion aus einem Zustand den Nachfolgezustand berechnet
- $s_0 \in S$: Initialzustand
- $G \subseteq S$: Zielzustände

Ein Planungsproblem ist also auf einen Startzustand s_0 initialisiert. Danach kann eine anwendbare Aktion benutzt werden, um mit Hilfe ihrer Effekte die Änderungen, die er auf dem Zustand bewirkt, und damit den Folgezustand s', zu berechnen. Dieses lässt sich beliebig oft wiederholen, bis man in einem Zielzustand ist. Eine Sequenz von n Aktionen $a_1, \ldots, a_n$, die der Reihe nach von s_0 ausgehend einen Zielzustand s_n erzeugen, wird als *Plan* der Länge n bezeichnet. Man beachte hier die Paralellen zu allgemeinen Ein-Personen-Spielen, wie sie oben beschrieben wurden.

4.3 Transformation

Im Folgenden wird die Transformation von GDL nach PDDL beschrieben und formal definiert sowie im Anschluss ihre Korrektheit bewiesen.
Gegeben ist also eine GDL-Spezifikation $\Gamma = (AF^1, R, s_0^1, \mathcal{T})$ basierend auf einer GDL-Signatur $(V_1, F_1, P_1, \alpha_1)$ eines Ein-Personen-Spiels, welche in eine PDDL-Spezifikation $\Pi = \left(AF^2, A, s_0^2, G\right)$ basierend auf einer PDDL-Signatur $(V_2, F_2, P_2, \alpha_2)$ transformiert werden soll.

Es wird zunächst die Transformation $\mathit{compileAction}(a(\vec{x}))$ (kurz $cA(a(\vec{x}))$) der Transitionsfunktion von GDL nach PDDL beschrieben. Die Menge der GDL-Aktionen sei $\mathit{Actions}_\Gamma$ wie im obigen Abschnitt definiert.
Sei $G = \{r_1, \ldots, r_n\}$ die Menge der Datalogregeln der GDL-Spezifikation, wobei die Regeln die Form

$$r_i \equiv \forall \vec{x} : \left(h_{r_i}(\vec{x}) \leftarrow \exists \vec{y} : \underbrace{\bigwedge_j b^j_{r_i}(\vec{x}, \vec{y})}_{b_{r_i}(\vec{x}, \vec{y})} \right), \; h_{r_i}(\vec{x}), b^j_{r_i}(\vec{x}, \vec{y}) \in L_{GDL}$$

haben, wobei $\vec{x}$ und $\vec{y}$ gerade die freien Variablen des Kopfes und die verbleibenden freien Variablen des Rumpfes sind.
In einem ersten Schritt werden alle Regeln, deren Kopfatom das Prädikat *NEXT* beinhaltet, oBdA in der Form $(\mathit{NEXT}\ \varphi(\vec{v})) =: \varphi'(\vec{v})$, und welche in ihrem Rumpf ein Literal mit *DOES* enthalten, oBdA in der Form $(\mathit{DOES}\ a(\vec{z})) =: \delta(a(\vec{z}))$, bezüglich dieses Kopfatoms und der dazugehörigen Aktion sortiert. Sollte ein Regel mit *NEXT* im Kopf kein *DOES* in ihrem Rumpf enthalten, so wird sie bezüglich ihres Kopfes und einer nichts-tun-Aktion *noop* sortiert.

$$N(\varphi, a) := \{r_i \in R \mid h_{r_i}(\vec{v}) = \varphi'(\vec{v}) \text{ und } \exists j = 1, \ldots, n_i : b^j_{r_i}(\vec{v}, \vec{z}) = \delta(a(\vec{z}))\}$$

Weiterhin müssen sowohl die allquantifizierten wie auch die existenzquantifizierten Variablen mitgeführt werden:

$$\vec{x} := \{fr(h_{r_i}(\vec{x})) \mid r_i \in N(\varphi, a)\}$$

$$\vec{y} := \{fr(b_{r_i}(\vec{x}, \vec{y})) \setminus fr(h_{r_i}(\vec{x})) \mid r_i \in N(\varphi, a)\}$$

Sei $\sigma(\varphi, a) := mgu\{h_{r_i} \mid r_i \in N(\varphi, a)\}$ der allgemeinste Unifikator (most general unifier) für alle Köpfe der Regeln mit demselben Prädikat im Kopf sowie der gleichen Aktion im Rumpf. Die Substitution wird zunächst auf die beiden Mengen von quantifizierten Variablen angewendet werden:

$$\vec{x}' := x[\sigma(\varphi, a)] \text{ und } \vec{y}' := y[\sigma(\varphi, a)]$$

Weiterhin ist dann

$$h_{\varphi,a}(\vec{x}') := h_{r_i}(\vec{x})[\sigma(\varphi, a)], \text{ mit } r_i \in N(\varphi, a)$$

der substituierte Kopf der Regeln r_i in der „Kategorie“ φ, a, deren freien Variablen gerade die ebenfalls substituierte Menge der vorherigen freien Variablen ist.
Danach können alle Regeln derselben Kategorie in einer großen kombinierten Regel K^+ zusammengefasst werden, wobei auch die (zusammengefassten, da substituierten) quantifizierten Variablen mit hinzugenommen werden. Dazu wird zunächst der allgemeinste Unifikator auch noch auf alle Rümpfe dieser Regeln angewendet und die Rümpfe der Regeln in einer Disjunktion vereint:

$$b_{\varphi,a}(\vec{x}', \vec{y}') := \bigvee_{r_i \in N(\varphi,a)} b_{r_i}(\vec{x}, \vec{y})[\sigma(\varphi, a)])$$

Dann ist die kombinierte Regel definiert als:

$$K^+_{\varphi,a}(\vec{x}', \vec{y}') := \forall \vec{x}' : \left(h_{\varphi,a}(\vec{x}') \leftrightarrow \exists \vec{y}' : \underbrace{b_{\varphi,a}(\vec{x}', \vec{y}')}_{=:p^+_{\varphi,a}(\vec{x}', \vec{y}')} \right)$$

Nachdem dies für alle sortierten Regeln (Kombinationen von φ, a) getan wurde, werden alle $K^+_{\varphi,a}(\vec{x}', \vec{y}')$ negiert, und zwar derart, dass wenn der negierte Rumpf gilt, die negierte Aussage der Regel folgt, der Kopf. Die entstehende Regelmenge $K^-_{\varphi,a}(\vec{x}', \vec{y}')$ ist wie folgt definiert:

$$K^-_{\varphi,a}(\vec{x}', \vec{y}') := \forall \vec{x}' : \left(\neg h_{\varphi,a}(\vec{x}') \leftrightarrow \forall \vec{y}' : \underbrace{\neg b_{\varphi,a}(\vec{x}', \vec{y}')}_{=:p^-_{\varphi,a}(\vec{x}', \vec{y}')} \right)$$

Nun hat man also für jedes Prädikat, welches in einer *NEXT*-Regel vorkommt, und die zugehörige Aktion einen großen Rumpf von Bedingungen, die gelten müssen, damit das Prädikat auch tatsächlich im nächsten Zustand wahr wird, sollte die Aktion gewählt werden. Aus der gesamten Menge der kombinierten Regeln, also für alle $K^+_{\varphi,a}(\vec{x}', \vec{y}')$, sollen jetzt die PDDL-Aktionen konstruiert werden. Dabei wird die Kompilierung $cA(a)$ wie von Röger, Helmert und Nebel [11] beschrieben umgesetzt, um die PDDL-Aktionen $\langle name,\ pre,\ effects \rangle$ zu erzeugen.
Als Name wird der Name der jeweiligen GDL-Aktion verwendet, $name := a$.
Um die Vorbedingung für die PDDL-Aktion zu bestimmen, werden zunächst alle Regeln, deren Kopfatom das Prädikat $(LEGAL\ a(\vec{v}))$ beinhaltet, in einer Menge gesammelt:

$$leg(a) := \{r_i \in R \mid h_{r_i}(\vec{v}) = (LEGAL,\ a(\vec{v}))\}$$

Nach anschließender Substitution mit $\sigma(a) := mgu\{h_{r_i} \mid r_i \in leg(a)\}$, analog zur oben beschriebenen Vorgehensweise, gibt es dann eine neue kombinierte Regel, von welcher nur der Rumpf $b_{leg}(\vec{v}', \vec{z}') := \bigvee_{r_i \in leg(a)}(TRUE\ b_{r_i}(\vec{v}', \vec{z}'))$ von Interesse ist. Es gilt dann insbesondere, dass $(LEGAL\ a(\vec{v}'))$ genau dann gilt, wenn $b_{leg}(\vec{v}', \vec{z}')$ gilt. Als Vorbedingung *pre* für die neue PDDL-Aktion wird dann diese Formel des Rumpfes, ohne die *TRUE*-Prädikate, eingesetzt: $pre := \bigvee_{r_i \in leg(a)} b_{r_i}(\vec{v}', \vec{z}')$.
Um die Effektmenge zu bilden, werden zunächst paarweise Effekte konstruiert. Für jede

Regel $K^+_{\varphi,a'}(\vec{x}\,', \vec{y}\,')$ mit $a' = a$ – also die kombinierten Regeln, die der Aktion a zugeordnet sind – wird ihr Rumpf $p^+_{\varphi,a'}(\vec{x}\,', \vec{y}\,')$ als Effektbedingung und ihr Kopf $h_{\varphi,a'}(\vec{x}\,', \vec{y}\,')$ als tatsächlicher Effekt verwendet. Gleichzeitig dazu wird für diese Regel auch ihr Pendant $K^-(\varphi, a')$ auf analoge Weise verwendet, um die negativen Effekte hinzuzufügen. Diese Paar bildet zusammen dann einen allquantifizierten Effekt:

$$\mathit{effects}(\varphi, a') := \{ef^{a'}_{\varphi} \mid K^+_{\varphi,a'}(\vec{x}\,', \vec{y}\,') \neq \emptyset\} \cup \{ef^{a'}_{\neg\varphi} \mid K^-_{\varphi,a'}(\vec{x}\,', \vec{y}\,') \neq \emptyset\}$$

$$ef^{a'}_{\varphi} := \forall \vec{x}\,' : \left(\exists \vec{y}\,' : p^+_{\varphi,a'}(\vec{x}\,', \vec{y}\,') \triangleright h_{\varphi,a'}(\vec{x}\,', \vec{y}\,')\right)$$

$$ef^{a'}_{\neg\varphi} := \forall \vec{x}\,' : \left(\forall \vec{y}\,' : p^-_{\varphi,a'}(\vec{x}\,', \vec{y}\,') \triangleright \neg h_{\varphi,a'}(\vec{x}\,', \vec{y}\,')\right)$$

Die Effektmenge einer PDDL-Aktion besteht dann mindestens aus einem solchen Paar von bedingten Add- und Delete-Effekten:

$$\mathit{effects} := \{\mathit{effects}(\varphi, a') \mid \varphi \in AF \text{ und } a \in \mathit{Actions}_\Gamma\}$$

Die Kompilierung einer GDL-Aktion a ist dann das Ergebnis $a' := cA(a)$ auf die soeben beschriebene Weise. Weiterhin sei $cA(A) := \{cA(a) \mid a \in \mathit{Actions}_\Gamma\}$
Das ganze Prozedere muss für jede Aktion durchgeführt werden. Die konstruierten Aktionen bilden dann die Aktionsmenge A der entstehenden PDDL-Spezifikation.

Sei S_1 die Zustandsmenge des Transitionssystems der GDL-Spezifikation Γ. Ein Zustand eines GDL-Spiels hängt von den gültigen Prädikaten ab, die mit *TRUE* in diesem Zustand (einer Menge von atomaren Formeln) erscheinen. Insbesondere spielen die restlichen vordefinierten Prädikate keine Rolle. In PDDL ist ein Zustand ebenfalls eine Teilmenge der atomaren Formeln, nur dass hier keine vordefinierten Prädikate wie *TRUE* auftreten. Eine naheliegende Transformation für Zustände ist also, gerade die Menge der wahren Prädikate aus GDL zu übernehmen. Da weiterhin die Menge der atomaren Formeln AF ebenfalls noch transformiert werden muss und Zustände Teilmengen dieser Menge sind, wird eine allgemeine Kompilierung definiert, die sowohl für Zustände als auch die gesamte Menge der atomaren Formeln definiert ist. Diese Transformationsabbildung für Zustände ist eine Funktion, die alle geltenden Atome (mit *TRUE* davor) in einer Menge zusammenfasst und somit einen PDDL-Zustand beschreibt:

$$\mathit{compileStateAtoms}(s) := cSA(s) = \{\varphi(\vec{x}) \mid (\mathit{TRUE}\ \varphi(\vec{x})) \in s\}, s \subseteq AF$$

Insbesondere kann für s also auch AF eingesetzt werden. Außerdem sei $cSA(S_1) := \{cS(s) \mid s \in S_1\}$.

Die Transformation der Rewardfunktion, die in GDL-Spezifikationen stest auch implizit vorhanden ist, ist trivial, sie wird einfach übernommen.

Bis auf syntaktische Ergänzungen, damit korrektes PDDL entsteht, ist somit eine Transformation Φ einer GDL-Spezifikation $\Gamma = (AF^1, R, s^1_0, T)$ zu einer PDDL-Spezifizierung $\Pi = \left(AF^2, A, s^2_0, G\right)$ definiert. Die Aktionsmenge $\mathit{Actions}_\Gamma$ der GDL-Aktionen sei wie

im Abschnitt 4.1.2 über Semantik von GDL durch die Regelmenge R definiert. Die vollständige Transformation Φ ist dann wie folgt definiert:

$$\Phi(\Gamma) := \Big(cSA(AF^1), cA(Actions_\Gamma), cSA(s_0^1), cSA(\mathcal{T})\Big) =: \Pi$$

Es bleibt die Korrektheit zu beweisen.

Satz 1 (Korrektheit der Transformation Φ)**.** *Sei eine GDL-Spezifikation* Γ *wie oben gegeben. Dann gilt:*

$$cSA(applyGDL(a_1(\vec{x}), s_1)) = applyPDDL(cA(a_1(\vec{x})), cSA(s_1))$$

Laut Behauptung spielt es also keine Rolle, ob zunächst der Folgezustand innerhalb des Transitionssystems berechnet und danach die Transformation angewendet wird, oder ob man erst die Transformation berechnet, danach die transformierte Aktion auf das Ergebnis anwendet. In beiden Fällen erhält man denselben Zustand. Insbesondere ist dann die Transformation korrekt.
Anschauchlich dargestellt ist der Sachverhalt in Abbildung 1.

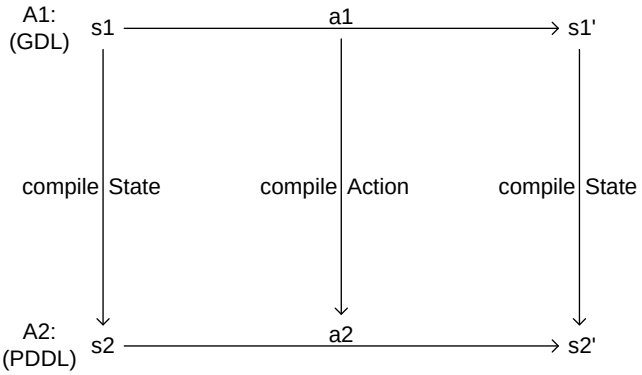


Abbildung 1: Transformation

Beweis. Sei s_1 ein Zustand im GDL-Transitionssystem $Tr(\Gamma)$ und sei $a_1(\vec{t}) \in Actions_\Gamma$ eine Grundaktion der GDL-Spezifikation. Zunächst wird gezeigt, dass $a_1(\vec{t})$ in s_1 genau dann anwendbar ist, wenn $cA(a_1)$ in $cSA(s_1)$ anwendbar ist:
Sei $a_1(\vec{t})$ in s_1 anwendbar, also $s_1 \models (LEGAL\ a_1(\vec{t}))$. Während der Konstruktion von $cA(a_1)$ wird als Vorbedingung $pre := \bigvee_{r_i \in leg(a)} b_{r_i}(\vec{t}, \vec{t'})$ verwendet. Diese Formel gilt genau dann wenn $(LEGAL\ a_1(\vec{t}))$ gilt, also $a_1(\vec{t})$ in s_1 anwendbar ist. Es gilt also $s_1 \models (LEGAL\, a_1\vec{t})$ genau dann wenn $cSA(s_1) \models pre$, da weiterhin bei der Zustandstransformation lediglich die $TRUE$-Prädikate wegfallen. $cSA(s_1) \models pre$ stellt aber gerade die Bedingung zur Anwendung von $cA(a_1)$ in $cS(s_1)$ dar.
Da überall genau-dann-wenn-Beziehungen gelten und die Transformation des entfernen

von *TRUE* vor jedem Atom auch rückwärts genauso funktioniert (also das Hinzufügen von *TRUE*), ist die Rückrichtung bzw. der Fall, dass a_1 in s_1 nicht anwendbar ist, schon gezeigt.
Es folgt sofort, dass a_1 in s_1 anwendbar ist, genau dann wenn $cA(a_1)$ in $cS(s_1)$ anwendbar ist.

Nun muss noch bewiesen werden, dass die entstehenden Folgezustände sich auch mittels Transformation ineinander überführen lassen. Es ist also zu zeigen, dass für jedes Grundatom $\varphi(\vec{t}) := \varphi(t_1, \ldots, t_n)$, t_i Grundterme, gilt:

$$\begin{aligned} &\mathit{applyGDL}(a(\vec{u}), s) \models (\mathit{TRUE}\ \varphi(\vec{t})),\ \vec{u} \text{ Grundterme, genau dann, wenn} \\ &\mathit{applyPDDL}(cA(a(\vec{u})), cS(s)) \models cS((\mathit{TRUE}\ \varphi(\vec{t}))). \end{aligned}$$

Man betrachte das Atom $\varphi(\vec{t})$. Dieses kann in s gelten oder nicht gelten. Für den Moment spielt dies keine Rolle, es sei nur angenommen, dass es eine GDL-Regel gibt, welche dieses Atom in ihrem Kopf enthält, in der Form $(\mathit{NEXT}\ \varphi(\vec{x}))$. Sollte es keine solche Regel geben, so wird das Atom $\varphi(\vec{t})$ nach Anwendung der Aktion a im Folgezustand entweder falsch oder wahr bleiben, je nach Ausgangszustand. Insbesondere wird dann auch die Kompilierung des Zustandes diese Tatsache beibehalten, weshalb sofort die Behauptung folgt.
Angenommen, eine solche GDL-Regel existiert. Dann hat sie die Form

$$r = \forall \vec{x} : \left(h(\vec{x}) \leftarrow \exists \vec{y} : \underbrace{\bigwedge_i b_i(\vec{x}, \vec{y})}_{b} \right)$$

und es muss eine Substitution σ geben, so dass $h(\vec{x})\sigma = (\mathit{NEXT}\ \varphi(\vec{t}))$ und $b_i(\vec{x}, \vec{y})\sigma = (\mathit{DOES}\ a(\vec{u}))$ für ein $i = 1, \ldots, k$. Diese Regel r wird dann in die entsprechende Gruppe von Regeln „einsortiert“: $r \in N(\varphi, a)$ und mit eventuell anderen vorhandenen zu einer großen Regel $K^+_{\varphi,a}(\vec{x}, \vec{y})$ kombiniert, sowie die negierte Version $K^-_{\varphi,a}(\vec{x}, \vec{y})$ gebildet. Die konstruierte PDDL-Aktion $cA(a)$ hat dann ein Paar bedingter Effekte, wie in der Transformation beschrieben:

$$\mathit{effects}(\varphi, a) := \{ef^a_\varphi \mid K^+_{\varphi,a}(\vec{x}, \vec{y}) \neq \emptyset\} \cup \{ef^a_{\neg\varphi} \mid K^-_{\varphi,a}(\vec{x}, \vec{y}) \neq \emptyset\}$$

$$ef^a_\varphi := \forall \vec{x} : \left(\exists \vec{y} : p^+_{\varphi,a}(\vec{x}, \vec{y}) \triangleright h_{\varphi,a}(\vec{x}, \vec{y})\right)$$

$$ef^a_{\neg\varphi} := \forall \vec{x} : \left(\forall \vec{y} : p^-_{\varphi,a}(\vec{x}, \vec{y}) \triangleright \neg h_{\varphi,a}(\vec{x}, \vec{y})\right)$$

wobei $p^+_{\varphi,a}(\vec{x}, \vec{y})$ bzw. $p^-_{\varphi,a}(\vec{x}, \vec{y})$ gerade dem Rumpf bzw. dem negierten Rumpf der kombinierten Regel entspricht, welche r enthält. Es gilt also im Allgemeinen $s \models p^+_{\varphi,a}(\vec{x}, \vec{y}) \vee p^-_{\varphi,a}(\vec{x}, \vec{y})$.

Unter der Annahme $applyGDL(a(\vec{x}), s) \models (TRUE\ \phi(\vec{t}))$ muss s gerade den Rumpf der positiven Regel r erfüllt haben, also muss s auch Modell des Rumpfes der kombinierten Regel $K^+_{\varphi,a}(\vec{x}, \vec{y})$ sein, da der Rumpf der Regel r dort als ein Disjunktionsglied vorkommt: $s \models p^+_{\varphi,a}(\vec{x}, \vec{y})$. Weiterhin ist nach obigem Abschnitt $cA(a)$ in $cSA(s)$ anwendbar, wenn a in s anwendbar ist. Das heißt, es wird insbesondere der Add-Effekt tatsächlich ausgeführt und es folgt, dass $applyPDDL(cA(a(\vec{x})), cSA(s)) \models cS((TRUE\ \varphi(\vec{t})))$.

Sollte $applyGDL(a(\vec{x}), s) \not\models (TRUE\ \phi(\vec{t}))$ gelten, so muss s gerade den Rumpf der negierten Regel r erfüllt haben, welche als Disjunktionsglied im Rumpf von $K^-_{\varphi,a}(\vec{x}, \vec{y})$ vorkommt. Es gilt also $s \models p^-_{\varphi,a}(\vec{x}, \vec{y})$. Nach wie vor ist $cA(a)$ in $cSA(s)$ anwendbar, was zur Folge hat, dass der Delete-Effekt tatsächlich zur Geltung kommt. Es folgt $applyPDDL(cA(a(\vec{x})), cSA(s)) \not\models cS((TRUE\ \varphi(\vec{t})))$. □

5 Modifizierung des FastDownward-Planers

Nachdem im vorangehenden Abschnitt eine Transformation von GDL nach PDDL beschrieben wurde, kann nun ein Planungssystem (mit leichten Modifizierungen) zur Lösung verwendet werden. Die Wahl fiel für diese Arbeit auf den FastDownward-Planer. Dieses zweite Hauptkapitel der Bachelorarbeit beschreibt nun die Funktionsweise des FastDownward-Planers und die Modifizierung, die es ermöglicht, mit diesem Planungssystem allgemeine Ein-Personen-Spiele zu lösen. Weiterhin ist es das Ziel dieser Modifizierung, das Lösen von Ein-Personen-Spiele in einer Art Anytime-Umgebung, ähnlich der des GGP-Wettbewerbs (Abfrage des nächsten Zuges nach kurzen Zeitintervallen) zu simulieren, um den Planer später auf einfache Art auch für Mehrpersonenspiele erweitern zu können. Im Zuge dieser Modifizierung wird eine iterative „Tiefenbestensuche“ in Kombination mit der JO-Berechnungsfunktion [1] verwendet.

5.1 FastDownward-Planer

FastDownward wurde von Malte Helmert an der Uni Freiburg entwickelt [8] und ist in C++ implementiert. Es ist ein klassisches, auf heuristischer Suche basierendes Planungssystem. Es arbeitet mit allgemeinen Planungsproblemen, kodiert in propositionalem PDDL2.2, inklusive erweiterter ADL-Features wie bedingte Effekte und abgeleitete Prädiakte (Axiome). FastDownward ist wie auch viele andere bekannte Planungssysteme ein progressiver Planer, das heißt die Suche im Zustandsraum geschieht in Vorwärtsrichtung. Ein Unterschied zu den meisten anderen Planungssystemen besteht allerdings in der Repräsentation von Planungsaufgaben, die hier nämlich nicht mehr in PDDL gehalten wird, sondern in einem ersten Schritt (dem *Translate*-Teil) in eine andere, mehrwertige Repräsentation transformiert wird. In dieser Darstellung werden viele der impliziten Constraints explizit dargestellt, um danach eine hierarchische Struktur der Planungsaufgabe zu bilden. Diese wird dann genutzt, um einen Kausalgraphen zu generieren, mit dessen Hilfe die Kausalgraph-Heuristik arbeitet, das Kernstück des Suchalgorithmus des FastDownward-Planers. Als Suchalgorithmus ist in FastDownward eine Bestensuche implementiert, die viele weitere Techniken zur Verbesserung benutzt, wie zum Beispiel *Preferred Operators* (hilfreiche Aktionen werden zunächst in einer lokalen Suche verwendet) und verzögerte Heuristikauswertung (schwächt den Effekt von hohen Verzweigungsgraden der Suche ab). Eine von vielen weiteren Optionen ist eine Multiheuristiksuche, die mehrere Heuristiken kombiniert.

FastDownward als sehr umfangreiches und komplexes Planungssystem bringt viele Strukturen bereits mit sich, auf denen aufgebaut werden kann. So gibt es ein Modul für Heuristiken, welches es erlaubt, dem Planungssystem auf einfache Art neue Heuristiken zur Verfügung zu stellen und zu verwenden. Außerdem stehen bereits einige vorhandene Heuristiken zur Verfügung, wie zum Beispiel die Kausalgraph-Heuristik, die Blind-Search-Heuristik, die Goal-Count-Heuristik oder auch die FastForward-Heuristik. Außerdem stehen sehr effiziente Implementierungen von Open- und ClosedLists zur Verfügung. Die OpenList ist als eine „Double Ended Queue“ (deque in C++) implementiert. Die Clo-

sedList ist eine Map, welche einem Zustand seinen „erzeugenden Operator“ zuordnet, also den Operator, der diesen Zustand aus einem Vorgänger generiert hat. Weiterhin gibt es ein Modul Searchengine, das den Suchalgorithmus als solchen übernimmt, ohne die Detailschritte zu implementieren. Auf diese Art lässt sich hier schnell eine modifizierte Version einer Suche einfügen. Zuletzt verwendet FastDownward eine spezielle, effiziente Art der Zustandsexpandierung: Zunächst werden die nicht anwendbaren Operatoren aussortiert, und danach der Zustand derart expandiert, dass beim Einfügen in die OpenList der neue Zustand generiert wird und mitsamt seinem Vorgänger und dem erzeugenden Operator sowie relevanten Suchinformationen wie Heuristikwert oder Tiefe im Suchbaum gespeichert. Weitere detaillierte Informationen über FastDownward findet man in [8].

5.2 Anytime-Aspekt und Iterative Tiefensuche

Aufgrund der Anpassung der PDDL-Kodierung an Spiele und der damit verbundenen Erweiterung auf mehrere Ziele mit unterschiedlichen Nutzenwerten musste zunächst das Einlesen angepasst und einige globale Datenstrukturen angepasst werden. Ansonsten konnten meistens die vorhanden Strukturen, eventuell mit durch die Unterschiede zwischen Ein-Personen-Spielen und Planungsproblemen bedingten Modifzierungen versehen, verwendet werden.
Zunächst galt es natürlich zu berücksichtigen, dass GDL-Spiele, vor allem im Bezug auf den Wettbewerb, auf eine andere Weise angegangen werden müssen als klassische Planungsprobleme. Eine große Änderung ist hier vor allem die Tatsache, dass die Lösung nicht als Komplettlösung nach einer eher längeren Zeit gewünscht wird, sondern schrittweise in relativ kurzen Zeitintervallen die jeweils besten Spielzüge ermittelt werden sollen, damit diese dann in einem Wettbewerb gegen andere Spieler gespielt werden können und somit der nächste Zustand überhaupt erst berechnet werden kann. Es geht also nicht darum, einen fertigen Plan zu ermitteln, sondern stets flexibel auf neue Situationen reagieren zu können. Der Spieler sollte insbesondere zu bestimmten, meistens recht kurzen Zeitintervallen bereit sein, seinen nächsten Spielzug an den GM übermitteln zu können, und in seinem eigenen Interesse sollte dies natürlich ein für ihn möglichst guter sein. Dieser Anforderung wurde im Rahmen dieser Arbeit mit dem Anytime-Aspekt begegnet, der es ermöglicht, zu jedem beliebigen Zeitpunkt den nächsten Spielzug abzufragen. Dies erleichtert den Ablauf des Suchalgorithmus dahingehend, dass man nicht permament kontrollieren muss, wie viel Zeit bis zur nächsten Abfrage noch verbleibt.

Um diesen Anytime-Aspekt umzusetzen, wird in der gewählten Implementierung eine iterative Tiefensuche mit einer Bestensuche kombiniert, genannt *Tiefenbestensuche*. Grob gesagt wird bis zu einer gewissen Tiefe zunächst mit Bestensuche gesucht, sobald diese aber an der aktuellen Suchtiefe angelangt ist, werden auch die restlichen Teilbereiche des Suchbaumes durchsucht, bis die gesamte aktuelle Tiefe durchsucht wurde. Danach startet die Prozedur von vorne, mit einem erhöhten Suchmaximum.
Diese Tiefenbestensuche wurde komplett neu implementiert. Um eine Bestensuche verwenden zu können, benötigt man Informationen über die Güte der Knoten in der Open-

List, um den besten zuerst expandieren zu können. Über die Art der Berechnung und die Verwendung von Heuristiken wird im nächsten Abschnitt berichtet; für den Moment sei eine Bewertungsfunktion für Suchzustände gegeben. Zunächst wird hier der Tiefensuchcharakter des Algorithmus vorgestellt.

```
IterativeDeepeningSearch(Node)
{
  IterationDepth := 1;
  while (IterationDepth < infinity)
  {
    DepthLimitedBestFirstSearch(Node, IterationDepth);
    IterationDepth := IterationDepth + 1;
  }
}
```

Die iterative Tiefenbestensuche startet mit einer Suchtiefe von 1 und erhöht diese schrittweise um 1, so lange die auf die aktuelle Tiefe beschränkte Bestensuche noch kein Ergebnis erzielt hat. Letztere wurde folgendermaßen implementiert:

```
DepthLimitedBestFirstSearch(Node, IterationDepth)
{
  ActualDepth := 0;
  Searchstep();
  if (DepthExplored) {stop DepthLimitedSearch};
  if (SpaceExplored) {stop IterativeDeepeningSearch};
}
```

Hierbei überprüft `Searchstep()` zunächst, ob der aktuelle Knoten bereits expandiert wurde. Ist dies nicht der Fall, so wird die Suchtiefenvariable inkrementiert und der Knoten expandiert, wobei alle Folgezustände in die OpenList eingefügt werden und dabei mit Hilfe des gegebenen Bewertungskriteriums der beste Knoten bestimmt wird, der als nächstes betrachtet werden soll. Dabei wird auch gegebenenfalls der Operator gespeichert, den man im Startzustand der aktuellen Suche anwenden muss, um in den richtigen Teilbaum zu gelangen. Da die tiefenbeschränkte Bestensuche mit Tiefe 1 beginnt, ist garantiert, dass unverzüglich nach Beginn der Suche zumindest ein momentan bester nächster Operator bestimmt wird. Dieser kann sich im Verlauf der Bestensuche mit höherer Tiefe noch ändern und dabei das erwartete Ergebnis verbessern. Sollte die tiefenbeschränkte Bestensuche die komplette Tiefe durchlaufen haben, wird sie beendet und eine neue Bestensuche, mit höherer Suchtiefe, wird gestartet. Sollte in einem Suchdurchlauf gar der ganze Zustandsraum durchsucht worden sein, so kann die iterative Tiefensuche beendet werden, da in diesem Fall der optimale Spielzug bekannt ist. Sollte der GM schon vorher den nächsten Spielzug übermittelt bekommen wollen, so wird die aktuelle Suche unterbrochen und nach Übermittlung des neuen Spielstandes mit neuem Startzustand neugestartet. Der Vorteil der Anwendung einer Bestensuche in jedem tiefenbeschränkten Durchlauf im Vergleich zu einer klassischen Tiefensuche kommt dann

zur Geltung, wenn die aktuelle Suchtiefe zum Zeitpunkt der Spielzugabfrage des GMs noch nicht vollständig durchsucht werden konnte. In diesem Fall kann es sein, dass die Bestensuche ein besseres Ergebnis erzielt als eine normale Tiefensuche.

Im Fall von Ein-Personen-Spielen kommt der Anytime-Aspekt noch nicht so deutlich zum Tragen, weil keine Spielzüge anderer Spieler berücksichtigt werden müssen und man somit prinzipiell immer davon ausgehen kann, dass man mit Hilfe seines gewählten Spielzuges im Voraus weiß, wie der Folgezustand aussehen wird. Um die Erweiterung der Implementierung auf Mehrpersonenspiele zu gewährleisten und dem Anytime-Aspekt besser gerecht zu werden, werden deshalb bereits in der jetzigen Implementierung dafür einige Grundlagen geschaffen. Der GM wird durch den Einsatz eines parallel laufenden Threads simuliert, der den Spielfluss steuert und regelmäßig nach bestimmten Zeitintervallen vom Suchalgorithmus, der in einem anderen Thread läuft, einen Spielzug übermittelt bekommen will. Kurz darauf wird er genau diesen Spielzug als „gewählten Spielzug“ zurückgeben, so dass der Suchalgorithmus den neuen Zustand berechnen und die Suche neustarten kann. Konkret wurde diese Umgebung mit Hilfe der Bibliothek QT[3] implementiert.

5.3 Zielzustände, Rewards und Heuristiken

Ein weiterer großer Unterschied zwischen Ein-Personen-Spielen und klassischen Planungsproblemen ist die unterschiedliche Definition von Zielzuständen. Wird in der normalen Handlungsplanung zwischen den Zielzuständen nicht unterschieden – jeder erreichte Zielzustand ist genauso wertvoll wie ein anderer –, so gibt es in GDL-Spielen verschiedene Gruppen von Terminalzuständen, nach ihren Bewertungen durch die Rewardfunktion geordnet. Dies hat zur Folge, dass manche nahen Zielzustände mit niedrigem Nutzen schlechter zu bewerten sind als weiter entfernte Zustände mit hohem Nutzen. Eine weitere Konsequenz ist die Tatsache, dass „normale“ Planungsheuristiken zur Bewertung von Zuständen nicht sinnvoll sind, sondern andere Funktionen gefunden werden müssen.

Die verwendeten Strukturen und die Bewertungsfunktion werden im Folgenden an Hand der Suche erläutert. Wenn der aktuelle Zustand noch nicht expandiert wurde, sich also noch nicht in der ClosedList befindet, so wird er expandiert, indem er in die ClosedList eingefügt wird und seine Nachfolger mittels `GenerateSuccessors()` erzeugt. Danach wird der nächste zu betrachtende Zustand aus der OpenList geholt, wobei die aktuelle Suchtiefe berücksichtigt werden muss. Die originale OpenList des FastDownward-Planers wurde dazu dahingehend angepasst, dass zusätzlich die Tiefe des Knotens im Suchbaum wie auch die Bewertung des Knotens gespeichert wird. Die Methode zur Erzeugung der Nachfolger sieht wie folgt aus:

```
GenerateSuccessors(ParentState)
{
```

[3] http://qt.nokia.com/

```
  Operators[] := GenerateApplicableOps(CurrentState, AllOperators);
  hValues := GetHeuristicValues();
  Reward := CalculateReward(hValues);
  if (Reward >= CurrentBestReward)
  {
    CurrentBestReward := Reward;
    update(CurrentBestNextOperator);
  }
  for i = 0 to Operators.size-1 do
    insert (ParentState, Operators[i]) into OpenList;
  end for;
}
```

Beim Expandieren eines Zustandes werden also zunächst die anwendbaren Operatoren erzeugt und danach für alle möglichen Zielzustandsmengen mit demselben Reward ein separater Heuristikwert für den Abstand zum nächsten Zustand dieser Menge berechnet und in `hValues` gespeichert. Die dazu verwendete Heuristik ist die Kausalgraph-Heuristik, die nur minimal modifiziert werden musste, um mit den veränderten globalen Datenstrukturen arbeiten zu können. Um ihre Funktionsweise zumindest grob beschreiben zu können, bedarf es zwei weiterer Definitionen:

Definition 27 (Kausalgraph)**.** Sei Π eine Planungsaufgabe mit Variablenmenge V und Aktionsmenge A. Ihr *Causal Graph* $CG(\Pi)$ ist der Digraph (V, E), welcher genau dann eine Kante (u, v) enthält, wenn $u \neq v$ und eine Aktion $a \in A$ mit $a = \langle name,\ pre,\ \mathit{eff} \rangle$ existiert, so dass $\mathit{eff}(v)$ und entweder $pre(u)$ oder $\mathit{eff}(u)$ definiert sind.

Definition 28 (Domain Transition Graph)**.** Sei Π eine Planungsaufgabe mit Variablenmenge V und sei $v \in V$. Sei weiterhin D_v der Wertebereich der Variable v.
Der *Domain Transition Graph* $DTG(v)$ ist der beschriftete Digraph mit Knotenmenge D_v, welcher eine Kante (d, d') genau dann enthält, wenn es eine Aktion
$\langle name,\ pre,\ effects \rangle$ gibt mit $pre(v) = d$ oder $pre(v)$ undefiniert und $\mathit{eff}(v) = d'$. Die Kante wird mit $pre|(V \setminus \{v\})$ beschriftet.
Für jede Kante (d, d') mit Label L hat v eine *Transition* von d nach d'.

Die Kausalgraph-Heuristik $h_{cg}(s)$ basiert auf einer Schätzung der Anzahl an Aktionen, die benötigt werden, um das Ziel ausgehend von s zu erreichen. Dabei werden die Kosten summiert, die man aufwenden muss, um alle im Zustand s vorkommenden Variablen v zu ändern, so dass sie ihren Wert v_s zu einem Wert v_* ändern, den sie im Ziel haben müssen.

Definition 29 (Kausalgraph-Heuristik)**.** Die *Kausalgraph-Heuristik* ist definiert als

$$h_{cg}(s) = \sum_{v \in s^*} cost_v(v_s, v_*)$$

Dabei wird $cost_v(d, d')$ mit Hilfe der zwei oben definierten Strukturen berechnet: Der Domain Transition Graph $DTG(v)$, der die Struktur der Domäne D_v, die mit jeder

Variable v verknüpft ist, beschreibt, und der Kausalgraph $CG(\Pi)$, der die Relationen der Variablen v in der Planungsaufgabe Π beschreibt.

Für ausführlichere Beschreibungen und Hintergründe zur Kausalgraph-Heuristik siehe [5] und [7].

Nachdem also der Abstand der verschiedenen Zielzustandsmengen mit der Kausalgraph-Heuristik geschätzt wurde, wird im nächsten Schritt dann die Bewertung für den aktuellen Zustand berechnet und in `Reward` gespeichert. Die Bewertung hängt dabei, wie oben erwähnt, nicht mehr nur vom Abstand zu einem Ziel, sondern auch vom Nutzenwert Ziels ab. Hierzu wird die JO-Berechnungsfunktion verwendet, wie sie in [1] beschrieben wird. Zunächst werden von den Rewards, die mit der normalen Bewertungsfunktion $g : \mathcal{T} \to \{0, \ldots, 100\}$ berechnet werden, je 50 abgezogen. Der Grund dafür liegt in der Berechnung des Heuristikwertes. Diese bildet die mit den Rewards gewichtete Summe aus den Kehrwerten der Distanz zum jeweiligen Terminalzustand. Dadurch bekommen nähere Terminalknoten ein erheblich höheres Gewicht als weiter entferntere, und die Gewichtung durch die negativen oder positiven Rewards bewirkt, dass unvorteilhafte Zustände (in denen man verliert) die Gesamtbeurteilung des aktuellen Zustandes verschlechtern, also insbesondere wenn sie recht nah sind. Sei g die in 4.1.2 beschriebene Rewardfunktion.

Definition 30 (Modifizierte Rewardfunktion)**.** Die *modifizierte Rewardfunktion* g' ist dann definiert als $g'(t) := g(t) - 50$. Somit ist g' dann eine Abbildung mit folgender Signatur: $g' : \mathcal{T} \to \{-50, \ldots, 50\}$.

Definition 31 (JO-Bewertungsfunktion)**.** Seien $\mathcal{T}_q := \{s \in \mathcal{T} \mid g'(s) = q\}$ für alle $q = -50, \ldots, 50$ die nach ihren Gewinnen gruppierten Terminalzustände aus $\mathcal{T}$; $\mathcal{T}_{50}$ sind zum Beispiel alle Terminalzustände mit Reward 50. Dann ist die *JO-Bewertungsfunktion* für einen Zustand s wie folgt definiert:

$$h_{JO}(s) = \begin{cases} \sum_{q \in \{-50,\ldots,50\}} \frac{1}{\texttt{dist}(s, T_q)} \cdot q, & \text{wenn } s \notin T \\ g(s), & \text{sonst} \end{cases}$$

$\texttt{dist}(s, \mathcal{T}_q)$ ist dabei der Abstand vom zu bewertenden Zustand s zum nächsten Zustand aus den Terminalzuständen $\mathcal{T}_q$.

$\texttt{dist}(s, \mathcal{T}_q)$ wird dabei gerade durch die Kausalgraph-Heuristik berechnet.

Insgesamt ist die iterative Tiefenbestensuche in Kombination mit der JO-Berechnungsfunktion ein geeigneter Algorithmus zur Umsetzung der benötigten Änderungen, damit ein Ein-Personen-Spiel von einem Planungssystem gelöst werden kann. Mögliche Erweiterungen oder Verbesserungen könnten im Bereich der Suche ansetzen, indem man zum Beispiel bestimme Teilbäume während der Suche mit höherer Priorität als andere durchsucht, oder sich eine geeignete Datenstruktur überlegt, mit deren Hilfe man Teilbereiche, die man schon expandiert hat, in die nächste Spielrunde (Neustart der Suche) mit übernehmen kann. Jedoch ist bereits klar, dass es mit dem vorgestellten Ansatz möglich ist, mit Planungssystemen Ein-Personen-Spiele zu lösen.

6 Zusammenfassung

In dieser Arbeit wurden zunächst die Grundlagen von General Game Playing und Handlungsplanung erläutert.

Im General Game Playing und dem dazugehörigen Wettbewerb versuchen Programme allgemein kodierte Spiele erfolgreich zu spielen. Der Schlüssel ist hierbei, stets einen besten Spielzug zur Verfügung zu haben, der auch die möglichen Züge der anderen Spieler berücksichtigt. Insbesondere ist man an keiner Komplettlösung interessiert, sondern immer nur an der temporal besten Spielmöglichkeit. Viele Ansätze zur Lösung von allgemeinen Spielen beruhen auf Suchalgorithmen zur Durchsuchung des Spielbaums mit Heuristiken. Es gibt auch andere Ansätze, die zum Beispiel eine gewisse Randomisierung bei der Spielzugauswahl verwenden.
Im Zusammenhang mit GGP wurden weiterhin die Grundlagen der Syntax sowie der Semantik von GDL erläutert. Eine Zentrale Rolle spielen die Datalogregeln, mittels denen sich die Spielwelt entwickeln kann. Datalogregeln spiegeln das Konzept von Successor State Axiomen wider. Diese berechnen einen Folgezustand, indem sie sämtliche gültige Prädikate neu bestimmen, weshalb es viele FrameAxiome geben muss, die Aussagen über unveränderte Prädikate treffen. Die Semantik von GDL basiert auf logischen Modellen und lässt sich anschaulich als Transitionssystem darstellen, zum Beispiel in Automatenform.

Mit ähnlichen Problemstellungen wie das GGP beschäftigt sich auch die Handlungsplanung. Hier werden ebenfalls Planungsaufgaben allgemein kodiert an Programme übermittelt. Im Unterschied zu GGP ist hier nur eine Komplettlösung in Form eines Plans von Bedeutung, Zwischenschritte spielen kaum eine Rolle, da auch keinerlei Interaktion mit anderen Agenten zu berücksichtigen ist. Weiterhin ist irrelevant, welcher Zielzustand erreicht wird, da den Zielzuständen keine unterschiedliche Bewertungen zugeordnet werden. Auch in der Handlungsplanung verwenden bekannte Programme Suchalgorithmen mit Heuristiken zur Lösung der Planungsprobleme.
Weiterhin wurden die Syntax und Semantik von PDDL, analog zum GDL-Teil, beschrieben. Im Unterschied zu GDL verwendet PDDL keine Regelmengen, sondern direkt Aktionen, welche die Veränderung der Planungswelt und das Erreichen von Zielen ermöglichen. Das dahinter liegende Konzept sind die State Update Axiome, welche im Wesentlichen den allquantifizierten bedingten Effekten von PDDL-Aktionen entsprechen. Sie berechnen aus einem bekannten Zustand einen neuen, indem bestimmte Eigenschaften entfernt und andere hinzugenommen werden; es wird insbesondere nicht der komplette Zustand neu konstruiert. Somit lösen State Update Axiome auch das Frame-Problem der Successor State Axiome. Die Semantik von PDDL lässt sich ebenfalls als Transitionssystem in Automatenform darstellen.

Ähnlichkeiten im Spezialfall von Ein–Personen–Spielen mit der Handlungsplanung waren offensichtlich und begründeten den in der Arbeit verfolgten Ansatz, mit Hilfe von Planungssystemen Spiele zu lösen. Damit dieses Ziel der Arbeit umgesetzt werden konnte, musste zunächst die GDL-Beschreibung in eine PDDL-Beschreibung übersetzt werden.

Einige Probleme im Zusammenhang mit der logischen Transformation wie das Mitführen bestimmter Informationen wurden dabei erfolgreich gelöst. Anschließend wurde die Korrektheit der Transformation bewiesen und gezeigt, dass Ein-Personen-Spiel-Kodierungen in äquivalente Planungskodierungen übersetzt werden können.

Im Anschluss an die Transformation konnte dann der modifizierte FastDownward-Planer zum Einsatz kommen. Dieser wurde mit einem neuen Suchalgorithmus ausgestattet, der den Anytime-Aspekt verwirklicht, welcher eine gute Möglichkeit darstellt, den Anforderungen im GGP-Wettbewerb zu genügen. Konkret wurde eine iterative Tiefenbestensuche in Kombination mit der Kausalgraph-Heuristik verwendet. Zur Evaluierung eines Zustandes reichte eine normale Heuristik aber nicht mehr aus, da auch die unterschiedlichen Rewards der möglichen Zielzustände berücksichtigt werden musste. Hierfür wurde die JO-Berechnungsfunktion verwendet, die einen Zustand mit Hilfe gewichteter Summen bewertet. Schließlich wurde noch ein Spielmanager in den Planer eingebaut, der mittels paralleler Threadausführung einen GDL-Wettbewerbsbetrieb simulieren kann und somit auch die modulare Erweiterung auf Mehrpersonenspiele gewährleistet.

Die erfolgreiche Transformation der GDL-Kodierung in eine PDDL-Kodierung sowie die anschließende Ausführung des FastDownward-Planers hat gezeigt, dass Planungssysteme in der Lage sind, Ein-Perseonen-Spiele zu lösen. Es bedarf dazu einiger Anpassungen, welche in dieser Arbeit dargestellt wurden. Es wird ohne viel Aufwand möglich sein, den FastDownward-Planer derart zu erweitern, dass auch Mehrpersonenspiele gelöst werden können. Auf die Weise ist das System auch geeigent, am GGP-Wettbewerb teilzunehmen. Vorstellbar ist weiterhin, dass diese Art der Modifizierung auch auf andere, gute Planungssysteme übertragen werden kann, um mit diesen auch im GGP-Wettbewerb erfolgreich teilnehmen zu können. Ich halte es für wahrscheinlich, dass man durch Weiterverfolgung des hier beschriebenen Ansatzes erfolgreich allgemeine Spiele lösen kann.

Literatur

[1] Aldinger, Johannes: *Lösen allgemeiner Spiele durch heuristische Suche*. April 2009

[2] Campbell, Murray ; Hoane, A. J. ; Hsu, Feng hsiung: *Deep Blue*. August 2001

[3] Ebbinghaus, Heinz-Dieter ; Flum, Jörg ; Thomas, Wolfgang: *Einführung in die mathematische Logik*. Spektrum-Akademischer Verlag, August 1996. – 310 S

[4] Edelkamp, Stefan ; Hoffmann, Jörg: PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition (PDF) / Albert-Ludwigs-Universität Freiburg, Institut für Informatik. 2004 (195). – Forschungsbericht

[5] Geffner, Héctor: *The Causal Graph Heuristic is the Additive Heuristic plus Context*. Association for the Advancement of Artificial Intelligence. 2007

[6] Genesereth, Michael ; Love, Nathaniel: *General Game Playing: Overview of the AAAI Competition*. March 2005

[7] Helmert, Malte: *A Planning Heuristic Based on Causal Graph Analysis*. ICAPS-04. 2004

[8] Helmert, Malte: The Fast Downward Planning System. In: *Journal of Artificial Intelligence Research* 26 (2006), July, S. 191–246

[9] Love, Nathaniel ; Hinrichs, Timothy ; Haley, David ; Schkufza, Eric ; Genesereth, Michael: *General Game Playing: Game Description Language Specification*. March 2008

[10] Nebel, Bernhard ; Hoffmann, Jörg: The FF Planning System: Fast Plan Generation Through Heuristic Search. In: *Journal of Artificial Intelligence Research* 14 (2001), S. 253–302

[11] Röger, Gabriele ; Helmert, Malte ; Nebel, Bernhard: On the Relative Expressiveness of ADL and Golog / Institut für Informatik, Albert-Ludwigs-Universität Freiburg, Germany. – Forschungsbericht

[12] Richter, Sylvia ; Westphal, Matthias: *The LAMA Planner Using Landmark Counting in Heuristic Search*. International Planning Competition 2008. 2008

[13] Thielscher, Michael: From situation calculus to fluent calculus: State update axioms as a solution to the inferential frame problem. In: *Artificial Intelligence* 111 (1999), S. 277–299