**Johannes Aldinger**
**Matr.Nr. 1321288**

ALBERT-LUDWIGS-
UNIVERSITÄT FREIBURG

Lehrstuhl Grundlagen der KI
Prof. Dr. Bernhard Nebel

Freiburg, 29. April 2009

# Lösen allgemeiner Spiele durch heuristische Suche

# 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 29. April 2009

_____

Johannes Aldinger

# Inhaltsverzeichnis

# 1 Einführung

Spiele stehen seit jeher im Fokus der künstlichen Intelligenz. Dem Schachspiel kam dabei ein besonders großes Interesse zu. Computerschach sollte zur Unterhaltung und zur Analyse von Schachpartien dienen, und außerdem einen Einblick in das menschliche Denken und Handeln bringen. Mit der Zeit wurden die Programme immer stärker, und heute sind Schachprogramme für Heimcomputer[1] besser als die menschlichen Großmeister. Dennoch sind die Einblicke in die menschliche Denkweise dürftig. Zwar spielen Schachcomputer sehr gut Schach, aber eben nur Schach. Andere Spiele, selbst einfache wie Tic-Tac-Toe, können von ihnen nicht gelöst werden. Ein Spezialgebiet konnte mit Hilfe von Expertenwissen soweit perfektioniert werden, dass der Computer seine schnellere Rechenzeit gegenüber dem Menschen ausspielen kann, ein tieferer Einblick bleibt aber versagt. Ein Schritt in Richtung eines allgemeineren Denkens sind Computerprogramme, welche nicht nur *ein* Spiel spielen können, sondern *jedes* Spiel. Dem Computer werden dazu lediglich die Spielregeln eines Spiels vermittelt. Die Forschung in diesem Gebiet nennt sich General Game Playing (GGP).

Ein erfolgversprechender Ansatz um allgemeine Spiele zu lösen wird hier vorgestellt. Planungsprobleme können, nicht zuletzt auch auf Grund äußerst erfolgreicher Forschung der Universität Freiburg [HN01][Hel06][RW08], sehr fortschrittlich gelöst werden. Einspieler-Knobelspiele sind im Grunde genommen nichts anderes als Planungsprobleme. Lediglich die Eingabesprache (Game Description Language (GDL) statt Planning Domain Description Language (PDDL)) unterscheidet sich ein wenig. Mehrpersonenspiele erweitern die Problemstellung um die Ungewissheit der Aktionen der Mitspieler, aber dennoch gibt es Möglichkeiten, diese Teilgebiete miteinander zu verbinden.

In Kapitel 2 wird der Rahmen beschrieben, in welchem die GGP-Forschung stattfindet. In Kapitel 3 werden theoretische Vorüberlegungen zum allgemeinen Lösen von Spielen vermittelt und Anpassungsmöglichkeiten auf die GGP-Problemstellung aufgezeigt. Kapitel 4 beschreibt schließlich die praktische Realisierung dieses Ansatzes in Java. Die hieraus gewonnenen Ergebnisse werden schließlich im Kapitel 5 beschrieben.

1 Deep Fritz - Kramnik 4:2 (2006)

# 2 Grundlagen

GGP [GL05] ist ein aktueller Forschungsbereich, der insbesondere durch seine allgemeine Ausrichtung interessant ist. Entgegen den bisherigen Experten-Spielprogrammen können allgemeine Spielprogramme viele verschiedene Spiele spielen. Aber auch andere Aufgaben wie z.B. Geschäftsmodellmanagement, Automatisierungsprozesse im elektronischen Handel sowie Verifikationsprobleme lassen sich als „Spiel“ modellieren [GL05]. Um verschiedene Ansätze miteinander vergleichen zu können, ist ein Rahmen nötig, welcher einen gemeinsamen Standard als Grundlage schafft. Dieser wird durch den GGP-Wettbewerb der Stanford University gegeben, welcher ein Teil der AAAI-Konferenz ist. Im Folgenden wird der so definierte Rahmen beschrieben. Dies sind einerseits der Wettbewerb selbst, andererseits die Spielmodellierung und die Spielregelbeschreibungssprache GDL.

## 2.1 Wettbewerb

Von der Association for the Advancement of Artificial Intelligence (AAAI) wird regelmäßig ein offener Wettbewerb im Bereich GGP veranstaltet, bei dem die Teilnehmer ihre allgemeinen Spielprogramme in einem Turnier gegeneinander antreten lassen. Das beste Programm wird mit einer Siegprämie von 10.000 US$ belohnt.
Sowohl Normierungen wie die Sprache GDL als auch die Kommunikations- und Kontroll-Plattform Game Master (GM) werden zur Verfügung gestellt. Die Kommunikation mit GM erfolgt über das Internet mit dem Hyper Text Transfer Protocol (HTTP). Der GM regelt die Kommunikation der Spieler, misst die Zeit und fungiert als Schiedsrichter. So wählt er beispielsweise zufällige Züge aus, falls die zur Verfügung stehende Zeit abgelaufen ist oder ein illegaler Zug übermittelt wurde. Zudem verfügt er über ein grafisches Interface, so dass die Spiele des Wettbewerbs anschaulich mitverfolgt werden können.

## 2.2 Spielmodell

Die Spiele, welche im Rahmen von GGP betrachtet werden, sind endliche, synchrone Spiele. Endlich bedeutet, dass sowohl die Anzahl der Spieler, die Anzahl der möglichen Züge und der Zustände endlich ist. Synchron bedeutet, dass in jeder Runde alle Spieler gleichzeitig einen Zug auswählen. Das ist keine Einschränkung, da in vielen sequentiellen Spielen der einzig legale Zug eines Spielers „noop“ (also die „mache-nichts“-Aktion) ist, wenn er nicht an der Reihe ist. Im Gegensatz zu spezialisierten Spielprogrammen müssen GGP-Spieler sowohl einfache (Tic-

Tac-Toe) wie auch komplexe Spiele (Schach) spielen können. Die Welt kann sowohl statisch als auch dynamisch formuliert werden. Spiele mit partieller Information werden bisher noch nicht unterstützt, eine Erweiterung der Spielbeschreibungssprache GDL ist aber möglich [LHH+08]. Ein Spiel besteht formal aus den folgenden Komponenten:

| | |
|---|---|
| • eine Menge von Zuständen | $S$ |
| • $n$ Spieler, sogenannten „Rollen“ | $r_1, \ldots, r_n$ |
| • $n$ Aktionsmengen, eine für jeden Spieler | $I_1, \ldots, I_n$ |
| • legale Züge für jeden Spieler in einem Zustand | $l_1, \ldots, l_n$, $l_i \subseteq I_i \times S$ |
| • eine partielle Zustandübergangsfunktion, die aus den Aktionen der Spieler und dem aktuellen Zustand den Nachfolgezustand berechnet. | $u : I_1 \times \cdots \times I_n \times S \to S$ |
| • einem Initialzustand | $s_0 \in S$ |
| • eine Menge von Terminalzuständen | $T \subseteq S$ |
| • eine Funktion pro Spieler, die den Gewinn (Nutzen) in einem Terminalzustand berechnet. | $g_1, \ldots, g_n$, $g_i : T \to \{0, \ldots, 100\}$ |

Ein Spiel startet in Zustand $s_0$ und wird solange gespielt, bis ein Terminalzustand $t \in T$ erreicht ist. In jeder Runde wählt jeder Spieler einen legalen Zug aus. Endet das Spiel, so erhält jeder Spieler den durch $g_i$ beschriebenen Gewinn. Die Spiele werden hier nicht notwendigerweise als Nullsummenspiele modelliert. In einem Terminalzustand $t$ ist die Summe der Spielergewinne $\sum_{i=1}^{n} g_i[t]$ also nicht unbedingt 0, wie dies z.B. bei Schach ($g_{Sieger} = 1, g_{Verlierer} = -1$, oder 0 bei einem Unentschieden) der Fall ist. Die Zustandsbeschreibung kann in Automatenform repräsentiert werden (Abb. 1). Diese erlaubt synchrone Züge, und mehrfach erreichbare Zustände müssen entgegen anderen Darstellungsformen (wie z.B. einer Baumdarstellung) nur einmal angegeben werden. In Abbildung 1 ist $s_0 = a$, $T = \{c, e, i, k\}$ und $I_1 = I_2 = \{x, y\}$. Die unterschiedliche Schattierung symbolisiert unterschiedlich hohe Gewinne. An den Kanten sind die möglichen Aktionen der Spieler angegeben.

Alle Spiele sind zudem schwach gewinnbar, was bedeutet, dass es für jeden Spieler eine Sequenz von Spielzügen aller Spieler gibt, so dass dieser Spieler seinen maximalen Gewinn erhält. Gibt es sogar eine Sequenz von Spielzügen eines Spielers[2], bei der dieser seinen maximalen Gewinn

2 übernommen von [GL05], intuitiver wäre die Existenz einer Strategie (bedingter Plan), und nicht die stärkere Forderung nach einem konformanten Plan

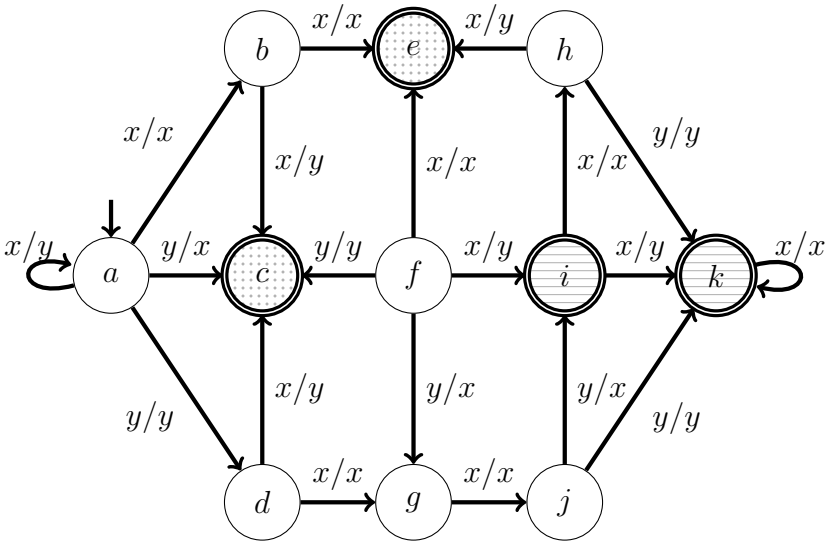


Abbildung 1: $u$ als Zustandsautomat

erreichen kann, unabhängig wie die anderen Spieler spielen, so bezeichnet man ein Spiel als stark gewinnbar, und dieser Spieler hat eine Gewinnstrategie. Bei Mehrpersonenspielen gibt es diese im Allgemeinen allerdings nicht. Damit die Spiele wohlgeformt sind, müssen sie zudem terminieren. Alle Spiele lassen sich theoretisch durch Spielen aller möglichen Kombinationen auf Wohlgeformtheit überprüfen. Alle Spiele des Wettbewerbs können allerdings ohne Beweis als wohlgeformt vorausgesetzt werden.

## 2.3 Datalog und GDL

Die Spielregeln der GGP-Spiele müssen dem Computerprogramm in irgendeiner Form vermittelt werden. Natürliche Sprache bietet sich hierfür aus mehreren Gründen nicht an. Einerseits ist sie oft unpräzise, so dass Konstellationen auftreten können, die nicht eindeutig in den Regeln abgedeckt sind. Auf der anderen Seite ist natürliche Sprache sehr komplex. Das Spiel intern so zu repräsentieren, dass der Computer das Spiel „versteht“, stellt eine computerlinguistische Herausforderung dar, welche mit dem eigentlichen Fokus von GGP wenig zu tun hat. Stattdessen verwendet der GGP-Wettbewerb die logische Sprache GDL. GDL basiert auf Datalog, mit einigen problemspezifischen Anpassungen. Datalog ist eine an Prolog angelehnte Datenbanksprache, welche eine kompakte Repräsentation des Zustandsraums ermöglicht.

GDL baut auf Datalog auf, verzichtet allerdings auf einen Gleichheitstest und fügt Funktionen ein. Ferner werden in Kap. 2.3.4 einige Relationen mit einer festen Bedeutung definiert. Gleichheit ist in Datalog auf einer syntaktischen Ebene definiert. ($a = a$ und $a \neq b$). Statt der

Datalog-Tests für „=“ und „≠“ wird in GDL lediglich ein Test auf Ungleichheit (`distinct`) benötigt. Ein Gleichheitstest ist nicht notwendig, da man beim Erstellen der Spielbeschreibung einfach die gleichen Variablen[3] verwenden kann.

### 2.3.1 Die Syntax von GDL

Die Syntax von Datalog, und die Unterschiede zu GDL, werden in [LHH+08] beschrieben. Da die Unterschiede für uns nicht relevant sind, wird gleich die für GDL angepasste Syntax vorgestellt.

**Definition 1 (Vokabular)** *Das Vokabular von GDL besteht aus*

- *Relationsbezeichner: Konstante mit Stelligkeit, z.B.* `adjacent/2`, `terminal/0`
- *Funktionsbezeichner: Konstante mit Stelligkeit, z.B.* `cell/3`
- *Objekten: auch Objekt-Konstanten, z.B.* `a`, `1`, `blank`
- *Variablen: z.B.* `X`, `Y`

*Aus einem Vokabular lässt sich eine* Sprache *bilden.*

**Definition 2 (Term)** *Als Terme bezeichnet man*

- *Objekt-Konstanten oder*
- *(Objekt-)Variablen oder*
- *Funktionsbezeichner mit Stelligkeit n angewandt auf n Terme*
  *z.B.* `cell(a, 2, white_knight)`

Selbst das Lösen einfacher genereller Spiele stellt eine große Herausforderung dar. Der Fokus dieser Arbeit liegt bei sequentiellen Zweipersonenspielen. Parallele Spiele sowie Spiele mit drei oder mehr Spielern werden hier nicht behandelt. In der Spezifikation der GDL-Spielbeschreibungen können zudem Funktionen beliebiger Schachtelungstiefe vorkommen. Hier (wie auch in allen bisherigen Wettbewerben) werden allerdings nur Funktionen mit einer Schachtelungstiefe von 1 (d.h. Funktionen, deren Parameter Objekte, aber nicht selbst wieder Funktionen sind) behandelt.

3 Unique name assumption

Wie im Kapitel 2.2 erwähnt soll GDL endliche Zustandsautomaten beschreiben. Die Berechnung der Zustandsübergangsfunktion ist immer entscheidbar und die Lösung ist immer einer von endlich vielen Zuständen. Auch GDL soll diese Eigenschaften haben. Durch die Hinzunahme von Funktionen können aber bei rekursiven Funktionen Probleme auftauchen.

**Definition 3 (Abhängigkeitsgraph)** *Sei $\Delta$ eine Menge von Datalogregeln. Dann sind die Knoten des Abhängigkeitsgraphen die Relationsbezeichner des Vokabulars. Er enthält eine Kante von $r_b$ zu $r_h$, wenn es eine Regel $R \in \Delta$ gibt, in welcher $r_h$ im Kopf und $r_b$ im Rumpf vorkommt. Wenn $r_b$ für ein negatives Literal steht, wird diese Kante mit „$\neg$“ beschriftet.*

**Definition 4 (Rekursionsrestriktion)** *Sei $\Delta$ eine Menge von Regeln und $G$ der Abhängigkeitsgraph von $\Delta$, und $\Delta$ enthält eine Regel*

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

*bei welcher $q$ in einem Zykel mit $p$ in $G$ vorkommt. Dann muss entweder für alle $j \in \{1, \ldots, k\}$ der Term $v_j \in \{t_1, \ldots, t_n\}$ variablenfrei sein, oder es existiert ein $i \in \{1, \ldots, m\}$, so dass $b_i = r(\ldots, v_j, \ldots)$, wobei $r$ nicht in einen Zykel mit $p$ vorkommt.*

Dadurch wird sichergestellt, dass Funktionsterme eine endliche Größe haben.

**Definition 5 (Atom, atomarer Satz)** *Ein Atom, atomarer Satz oder auch eine Relation ist ein Relationsbezeichner der Stelligkeit $n$ angewandt auf $n$ Terme.*

*z.B.* `adjacent(a, b)`

**Definition 6 (Literal)** *Ein Literal ist ein Atom, oder dessen negierte Form.*

*z.B.* $\neg$`adjacent(a, c)`, `successor(1, 2)`

Die im nächsten Kapitel 2.3.3 vorgestellte Notation verwendet das Knowledge Interchange Format (KIF). Dort wird die Disjunktion „$\vee$“ verwendet, und beschrieben, dass diese durch den $s$-stelligen logischen Operator „`or`“ ausgedrückt wird. Dessen Definition fehlt in der offiziellen Spezifikation [LHH+08], obwohl „`or`“ in Spielbeschreibungen verwendet wird.

**Definition 7 (Datalog-Regel)** *Eine Datalogregel ist eine Implikation der Form*

$$h \Leftarrow b_1 \wedge \cdots \wedge b_n$$

*dabei ist*

- *der Kopf $h$ der Implikation ein Atom,*
- *die $b_i$, $1 \leq i \leq n$, im Rumpf sind (möglicherweise einstellige) Disjunktionen von Literalen*
- *Jede Variable, die im Kopf oder in einem negativen Literal vorkommt, muss im Rumpf auch in einem positiven Literal vorkommen.*
- *Die Rekursionsrestriktion (Def. 4) muss erfüllt sein.*

Die letzte Einschränkung verhindert Probleme mit unendlich großen Modellen, die im nächsten Abschnitt über die Semantik von Datalog 2.3.2 klar werden.

**Definition 8 (Grundausdruck)** *Enthält ein Ausdruck (Term, Atom, Literal oder Regel) keine Variablen, so bezeichnet man ihn als Grundausdruck (Grundterm, Grundatom, Grundliteral oder Grundregel).*

### 2.3.2 Die Semantik

**Definition 9 (Modell)** *Ein Modell ist eine Menge an Literalen, welche aus einer Sprache L gebildet werden können.*

**Definition 10 (Erfüllbarkeit)** *Sei $M$ ein Modell und der jeweilig betrachtete Satz eine explizit allquantifizierte Datalogregel. Dann gilt*

- $M \models t_1 = t_2$ *genau dann, wenn* $t_1$ *und* $t_2$ *syntaktisch der gleiche Term sind.*
- $M \models p(t_1, \ldots, t_n)$ *genau dann, wenn* $p(t_1, \ldots, t_n) \in M$.
- $M \models \neg\phi$ *genau dann, wenn* $M \not\models \phi$.
- $M \models \phi_1 \wedge \cdots \wedge \phi_n$ *genau dann, wenn* $M \models \phi_i$ *für alle* $i$.
- $M \models h \Leftarrow b_1 \wedge \cdots \wedge b_n$ *genau dann, wenn* $M \not\models b_1 \wedge \cdots \wedge b_n$ *oder* $M \models h$
- $M \models d_1 \vee \cdots \vee d_s$ ***genau dann, wenn*** $M \models d_i$ ***für ein*** $i$

- $M \models \forall X.\phi(X)$ *genau dann, wenn* $M \models \phi(t)$ *für alle Grundterme* $t$*, die aus der Sprache gebildet werden können.*

Sind die Regeln nicht explizit allquantifiziert, werden sie als implizit allquantifiziert angenommen.

Es ist wünschenswert, ein *minimales* Modell zu finden. Ein Modell $M$ ist minimal mit der Eigenschaft, dass $M \models \phi$, wenn für kein $N \subsetneq M$ gilt $N \models \phi$. In Datalog ohne Negation ist dieses minimale Modell eindeutig und leicht zu finden. Es ist genau die Bedeutung der Regeln. Für minimale Modelle in Datalog mit Negation werden einige weitere Definitionen benötigt.

**Definition 11 (Stratifizierte Datalogregeln)** *Eine Menge von Datalogregeln* $\Delta$ *heißt stratifizierbar genau dann, wenn der zugehörige Abhängigkeitsgraph keine Zykel enthält, in denen mindestens eine Kante mit* $\neg$ *beschriftet ist.*

**Definition 12 (Stratum)** *Sei* $\Delta$ *eine Menge stratifizierbarer Datalogregeln und betrachtet man den Abhängigkeitsgraphen von* $\Delta$*. Ein Relationsbezeichner* $r$ *ist in Stratum* $i$ *genau dann, wenn die maximale Anzahl der mit* $\neg$ *beschrifteten Knoten auf einem Pfad, der in* $r$ *beginnt,* $i$ *ist.*

*Eine Regel ist in Stratum* $i$*, wenn der Relationsbezeichner im Kopf der Regel in Stratum* $i$ *ist.*

Da die Regeln stratifiziert sind, kann es keine Zykel mit Negation geben, und jede Regel gehört zu genau einem (endlichen) Stratum. Außerdem gibt es mindestens eine Regel in Stratum 0. Jetzt ist es auch möglich, eine Analogie zum minimalen Modell in Datalog ohne Negation zu definieren.

**Definition 13 (Stratifizierte Datalogsemantik)** *Sei* $\Delta$ *eine Menge stratifizierbarer Datalogregeln, und* $M_0$ *das minimale Modell, welches die Regeln in Stratum 0 erfüllt. Um* $M_i$ $(i > 0)$ *zu berechnen, initialisiere* $M_i$ *mit* $M_{i-1}$*.*

*Füge den Kopf* $h\sigma$ *jeder Substitution* $\sigma$ *einer Regel* $h \Leftarrow b_1 \wedge \cdots \wedge b_n$ *aus Stratum* $i$*, für die* $M_i \models (b_1 \wedge \cdots \wedge b_n)\sigma$ *gilt, zu* $M_i$ *hinzu, und wiederhole diesen Prozess, bis keine Änderungen mehr vorkommen.*

*Sei* $k$ *das größte vorkommende Stratum der Regeln in* $\Delta$*. Die stratifizierte Datalogsemantik von* $\Delta$ *ist* $M_k$*.*

Das Hinzufügen der Köpfe $h\sigma$ terminiert, da das Modell $M_{i-1}$ endlich ist. Wegen der Rekursionsrestriktion (Def. 4) können auch die frisch hinzugefügten Köpfe keine unendlichen Modelle produzieren.

Man kann eine Aussage darüber machen, ob ein Literal $\phi$ bereits aus einer Regelmenge folgt.

**Definition 14 (Logische Folgerung)** *Sei $\Delta$ eine Menge stratifizierter Datalogregeln und $M$ die stratifizierte Datalogsemantik für $\Delta$. Dann gilt $\Delta \models \phi$ genau dann, wenn $M \models \phi$.*

Selbst bei einer stratifizierten Datalogsemantik könnte sich aber ein unendliches Modell ergeben, wenn man auf die Sicherheits-Einschränkung verzichten würde. Jede Variable, die im Kopf oder in einem negativen Literal vorkommt, muss im Rumpf auch in einem positiven Literal vorkommen. Betrachten wir beide Fälle:

- $p(X, Y) \Leftarrow q(X)$. Angenommen $q(a)$ ist wahr. Also ist auch $p(a, Y)$ für alle $Y$ wahr. Und wenn die Sprache unendlich viele Objektkonstanten enthält, die für $Y$ eingesezt werden können, dann wäre auch das Modell unendlich groß, etwa $\{p(a, a), p(a, b), p(a, c), \dots\}$, obwohl die Anzahl der Regeln $\Delta$ endlich ist.
- Im anderen Fall $s(a) \Leftarrow \neg r(X)$ ist nicht unbedingt klar, ob dies aussagt, dass $s(a)$ gilt, wenn es ein $X$ gibt, so dass $r(X)$ falsch ist, oder dass $s(a)$ gilt, wenn $r(t)$ für keinen Grundterm $t$ gilt.

Mit dieser Einschränkung kann man bei einer Evaluierung zunächst die wahren Literale instantiieren, und danach gefahrlos die negativen Literale auswerten.

### 2.3.3 KIF

Datalog verwendet eine mathematische Notation, bei der z.B. Variablen Großbuchstaben sind. Beim GGP-Wettbewerb sollen Programme von verschiedenen Plattformen miteinander verglichen werden, so dass es sinnvoller ist, eine Notation zu verwenden, die Probleme mit Groß- und Kleinschreibung sowie mit Sonderzeichen („$\vee$“, „$\wedge$“, „$\Leftarrow$“) umgeht. GDL verwendet deshalb die Knowledge-Interchange-Format-Notation KIF [GF+92]. Variablen werden dort mit einem „?“ annotiert, und alle Terme sind Lisp-S-Ausdrücke. KIF verwendet eine Präfixnotation. Logische Operatoren, Implikationen und Funktionen werden in Präfixschreibweise notiert. Die Operatoren „$\Leftarrow$“, „$\neg$“ und „$\neq$“ sowie „$\vee$“ werden zu „`<=`“, „`not`“, „`distinct`“ und „`or`“.

- $X$ wird zu `?x` (und gleichbedeutend `?X`)
- $adjacent(a, b)$ wird zu `(adjacent a b)`
- $true(cell(a, 1, PIECE))$ wird zu `(true (cell a 1 ?piece))`
- $p(X) \Leftarrow q(X) \wedge (\neg r(a) \vee (X \neq a))$ wird zu

```
(<=  (p ?x)
     (q ?x)
     (or (not (r a))
         (distinct ?x a)
     )
)
```

Die Vorbedingungen einer Implikation sind dabei eine Formel in konjunktiver Normalform. Die Konjunktionen werden hier nicht explizit hingeschrieben, sondern durch die Klammerung implizit gegeben. Disjunktionen hingegen werden explizit durch das Schlüsselwort „`or`“ ausgedrückt.

### 2.3.4 Vordefinierte Relationen in GDL

Folgende Relationen werden in GDL vordefiniert: „`role`“, „`true`“, „`init`“, „`next`“, „`legal`“, „`does`“, „`goal`“, „`terminal`“ und zudem noch die logische Relation „`distinct`“.

Als Beispiel wird hier das Spiel Tic-Tac-Toe verwendet.

**GDL-Relation 1 (role)** *`role`/1 [SPIELERNAME].*
*SPIELERNAMEN sind spezielle Objekte, welche für die Namen der Spieler stehen.*

```
(role xplayer)
(role oplayer)
```

In Tic-Tac-Toe gibt es zwei Spieler, einen mit Namen „`xplayer`“ und einen „`oplayer`“.

**GDL-Relation 2 (true)** *`true`/1 [FAKT].*
*FAKTEN sind Objekte oder Funktionen (bzw. Funktionsterme) der Schachtelungstiefe 1. Der aktuelle Zustand des Spiels wird von Grundtermen in relationaler Logik geformt.*

*`(true (cell a 1 blank))` bedeutet, dass `(cell a 1 blank)` im aktuellen Zustand gilt.*

Parameterlose, nullstellige Funktionen entsprechen den Objekten. Die Argumente einer Funktion der Schachtelungstiefe 1 sind Funktionen, deren Argumente Objektkonstanten sind.

**GDL-Relation 3 (init)** *`init`/1 [FAKT].*
*Analog zu `true`, mit dem Unterschied, dass eine Aussage über den Wahrheitswert von Fakten im Initialzustand gemacht wird.*

```
(init (cell a 1 blank))
(init (cell b 1 blank))
(init (cell c 1 blank))
(init (cell a 2 blank))
(init (cell b 2 blank))
(init (cell c 2 blank))
(init (cell a 3 blank))
(init (cell b 3 blank))
(init (cell c 3 blank))
(init (control xplayer))
```

Tic-Tac-Toe startet mit einem leeren Spielfeld, und Spieler „`xplayer`“ ist am Zug.

**GDL-Relation 4 (next)** *`next`/1 [FAKT].*
*Analog zu `true`, mit dem Unterschied, dass eine Aussage über den Wahrheitswert von Fakten im Folgezustand gemacht wird.*

```
(<= (next (control xplayer))
    (true (control oplayer)))

(<= (next (control oplayer))
    (true (control xplayer)))
```

Wenn im aktuellen Zug „`xplayer`“ an der Reihe ist, wird im nächsten Zug „`oplayer`“ an der Reihe sein, und umgekehrt.

**GDL-Relation 5 (legal)** *`legal`/2 [SPIELERNAME, SPIELZUG].*
*Ein SPIELZUG ist ein Funktionsterm, welcher eine Aussage trifft über die Aktion, die ein Spieler machen kann.*

```
(<= (legal ?player (mark ?x ?y))
    (true (cell ?x ?y blank))
    (true (control ?player)))

(<= (legal xplayer noop)
    (true (control oplayer)))

(<= (legal oplayer noop)
    (true (control xplayer)))
```

Der Spieler, der am Zug ist, darf ein Feld markieren, wenn es frei ist. Wenn der andere Spieler an der Reihe ist, darf man nichts tun (also „noop“ spielen).

**GDL-Relation 6 (does)** *`does`/2 [SPIELERNAME, SPIELZUG].*
*Analog zu `legal`, mit dem Unterschied, dass der Spieler den Zug auch tatsächlich ausgeführt hat. Meistens kommt `does` im Rumpf von Datalogregeln mit der Kopfrelation `next` vor.*

```
(<= (next (cell ?x ?y x))
    (does xplayer (mark ?x ?y)))
```

*Außerdem werden Rahmenaxiome auf diese Art und Weise angegeben (wahr ist, was wahr war und nicht falsch gemacht wurde).*

```
(<= (next (cell ?x ?y blank))
    (true (cell ?x ?y blank))
    (does xplayer (mark ?m ?n))
    (or (distinct ?m ?x)
        (distinct ?n ?y)))
```

Markiert „`xplayer`“ ein Feld, dann wird es im Folgezustand ein „`x`“ enthalten. Alle anderen Felder bleiben, wie sie sind.

**GDL-Relation 7 (goal)** *goal/2 [SPIELERNAME, BELOHNUNG].*
*BELOHNUNG ist ein Objekt, welches für einen numerischen Wert zwischen 0 und 100 steht.*

```
(<= (goal xplayer 100)
    (line x))
```

Spieler „xplayer“ erhält den (maximalen) Nutzen von 100, wenn es ihm gelingt, eine Reihe mit „x“en zu erzeugen.

**GDL-Relation 8 (terminal)** *terminal/0.*
*Wenn ein Terminalzustand erreicht ist, wird die null-stellige Relation „terminal“ wahr.*

```
(<= terminal
    (line ?symbol)
    (distinct ?symbol blank)
)
```

Das Tic-Tac-Toe-Spiel endet (unter anderem), wenn auf dem Spielfeld eine Reihe von gleichen Symbolen ist.

### 2.3.5 Restriktionen der GDL-Relationen

Um die beabsichtige Semantik der vordefinierten Relationen sicherzustellen, sind einige weitere Einschränkungen nötig. Eine Abweichung von der offiziellen Version [LHH+08], welche aber in der Praxis vorkommt, ist **fett** gedruckt.

**Definition 15 (GDL Restriktionen)** *Sei $\Delta$ eine GDL-Spielbeschreibung und $G$ der Abhängigkeitsgraph von $\Delta$. Dann muss jede der folgenden Bedingungen gelten:*

- *Die Relation role kommt nur in Grundatomen vor. Entweder als Kopf einer Datalogregel ohne Rumpf,* ***oder im Rumpf von anderen Regeln.***
- *Die Relation init kommt nur im Kopf von Datalogregeln vor, und in $G$ ist der init-Knoten nicht in derselben Zusammenhangskomponente wie true, does, next, legal, goal or terminal.*

- *Die Relation* `true` *kommt nur im Rumpf von Datalogregeln vor.*
- *Die Relation* `next` *kommt nur im Kopf von Datalogregeln vor.*
- *Die Relation* `does` *kommt nur im Rumpf von Datalogregeln vor, und in $G$ gibt es keinen Pfad zwischen dem* `does`*-Knoten und* `legal`*,* `goal` *oder* `terminal`*.*

Nochmals erwähnt sei hier zudem, dass eine Spielregelbeschreibung ausschließlich aus Datalogregeln besteht. Atomar vorkommenden Relationen sind eine Kurzschreibweise für Implikation ohne Rumpf. Demzufolge muss für sie auch die die Sicherheitseinschränkung gelten. Zum Beispiel steht `(adjacent a b)` für `(<= (adjacent a b))`.

# 3 Bestandteile eines GGP-Systems

Im vorherigen Kapitel wurde der Rahmen, in dem GGP-Probleme gestellt werden, beschrieben. Die theoretischen Vorüberlegungen für die Entwicklung eines GGP-Programms werden in diesem Kapitel vorgestellt. Zunächst werden die Anforderungen eines GGP-Programms betrachtet. Dabei wird deutlich, dass ein solches Programm ein Planungsmodul sowie ein Bewertungsmodul benötigt. In den darauf folgenden Kapiteln 3.2 und 3.3 werden diese Module behandelt.

## 3.1 Anforderungen

Um die in der logischen Sprache GDL geschrieben Spielregeln zu verarbeiten, muss ein Programm logisch schließen können, sei es direkt auf der Sprache, oder auf einer daraus gewonnenen Repräsentation. Die bisher verfügbaren Programme eignen sich dazu allerdings nur bedingt, da im Bereich GGP einige spezifische, aber immer wiederkehrende Probleme auftauchen. Bei einfachen Einpersonenspielen entspricht die Problematik dem, was herkömmliche Logikplaner schon sehr gut lösen können. Wenn es gelingt, die GDL-Beschreibung in eine bekannte Repräsentation zu übersetzen, kann man hier auf eine Fülle an Expertise und vorhandenen Implementierungen zurückgreifen. Mehrpersonenspiele, bei denen die Aktionen der Gegner nicht im Voraus bekannt sind, stellen allerdings neue Herausforderungen dar. Konditionales Planen, insbesondere bei komplexeren Spielen, wird schnell zu einem sehr (und für unsere Ansprüche zu) rechenintensiven Problem. Und selbst wenn man unbegrenzte Rechenzeit hätte, könnte man nicht unbedingt „den besten“ Plan finden. Nicht alle Spiele sind stark gewinnbar, viele enden bei korrektem Spiel beider Spieler unentschieden. Dennoch kann es Züge geben, welche eine Gewinnstrategie vom neuen Spielzustand aus ermöglichen, falls der Gegner einen suboptimalen Zug spielt. Eine mögliche Strategie wäre es hier zum Beispiel, nach Zügen zu suchen, welche dem Gegner einen großen Suchraum bieten. Es ist so wahrscheinlicher, dass der Gegner in der ihm zur Verfügung stehenden Zeit keinen guten Zug findet. Durch die Größe des Zustandsraums bei komplexeren Spielen (Schach hat ca. $10^{30}$ Zustände) ist es nur theoretisch möglich, den Zustandsraum komplett zu durchsuchen. Es ist dann notwendig, Näherungsfunktionen zur Einschätzung der Qualität eines Zustandes zu berechnen. Diese Näherungsfunktionen nennt man Heuristiken oder Bewertungsfunktionen.

Eine weitere Möglichkeit, bessere Züge zu finden, ergibt sich aus dem wiederholten Spielen eines Spiels gegen den gleichen Gegner. So sind auch Module zur Gegnermodellierung denkbar. Angenommen, man spielt das Spiel „Schere-Stein-Papier“, und stellt fest, dass der Gegner jedes

Mal die Aktion „Schere“ spielt. Dann würde ein gutes Programm dieses Verhalten erkennen, und selbst „Stein“ spielen, um das suboptimale Verhalten des Gegners auszunutzen. Um ein GGP-Programm zu schreiben, benötigt man demnach zumindest ein Planungsmodul, welches nach einem guten Spielzug sucht, und ein Heuristikmodul, welches für große Zustandsräume schnell Näherungen von Bewertungskriterien berechnen kann.

Die Spiele des GGP sind zeitbeschränkt, und nur selten kann man das Spiel komplett durchrechnen. Ein Planungsmodul sollte zu jedem Zeitpunkt den als besten bekannten Zug zurückliefern können. Je mehr Zeit zur Verfügung steht, umso besser sollte dieser Zug auch sein. Auch wünschenswert ist es, dass ein optimaler Zug zurückgeliefert wird, wenn die zur Verfügung stehende Zeit unbeschränkt ist. Das in Kapitel 4 vorgestellte Programm „CoolerJo“ berechnet einen der Züge, welche zum höchsten sicher erreichbaren Nutzen führen. Wie weiter oben diskutiert, kann es davon allerdings mehrere Züge geben, von denen manche erstrebenswerter sind als andere, weil der Gegner in eine Situation mit erhöhter Fehleranfälligkeit gebracht wird. Dafür wären dann allerdings weitere Module notwendig, welche z.B. das Verhalten des Gegners miteinbeziehen. Da diese Analysemodule auch Zeit benötigen, ist es fraglich, ob und inwieweit dies die Leistung des Gesamtsystems tatsächlich verbessern könnte. In dieser Arbeit liegt der Fokus auf den zuerst genannten Modulen, welche in den nächsten Unterkapiteln 3.2 und 3.3 vorgestellt werden.

## 3.2 Planungsmodul

Das Planungsmodul oder auch Suchmodul hat die Aufgabe, einen möglichst erstrebenswerten Zielzustand, also einen mit hohem Nutzen, zu erreichen. Insbesondere bei Mehrpersonenspielen muss dabei die Unbestimmtheit, welche sich aus den verschiedenen Optionen des Gegners ergibt, miteinbezogen werden.

Im Falle von Einpersonenspielen bietet sich eine heuristische Bestensuche (Kap. 3.2.1) an. Um ein Zweipersonenspiel optimal zu lösen, gibt es den Minimax-Algorithmus, welcher weiter unten (Kap. 3.2.2) vorgestellt wird. Nach einigen Anpassungen (Kap. 3.2.3) eignet er sich auch für GGP. Dank der Verwendung des Bewertungsmoduls kann man die Suche jederzeit abbrechen, und erhält den besten bisher bekannten Zug.

### 3.2.1 Bestensuche für Einpersonenspiele

Das Planungsmodul muss sich je nach Spieleranzahl unterschiedlich verhalten. Kommt in der GDL-Beschreibung nur ein Spieler vor, wird eine Bestensuche gestartet. Normale Bestensuche sucht einen möglichst kurzen Weg zu „dem“ Ziel. Die Heuristik der Bestensuche schätzt dabei normalerweise die Distanz zum Ziel. Beim GGP gibt es allerdings viele Ziele, die zudem unterschiedlich erstrebenswert sind. Hier wird stattdessen die Evaluierungsfunktion $f_{JO}$ benutzt, die später (Kapitel 3.4) vorgestellt wird. Diese schätzt die Güte eines Zustandes ab. Entgegen der klassischen Bestensuche wird in diesem Fall über die Heuristikwerte maximiert. Der Algorithmus in Pseudocode:

```
GGP-Best-First-Search (One-Player-Game) returns BestMoveSoFar
  initialState.determineHeuristicValue
  Queue = [initialState]
  BestTerminalNode = nil (with utility -infinity)
  while (timeLeft AND NOT Queue.empty) do
     node = Queue.removeElementWithBestHeuristicValue
     if node.isTerminal AND node.utility > BestTerminalNode.utility then
       BestTerminalNode = node
     else
       forall successors s of node do
         s.determineHeuristicValue
         Queue.enqueue s
       end for
     end if
  end while
  return moveToReach(max(Queue.elementWithBestHeuristicValue,
                         BestTerminalNode.heuristicValue))
```

Der Heuristikwert für Terminalknoten entspricht genau dem Nutzen, den diese Knoten haben. Wenn beliebig viel Zeit zur Verfügung gestellt wird, leert sich die Warteschlange `Queue`, und der tatsächlich beste Terminalknoten ist in `BestTerminalNode` gespeichert. Die nächste Aktion, die ausgeführt werden muss, um einen bestimmten Zustand zu erreichen, wird von `moveToReach(node)` zurückgeliefert.

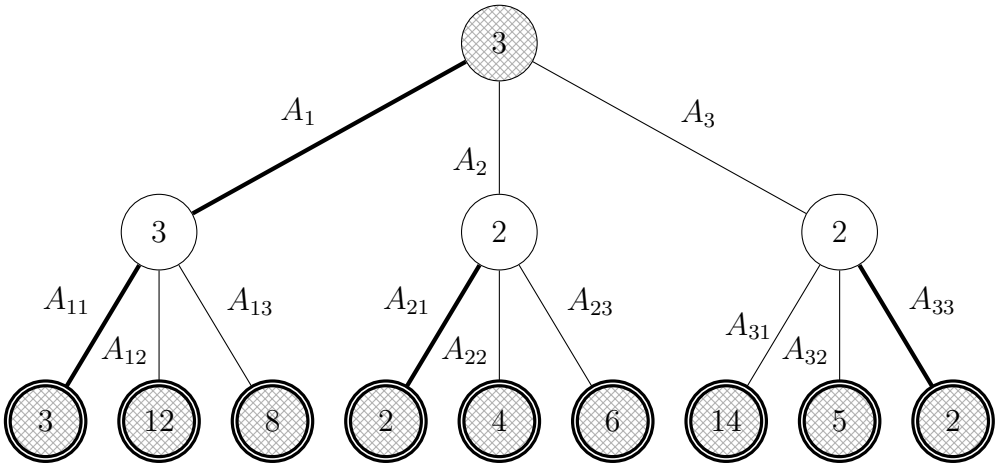


Abbildung 2: Minimax

### 3.2.2 Minimax für Zweipersonenspiele

Der Minimax-Algorithmus [RN95, Kap. 5] wurde entwickelt, um eine *optimale* Strategie in einem Zweipersonenspiel zu spielen. Der hier vorgestellte klassische Minimax-Algorithmus ist eigentlich ein Spezialfall für Nullsummenspiele. Die allgemeinere Version, angepasst auf die GGP-Problemstellung, wird im nächsten Kapitel vorgestellt. Minimax berechnet die optimale Strategie, *gegeben der Gegner spielt auch optimal*. Bei beschränkter zur Verfügung stehender Zeit ist letztere Annahme nicht immer berechtigt. Dennoch ist es interessant denjenigen Zug zu spielen, der bei optimaler Gegnerreaktion der beste wäre. Falls der Gegner von der optimalen Strategie abweicht, kann der so erreichte Nutzen nur steigen. Versucht man die Fehler des Gegners auszunutzen, wäre es möglich, dass der Gegner plötzlich von seiner unvorteilhaften Strategie abweicht, und zufällig doch den optimalen Zug macht. Dadurch könnte man einen schlechteren als den theoretisch erreichbaren Nutzen bekommen. Des Weiteren kostet die Analyse der gegnerischen Strategie auch Zeit, die man stattdessen damit verbringen könnte, den Suchbaum tiefer zu evaluieren.

Die Spieler des Minimax-Algorithmus ziehen abwechselnd. Am Ende wird der Ausgang des Spiels mit einem Nutzen bewertet. Einer der Spieler, genannt Max, möchte diese Punktzahl maximieren, wohingegen der Gegner, genannt Min, den gegnerischen Nutzen minimieren möchte.

Abbildung 2 zeigt ein sehr einfaches Spiel. Max ist hier der Spieler mit den grau schattierten Knoten, und Min der Spieler der weißen Knoten. Die Terminalknoten enthalten den Nutzen aus Sicht von Spieler Max. Die anderen Knoten enthalten den vom Minimax-Algorithmus be-

rechneten Wert. Die dicken Kanten markieren den jeweils optimalen Zug des aktiven Spielers. Die Aktionen von Spieler Max sind $A_1$, $A_2$ und $A_3$ und die Antworten von Min $A_{11}$ bis $A_{33}$.

Die einzelnen Schritte des Minimax-Algorithmus sind folgende:

- Generiere den gesamten Spielbaum bis zu den Terminalzuständen.
- Berechne den Nutzen jedes Terminalzustandes.
- Berechne von den Terminalzuständen ausgehend den Nutzen des darüberliegenden Knotens. Ist dieser Knoten ein Max-Knoten, wähle den höchsten Nutzen aller Kinder, im Falle eines Min-Knotens den kleinsten.
  *Im linken untersten Min-Knoten in Abb. 2 hat Min z.B. die Wahl zwischen $A_{11}, A_{12}$ und $A_{13}$. Durch Spielen der Aktion $A_{11}$ erreicht Min den minimalen Nutzen von* 3. *Man kann den Knoten demnach mit* 3 *bewerten, unter der Annahme, dass Min richtig spielen würde. Analog tragen die anderen beiden Min-Knoten den Nutzen* 2.
- Berechne solange den Nutzen der jeweils nächsthöheren Knotenschicht, bis man an der Wurzel ankommt.
- Wähle die Aktion, welche zu dem Nachfolgerknoten mit dem größten Nutzen führt.
  *In Abb. 2 ist das Aktion $A_1$, welche einen Nutzen von* 3 *garantiert. Auch in tieferen Ebenen ist die jeweils beste Aktion durch eine fette Kante gekennzeichnet.*

In dieser Form ist der Minimax-Algorithmus nur von theoretischem Interesse. Die Anzahl aller Stellungen eines Spiels ist exponentiell in der Tiefe des Spielbaums, was auch eine exponentielle Berechnungszeit mit sich bringt. Damit man den Minimax-Algorithmus praktisch nutzen kann, ist es nicht möglich das Spiel bis zu einem Terminalzustand zu berechnen. Stattdessen muss man auf Bewertungsfunktionen zurückgreifen. Dabei sollten der Schätzwert und der tatsächliche Nutzen bei optimalem Spiel beider Spieler stark korrelieren. Zudem müssen sie schnell zu berechnen sein. Um eine solche Heuristik konkret zu erstellen, wird normalerweise spielspezifisches Expertenwissen eingesetzt. Eine weit verbreitete gewichtete lineare Evaluationsfunktion (siehe Kap. 3.3) für das Schachspiel ist zum Beispiel die Materialbewertung in Bauerneinheiten: ein Bauer hat einen Wert von 1, Springer und Läufer je 3, ein Turm 5 und die Dame 9. Durch Summieren aller Materialwerte auf jeder Seite lässt sich die Qualität einer Stellung beurteilen. Ein Vorsprung von einem Bauern deutet auf eine stark erhöhte Gewinnwahrscheinlichkeit für den entsprechenden Spieler hin, ein Vorsprung von zwei Bauerneinheiten entspricht einem fast sicheren Sieg.

Heuristiken lassen sich auch für andere Spiele finden. Die Heuristiken für GGP können natürlich nicht mit spielspezifischem Wissen erstellt werden, sondern müssen, wie in Kapitel 3.3 vorgestellt, aus den Spielregeln hergeleitet werden.

Mit Hilfe einer Heuristik ist es jetzt möglich, die Minimax-Berechnung vorzeitig abzubrechen. Damit die zur Verfügung stehende Zeit voll ausgenutzt wird, und zudem jederzeit der beste bisher bekannte Zug zurückgeliefert werden kann, bietet sich iterative Tiefensuche an. Dabei wird die Minimax-Suche nach einer Tiefe $d$ abgebrochen, und für jeden Zustand dieser Tiefe werden die Heuristikfunktionen berechnet. Der Minimax-Baum wird wie im ursprünglichen Algorithmus berechnet, nur dass man die Suche nicht an den Terminalknoten, sondern an den heuristisch bewerteten Knoten beginnt. Steht noch mehr Suchzeit zur Verfügung, wird die Tiefe $d$ um 1 erhöht, und die heuristischen Schätzfunktionen der tiefer liegenden Ebene berechnet.

Eine häufig vorkommende Gefahr sei hier noch erwähnt. Nehmen wir zum Beispiel eine ausgewogene Stellung beim Schachspiel an, in der es einige Züge später unweigerlich dazu kommt, dass die Damen abgetauscht werden. Fällt jetzt das Schlagen der ersten Dame genau auf die Tiefe $d$, so wird die Stellung mit +9 Bauerneinheiten bewertet, obwohl sie ausgewogen ist, weil der Gegner im direkten Gegenzug problemlos die andere Dame schlagen kann. Stellungen, in denen keine extremen Umschwünge des Heuristikwerts vorkommen, heißen „ruhig“. In nichtruhigen Positionen muss zur Vermeidung dieser Probleme noch weitergesucht werden, bis eine ruhige Stellung erreicht wird.

### 3.2.3 GGP-Anpassung von Minimax

Im letzten Abschnitt wurde festgestellt, dass man den Minimax-Algorithmus mit einer Bewertungsfunktion ausstatten muss, um große Zustandsräume zu durchsuchen. Außerdem ist der Minimax-Algorithmus für sequenzielle Zweipersonenspiele entworfen. Somit kann er nicht ohne weiteres als Planungsmodul für ein GGP-Programm benutzt werden. Das Planungsmodul sollte sich allerdings wie ein Minimax-Algorithmus mit iterativer Tiefensuche verhalten, wenn die Voraussetzungen der Spielbeschreibung dies zulassen. Dabei wird das sogenannte $\alpha$-$\beta$-Pruning verwendet. Knoten des Minimaxbaumes, welche auf das Ergebnis keinen Einfluss haben, können vorzeitig abgeschnitten werden. Die genaue Funktionsweise ist z.B. in [RN95, Kapitel 5.4] nachzulesen. Eine weitere Verbesserung für Nicht-Nullsummenspiele und damit insbesondere für kooperative Spiele wird erreicht, indem man in den Min-Schritten nicht über die Knotenwerte minimiert, sondern stattdessen die Heuristik aus Sicht des Min-Spielers be-

rechnet, und deshalb auch hier maximiert. In Nullsummenspielen entspricht dies genau dem klassischen Minimax-Algorithmus.

Bei mehr als zwei Spielern kommt eine weitere Schwierigkeit hinzu. Angenommen, die Spieler $A$, $B$ und $C$ spielen ein Spiel, in dem unser Programm Spieler $A$ steuert, und $C$ am Zug ist. Die Heuristikwerte der Nachfolgezustände jedes Spielers seien $(-1, 3, 4)$ und $(3, -1, 4)$. Da beide Aktionen einen aus Sicht von $C$ mit 4 bewerteten Zustand erreichen, ist es nicht klar, welchen der beiden Züge ein optimal spielender $C$ wählen würde[4]. Ebenso stellen synchrone Spiele zusätzliche Probleme, die aber hier nicht weiter vertieft werden und deshalb nicht im Fokus dieser Arbeit stehen.

Es bleibt die Berechnung der Evaluierungsfunktion. Da kein spielspezifisches Wissen zur Verfügung steht, muss die Heuristik aus der Spielbeschreibung erstellt werden.

## 3.3 Heuristikmodul

Die Spiele des GGP sind in der Praxis nur in sehr einfachen Fällen komplett berechenbar. Schnell reicht die zur Verfügung stehende Zeit nicht mehr aus, um den optimalen Zug zu finden. Der Ausweg besteht darin, sich mit Funktionen zu behelfen, welche die Güte einer Spielstellung beurteilen können. Diese Funktionen müssen schnell berechenbar sein, und dem tatsächlichen Nutzen möglichst nahe kommen.

### 3.3.1 Evaluierungsfunktionen für Spiele

Bei vielen Spielen werden gewichtete lineare Evaluierungsfunktionen verwendet. Sie haben die Form $\sum_{i=1}^{n} w_i f_i$, wobei $w_i$ die Gewichte und $f_i$ spezielle Merkmale eines Spiels sind, welche über den Ausgang entscheiden[5]. Beim Schach verwendet man üblicherweise als $f$'s unter anderem die Anzahl der Spielfiguren des jeweiligen Typs. Die Gewichte $w$ sind dabei 1 für jeden Bauern, 5 für den Turm etc. Bisher ist es noch nicht praktikabel möglich, solche Merkmale automatisch zu generieren. Die Gewichte hingegen lassen sich durch vielmaliges Spielen des Computers gegen sich selbst recht gut selbständig erkennen.

Neben dem Erkennen von spielrelevanten Merkmalen bieten auch nicht-lineare Funktionen weiteren Raum, um Expertenwissen zur Heuristikgewinnung einzusetzen. Beim GGP hinge-

4 Lösungsansätze hierfür sind paranoider Minimax oder SoftMax
5 Cluneplayer [Clu07] ist ein GGP-Programm, was nach solchen Merkmalen sucht

gen ist es nicht möglich, auf Spezialwissen zurückzugreifen. Zudem sind die Spiele zeitbegrenzt, und selbst wenn man die Merkmale einer spielspezifischen Evaluierungsfunktion kennen würde, hätte man wohl kaum Zeit genug, um auch die Gewichte auszurechnen. Der klassische Ansatz zur Beurteilung von Spielstellungen ist in diesem Fall daher nicht praktikabel. Evaluierungsfunktionen werden auch in anderen Bereichen, in denen große Räume schnell durchsucht werden müssen, eingesetzt. Die allgemeine Herangehensweise ist die Suche eines ähnlichen, aber leichter zu lösenden Problems. Ist man zum Beispiel am kürzesten Weg von einer Stadt zur anderen interessiert, ist der tatsächliche Weg, den man auf einer Straße zurücklegen muss, nicht schnell berechenbar. Die Luftliniendistanz der Städte ist hingegen einfach zu berechnen und ein guter Schätzwert. Auch zwischen Zuständen einer logischen Beschreibung kann man mit Hilfe vereinfachter Probleme Schätzfunktionen finden und berechnen. Ein Zustand wird durch Literale beschrieben, die entweder wahr oder falsch sein können. Eine sehr einfache Schätzfunktion, die sogenannte Hammingdistanz [GNT04], zählt die Literale, deren Polarität sich im aktuellen Zustand von der im Terminalzustand unterscheidet. Der Hammingabstand $d$ zwischen zwei Literalbelegungen $K = k_1 \dots k_n$ und $L = l_1 \dots l_n$ ist $d = |i; k_i \neq l_i|$. Diese Schätzung ist allerdings zu grob, was an folgendem Beispiel deutlich wird:

**Beispiel 1** *Man betrachte das Literal des Zustandsraums, welches das nullstellige* `terminal`*-Prädikat repräsentiert. Der Hammingabstand von jedem nicht-Terminalzustand zu diesem Literal ist 1. Das Problem: tatsächlich müssen wahrscheinlich sehr viele Aktionen durchgeführt werden, damit ein Terminalzustand erreicht wird.*

In der Planungstheorie wurden in den letzten Jahren viele gute Heuristiken entwickelt, die hier herangezogen werden können. Ein Unterschied der Spiele zu klassischen Planungsproblemen ist allerdings die Anzahl der Ziele. Während Planer einen möglichst schnellen Weg zu *dem* Ziel suchen, gibt es beim GGP mehrere Ziele, die unterschiedlich erstrebenswert sind. Ein aktueller Forschungsbereich der Planungstheorie ist „planning with preferences“ [GM07], wobei man einzelne Teilziele gewichtet. Dieses Gebiet teilt sich in zwei Bereiche: Planen mit qualitativen Präferenzen und Planen mit quantitativen Präferenzen. Letzteres ähnelt der Aufgabenstellung des GGP, so dass man die Fortschritte in diesem Bereich im Auge behalten sollte.

Für eine GGP-Heuristik ist der Abstand zu einem Terminalzustand als Schätzwert alleine nicht ausreichend. Die Evaluierungsfunktion $f_{JO}$, welche in Kap. 3.4 vorgestellt wird, benötigt allerdings einen schnell berechenbaren Abstand, welcher durch eine Planungsheuristik berechnet werden kann. In dieser Arbeit wird dafür eine modifizierte Version der von Hoffmann und Nebel entwickelten $h_{FF}$-Heuristik [HN01] verwendet.

### 3.3.2 Heuristiken für Planungsprobleme

In diesem Kapitel werden Planungsheuristiken vorgestellt, um später (Kap. 3.4) damit Heuristiken für GGP zu entwickeln. Ein Planungsproblem $\mathcal{P} = \langle \mathcal{A}, \mathcal{I}, \mathcal{O}, \mathcal{G} \rangle$ besteht aus einer Menge von Zustandsvariablen $\mathcal{A}$, einem Initialzustand $\mathcal{I} : \mathcal{A} \to \{\top, \bot\}$, einer Menge von Regeln $\mathcal{O} = e \xleftarrow{c} p$ und einer Zielformel $\mathcal{G}$. Analog zu den Spieleraktionen beim GGP gibt es in der Planungstheorie Operatoren. Diese sind Regeln $\mathcal{O}$ mit Vorbedingungen $p$, Effekten $e$ und Operatorkosten $c \in \mathbb{R}_{\geqslant 0}$. Die Vorbedingung $p$ ist dabei eine Formel, die erfüllt sein muss, um den Operator anzuwenden. Die Effekte $e$ sind eine Konjunktion der Literale, die bei Anwendung des Operators wahr gemacht werden.

Durch das Lösen ähnlicher vereinfachter Probleme (siehe Kap. 3.3.1) lassen sich Heuristiken entwerfen. Bei Planungsheuristiken ist es hierfür sinnvoll, die Repräsentation des Problems in positive Normalform zu bringen, was in polynomieller Zeit möglich ist. In positiver Normalform bestehen sowohl die Vorbedingungen der Operatoren als auch der Terminalzustand nur aus positiven Literalen. Zur Umwandlung in positive Normalform wird für jedes negativ vorkommende Literal eine neue Variable erzeugt. Zum Beispiel würde das Literal $\neg$`locked` durch ein neues Literal `unlocked` ersetzt. In allen Effekten und im Initialzustand wird jedes Vorkommen dieses Literals (im Beispiel `locked`) durch das Literalpaar aus altem und neuem Literal ersetzt (`locked` wird zu `locked` $\wedge\neg$`unlocked`).

Ein vereinfachtes Problem ergibt sich jetzt, wenn man alle negativen Operatoreffekte ignoriert. In diesem relaxierten Problem ist ein Zustand besser als ein anderer, wenn in ihm zusätzliche Literale wahr sind. So sind mehr Operatoren anwendbar oder mehr Literale des Zielzustands erreicht. In diesem vereinfachten Problem ist es leicht zu sehen, ob ein Operator nützlich ist, indem man prüft, ob er bisher falsche Literale erreicht. Ein sehr einfacher Algorithmus mit polynomieller Laufzeit kann solange beliebige Operatoren anwenden, welche neue Literale wahr machen, bis ein Zielzustand erreicht ist. Falls dies nicht gelingt, gibt es auch im ursprünglichen Problem keinen Weg vom aktuellen Zustand zum Ziel.

Um für das ursprüngliche Problem einen Schätzwert zu erhalten, kann man beispielsweise die Anzahl der Operatoren zählen, die im relaxierten Plan angewendet wurden. Die Anzahl der von diesem einfachen Algorithmus gefundenen Operatoren kann allerdings stark variieren, was damit zusammenhängt, dass die Operatoren mehrere Effekte haben. Die Suche eines *optimalen* relaxierten Plans ist ein NP-schweres Problem [Byl94], und falls P$\neq$NP, nicht effizient berechenbar.

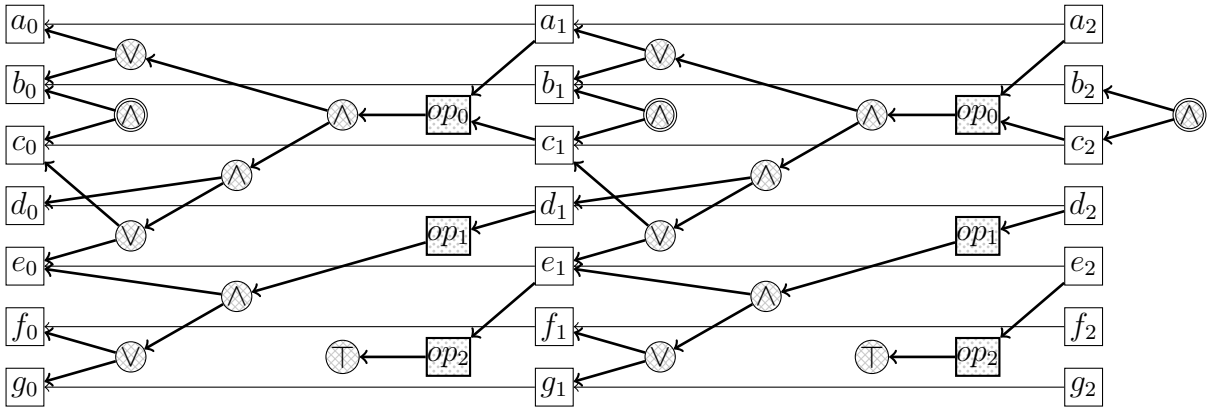


Abbildung 3: Graph eines parallelen Plans

Um bessere Heuristiken zu erstellen, die aber trotzdem in polynomieller Zeit lösbar sind, kann man parallele relaxierte Pläne betrachten. In jedem Schritt werden dabei alle anwendbaren Operatoren gleichzeitig angewendet. Die Heuristiken hängen dadurch nicht mehr von der Reihenfolge der Operatoranwendung ab, und die starken Schwankungen der Schätzwerte werden reduziert.

### 3.3.3 Parallele Pläne

In parallelen Plänen werden im Allgemeinen mehrere Operatoren gleichzeitig angewendet. Die Zustandsvariablen ändern sich in jedem Zeitschritt und bilden so eine Schichtung. Abbildung 3 zeigt den Graph des folgenden parallelen Plans:

**Beispiel 2** *Ein Planungsproblem bestehend aus den Operatoren* $op_0$, $op_1$ *und* $op_2$ *mit*

$$op_0 = \langle (a \vee b) \wedge (d \wedge (c \vee e)), a \wedge c \rangle$$
$$op_1 = \langle e \wedge (f \vee g), d \rangle$$
$$op_2 = \langle -, e \rangle$$

*und Ziel* $t = b \vee c$.

Die Tiefe des parallelen Plans, also die Schicht, in welcher der Zielzustand erstmals durch die Zustandsvariablen erfüllt ist, ist eine zulässige Heuristik, welche $h_{max}$ genannt wird. Da in jedem Schritt aber mehrere Operatoren angewendet werden können, unterschätzt sie die tatsächlichen Kosten zu sehr, um informativ zu sein. Eine andere Heuristik, welche sich aus

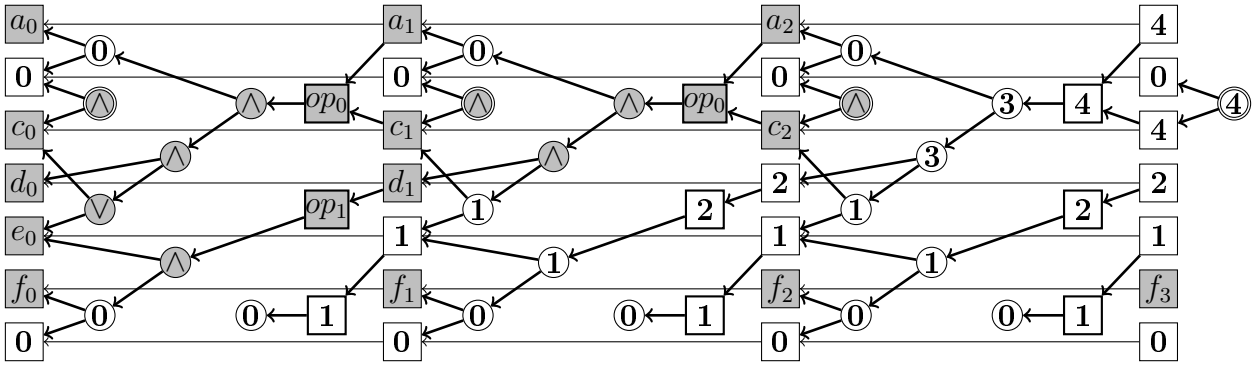


Abbildung 4: Berechnung von $h_{add}$

dem parallelen Plan ergibt, ist $h_{add}$. Diese ergibt sich als Summe der Kosten aller Operatoranwendungen, die ausgeführt werden müssen, um den Zielzustand des parallelen Plans zu erreichen. Um diese Summe zu berechnen, kann man das Planungsproblem wie einen logischen Schaltplan auffassen. Die Kosten jedes Knotens des Initialzustands sind 0, und jeder Operator erhöht die Kosten um 1. Ein Konjunktionsknoten summiert die Kosten der beiden Vorgänger, ein Disjunktionsknoten bildet das Minimum. Die von $h_{add}$ berechneten Werte für Beispiel 2 mit Operatorkosten von 1 zeigt Abbildung 4.

Diese Heuristik überschätzt die tatsächlichen Kosten allerdings zu stark. Für jedes von einem Operator erreichte Teilziel wird dieser für die Heuristikberechnung gezählt, obwohl er unter Umständen nur einmal angewendet werden müsste. In Beispiel 2 macht die Anwendung von $\mathtt{op}_2$ das Literal $e$ wahr, welches eine Vorbedingung von $\mathtt{op}_0$ sowie von $\mathtt{op}_1$ ist. Die Kosten zum Erreichen von $e$ werden doppelt gezählt, obwohl $\mathtt{op}_2$ nur einmal angewendet werden muss. Der Heuristikwert im obigen Beispiel ist 4, obwohl nur 3 Operatoren angewendet werden müssen. Der Mittelweg zwischen diesen beiden ist die im Folgenden beschriebene $h_{FF}$-Heuristik, welche genau dieses Mehrfachzählen zu umgehen versucht.

### 3.3.4 FF-Heuristik

Die $h_{FF}$-Heuristik, Herzstück des erfolgreichsten Planers der International Planning Competition 2000 [HN01], gibt eine Schätzung für den Abstand zwischen aktuellem Zustand und Zielzustand an. Wie auch die weiter oben vorgestellte Heuristik $h_{add}$ arbeitet $h_{FF}$ auf relaxierten Plänen. Allerdings werden die Operatoren nicht als unabhängig angenommen.

Zur Berechnung von $h_{FF}$ wird in einem Vorwärtsschritt ein relaxierter Plan aufgebaut. In einem Rückwärtsschritt werden in diesem Plan dann nur die Operatoren gezählt, die auch

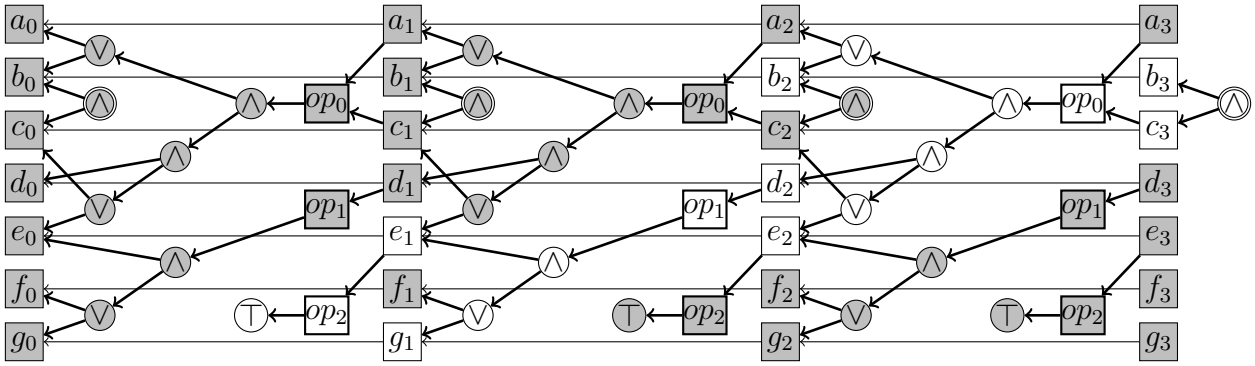


Abbildung 5: Subgraph des von $h_{FF}$ gefundenen Plans (weiße Knoten)

angewendet werden müssen, um das Ziel zu erreichen. Abbildung 5 zeigt den im Rückwärtsschritt extrahierten Subgraphen aus Beispiel 2. Es kann allerdings vorkommen, dass ein Teilziel auf verschiedenen Wegen erreicht werden kann. Je nachdem, auf welchem Weg man die Operatoren zählt, ergeben sich unterschiedliche Heuristik-Werte. Das ist auch der Grund dafür, dass die $h_{FF}$-Heuristik nicht zulässig ist. Allerdings ist es wiederum ein NP-schweres Problem [Byl94], einen *optimalen* relaxierten Plan zu finden. Deshalb behilft man sich mit einer Heuristik innerhalb der Heuristik:

- Persistenz-Aktionen zuerst: Damit keine Operatoren angewendet werden, um bereits wahre Fakten nochmal wahr zu machen, wird zunächst geprüft, ob das entsprechende Literal schon im letzten Schritt erfüllt war.
- Operatoren mit wenigen Vorbedingungen sind im nicht-relaxierten Plan leichter anwendbar als solche mit vielen Vorbedingungen. Deshalb werden sie bevorzugt gewählt.

Die $h_{FF}$-Heuristik extrahiert einen Plan, um das gesuchte Ziel zu erreichen. Der Weg dorthin führt über einen der in der ersten Schicht angewendeten Operatoren. Der FF-Planer kann so schnell eine dieser Aktionen ausführen. Ob diese hilfreichen Aktionen auch für GGP nutzbar sind, ist aber eher fraglich, da die Heuristikwerte für die Zustände nach der Anwendung einer Aktion berechnet werden. Inwieweit sich die $h_{FF}$-Heuristik als Abstandsfunktion in Spielen verwenden lässt, wird im folgenden Kapitel behandelt.

## 3.4 Die Evaluierungsfunktion $f_{JO}$

Um allgemeine Spiele mit Hilfe der $h_{FF}$-Heuristik zu lösen, sind einige Anpassungen nötig. Beim GGP gibt es verschiedene, unterschiedlich erstrebenswerte, Ziele. Der kürzeste Abstand zum Ziel kann beispielsweise in einen Terminalzustand mit einem geringen Nutzen (in dem

man das Spiel verliert) führen. Der Nutzen in einem Terminalzustand ist für jeden Spieler unterschiedlich. Die Evaluierungsfunktion $f_{JO}$ wird aus Sicht eines Spielers berechnet. Beim GGP liegen die Nutzen zwischen 0 und 100. Wie später deutlich wird, ist für uns ein Nutzen von $-50$ bis 50 vorteilhafter, damit sich verlorene und gewonnene Spielstellung bereits aus dem Vorzeichen ersehen lassen. Vom den in Kapitel 2.2 beschriebenen Gewinnwerten $g(s)$ werden hier 50 subtrahiert.

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

Dabei ist $s$ der Zustand, dessen Nutzen geschätzt werden soll. Die Terminalzustände $t$ aus $T$ werden nach ihren Gewinnen $q = g(t)$ gruppiert. $T_{100}$ ist z.B. die Menge aller Terminalzustände, in denen der Spieler einen Nutzen von 100 erhält. Für die Formel muss sichergestellt sein, dass der Abstand zu einem Terminalzustand nicht mit 0 geschätzt wird (Division durch 0). Zudem sollten nicht erreichbare Ziele, wie Zielformeln $T_q$, deren Nutzen $q$ nicht in der Spielbeschreibung vorkommen, einen Abstand von unendlich haben. Wenn $\frac{1}{\infty} = 0$, fließen nicht-erreichbare Ziele auch nicht in die Summe ein. In einer praktischen Implementierung ist es möglicherweise besser, über die tatsächlich vorkommenden Nutzenwerte $q$ zu summieren.

Die Abstandsfunktion $\texttt{dist}(s, T_q)$ kann entweder tatsächlich ausgerechnet oder mit Hilfe einer Heuristik geschätzt werden. In dieser Arbeit wird eine modifizierte $h_{FF}$-Heuristik verwendet. Je näher ein Zustand an einem Ziel ist, desto größer wird der Einfluss, den dieses Ziel auf den Zustand hat. Deswegen fließt der Kehrwert des Abstandes in die Heuristik ein. Der andere Faktor, der Nutzen des Zustands, wird normalisiert. Damit ein langer, aber gefahrloser Weg zu einem positiven Ziel einen Einfluss auf die Heuristik hat, muss der geschätzte Nutzen negativ für verlorene Stellungen und positiv für gewonnene sein. Nach maximal 101 Heuristik-Aufrufen ist die Güte eines Zustands beurteilt.

## 3.5 Übersetzung von GDL nach FF

Eine GDL-Beschreibung ist eine Menge von GDL-Datalogregeln. Die Komponenten eines Planungsproblem $\mathcal{P}$ (siehe Kap. 3.3.2) werden in GDL mit den Datalogregeln beschrieben. Diese enthalten allerdings im Normalfall Variablen. FF arbeitet aber auf einer aussagenlogischen Ebene. Deshalb müssen die GDL-Regeln zunächst in eine variablenfreie Form umgewandelt und danach übersetzt werden.

Als „Grounding“ bezeichnet man den Vorgang, Variablen durch die von ihnen repräsentierten Objekte zu ersetzen. Kommen in einer Datalogregel mehrere Variablen vor, dann vergrößert sich dadurch die Regelmenge der neuen, „gegroundeten“ Datalogregeln exponentiell, da für jede Kombination von Variablenbelegungen eine eigene Regel erstellt werden muss. Dies lässt sich zwar theoretisch nicht vermeiden, allerdings ist auch die Größe der Basis entscheidend. Die Anzahl der für die Variablen eingesetzten Objekte sollte demnach so klein wie möglich sein, um nur diejenigen Regeln zu erzeugen, die auch tatsächlich vorkommen können. In den Kapiteln 4.2 und 4.3 werden verschiedene Wege vorgestellt, um gegroundete Datalogregeln zu erhalten. Regeln, in deren Vorbedingung eine Gleichheitsverletzung mit „`distinct`“ auftritt, werden nicht in FF-Regeln umgewandelt. Andernfalls kann „`distinct`“ entfernt werden.

Sei im Folgenden $\widehat{\Delta}$ eine Menge gegroundeter Datalogregeln $\widehat{h} \leftarrow \widehat{b_1} \wedge \cdots \wedge \widehat{b_n}$. Die Übersetzung der Datalogregeln in ein Planungsproblem $\tau(\widehat{\Delta}) \rightarrow \mathcal{P}$ wird im Folgenden beschrieben.

### 3.5.1 Zustandsvariablen

Um von Prädikatenlogik erster Stufe zu Aussagenlogik überzugehen, werden den Propositionen von GDL Aussagenvariablen zugeordnet. Diese entsprechen den Grundatomen, die im minimalen Modell der variablenfreien Datalogregeln $\widehat{\Delta}$ vorkommen. Die erreichbaren Grundatome entsprechen genau den Köpfen $\widehat{h}$ der erfüllbaren Datalogregeln. Allerdings werden in GDL auch Initialzustand, Zielformeln und Spieleraktionen durch Datalogregeln beschrieben, so dass es nicht unbedingt sinnvoll ist, *alle* GDL-Propositionen direkt den Zustandsvariablen $\mathcal{A}$ von FF zuzuordnen. Die Atome mit vordefiniertem Relationsbezeichner müssen genauer untersucht werden. Sowohl „`init`“ als auch „`next`“ sowie „`true`“ machen Aussagen über die Gültigkeit von Zuständen, sollten also in $\mathcal{A}$ repräsentiert werden. Auch „`legal`“- und „`does`“-Relationen können als FF-Zustandsvariablen erstellt werden, was die Umsetzung von Datalogregeln $\widehat{\Delta}$ zu FF-Regeln $\mathcal{O}$ vereinfacht (Kap. 3.5.3). Die vordefinierten Relationen „`role`“ und „`terminal`“ können auch als Zustandsvariablen eingeführt werden. „`goal`“-Atome beschreiben dagegen keine Zustände, sondern sind Metainformationen des Spiels.

Um den Zustandsraum möglichst klein zu halten, können die drei Relationen „`init`“, „`next`“ und „`true`“ zusammengefasst werden. Ebenso gilt dies für „`legal`“ und „`does`“. Da ein Zustandsraum beschreibt, was gerade gilt oder gerade gemacht wird, werden die Zustandsvariablen hier nach „`true`“ und „`does`“ benannt. In einer tatsächlichen Implementierung sind Zustandsvariablen allerdings eine Identifikationsnummer mit einem Wahrheitswert.

**Beispiel 3** *Den GDL-Atomen werden Zustandsvariablen zugeordnet.*

- *(init (cell a 1 blank))* *wird zu* *true_cell_a_1_blank*
- *(true (cell b 2 x))* *wird zu* *true_cell_b_2_x*
- *(next (on b c))* *wird zu* *true_on_b_c*
- *(legal robot move)* *wird zu* *does_robot_move*
- *(does xplayer (mark a 1))* *wird zu* *does_xplayer_mark_a_1*
- *(terminal)* *wird zu* *terminal*
- *(adjacent a b)* *wird zu* *adjacent_a_b*

### 3.5.2 Initialzustand

Der Initialzustand $\mathcal{I}$ des Planungsproblems ist eine Belegung der Variablen aus $\mathcal{A}$. Hier werden alle Zustandsvariablen, die für statischen Prädikate stehen (z.B. `adjacent_a_b`) sowie alle Zustandsvariablen, die für `role`- und `init`-Relationen stehen, mit $\top$, alle anderen mit $\bot$ initialisiert.

### 3.5.3 Operatoren und Axiome

Die meisten Datalogregeln haben die Form $\widehat{h} \leftarrow \widehat{b_1} \wedge \cdots \wedge \widehat{b_n}$, wobei $\widehat{b_i} = \widehat{l_1} \vee \cdots \vee \widehat{l_m}$, und die Literale $\widehat{l_k}$ sind entweder Datalogatome $\widehat{d}$ oder deren mit dem logischen Operator `not` negierte Form. Die Datalogregeln sollen durch FF-Regeln $\mathcal{O}$ der Form $e \overset{c}{\leftarrow} p$ dargestellt werden. Eine Ausnahme bilden `goal`-Regeln, die nur für die Erstellung der Zielformel benötigt werden. Alle anderen Datalogregeln lassen sich einem der folgenden vier Typen zuordnen:

1. `legal`-Regeln stehen für die legalen Züge eines Spielers.
2. `next`-Regeln mit `does` beschreiben die Effekte der Spieleraktionen.
3. `next`-Regeln ohne `does` werden für dynamische Änderungen der Welt verwendet.
4. Abgeleitete Prädikate, also Regeln, deren Kopf keine der vordefinierten Relationen enthält.

Die Propositionen der Regeln, die übersetzt werden sollen, sind alle als Zustandsvariablen vorhanden. So lassen sich diese vier Regeln recht direkt übertragen.

- Den Literalen $\widehat{l_k}$ werden Literale $z$ über $\mathcal{A}$ zugeordnet.

  $d \quad \rightarrow \quad z$

  `(not` $d$ `)` $\rightarrow \quad \neg z$

- Die Rumpfformel $\widehat{b}$ wird entsprechend übersetzt; dabei wird „`or`“ zu „$\vee$“ und die implizite Konjunktion zu „$\wedge$“.

  $\widehat{b} \quad \rightarrow \quad p$

- Der Kopf $h$ setzt die entsprechende Variable wahr.

  $\widehat{h} \quad \rightarrow \quad e$

- Die Kosten $c$ werden je nach Regeltyp angepasst.

**Beispiel 4** *Umsetzung einer Datalogregel in $\widehat{\Delta}$ zu einer FF-Regel in $\mathcal{O}$*

- ```
  (<= (legal xplayer (mark a 1))
      (true (control xplayer))
      (not (true (cell a 1 o)))
  )
  ```
  *wird zu* $does\_xplayer\_mark\_a\_1 \xleftarrow{1} true\_control\_xplayer \wedge \neg true\_cell\_a\_1\_o$

- ```
  (<= (next (cell a 1 x))
      (or (does xplayer (mark a 1))
          (true (cell a 1 x))
      )
  )
  ```
  *wird zu* $true\_cell\_a\_1\_x \xleftarrow{0} does\_xplayer\_mark\_a\_1 \vee true\_cell\_a\_1\_x$

In der klassischen $h_{FF}$-Heuristik werden durch $\mathcal{O}$ Operatoren beschrieben. Die Kosten der Regelanwendungen $c$ sind aber nicht notwendigerweise 1, und können nach Bedarf angepasst werden. Für GGP müssen diese Kosten je nach Regeltyp unterschiedlich sein, damit die Datalogregeln direkt umgewandelt werden können.

Datalogregeln von Typ 1, also die legalen Spieleraktionen, entsprechen den Vorbedingungen der klassischen Operatoren von FF. Der Effekt ist hier eine atomare Zustandsvariable, da die

Aktionseffekte durch `next`-Relationen beschrieben werden. Ein Operator in der klassischen Planung hat gewöhnlich Kosten von 1. Um diese auf GDL zu übertragen, müssen die Kosten von `legal` und `does` zusammen 1 ergeben. Da auf eine legale Aktion mehrere `does`-Effekte kommen können, werden die Kosten $c$ der FF-Regel hier mit 1 gewählt. Für Regelkosten von Aktionen des Gegners sowie für „`noop`"-Aktionen können je nach Implementierung allerdings durchaus Kosten von 0 sinnvoll sein. Der Abstand zu einem Zustand soll der Anzahl der Spielzüge entsprechen, die es dauert, bis dieser Zustand erreicht werden kann. Die „`noop`"-Aktionen dienen aber eher zur Realisierung synchroner Züge, weswegen sich hier Kosten von 0 anbieten.

Regeln von Typ 2 beschreiben die Effekte, welche Spieleraktionen haben. Um eine Aktions-Zustandsvariable wahr zu machen, müssen die Kosten bereits von einer Regel des ersten Typs bezahlt werden. Aus diesem Grund haben Typ-2-Regeln Kosten von 0. Auch alternative Aufteilungen der FF-Operatorkosten auf Typ-1- und Typ-2-Regeln sind denkbar.

Dynamische Änderungen der Welt, wie beispielsweise das Hochzählen des Spielzugs, sind je nach Implementierung verschieden zu behandeln. Wenn sichergestellt ist, dass in jedem Heuristik-Schritt jede der Regeln aus $\mathcal{O}$ nur einmal angewendet wird, ist es möglich, die Kosten mit 0 zu bewerten. Andernfalls können diese Regeln 1 kosten, womit sichergestellt ist, dass z.B. in Spielen, in welchen nach einer bestimmten Spielzugzahl das Spiel beendet ist, der Abstand zum Ziel nicht auf Grund von Typ-3-Regeln mit 0 geschätzt wird.

Der letzte Typ, die statischen Axiome, sind abgeleitete Prädikate, die die Sprache vereinfachen. Die Kosten sind hier mit 0 zu wählen.

### 3.5.4 Zielformel

Die Zielformel $\mathcal{G}$ der FF-Heuristik wird aus `terminal`- und `goal`-Relationen erstellt. Der Rumpf der Datalogregeln besteht aus Literalen oder `or`-Operatoren. Somit liegt der Rumpf bereits in konjunktiver Normalform vor. Durch die Einführung einer Zustandsvariable für die `terminal`-Relation ist es möglich, die Zielformeln für die unterschiedlichen Nutzen zu erhalten, indem man der `goal`-Zielformel ein Konjunktionsglied „`terminal`" anhängt.

Die Zielformeln entsprechen den $T_q$ aus Kapitel 3.4, wobei der Spieler und der Nutzen $q$ im Kopf der `goal`-Relation zu finden sind.

Diese Übersetzung der GDL in das Planungsproblem $\mathcal{P}$ kann jetzt in positive Normalform überführt werden, um damit die Heuristikwerte zu berechnen. Die unterschiedlichen Kosten der Regeln bieten dabei einen Spielraum, um die Heuristikergebnisse zu verbessern.

# 4 Implementierung

Im letzten Kapitel wurden theoretische Vorüberlegung zur Erstellung eines GGP-Programms, welches Spiele mit Hilfe von heuristischer Suche lösen kann, getroffen. Einige theoretische Probleme wurden erst als solche identifiziert, als ein scheinbar einfacherer Ansatz der Implementierung an seine Grenzen stieß. Im Folgenden werden diese Probleme sowie die aktuell verwendete Implementierung beschrieben.

## 4.1 Vorgehen und Überblick

Als Grundlage für das GGP-Programm dient die bereits verfügbare Implementierung der Stanford University, „Jocular“. Diese Java-Implementierung stellt bereits die Kommunikation mit dem Game Master (GM) zur Verfügung. Auch ist ein Parser vorhanden, der viele Metainformationen bereitstellt. Als Anlehnung daran wurde „CoolerJo“ entwickelt. Im Verlauf der Programmierung entfernte sich CoolerJo jedoch von Jocular.

In Kapitel 3.5 wurde bereits gezeigt, wie von GDL aus eine aussagenlogische Repräsentation des Problems erreicht werden kann. Für die FF-Heuristik selbst bestehen schon Implementierungen, die verwendet werden können. Die Übersetzung der GDL-Regeln stellt deshalb einen großen Teil des Programmieraufwandes dar.

Während der Implementierung taten sich einige Möglichkeiten auf, die Performanz des Programms zu verbessern. Für diese Bachelorarbeit waren allerdings zunächst die theoretischen Ergebnisse interessant. Mit der erworbenen Expertise kann in Zukunft eine wettbewerbstaugliche Version in einer performanteren Programmiersprache (etwa C++) benutzt werden.

CoolerJo baut zunächst einen eigenen Parser auf, um die Informationen aus der Spielbeschreibung zu extrahieren. Der Parser, welcher viele Metainformationen (Spieleranzahl und Namen, sowie Informationen über Initial- und Zielzustände) speichert, führt nacheinander folgende Schritte aus:

1. Erstellen der Zustandsvariablen (Kap. 4.2)
2. Erstellen von Initialzustand und den Terminal-Zuständen (Kap. 4.4)
3. Erstellen der Operatoren und Axiome (Kap. 4.5)
4. Initialisieren der Heuristik
5. Start des Planers (Kap. 4.6)

## 4.2 Erstellen der Zustandsvariablen

Die Zustandsbeschreibung eines Spiels lässt sich aus den gegroundeten Datalogregeln bestimmen (Kap. 3.5.1). Das Grounding einer Spielbeschreibung führt zu einer exponentiellen Vergrößerung, wenn verschiedene Variablen in einer Datalogregel vorkommen. Allerdings kann ein Großteil dieser Zustände niemals erreicht werden. Im Idealfall werden nur die Zustandsvariablen erstellt, welche im Verlauf eines Spiels tatsächlich benötigt werden können.

CoolerJo wandelt zunächst die KIF-Datei in eine Baumstruktur um. Auf einer semantischen Ebene sind sowohl Funktionen wie `(cell a 1 blank)` als auch Relationen wie `(adjacent a b)` Prädikate, die eine Zustandsinformation tragen. Durch Analyse der Domänen dieser Zustandsvariablen kann ein Großteil der nicht benötigten Variablen erkannt werden. In Kapitel 4.3 wird vorgestellt, wie man die syntaktischen Unterschiede zwischen Relationen und Funktionen dazu nutzen kann, einen noch kompakteren Zustandsraum zu erzeugen.

CoolerJo erkennt in seiner Erreichbarkeitsanalyse die Domänen der Prädikate. Die Domäne des Prädikats `cell`/3 aus Tic-Tac-Toe ist z.B. `cell: [a, b, c] [1, 2, 3] [blank, x, o]`, wodurch 27 Zustandsvariablen erzeugt werden.

Die Domänenanalyse ist eine Fixpunktiteration, bei welcher den Prädikaten zunächst die Domänen der statischen Fakten und `init`-Relationen zugewiesen werden. Dann werden die Datalogregeln iterativ nach Variablen durchsucht. Den Domänen der Kopf-Prädikate der Datalogregeln werden dabei die Domänen der Rumpf-Prädikate zugeordnet, in welchen sich Variablenpartner befinden. So würde beispielsweise `(<= (row ?x ?t) (true (cell ?x 1 ?t)) ...)` die Domäne von `row`/2 auf `row: [a, b, c] [blank, x, o]` vergrößern, wenn die Domäne von `cell` bereits der aus dem vorherigen Beispiel entspricht.

Nachdem sich keine Domäne mehr erweitert hat, werden die FF-Zustandsvariablen erstellt. Auf diese Art werden allerdings auch Variablen wie z.B. `successor_5_3` erzeugt, was im Weiteren auch die Anzahl der FF-Regeln $\mathcal{O}$ vergrößert. Für das Resultat der Heuristikberechnung spielen diese überflüssigen Variablen allerdings keine Rolle, da sie niemals wahr werden können. Für ein performanteres Programm ist allerdings die Erreichbarkeitsanalyse des nächsten Abschnitts 4.3 vorzuziehen.

## 4.3 Syntax-basierte Erreichbarkeitsanalyse

Um den Zustandsraum zu erstellen, kann man auch die Syntax von Datalog heranziehen. Dadurch werden sehr viele unerreichbare Zustandsvariablen, wie z.B. `successor_2_5`, erst gar nicht erstellt. Im Gegensatz zu der Erreichbarkeitsanalyse aus Kap. 4.2 werden nicht die Domänen der Prädikate, sondern die tatsächlich erreichbaren Grundatome gesucht. Zunächst wird die GDL-Beschreibung nach Grundausdrücken und Regeln mit Variablen sortiert. Die Liste der Grundausdrücke enthält `init`-Regeln und statische Fakten (Grundatome) wie z.B. `(adjacent a b)`. Diese werden nacheinander mit der Liste der Regeln mit Variablen abgeglichen. Kann ein Atom in eine Regel eingesetzt werden, wird diese kopiert. Falls in ihr nach der Ersetzung weitere Variablen enthalten sind, wird die neue Regel der Liste der Regeln hinzugefügt. Andernfalls wird der jetzt variablenfreie Kopf der Regel der Liste der Grundausdrücke angehängt.

Nachdem ein Grundausdruck überall eingesetzt wurde, kann man ihn aus der Liste der Grundausdrücke entfernen und für ihn eine Zustandsvariable erzeugen. Dieser Prozess wird für jedes Grundatom durchgeführt. Da in GDL rekursive Regeln beschränkt sind, terminiert er auch. Das Ergebnis sind genau die Zustandsvariablen, die durch die Anwendung der Datalogregeln wahr werden können. Bei Berücksichtigung der `distinct`-Relation werden so ausschließlich syntaktisch erreichbare Zustände erstellt.

## 4.4 Initial- und Terminalzustand

Die Zustandsvariablen sind Boolesche Variablen. Um sich Zustandsraum-Optimierungen mit mehrwertigen Variablen offen zu halten, werden sie als Ganzzahlen gespeichert. Den Prädikatknoten des Parsebaums werden Zustandsvariablen, die durch eine Identifikationsnummer repräsentiert sind, zugeordnet. Die Zustandsvariablen aller Prädikatsknoten, die als statische Fakten oder als `init`-Prädikate vorkommen, werden mit 1 initialisiert. Alle anderen Zustandsvariablen werden auf 0 gesetzt.

Etwas schwieriger gestaltet sich das Erkennen der Zielzustände. Eine Zielformel ist ein Tripel bestehend aus Spieler, Nutzen und einer Formel, welche genau für die Zustände erfüllt ist, die Terminalzustände mit dem entsprechenden Nutzen für den Spieler sind. In CoolerJo werden die Terminalformeln in disjunktive Normalform (DNF) umgewandelt. Dabei wurde `terminal` nicht als Zustandsvariable initialisiert, was dazu führt, dass die Zielformeln $T_q$ aus `goal`- und terminal-Relationen erstellt werden. Der Vorteil dabei ist der, dass widersprüchliche Literale

$(A \wedge \neg A)$ vorzeitig entfernt werden. Die Umwandlung in DNF ist exponentiell in Zeit und Platz. Darum besteht die Gefahr, dass die Berechnung der Zielformel lange dauert.

## 4.5 Operatoren und Axiome

Die Spieleraktionen des GGP entsprechen den Operatoren beim Planen. Mit Hilfe der unterschiedlichen Regeltypen aus Kapitel 3.5.3 ist es möglich, eine Datalog-nahe Regelerstellung zu erreichen. Die hier gewählten Kosten sind 1 für Spieleraktionen, 0 für „noop"-Aktionen, 0 für Spielereffektaxiome sowie für abgeleitete Regeln, und 1 für dynamische-Welt Axiome.

## 4.6 Der Planer

Die Aufgaben des Planungsmoduls wurden bereits in Kapitel 3.2 beschrieben. CoolerJo zählt die `role`-Relationen, um die Spieleranzahl zu ermitteln. Einpersonenspiele werden mit der Bestensuche aus Kapitel 3.2.1 gelöst. Zweispielerspiele verwenden eine Variation des Minimax-Algorithmus, wobei aber in jedem Schritt die Heuristik aus Sicht des aktiven Spielers berechnet wird.

Das Management der zur Verfügung stehenden Restzeit sowie die Anbindung an den GM regeln die von Jocular zur Verfügung gestellten Schnittstellen. Spiele mit drei oder mehr Spielern sowie synchrone Spiele werden nicht unterstützt.

# 5 Experimente

Im letzten Kapitel wurde das Programm „CoolerJo“ vorgestellt, mit welchem allgemeine Spiele gelöst werden sollen. Hier werden die Ergebnisse beschrieben, die „CoolerJo“ beim Spielen einiger Spiele der vergangenen GGP-Wettbewerbe erreicht hat. Die Spiele, die getestet wurden, sind:

1. Maze (1 Spieler): Ein Roboter startet in Feld $A$ (Abb. 6) und muss das Gold aus Feld $C$ zurück zum Eingang $A$ bringen. Zur Verfügung stehen die Aktionen „`move`“ (ein Schritt im Uhrzeigersinn), „`grab`“ und „`drop`“.
2. Simple Blocksworld (1 Spieler): Aus den Blöcken links (Abb. 7) soll der Turm rechts gebaut werden.
3. Tic-Tac-Toe (2 Spieler): Ziel dieses Spiels (Abb. 8) ist es eine Reihe aus X oder O zu erhalten.
4. Minichess: (2 Spieler): Auf einem $4 \times 4$ Schachbrett (Abb. 9) versucht Weiß, mit Turm und König den schwarzen Spieler mit König in 10 Zügen matt zu setzen.

In den folgenden Abschnitten werden die Spiele genauer betrachtet.

## 5.1 Maze

Maze (Abb. 6) ist ein sehr einfaches Einpersonenspiel. Da das Problem so klein ist, kann der Spielbaum in der zur Verfügung stehenden Zeit bis zu den Terminalknoten berechnet werden, und die richtige Lösung [`move`, `move`, `grab`, `move`, `move`, `drop`] wird gespielt. Interessant sind aber auch die Werte, welche $f_{JO}$ auf dem Weg zu diesem Ziel berechnet.

Der Abstand zum Ziel mit Nutzen 50 wird von $h_{FF}$ anfangs mit 3 geschätzt. Der relaxierte Plan ist dabei [`move`, `grab`, `drop`]. Da eine einmal wahr gemachte Variable im relaxierten Plan nie mehr falsch gemacht werden kann, muss „`move`“ nur einmal angewendet (und somit bezahlt) werden, um alle Zustandsvariablen wahr zu machen, die für die Position des Roboters stehen. Nach Ausführen der einzigen legalen Aktion „`move`“ verändert sich der geschätzte Abstand nicht, und der relaxierte Plan bleibt [`move`, `grab`, `drop`]. Mit dem Erreichen des Goldes in Feld $C$ kommt ein weiterer anwendbarer Operator „`grab`“ hinzu. Nach dem Aufheben des Goldes reduziert sich der geschätzte Abstand auf 2 [`move`, `drop`], wohingegen er weiterhin bei 3 bleibt, wenn der Roboter weiterläuft [`move`, `grab`, `drop`]. Leider ist der von $f_{JO}$ berechnete Heuristikwert dennoch gleich für beide Aktionen. Grund ist der Abstand zum Ziel mit negativem

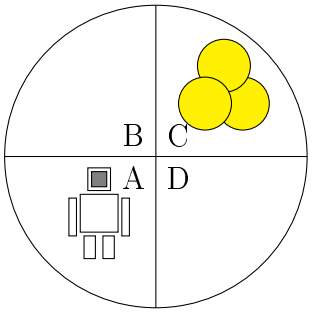


Abbildung 6: Maze

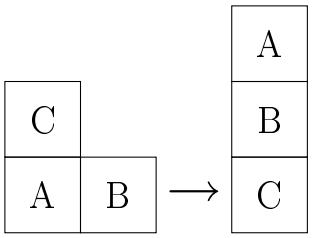


Abbildung 7: Blocksworld

Nutzen. Das Spiel endet, wenn `gold_a` erreicht ist. Die Zielformel für einen Nutzen von $-50$ enthält das Disjunktionsglied `gold_a` $\wedge$ `gold_d`. Im relaxierten Plan können beide Variablen gleichzeitig wahr werden. Beim Aufheben des Goldes mit einem Abstand von 2 [`drop`, `move`], beim Weiterlaufen mit 3 [`move`, `grab`, `drop`]. Dadurch berechnet $f_{JO}$ für beide Aktionen den selben Wert $\frac{1}{2} \cdot (-50) + \frac{1}{2} \cdot 50 = 0 = \frac{1}{3} \cdot (-50) + \frac{1}{3} \cdot 50$.

Dieses Problem bleibt bestehen, bis sich der Roboter mit Gold in Feld $A$ befindet. Dann ist der Heuristikwert 1 [`drop`] für einen Nutzen von 50 und 2 für $-50$. Die Evaluierungsfunktion wird bei Maze demnach erst dann informativ, wenn die Suche die Terminalzustände schon fast erreicht hat.

Interessant ist noch, dass die FF-Werte den tatsächlichen Zügen entsprechen, wenn das gleiche Spiel mit `(legal ?player (move ?x ?y) ...)` statt einer immer anwendbaren `move`-Aktion formalisiert wird. Allerdings bleibt das Problem bestehen, dass Feld $D$ „`robot_d`“ relaxiert wahr bleibt, und somit das Gold durch `drop` in mehreren Feldern gleichzeitig abgelegt werden kann.

## 5.2 Blocksworld

Blocksworld (Abb. 7) ist ein klassisches Planungsproblem. Auch hier kann das Programm das Spiel komplett berechnen und so zur richtigen Lösung gelangen. Allerdings gibt es hier keine negative Interaktion mit dem Zielzustand, da sich Ziele mit negativem Nutzen nur durch Ablauf der zur Verfügung stehenden Züge erreichen lassen. Der Heuristikwerte für `unstack_c_a` ist 2 mit Plan [`stack_b_c`, `stack_a_b`], der für `stack_b_c` aber 3 [`unstack_b_c`, `unstack_c_a`, `stack_a_b`]. Somit ergeben sich hier schon nach einem Schritt die richtigen Prioritäten in der Aktionswahl.

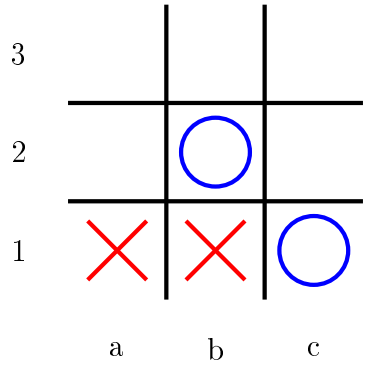


Abbildung 8: Tic-Tac-Toe

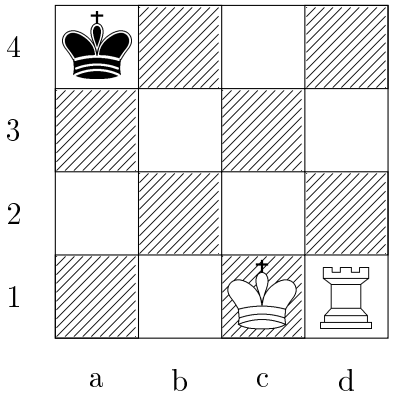


Abbildung 9: Minichess

## 5.3 Tic-Tac-Toe

Tic-Tac-Toe ist das wohl berühmteste Minispiel (Abb. 8), bei dem zwei Spieler X und O versuchen, auf einem $3 \times 3$ Feld eine Reihe aus X oder O zu markieren. Auch dieses Spiel kann von CoolerJo komplett berechnet werden. Die $f_{JO}$-Werte bei leerem Spielfeld sind $\frac{1}{2} \cdot 50 + \frac{1}{3} \cdot (-50) = \frac{25}{3}$ für jedes Feld. Nach 1. Xa1-Ob2 haben die Felder a2, a3, b1, c1 jeweils Schätzwerte von $\frac{1}{1} \cdot 50 + \frac{1}{2} \cdot (-50) = 25$, die anderen (b3, c2, c3) jeweils Schätzungen von $\frac{1}{2} \cdot 50 + \frac{1}{2} \cdot (-50) = 0$. Nach 2. Xb1-Oc1 ergibt sich die Stellung aus Abbildung 8. Dabei ergeben sich folgende Heuristikwerte der $f_{JO}$:

- a2: $\frac{1}{1} \cdot 50 + \frac{1}{1} \cdot (-50) = 0$
- a3: $\frac{1}{1} \cdot 50 + \frac{1}{2} \cdot (-50) = 25$
- b3: $\frac{1}{2} \cdot 50 + \frac{1}{1} \cdot (-50) = -25$
- c2: $\frac{1}{2} \cdot 50 + \frac{1}{1} \cdot (-50) = -25$
- c3: $\frac{1}{2} \cdot 50 + \frac{1}{1} \cdot (-50) = -25$

Somit würde 3. Xa3 gespielt, womit die Bedrohung der Linie für Spieler O abgewendet wird. CoolerJo versucht weiterhin seine begonnenen Reihen fertigzustellen. Bei Heuristikberechnungen der ersten Ebene (und ohne den Minimax-Suchbaum tiefer zu evaluieren) werden so schon vernünftige Züge ausgewählt. Allerdings können Bedrohungen für Spieler O, wie beispielsweise nach 1.Xa1-Ob2 2.Xc3, wo nach a3 oder c1 Spieler X mit einer Doppeldrohung das Spiel gewinnen kann, erst bei tieferer Analyse erkannt werden.

## 5.4 Minichess

Minichess ist in Abbildung 9 zu sehen. Weiß kann in zwei Zügen (1.Kb2 - Kb4, 2.Td4#) gewinnen. Durch die Domänenanalyse werden hier von CoolerJo leider so viele Zustandsvariablen erzeugt, dass die Erstellung der Operatoren aus Speicherplatzmangel misslingt.

## 5.5 Ergebnis

Die betrachteten Spiele konnten das Potential, welches in der Evaluierungsfunktion $f_{JO}$ steckt, zeigen. Oft sind die nach einem Suchschritt berechneten Heuristikwerte bereits akzeptabel.

Aber auch Schwächen des Ansatzes wurden deutlich. Durch die Relaxierung können mehrere Zustandsvariablen gleichzeitig wahr werden, wodurch wie im ersten Spiel schlechte Schätzwerte berechnet werden. Ein weiteres Problem stellen komplexere Spiele wie Minichess dar. Die Repräsentation des Zustandsraums muß weiter optimiert werden, um die an und für sich guten Ergebnisse nutzen zu können. Sollte sich herausstellen, dass die in Kapitel 4.3 vorgestellten Verbesserungen nicht ausreichen, kann immer noch nach Heuristiken gesucht werden, die auf Prädikatenlogik erster Stufe arbeiten.

# 6 Fazit

Allgemeine Spiele können mit Hilfe von heuristischer Suche von Computern gespielt werden. Damit diese Programme allerdings bei einem GGP-Wettbewerb erfolgreich teilnehmen, ist es notwendig, dass die einzelnen Module des Programms sehr effizient programmiert werden. Für das im Rahmen dieser Arbeit implementierte Programm wurden Verbesserungen vorgeschlagen, um besser mit großen Zustandsräumen komplexerer Spiele klarzukommen.

In Spielen ist die Welt sehr dynamisch, und bei der Problemmodellierung ist eine Relaxierung, welche davon ausgeht, dass bereits erreichte Teilziele auch erreicht bleiben, etwas zu optimistisch. Die in der Evaluierungsfunktion $f_{JO}$ berechneten Abstände können mit einer beliebigen Heuristik angenähert werden. Und möglicherweise werden mit anderen Heuristiken noch bessere Ergebnisse erreicht.

Andere Ansätze zum Lösen genereller Spiele sind ebenso denkbar. Der Sieger der letzten Wettbewerbe, Cadia Player [Fin07], verwendet Stichproben, und wählt seinen Zug auf Grund der besten gefundenen zufälligen Zugfolge aus. Diese sogenannten Monte-Carlo-Programme sind besonders dann erfolgreich, wenn sehr viele Stichproben zum Vergleich vorliegen. Bei kurzer Vorlaufzeit eines vorher unbekannten Spiels kann deshalb keine allzu große Stichproben-Datenbank erstellt werden. Dennoch gibt der Erfolg dem Cadia Player recht.

Für den Computer stellt das Spielen allgemeiner Spiele nach wie vor eine große Herausforderung dar. Heuristische Suche ist ein vielversprechender Ansatz, um diese zu lösen, und ich halte es für wahrscheinlich, dass künftige Wettbewerbe von Programmen gewonnen werden, die dem hier vorgestellten Ansatz nachkommen.

# Glossar

**AAAI** Association for the Advancement of Artificial Intelligence. 6

**Atom** auch atomarer Satz, Ein Relationsbezeichner angewandt auf $n$ Terme, wobei $n$ der Stelligkeit des Relationsbezeichners entspricht. 10, 11, 32, 46, 47

**Datalog** An Prolog angelehnte Datenbank-Programmiersprache, welche GDL als Grundlage dient. 8, 12, 13

**Datalogregel** Implikation mit den in Kapitel 2.3.1 beschriebenen Einschränkungen. 10–12, 31

**DNF** disjunktive Normalform. 39, 40

**Funktion** Beziehung zwischen Termen, der ein Wahrheitswert zugeordnet werden kann. 10, 14, 16, 47

**Funktionsbezeichner** Konstante, die für eine Funktion steht. 9

**GDL** Game Description Language. 5–10, 13, 14, 17, 19, 21, 31–33, 35, 36, 39, 46

**GGP** General Game Playing. 5, 6, 8, 13, 19–22, 24–27, 30, 31, 34, 37, 40, 41, 45, 46

**GM** Game Master. 6, 37, 40

**Hammingdistanz** Abstand $d$ zwischen zwei Literalbelegungen $K$ und $L$. 26

**HTTP** Hyper Text Transfer Protocol. 6

**Java** Objektorientierte Programmiersprache `http://java.sun.com`. 5, 37

**KIF** Knowledge Interchange Format. 10, 13, 38, 47

**Lisp-S-Ausdruck** Entweder ein Lisp-Atom, oder ein cons-Paar. Im GDL Kontext vereinfacht gesagt: es gibt nur geklammerte Ausdrücke. 13

**Literal** Ein Atom oder dessen negierte Form. In GDL z.B. `(true (cell a 2 blank))`, oder auch `(not (true (cell b 1 white_rook)))`. 10, 11, 13, 26, 35

**Nullsummenspiel** Ein Spiel, bei dem die Summe der Nutzen aller Spieler in jedem Terminalzustand 0 ergibt. Bei klassischen Spielen wird dieser oft mit $+1$ für den Sieger und $-1$ für den Verlierer modelliert. 7, 22, 24, 25

**Nutzen** Zahl, die eine Aussage darüber macht, wie erstrebenswert ein Zustand ist. Beim GGP liegt der Nutzen zwischen 0 und 100. 7, 17, 20, 22, 31, 35, 39, 41, 42, 46

**Objekt** In der Spielbeschreibung definierte Konstante. 9, 13–15, 47

**PDDL** Planning Domain Description Language. 5

**Prädikatenschema** Prädikat mit Stelligkeit. Instantiiert man die Stellen mit Objekte, erhält man eine Zustandsvariable. Steht entweder für eine Menge von Relationen (Atomen) oder von Funktionen. 47

**Relationsbezeichner** Konstante, die für eine Relation steht, mit der zugehörigen Stelligkeit. Wendet man den Relationsbezeichner auf Objekte entsprechend der Stelligkeit an, erhält man ein (Relations-)Atom. 9, 10, 12, 46

**Term** Ein Term ist entweder ein Objekt, eine Variable oder eine Funktion. 9, 10, 46

**Variable** (auch Objektvariable), Platzhalter für ein Objekt. In KIF-Notation beginnen Variablen mit einem „?“. 9, 47

**Zustandsvariable** (auch atomares Prädikat), Ein Prädikatenschema, bei dem alle Stellen mit Objekten instantiiert sind.. 28, 37, 39, 47

# Literatur


[Byl94] Bylander, Tom: *The computational complexity of propositional STRIPS planning.* Artificial Intelligence, 69:165–204, 1994.

[Clu07] Clune, James: *Heuristic Evaluation Functions for General Game Playing.* In: *AAAI*, Seiten 1134–1139. AAAI Press, 2007.

[Fin07] Finnsson, Hilmar: *CADIA-Player: A General Game Playing Agent*, 2007.

[GF+92] Genesereth, Michael, Richard Fikes et al.: *Knowledge Interchange Format*, 1992.

[GL05] Genesereth, Michael und Nathaniel Love: *General Game Playing: Overview of the AAAI Competition*, 2005.

[GM07] Giunchiglia, Enrico und Marco Maratea: *Planning as Satisfiability with Preferences.* 2007.

[GNT04] Ghallab, Malik, Dana Nau und Paolo Traverso: *Automated Planning — Theory and Practice*, 2004.

[Hel06] Helmert, Malte: *The Fast Downward Planning System.* Journal of Artificial Intelligence Research, 26:191–246, 2006.

[HN01] Hoffmann, Jörg und Bernhard Nebel: *The FF Planning System: Fast Plan Generation Through Heuristic Search.* 14:253–302, 2001.

[LHH+08] Love, Nathaniel, Timothy Hinrichs, David Haley, Eric Schkufza und Michael Genesereth: *General Game Playing: Game Description Language Specification*, 2008.

[RN95] Russel, Stuart und Peter Norvig: *Artificial Intelligence — A Modern Approach.* Prentice Hall, 1. Auflage, 1995.

[RW08] Richter, Silvia und Matthias Westphal: *The LAMA Planner.* 2008.